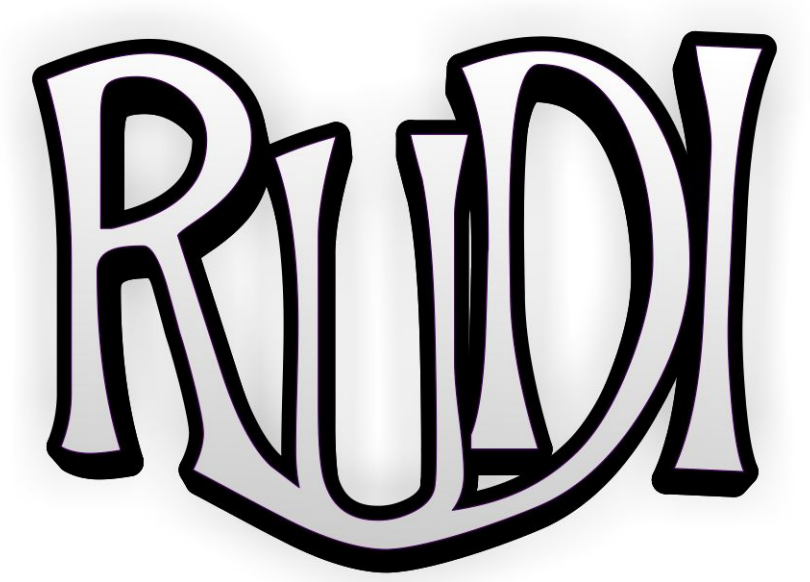


a game development workshop

by Christian Behrenberg

- deutsche Fassung -

Version: 0.1 (letzte Änderung: 24.12.2007)

Inhaltsverzeichnis

Autor: Christian Behrenberg
E-Mail: christian@behrenberg.de
http://www.christian-behrenberg.de

Vorwort - oder: die nackte Wahrheit

Vor ca. 4 Jahren trafen sich eine Hand voll jüngere und ältere Anhänger der Spieleprogrammierung in Bochum Innenstadt in den Räumen der Ruhr-IT zum ersten „3D Gamestudio Stammtisch“. Damals kam die A5 raus und Spiele, die damit gemacht wurden, wurden belächelt. Bis heute hat sich das Vorurteil gehalten, dass Spiele, die mit 3D Gamestudio gemacht werden, „zusammengeklickt“ sind. Zwar bietet das 3D Gamestudio mithilfe von Templates eine einsteigerfreundliche Möglichkeit, zu schnellen Ergebnissen zu kommen, aber mit der Zeit hat sich gezeigt, dass man mit der A4, dann A5, A6 und jetzt mit der A7 game engine komplexe Spiele mit einer tollen Grafik und wunderbaren Spielspaß machen kann, die sich gut verkaufen und auch Preise einheimsen, wie z.B. jüngst das offizielle Spiel zur dt. TV-Serie Stromberg, das mit dem deutschen Entwicklerpreis ausgezeichnet wurde.

Nunja, damals waren die Jungs der Ruhr-IT („Guerilla Tactix“) Veranstalter des Stammtisches. Dabei waren noch einige bekanntere Vertreter aus dem 3DGS Forum, die Jungs der Ruhr-IT (u.a. Olliver Berg) und eben ich. Man hat sich ausgetauscht, angeregt diskutiert und dann hatten wir die Idee, dass man es doch wohl schaffen sollte, innerhalb von einem Monat – besser gesagt: 30 Tage – mit einem Team, bestehend aus guten Programmierern, Grafikern und Designern was auf die Beine stellen kann, was auch „gut“ ist und Spaß macht. Das war im Oktober und da ja damals zu der Zeit Weihnachten anstand, haben wir uns auf ein Weihnachtsspiel geeinigt. Heraus kam dabei ein recht umfangreiches Arsenal an Ideen für ein Spiel, dass „RUDI“ heißen sollte...

Um es kurz zu fassen und um ehrlich zu sein – die Entwicklung des Spiels war ein Desaster. Demotivation, Unzuverlässigkeit und ein insgesamt zu lockeres und daher unkoordiniertes Team waren die drei Grundpfeiler des Misserfolgs. Zuletzt gab es nur noch Olliver und mich, die das Projekt weitergeführt hatten. Weil Weihnachten längst vorbei war, wollten wir das Spiel noch „transformieren“ und anders verwerten. Irgendwann kollabierte das Projekt und ich hatte dann auch nicht mehr den Drang, es fertigzustellen – verständlicherweise.

Vier Jahre später habe ich mich dann dazu entschlossen, mit der A7 bis Weihnachten das Ding wiederzubeleben. Den Entschluss und die ersten Codezeilen habe ich am 10. Oktober '07 ge- bzw. verfasst - hatte also genau 75 Tage dazu Zeit, das Projekt auf die Beine zu stellen. Nach 75 Tagen kann ich nun das Resümee ziehen, das es machbar ist! Zwar habe ich für das komplette Spiel das doppelte der veranschlagten Zeit von damals gebraucht, allerdings kann ich auch behaupten, dass die visuelle Qualität des Spiels das Erreichbare der A4/A5 zu jener Zeit um ca. das vierfache (grob geschätzt) übersteigt – man denke nur an Landschaften aus Blöcken... kein Kommentar.

Aber ich muss auch sagen, dass mir das Projekt gezeigt hat, wie bemitleidenswert die hoffnungslose Romantik von Spieleentwicklern ist (zu diese Art von Leuten zähle ich mich, glaube ich, auch). Der Workshop – der von Anfang an geplant war - war eigentlich als Adventskalendar gedacht. Es sollte jeden Tag ein neues Kapitel herauskommen. Also am Ende 24 Kapitel, die alle zu dem Spiel führen. Eine nette Idee.. fanden viele. Es hat sich herausgestellt, dass dies nicht machbar war. Ich wollte 3 Level anstatt nur einem in das Spiel einbauen. Es hat sich herausgestellt, dass dies total überzogen war. Ich wollte eigentlich noch den Workshop während des Schreibens ins Englische übersetzen, aber es hat sich gezeigt, dass das utopisch war. Trotz all dieser Dinge hat aber der Ehrgeiz ausgereicht, um es zu schaffen. Warum aber „hoffnungslose Romantik“ und warum ist das bemitleidenswert?

Das zeigt in der Erkenntnis, dass 90% aller Leute, die gesagt haben, sie wollten helfen, es nicht getan haben oder es einfach nicht konnten. Am Ende gingen aus der anfangs recht großen Schar hilfsbereiter Menschen nur ein, zwei Leute hervor und an dieser Stelle muss ich mich besonders bei Felix Caffier bedanken (im GS Forum als Broozar bekannt), der wirklich viel zum Projekt beigetragen hat. Ich glaube, ohne ihn wäre das ganze Ding nichts geworden. „Leider“ musste ich mich selber sehr viel um die Grafik kümmern (Hilfe!) und hätte ich mich auch noch um die vielen Charaktere kümmern müssen (inklusive modeling, texturing und Animation), wäre ich heute noch nicht mal ansatzweise so weit, das Spiel und den Workshop zu veröffentlichen.

Auch geht mein Dank an Caity für die Skizzen & Ideen, Sebastian für die noch nicht verwendeten Models ;-) , Nils für sein Shader-Know-How, Timo für die vielen privaten GameEdit Betas – ich glaube sonst wäre das Level nur Schrott – Spike für vieles, auch wenn leider, leider noch nicht viel dabei rumgekommen ist (wird schon!), Martin für das Gegenlesen, André, Dennis, Dirk und Simon für den tollen Abend in Essen, Kevin für die wunderbare Musik und Ed, weil er in letzter Minute das Spiel noch veredelt hat.

Liebe Grüße und viel Spaß mit dem Spiel, dem Source und dem Workshop - Christian („*HeelX*“)

Kapitel 1: Einführung in den Workshop

Herzlich Willkommen zum Game-Development-Workshop des Open-Source Computerspiels „RUDI“ (v0.1)!

Der Workshop und das Projekt

Dieser Workshop basiert auf dem Spiel „RUDI“, bzw. andersherum: wenn Sie den Workshop durcharbeiten, werden Sie das Spiel erstellt haben – so oder so ähnlich. Auf jeden Fall sind beide Dinge zeitgleich entstanden, wobei sie als eine Einheit oder separat gesehen werden können, je nachdem was Sie selber für Erwartungen haben. Wenn Sie nur daran interessiert sind, zu wissen, wie das Spiel „gebaut“ wurde, dann können Sie den Workshop gemütlich lesen um hier und da einen tieferen Einblick zu erhalten. Wenn Sie jedoch RUDI nachbauen wollen (mit den mitgelieferten Inhalten oder mit eigenen), dann können Sie sich vom Text an die Hand nehmen lassen und aktiv durcharbeiten. Das Spiel liegt als Source in der finalen Version vor. Die Entwicklung des Spiels ist am Workshop wiederzuerkennen, da einige Dinge früh programmiert wurden und dann später abgewandelt werden mussten – in der finalen Version des Sourcecodes sieht man das natürlich nicht.

Es kann sein, dass der Workshop einige Dinge des Source-Codes nicht dokumentiert.

Ich kann darauf keine Gewährleistung geben, weil Anwendersoftware, insbesondere Computerspiele, einem ständigen Wandel unterliegen. Es können Grafiken geändert, hinzugefügt oder entfernt werden oder auch eben der damit verbundene Quellcode. Der Workshop ist relativ groß geworden, sodass nicht immer jede Zeile geänderter Code auch seinen Weg in den Workshop finden konnte (z.B. Konstanten). Allerdings sollten Sie keine Probleme beim Abgleich von Workshop und Source haben – alle wichtigen und großen Änderungen sind dokumentiert!

In diesem Workshop wird die content-Creation nicht behandelt!

Als content-Creation bezeichnet man den Produktionsprozess, der dafür verantwortlich ist, dass Sie im Spiel 3D Objekte mit Texturen, Shadern und Animationen haben. Dies betrifft auch die Erstellung von 2D Grafiken und dem Design von z.B. den Menüs des Spiels. Die content-Creation umfasst auch den Designprozess des Spiels. Vor allem,

wenn ein Spiel neu entsteht, muss eine Menge Zeit in den Designprozess gesteckt werden, damit das Spiel einen einheitlichen und gut aussehenden Stil besitzt. Die Zeit war knapp, weshalb sowohl der Designprozess als auch die content-Creation zeitgleich entstanden – glücklicherweise hat es einigermaßen geklappt. Da Ihnen eventuell auch nicht die Programme, Ressourcen und Fähigkeiten zur Verfügung stehen wie uns, wäre es wahrscheinlich vergebens gewesen, die content-Creation ausführlichst zu dokumentieren, weil es eben nur wenige Menschen betrifft. Programmierung ist allerdings erlernbar und – in meinen Augen - durchaus wichtiger zu zeigen, als einzelne Grafiken anzufertigen. Damit Sie aber nicht die ganze Zeit trockenem Stoff ausgesetzt sind, hat Felix Caffier einen kleinen Exkurs zur Verfügung gestellt, indem er zeigt, wie Sie einen Schneemann modellieren und texturieren. Das Modell findet auch Verwendung im Spiel!

Die Texturen der Models sind nicht repräsentativ für moderne Licht & Schatten Szenen!

Ich habe mich dazu entschlossen aufgrund der Perspektive und der Entfernung der Kamera zu den Objekten mit dem sogenannten „texture baking" Verfahren zu arbeiten, um eine schattierte Umgebung zu besitzen. Dabei wird ein Objekt modelliert und in einem Renderprogramm beleuchtet. Der dabei entstehende Schatten auf dem Objekt oder auf anderen Objekten wird bei dem Verfahren dann in eine Textur „mitgebacken". Das bedeutet, die Modelle erhalten eine neue Textur, die sowohl die Textur als auch den Schatten zusammenfasst. Dies bedeutet aber auch gleichzeitig einen höheren Texturverbrauch, weil nun jeder Pixel der Textur nur einmal auf dem Model vorkommen darf! Für moderne Spiele, in denen viel Wert auf Details gelegt wird – wie z.B. first person shooter – ist dieses Verfahren absolut nicht geeignet! Sie benötigen desweiteren auch geeignete Tools um „texture baking" ausführen zu können.

Desweiteren verwende ich für die Texturen aller Models DDS Texturen. DDS Texturen sind komprimierte Bilder, die sehr schnell in den Grafikspeicher geladen werden können und im Vergleich zu Bitmaps oder TGA Bildern recht klein gehalten sind. Im Zusammenhang mit dem „texture baking" wird das Defizit von höherem Speicherverbrauch wieder wett gemacht, allerdings empfehle ich dieses Verfahren nur dann einzusetzen, wenn sie absolut sicher sind, dass es die grafische Darstellung ihres Spiels nicht beeinträchtigt! Benutzen Sie lieber kachelbare Texturen und ein zweites UV set für eine shadowmap bei statischen Schatten oder shaderbasierte Softshadows für dynamische Schatten. Für die Erstellung von DDS Texturen empfehle ich den *ATI Compressonator*.

Das Spiel ist noch nicht fertig!

Wie bereits erwähnt, hatte ich vor, wesentlich mehr Levels zu bauen und mehr Gameplayelemente zu programmieren. Allerdings entfällt auf eine Stunde schnelles, hochwertiges sauberes Programmieren etwa 2-3 Stunden Verschriftlichung. Es ist nicht geplant jetzt aufzuhören. Vielmehr wollen wir weitermachen und nächstes Jahr mehr Levels bauen und das Spiel erweitern. Inwiefern der Workshop erweitert wird, hängt alleine von der Resonanz und von der Einbringung von den Lesern ab!

Bringen Sie sich ein!

Es gibt 100 Möglichkeiten, sich in das Projekt mit einzubringen. Erzählen Sie Freunden und Verwandten davon, spielen Sie das Spiel bis zum umfallen, melden Sie Fehler des Spiels, schreiben Sie in Ihrem Blog darüber, verlinken Sie meine Seite, spenden Sie über Paypal ein oder zwei Euro (ID: christian@behrenberg.de) oder produzieren Sie selber content für das Spiel, designen Sie Levels.. Sie sehen, es gibt wirklich viel zu tun! Starten Sie noch heute! Studieren Sie den Code und erweitern Sie das Spiel und senden Sie mir den modifizierten Code, dann baue ich es in die offizielle nächste Version ein. Go go go!

Schrecken Sie nicht vor dem Workshop zurück!

Es gibt Leute, die sich schon bedrängt fühlen, wenn ein Dokument mehr als 10 Seiten besitzt.. dieser Workshop besitzt sogar an die 200 Seiten und er wird immer größer... um Himmels Willen! Aber keine Angst, er beißt Sie nicht. Vielmehr haben Sie dazu Gelegenheit, hiermit den – soweit ich informiert bin – ersten großen Workshop für A7 und Lite-C zu lesen. Wollen Sie mit dem 3D Gamestudio ein aufstrebender Stern am Himmel der Gamedesigner werden? Lesen Sie den Source, spielen Sie das Spiel, arbeiten Sie den Workshop durch, schlagen Sie im Handbuch nach – und bald sind Sie selber im Handumdrehen in der Lage Spiele selber zu schreiben! Garantiert!

(Anmerkung: aus zeitlichen Gründen fehlt die Dokumentation der Soundeffekte. Sie wird in den Dateien sound.c und sound.h geleistet. In der nächsten aktualisierten Version des Workshops wird das Kapitel hinzugefügt. Allerdings muss ich auch sagen, dass hier wirklich nur ganz rudimentär und einfach die Soundeffekte eingefügt worden sind.)

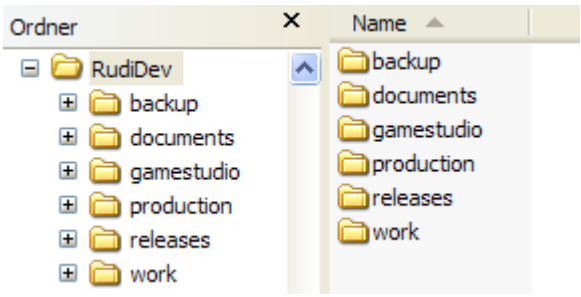
Die Struktur des Projektes

Im Folgenden wollen wir bereits den ganzen Entstehungsprozess überspringen und auf die Projektstruktur des finalen Spieles eingehen, indem wir die Organisation des Source-Codes betrachten. Im Gegensatz zur konventionellen Software-Entwicklung kann man bei Unterhaltungssoftware, besonders bei Spielen, keine allgemeingültige Struktur von solchen Projekten empfehlen, da es immer auf die Eigenheiten des Spiels ankommt und welche Technik man verwendet.

Leider hat jeder eine eigene Vorstellung davon, wie man Projekte anlegt, was man für Programme zur Unterstützung hinzuzieht, etc. Alle Methoden haben ihre Vor- und Nachteile und ich möchte an dieser Stelle eine Vorgabe – oder besser: eine Empfehlung – machen, womit die freie / ungeordnete Entwicklung von Spielen relativ entwicklerfreundlich sein kann. Mit „frei“ oder „ungeordnet“ meine ich den Verzicht auf spezielle Projektmanagementsoftware und Versionskontrollsysteme wie z.B. Subversion, CVS oder Alienbrain. Ich will nur an dieser Stelle darauf hinweisen, dass es so etwas gibt und ein Projekt durchaus sicherer und geordneter macht, wir an dieser Stelle aber darauf verzichten wollen.

Wir gehen davon aus, dass Sie sich bereits „etwas“ auf Ihrem PC bezüglich Ihrer Arbeiten organisiert haben und einen Ordner für solche Projekte angelegt haben. Ich neige dazu, meine Projekte nach Jahren zu sortieren. In meinem Ordner „projects/2007“ habe ich einen Ordner namens „RudiDev“ angelegt, um die Entwicklung für RUDI zu organisieren. Sie sollten erwägen, den Projekt-Ordner nicht auf der Festplatte zu installieren, auf der Ihr Betriebssystem installiert ist, z.B. D:\ , falls Windows auf C installiert ist. Wenn nämlich Windows crasht und eventuell neu installiert oder gar die Festplatte formatiert werden muss, sind Ihre Projektdaten komplett weg oder erheblich gefährdet. Externe Festplatten eignen sich auch dafür, jedoch sind die Zugriffs- und Datentransferzeiten via USB nicht besonders hoch.

In dem Projekt Ordner sind diverse andere Ordner organisiert, inklusive einem Arbeitsordner, indem sich die aktuelle Arbeitsversion des Spiels befindet, die sogenannte „working Copy“. Die Working Copy besteht neben anderen Ordnern, die so angelegt sind:



Ich sichere nach jeder erfolgreichen Erweiterung des Spiels oder anderer Daten den entsprechenden Ordner in dem Verzeichnis „backups“. Dokumente wie Konzepte, Skizzen, Listen, Email Verkehr, Kontaktdaten, bookmarks usw. speichere ich im Verzeichnis „documents“. Das Spiel wird mit 3D Gamestudio entwickelt. Ich habe einen Ordner angelegt, indem sich die aktuell stabilste veröffentlichte Version befindet, mit der ich arbeiten und rechtzeitig das Projekt veröffentlichen kann. Die Gamestudio Version befindet sich im Ordner „gamestudio“.

Berüchtigt – wie ich finde – ist die Tatsache, dass sehr viele Hobbyentwickler ihre Spielinhalte (also Texturen, Modelle, Levels, ...) direkt im Spielverzeichnis erstellen und speichern. Das ist aber keine gute Idee: die meisten Texturen beinhalten in ihren original-Dateien zum Beispiel sogenannte Layer (an und ausschaltbare Schichten in Ihrer Malsoftware, wie z.B. Adobe Photoshop) oder haben eventuell auch eine größere Auflösung als die fertigen Texturen im Spiel. Das gleiche gilt auch für 3D Modelle: meistens werden die 3D Modelle in anderen Programmen und Formaten editiert und werden dann später ins Zielformat konvertiert. Das kann man nun auf alle anderen Inhalte wie Musik, Sounds, Terrains, usw. übertragen. Es stellt sich heraus, dass es einfach nicht sinnvoll ist, die Originaldaten im Spielverzeichnis zu speichern.

Deshalb verwalte ich den „rohen“ content, aus dem die finalen Dateien für das Spiel hervorgehen, im Ordner „production“. Wenn ich also z.B. ein neues Modell fertig habe, ist es im Ordner „production“ gespeichert und

zusätzlich im Arbeistverzeichnis.

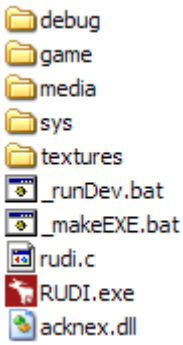
Wenn ich eigene Versionen, Beta-Versionen oder die finale Version des Spiels kompiliert habe, speichere ich jede Version davon im Ordner „releases“.

Der Ordner, indem wir das Spiel an sich anfertigen, nennen wir „work“ (für „working copy“). In diesem Ordner befindet sich das Spiel im Entwicklungsstatus, im Idealfall immer in einem spielbaren Zustand. Den Quellcode, den Sie sich zu RUDI herunterladen können, ist „fast“ eine exakte Kopie des Quellcodes. Es wurden einige Dateien entfernt, die im Rahmen meines Projektordners das Spiel starten und kompilieren. Darauf gehen wir jetzt ein.

Starten des Spiels

Dem Source ist bereits eine vorkompilierte EXE Datei beigelegt. Wenn Sie diese starten, wird das Spiel automatisch gestartet, ohne den Programmcode zu kompilieren. Wenn Sie allerdings Änderungen am Source-Code vornehmen, müssen Sie das Spiel immer per Hand über den Engine-Compiler starten.

Sie können nun im bereits angefertigten gamestudio-Ordner den WED öffnen und die Hauptdatei starten. Dies ist aber umständlich! Zur Abhilfe habe ich mir einige *.bat Dateien gebaut, die gewissen Aufgaben übernehmen.



Eine Datei namens „_runDev.bat“ startet zum Beispiel das Spiel, indem die acknex.exe, die sich im übergeordneten gamestudio-Ordner befindet, mit der Hauptdatei („rudi.c“) aufgerufen wird. Der Inhalt der Datei sieht so aus:

```
@echo off
call ..\gameStudio\acknex.exe "rudi.c" -diag
cls
```

Zudem wird auch noch die Diagnose mit -diag eingeschaltet. Die Datei „_makeEXE.bat“ ist eine weitere nützliche Datei, die es erlaubt, aus dem Quellcode eine kompilierte, ausführbare Datei zu erstellen. Der Code der Datei sieht so aus:

```
@echo off
call ..\gameStudio\acknex.exe "rudi.c" -exe -nc -cc
call ..\gameStudio\wed.exe -p ..\work\rudi.exe
cls
```

Dabei wird einerseits die *.exe Datei erzeugt und eine dazu passende „acknex.dll“. Kopieren Sie sich den kompletten Sourcecode in das work-Verzeichnis und starten Sie das Spiel über die rundev.bat und Sie spielen das Spiel, kompiliert aus dem source heraus!

Die Organisation des Sourcecodes

Im Work Ordner gibt es auf der Hauptebene (prinzipiell) nur die Hauptdatei „rudi.c“. Sie bietet im Prinzip den Einstieg ins Spiel für den Compiler. Der Rest des Spiels ist in den zahlreichen Ordnern und Unterordnern „versteckt“. In den folgenden Kapiteln wird immer mal wieder auf die Bedeutung der Ordner zurückverwiesen, und was sie für Unterordner besitzen. An dieser Stelle möchte ich eben kurz die Hauptordner beschreiben.

Der Debug-Ordner enthält einige Dateien und Code, die beim debuggen – also dem Entfernen von Fehlern – hilfreich sind. Wir werden in diesem Zusammenhang ein kleines, rudimentäres Debug-System in das Spiel integrieren. Weil Sie den Source-Code vor sich haben, bzw. weil ich in dem Workshop recht zielstrebig und (hoffentlich) fehlerlosen Code schreibe, werden wir nur hier und dort exemplarisch debuggen. Zumeist werden Sie selber nur dann darauf zugreifen, wenn Sie selber Code für RUDI schreiben oder den Quellcode erweitern.

Im game-Ordner finden Sie das eigentliche Spiel. Dort drin steckt die ganze Spielmechanik, die Grafik, die Levels, Effekte,... einfach alles. Im Ordner sys finden Sie hingegen das sogenannte „Framework“, also ein Rahmengerüst, dass einige system-typische Dinge tut. Wir werden in dem Workshop nur ein sehr kleines, rudimentäres Gerüst bauen, das nicht viel kann, aber das Spiel am Laufen hält. In der Regel sollte man versuchen, das Spiel und ein umschließendes technisches System – das Framework – voneinander zu trennen. Aber darauf gehen wir später nocheinmal ein.

Wir benutzen im Spiel u.a. auch Musik und Videos, die wir von der Festplatte streamen wollen (dabei wird immer nur ein kleiner Teil der Datei in den Arbeitsspeicher geholt, währenddessen die ganze Zeit auf die Datei zugegriffen wird). Wir richten für diese Dateien einen eigenen Ordner namens „media“ ein, in den wir diese Dateien platzieren.

Der Ordner „textures“ ist im Prinzip kein Teil des Sources, da wir dort Texturen aufbewahren, die von WMP (Level-) Dateien geteilt werden. Wenn Sie in ihrem Spiel keine Block-Level benutzen, benötigen Sie diesen Ordner nicht. Da ich aber aus Vollständigkeitsgründen die relative Zuordnung von Texturverzeichnissen für WMP Dateien erwähne, ist der Ordner mit einigen temporären Dateien mit im Source inbegriffen.

Weitere Details zu den Ordnern finden Sie im weiteren Workshop.

Kapitel 2: Das Projekt und die Spielidee

Wir wollen uns nun damit befassen, festzuhalten, was wir für ein Spiel überhaupt entwickeln wollen. Natürlich ist dieses Kapitel keine freie Diskussion, weil dies auch recht schwierig zu gestalten ist, wenn der Autor nur indirekt mit dem Leser sprechen kann. Deshalb präsentiere ich eher die Idee und gehe Schritt für Schritt die wichtigen Dinge durch, die umso wichtiger sind bei der Gestaltung und Planung eines Spiels. Wobei der Begriff „Spiel“ eher einzugrenzen ist: wir entwickeln zwar ein Spiel, aber im engeren Sinne ein Casual Game.

Der Plan – ein casual game

Casual Games sind Gelegenheitsspiele, die zum schnellen und vor allem einfachen Spielen anregen. Kurze Spielzeit und vor allem der Spielspaß stehen hierbei im Vordergrund. Casual Games findet man häufig auf Webseiten, z.B. als Flashspiele oder JAVA Applets. Viele bekannte Handyspiele sind ein sehr gutes Beispiel für das, was ein Casual Game leisten muss: sie sind kurzweilig, schnell zu spielen, jeder weiß intuitiv, wie das Spiel gespielt wird und ist daher auch leicht zu erlernen. Die meisten Casual Games basieren auf Brett- und Kartenspielen, Logikspiele; kleinere Actionspiele, Arcade- oder Reaktionsspielchen und eben auch Spieleklassiker. Handy- oder Handheldklassiker wie „Snake“ oder „Tetris“ sind sehr gute Beispiel dafür.

Wir wollen in diesem Workshop auch ein solches Casualgame herstellen. Alles andere – größere - ist nicht wirklich schaffbar. Zwar kann man Workshops über „größere“ Spiele (berühmt und berüchtigt sind hierbei Shooter, Strategiespiele und Rollenspiele, die in der Entwicklung zu oft von Anfängern unterschätzt werden) schreiben, aber irgendwie scheint dabei immer viel auf der Strecke zu bleiben. Ich hoffe, mein Workshop kann da in eine andere Richtung vordringen. Passend zur Jahreszeit (zu der Zeit des Erscheinens dieses Workshops) wollen wir ein Weihnachtsspiel entwickeln.

Dass Spieleentwicklung etwas mit Liebe zum Detail zu tun hat, ist klar, auch, dass solch ein Projekt viel mit Planung zu tun hat. Ein wichtiger Teil dieser Planung bei vielen Spieleproduktionen ist es, ein so genanntes Design Dokument auf die Beine zu stellen. In so einem Dokument fasst man wirklich alles zusammen, was das Spiel betrifft. Die kleinere Version des Design Dokumentes nennt man Game Abstract. Dieses wird vorwiegend „nur“

herumgezeigt, um den Leuten (z.B. potentiellen Publishern) das Spiel schmackhaft zu machen.

Anstatt jetzt zuviel Zeit in ein solches Dokument zu investieren, gehen wir den Mittelweg. Wir fassen all unsere Ideen zusammen und machen quasi ein „brainstorming“, bevor wir hier dem Planungstod erliegen ;) Im Anschluss steigen wir in die genauere Planung des ersten Levels ein.

Die Spielidee

Nun, wie gesagt, es soll ein Weihnachtsspiel sein und den Titel „RUDI“ tragen, weil es um Rudi das Rentier mit der roten Nase gehen soll. Die Geschichte ist ganz simpel: Die Elfen wollten kurz vor Heiligabend die letzten Geschenke noch schnell zum Weihnachtsmann bringen, damit er sie pünktlich an die Kinder verteilen kann. Doch dann ist ihnen ein Malheur passiert! Einige der Geschenke sind vom Schlitten gefallen! Der Weihnachtsmann weiß nicht, was er tun soll... Doch dann kommt Rudi und sagt, dass er sich drum kümmern würde. Der Spieler schlüpft in die Rolle von Rudi, dem Rentier mit der roten Nase und seine Aufgabe ist es, so viele Geschenke wie möglich einsammeln, damit jedes Kind auch ein schönes Weihnachten mit vielen Präsenten feiern kann.

So simpel wie die Geschichte ist auch der Spielablauf. Ähnlich wie in dem bekannten Spiel „Snake“ steuert man Rudi durch wunderschöne Winterlandschaften und sammelt die verlorenen Geschenke ein. Hinter Rudi haken sich dann Schlitten ein und auf diesen verstaut er die Geschenke. Je mehr Geschenke er sammelt, desto länger wird die Schlange. Wenn er durch seine eigene Schlitten-Schlange fährt, trennt er die restlichen Schlitten ab und die Geschenke gehen kaputt – das ist schlecht!

Rudi hat es eilig und ist immer in Bewegung! Wenn er gegen ein Hindernis läuft, fällt er ohnmächtig um und muss es nochmal versuchen. Er hat insgesamt 5 Versuche, danach ist das Spiel vorbei – der Weihnachtsmann hat die Geduld verloren! Erst wenn Rudi alle Geschenke in einem Level gefunden hat, kann er durch ein Weihnachtstor zum nächsten Ort. Der Weihnachtsmann hat Rudi für jedes eingesammelte Geschenk ein paar Lebkuchenherzen versprochen, weil Rudi die so gerne mag. Verlässt er ein Level, bekommt er dann für jedes Geschenk ein Lebkuchenherz gutgeschrieben. Also ist es nur in seinem Interesse, soviele Lebkuchenherzen wie möglich zu kriegen ;) Die Lebkuchenherzen kann man als „Ersatz“ für konventionelle Punkte betrachten.

Wir wollen uns nicht auf eine Steuerungsart festlegen. Rudi soll sowohl mit der Tastatur als auch mit einem Joypad steuerbar sein. Wir starten das Spiel und gelangen in ein simples Menü, das uns ein paar Optionen einstellen, das Spiel starten und verlassen lässt. Das sollen nicht zuviele Optionen sein, sondern nur die wesentlichen: ein paar Videoeinstellungen (Auflösung, Details, etc.), Soundeinstellungen und das war es. Das Spiel hat (vorerst) keine Speicheroption – das lohnt für ein Level nicht! Wenn der Spieler das Spiel startet, wird in einem kurzen Einleitungstext die Geschichte kurz erklärt (und damit auch das Spielprinzip) und dann geht es auch schon los mit dem ersten Level.

Die verwendete Software

Entwickelt wird das Spiel mit der 3D Gamestudio A7 Authoring Suite für Microsoft Windows. Gamestudio ist keine Game-Engine, kein Spiele-Editor und keine Spieleprogrammiersprache. Es ist alles zusammen. Das Entwicklungssystem beinhaltet alle notwendigen Werkzeuge zum Produzieren eines 3D- oder 2D-Spiels in kommerzieller Qualität. Integriert ist eine aktuelle 3D- und 2D-Engine, ein Leveleditor, ein Modell- und Terrain-Editor, ein Skript-Editor und Debugger sowie Tausende von einsatzfertigen Objekten wie Modellen, Level-Bauteilen oder Texturen. Falls Sie keine Lizenz für das 3D Gamestudio besitzen, ist das auch kein Problem: der Workshop ist größtenteils so geschrieben, dass er mit der kostenlosen 30 Tage Demoversion von Gamestudio A7 kompatibel ist. Lediglich während der Levelerstellung werden Shader benutzt, um eine schönere Grafik darzustellen. Sollte Ihre Edition diese nicht unterstützen, müssen Sie auf den Großteil dieser Shader verzichten – was aber nicht schlimm ist, da dass Spiel nicht abhängig davon ist und auch „so“ spielfähig ist.

Wenn Sie noch keine Version von 3D Gamestudio auf deinem Computer installiert haben oder noch mit der Vorgängerversion A6 arbeiten, gehen Sie bitte auf die Website von 3D Gamestudio und laden Sie sich die kostenlose Demo herunter. Hier ist der Link:

Website von 3D Gamestudio: http://www.3dgamestudio.com/

Was wir erreichen wollen

Da das Spiel weihnachtlich ist, als Casual Game definiert ist, Sie es eventuell nachprogrammieren und ihren Freunden und Verwandten schenken wollen, ist es besonders wichtig, dass das Spiel leicht zugänglich ist (Installation, Spielstart, Spielziel und Steuerung). Außerdem soll es wenn möglich fehlerfrei und vor allem spaßig sein! Also wollen wir natürlich schöne Musik, eine nette - und wenn möglich zuckersüße - Grafik und besonders wichtig: ein wenig Aufmerksamkeit des Spielers.

Ein weiterer wichtiger Punkt ist: es soll vollständig sein. Es soll mindestens einen komplett spielbaren Level haben und alle relevanten Optionen und Menüs erhalten, die das Spiel „rund“ und zu einem echten Computerspiel machen. Viel zu oft unterschätzen Leute diesen Faktor, was dazu führt, dass gute Ideen vielleicht auch gut umgesetzt werden, aber kein „Spiel“ darstellen, weil es zu nicht mehr als einen Prototypen reicht – und dann regelrecht „untergehen“.

Kapitel 3: Framework & Debugging

Wir erzeugen in diesem Kapitel die Anfänge eines Frameworks, also eines System, dass das Spiel umhüllt und wichtige Aufgaben übernimmt. Desweiteren erstellen wir ein flexibel einklinkbares Debugmodul. Wir lernen ein paar Compiler- und Include-Tricks und programmieren heute eine gute Basis für unser Spiel. Im Vergleich zum letzten Kapitel, das relativ leicht war, geht es nun richtig los mit dem Programmieren... also anschnallen und auf geht’s!

Unser Programmeinstiegspunkt

Die erste Codedatei, die wir für das Spiel erzeugen werden, ist die Datei "rudi.c". "RUDI" ist der Name unseres Spiels und daher nennen wir die Startdatei des Spiels eben "rudi.c" (und nicht "rudi.h", denn .c steht in der Terminologie der Sprache C in der Regel für eine Codedatei). Anstatt den Quelltext des gesamten Spiels in eine Datei zu schreiben, wollen wir den Code in zusammenfassende Module unterteilen und zwar in Form von seperaten Codedateien, die wir an einer Stelle der Hauptdatei einfügen. Dies können wir durch sogenannte includes erreichen – Anweisungen, die an einer bestimmten Stelle eine andere Datei reinladen.

Wir werden in der Datei „rudi.c“ alle nötigen includes vorbereiten und die Hauptfunktion – die als erste des gesamten Programms automatisch gestartet wird - einbauen, damit das Programm dort den Einstiegspunkt hat. Damit wir auch wissen, dass das Programm läuft, testen wir dies in dem wir eine Fehlermeldung mit dem Text "test" ausgeben:

rudi.c:

```
//------------------------------------------------------------------------
// pre-processor
//------------------------------------------------------------------------

   //- acknex & liteC linkage ---------------------------------------
      #include <acknex.h>
      #include <default.c>

   //- system libs --------------------------------------------------
      #include <stdio.h>

//------------------------------------------------------------------------
// main
//------------------------------------------------------------------------
void main ()
{
```

```
   error("test");
   wait(1);
}
```

Diese Datei ist die Hauptdatei des Spiels. Sie wird als erstes geladen. Von hier aus werden alle anderen Codeteile des Spiels geladen. Man kann außerdem schon einige sinnvolle Sachen entdecken, die ich hier eingeführt habe: einmalig sieht man hier den Präprozessor-Abschnitt für den Lite-C Compiler und die entsprechenden includes und die Hauptfunktion, die void main(). In diese haben wir erstmal nur ein wait(1); und ein error("test"); gesetzt (die Anweisung error gibt eine Fehlerbox mit einem benutzerdefinierten Text aus) - es passiert vorerst also gar nichts interessantes.

Die Datei habe ich im SED erstellt und im "work" Verzeichnis gespeichert. Unter "Options/Preferences" habe ich dann die Datei als Startdatei angegeben und des Weiteren -diag angeklickt, was mir die log Datei "acklog.txt" erzeugt. Um das Programm auszuführen, drücke ich F5 im SED und erwartungsgemäß wird mir die Fehlermeldung "test" ausgegeben. Sie können natürlich auch eine der *.bat Dateien benutzen – ich benutze diese eigentlich immer.

Das Framework

Wir wollen nun, bevor wir das eigentliche Spiel entwickeln, eine Art Codegerüst - ein Framework - aufbauen, was dem Spiel zugrunde liegt und unterstützt. Dafür wollen wir eine Systeminitialisierung schreiben, die das Spiel initialisiert und ein Debugtool, damit wir zur Laufzeit Daten auf unserem Bildschirm ausdrücken können, ähnlich einer Echolist (das kennen Sie vielleicht noch von MS-DOS, wo Befehle und Ausgaben listenartig untereinander ausgegeben worden sind – das nennt man „echolist“). Kümmern wir uns zunächst um das System.

Das System umfassen wir in 2 Codedateien: "sys.c" und der Headerdatei "sys.h". Genau wie alle anderen Codeteile des Spiels, müssen wir diese beiden Dateien in unser Spiel einfügen. Deshalb fügen wir in "rudi.c" hinter den lite-C includes eine neue Sektion hinzu, in der wir alle unseren spielspezifischen includes einfügen:

```
//- game ------------------------------------------------------

   //headers
      #include "sys.h" //framework

   //codefiles
      #include "sys.c" //framework
```

In diesem Fall trenne ich die header von den codefiles. Das mag komisch anmuten, hat aber einen Vorteil: wenn man das für das komplette Projekt durchzieht, stehen in allen Headerdateien alle Funktionsprototypen, Defines, globale Variablen, Struct Definitionen, Typedefs, initialisierte Structs, etc. **bevor** irgendein ein anderes Codefragment darauf zugreift.

Viele Amateure haben bei einer gewissen Projektgröße das Problem, dass irgendwelche Dateien auf Daten und Funktionen in anderen Codedateien zugreifen. Bei C-Script und Lite-C Projekten habe ich auf diese Art und Weise niemals ein Problem gehabt, weil man dadurch sehr sauber kontrollieren kann, was wann inkludiert wird.

Wir brauchen nun diese sys.c/.h Dateien, die – wie bereits erwähnt – dass System initialisieren sollen. Damit das auch so geschieht, schreiben wir in die main einen Aufruf "sys_main();".. das wird unser Start des Systems sein:

```
void main ()
{
   sys_main(); //start framework
}
```

Im Nachfolgenden ein Listing von sys.h gefolgt von sys.c, mit anschließendem Kommentar.

sys.h:

```
//---------------------------------------------------------------------
// prototypes
//---------------------------------------------------------------------
```

```
   void sys_main();
   void sys_init();
      void sys_video_init();
      void sys_keys_init();
         void sys_key_null();
   void sys_loop();
   void sys_close();

//------------------------------------------------------------------------
// defines
//------------------------------------------------------------------------

   #define DEF_VIDEO_RES   8  //1024x768
   #define DEF_VIDEO_DEPTH 32 //32 bit
   #define DEF_VIDEO_MODE  2  //window

   STRING* DEF_WINDOW_TITLE = "Rudi";
```

sys.c:

```
//------------------------------------------------------------------------
// system start
//------------------------------------------------------------------------
void sys_main()
{
   sys_init(); //system initialization
   sys_loop(); //system loop
}

//------------------------------------------------------------------------
// system initialization
//------------------------------------------------------------------------
void sys_init()
{
   randomize(); //seed the random() generator

   //system initializations
      sys_video_init();
      sys_keys_init();
}

   //- video ---------------------------------------------------------
   void sys_video_init ()
   {
      video_switch(DEF_VIDEO_RES, DEF_VIDEO_DEPTH, DEF_VIDEO_MODE); //init video adapter
      video_window(NULL, NULL, 0, DEF_WINDOW_TITLE); //set window title
      vec_set(screen_color, vector(5, 5, 5)); //no tearing
   }

   //- key mappings --------------------------------------------------
   void sys_keys_init ()
   {
      //disable default key bindings
         on_f2    = sys_key_null;
         on_f3    = sys_key_null;
         on_f4    = sys_key_null;
         on_f5    = sys_key_null;
         on_esc   = sys_key_null;

         on_close = sys_close;
   }

      //dummy key bind
      void sys_key_null () {wait(1);}

//------------------------------------------------------------------------
// system loop
//------------------------------------------------------------------------
void sys_loop()
{
   while (1) {
      //--- add frame functions here ---
      wait(1);
   }
}
```

```
//--------------------------------------------------------------------
// system exit (shutdown)
//--------------------------------------------------------------------
void sys_close()
{
   //--- add finishing processes here ---

   //shutting down
      sys_exit("");
}
```

Beginnen wir mit der "sys.c". Dort wird als erstes die sys_main() aufgerufen. Diese ruft wiederum die Funktionen sys_init() und sys_loop() auf. In der Init-Funktion wird das System initialisiert und in der Loop-Funktion läuft das System (nicht das Spiel!). In der Init Funktion wird der random Generator geseedet, was bedeutet, dass "echte" Zufallszahlen erzeugt werden. Daraufhin werden diverse andere Init-Funktionen aufgerufen. Zum einen eine Video-Funktion, die das Fenster einstellt und das tearing verhindert und eine Key-Funktion, die die standardmäßigen Tastaturbelegungen zurücksetzt - auch die ESC Taste, denn das Spiel wird später nur durch den Beenden-Knopf im Spielmenü zu verlassen sein. In der Entwicklungsphase benutzen wir den Windowed Mode. Das on_close -Event wird dann ausgelöst, wenn die engine beendet wird. Das passiert, wenn man z.B. das Kreuz oben rechts drückt - oder eben vom Spiel heraus die engine schließt. Wir weisen dem on_close eine benutzerdefinierte Funktion zu. Dort steht bisher nur die sys_exit Anweisung, aber da werden wir später noch einige Aufräumarbeiten einfügen.

Die sys_loop Funktion erscheint im Moment auch noch trivial, aber dort werden später noch einige Sachen ausgeführt, die das System betreffen. Der springende Punkt ist, dass die Funktionen, die in sys_loop aufgerufen werden, sogenannte Frame-Funktionen sind. Das heißt, die Funktionen werden direkt und nur in dieser einen Schleife ausgerufen und besitzen selber keine eigene Schleife, sonst wären das sogenannte Co-Routinen. Frame-Funktionen haben einen entscheidenen Vorteil gegenüber Co-Routinen: Die Ausführung ist kontrolliert und überwachbar.

Die Werte, die zum Beispiel in sys_video_init benutzt werden, stehen in der "sys.h". Dort werden erst alle Prototypen mit wiedergespiegelter Einrückungshierarchie in einer Prototypensektion aufgelistet. Dann folgen einige Defines, die in der "sys.c" verwendet werden. Wenn das Fenster einen anderen Titel haben soll, zum Beispiel "RUDI - das ultimative Weihnachtsgame für meine Oma", dann würde man den String dort einfach ändern, anstatt den gesamten Code abzusuchen. In der Abschlussphase des Projektes werden wir auch das Define DEF_VIDEO_MODE in 1 umändern, damit das Spiel im Vollbildmodus ausgeführt wird.

Diese Konstruktion ist nur dazu da, das Spiel zu unterstützen. Wir werden einige Systemfunktionen, -Variablen und dergleichen später noch schreiben müssen. Aber vom Spiel haben wir ja noch gar nichts gesehen! Wir haben das "Spiel" an sich auch noch gar nicht geschrieben, geschweige denn im Code eingefügt. Dafür erstellen wir 2 Dateien namens "game.c" und "game.h" und fügen diese in unsere include-Struktur in "rudi.c" ein. Das sieht dann so aus:

```
//- game ------------------------------------------------------

   //headers
      #include "sys.h"     //framework
      #include "game.h"    //game

   //codefiles
      #include "sys.c"     //framework
      #include "game.c"    //game
```

In der Datei "game.c" schreiben wir erstmal etwas ganz triviales rein, solange wir noch keinen echten Spiel-Code schreiben:

```
void game ()
{
   error("RUDI");
}
```

In der Datei "game.h" gehören natürlich die passenden Prototypen - den Kopf der beiden Dateien nicht vergessen! In der main rufen wir dann die Funktion game() auf, also quasi das Spiel:

```
void main ()
{
   sys_main(); //start framework
   game();     //start game
}
```

Wenn wir das Spiel nun starten, kriegen wir den Fehler "RUDI" in einer Error-Box, was auch vollkommen richtig ist!

Im Lauf der weiteren Kapitel werden wir dort ansetzen und das Spiel Stück für Stück weiterentwickeln. Aber bevor wir das machen, räumen wir erstmal im work-Verzeichnis auf und arbeiten dann noch an einem anderen Stück sehr hilfreichem Code.

Code in Unterordnern

In unserem Ordner "work" haben wir nun folgende Dateien:

- acklog.txt (falls -diag an war, wenn nicht, dann bitte anschalten!)
- game.c
- game.h
- rudi.c
- sys.c
- sys.h

Das sind jetzt nur einige Dateien, aber man kann sich sicherlich vorstellen, dass die Anzahl der Dateien im Laufe der Entwicklung rapide ansteigen wird. Wir wollen nun eine klare Trennung zwischen unserem framework und dem eigentlichen game-code durchführen. Deshalb erstellen wir nun 2 Ordner, einmal "sys" und "game" und verschieben die entsprechenden sys- und game-Dateien dorthin. Die Datei "rudi.c" bleibt im Hauptverzeichnis.

Damit das Programm die Dateien auch lädt, fügen wir ein define „PRAGMA_PATH“ vor die includes, um Verzeichnisse anzugeben, die der Compiler nach include Dateien absucht. Die Include-Hierarchie sieht also nun so aus:

```
//-----------------------------------------------------------------------
// pre-processor
//-----------------------------------------------------------------------

   //- acknex & liteC linkage ---------------------------------------
      #include <acknex.h>
      #include <default.c>

   //- system libs --------------------------------------------------
      #include <stdio.h>

   //- game ---------------------------------------------------------

      //paths for nested include files

         #define PRAGMA_PATH "sys"
         #define PRAGMA_PATH "game"

      //headers
         #include "sys.h"     //framework
         #include "game.h"    //game

      //codefiles
         #include "sys.c"     //framework
         #include "game.c"    //game
```

Wir sind jetzt in der Lage, jedweden Code in den sys- und game-Folder zu legen, sodass der Compiler diese Dateien findet. **Achtung**: im development mode funktioniert das PRAGMA_PATH define wie ein "normales" PATH statement in C-Skript. Das heißt, auch ganz normale Dateien wie models, bitmaps und sounds etc. werden gefunden. In der publishedten Version funktioniert das aber **nicht** und wir müssen die Ordner, die für Content

abgesucht werden, mit der Funktion „add_folder“ zum Dateisystem der engine hinzufügen.

(Anmerkung: in der Engine Version 7.07.5 wird dies nicht mehr so sein, dann werden Pfade, die über PRAGMA_PATH eingebunden werden, auch in der publishedten Version verfügbar sein und add_folder wird in diesem Zusammenhang obsolet!)

Das Debugmodul

3D Gamestudio verfügt über einige sehr gute Debugmöglichkeiten, man kann im SED Entities und Variablen beobachten, Daten in die acklog-Datei ausgeben und auch im singlestep-Verfahren durch den Code steigen. Da wir aber keine Debugdaten wie bei einer klassischen Konsolenanwendung dynamisch auf dem Bildschirm ausdrucken können und wir nicht exemplarische Bugs erzeugen wollen, um dann eines der genannten Verfahren austesten zu können, entwickeln wir ein einklinkbares Debugmodul. Die anderen Debugverfahren werden hier nicht weiter angesprochen, dies bleibt dem Leser vorbehalten, sich darin zu üben und auszutesten. Der Grund, warum wir dieses Modul einbauen, ist relativ einfach nachzuvollziehen: neben dynamischen Variablenauswertungen und dergleichen können wir auch todo-Nachrichten in den Code integrieren und merken immer im Spiel, wann noch was zu erledigen ist. Man kann damit auch den Code instrumentieren und vollstänge Verzweigungstests und so weiter ausführen. Auf jeden Fall ist das Ding sehr nützlich, also wollen wir es direkt mal anfassen, bzw. erstellen.

Damit wir das Debug-Modul eindeutig vom framework und vom Spielcode trennen können, richten wir den Ordner "debug" im "work"-Verzeichnis ein und fügen die Zeile

```
#define PRAGMA_PATH "debug"
```

hinter den anderen PRAGMA_PATH's ein. Das Debug-Modul soll einklinkbar sein, deshalb rufen wir nur eine Includedatei auf, nämlich die Datei "deb.c" (diese ruft dann automatisch die Header-Datei auf). Das Debug-Include wird vor allen anderen includes getätigt, damit alle Funktionen auf das Debugmodul zugreifen können:

```
//- game ------------------------------------------------------------

   //paths for nested include files

      #define PRAGMA_PATH "sys"
      #define PRAGMA_PATH "game"
      #define PRAGMA_PATH "debug"

   //debug

      #include "deb.c"     //debug

   //headers
      #include "sys.h"     //framework
      #include "game.h"    //game

   //codefiles
      #include "sys.c"     //framework
      #include "game.c"    //game
```

Wir erstellen nun im Ordner "debug" 2 Dateien: "deb.h" und "deb.c". Nachfolgend das listing dieser beiden Dateien, erst der header und dann die code Datei. Nachfolgend eine Erklärung.

deb.h:

```
//-------------------------------------------------------------------------
// prototypes
//-------------------------------------------------------------------------

   void deb_init ();
      void deb_clearList ();
      void deb_convoluteList (TEXT* _txt, int _dir);
      void deb_print(char* _in);
         void deb_print (STRING* _in);
         void deb_print (int _in);
         void deb_print (var _in);
         void deb_print (float _in);
```

```
      void deb_print(double _in);
   void deb_log (char* _in);

//------------------------------------------------------------------------
// ressources
//------------------------------------------------------------------------

   STRING* deb_str = "#128";
   FONT* deb_fnt = 0;
   TEXT* deb_txtList = 0;

//------------------------------------------------------------------------
// dynamic startup - no need to call it manually!
//------------------------------------------------------------------------
void deb_h_startup ()
{
   deb_init();
}
```

deb.c:

```
   #include "deb.h" //load header

//------------------------------------------------------------------------
// initializes the debug system
//------------------------------------------------------------------------
void deb_init ()
{
   #ifndef COMPILE

      add_folder("debug"); //add folder to engine filesystem
      deb_fnt = font_create("deb_fnt.tga"); //create font(s)

      //create echolist
         deb_txtList = txt_create(30, 999999);
         int i;
         for (i = 0; i < deb_txtList.strings; i++) { //string initialization
            (deb_txtList->pstring)[i] = str_create("#256");
         }

      //init text
         deb_txtList->font = deb_fnt;
         deb_txtList->pos_x = 40;
         deb_txtList->pos_y = 80;
         set(deb_txtList, VISIBLE);
   #endif
}

//------------------------------------------------------------------------
// ECHOLIST
//------------------------------------------------------------------------

   //------------------------------------------------------------------------
   // clears the echolist with empty strings in each row
   //------------------------------------------------------------------------
   void deb_clearList ()
   {
      #ifndef COMPILE
         int i;
         for (i = 0; i < deb_txtList->strings; i++) {
            str_cpy((deb_txtList->pstring)[i], "");
         }
      #endif
   }

   //------------------------------------------------------------------------
   // convolutes the echolist
   //------------------------------------------------------------------------
   void deb_convoluteList (TEXT* _txt, int _dir)
   {
      #ifndef COMPILE
         if (_dir == 0) {return;}
         int i;
         //push down
         if (_dir > 0) {
```

```
         for (i = _txt.strings - 1; i > 0; i--) {
            str_cpy((_txt->pstring)[i], (_txt->pstring)[i-1]);
         }
      } else {
         //pull up
         for (i = 0; i < _txt.strings - 1; i++) {
            str_cpy((_txt->pstring)[i], (_txt->pstring)[i+1]);
         }
      }
   #endif
}

//---------------------------------------------------------------------
// prints a char message into the echolist
//---------------------------------------------------------------------
void deb_print(char* _in)
{
   #ifndef COMPILE
      if (deb_fnt != 0) { //print it only, if font is created

         char buffer[128];

         deb_log(_in); //log debug str to acklog.txt
         sprintf(buffer, "%d: %s", total_frames, _in); //create entry
         deb_convoluteList(deb_txtList, 1); //push list down

         //copy new entry into the first row
         str_cpy((deb_txtList->pstring)[0], buffer);
      }
   #endif
}

//---------------------------------------------------------------------
// convenience functions
//---------------------------------------------------------------------

   void deb_print (STRING* _in) {
      #ifndef COMPILE
         deb_print(_in->chars);
      #endif
   }

   void deb_print (int _in) {
      #ifndef COMPILE
         char* buffer [64];
         sprintf(buffer, "%d", _in);
         deb_print(buffer);
      #endif
   }

   void deb_print (var _in) {
      #ifndef COMPILE
         str_for_num(deb_str, _in);
         deb_print(deb_str->chars);
      #endif
   }

   void deb_print (float _in) {
      #ifndef COMPILE
         str_for_num(deb_str, _in);
         deb_print(deb_str->chars);
      #endif
   }

   void deb_print(double _in) {
      #ifndef COMPILE
         str_for_num(deb_str, _in);
         deb_print(deb_str->chars);
      #endif
   }

//---------------------------------------------------------------------
// logs a char sequence into the acklog.txt
//---------------------------------------------------------------------
void deb_log (char* _in)
```

```
{
   #ifndef COMPILE
      char buffer[128];
      sprintf(buffer, "\nfrm(%d) = %s", (int)(total_frames), _in); //create entry
      diag(buffer); //print to acklog.txt
   #endif
}
```

Das Studium dieser beiden Codedateien bleibt dem Leser überlassen. Jedoch möchte ich auf die wichtigsten Dinge eingehen Zunächst startet das Modul ganz automatisch, indem in der Headerdatei eine "startup" Funktion ausgeführt wird. Das bedeutet, man muss nur die "deb.c" inkludieren und schon läuft das Modul. Dies ist auch recht nützlich für andere Projekte.

Desweiteren gibt es eigentlich nur 2 Funktionen, die von Interesse sind: deb_print und deb_log, wobei deb_log immer aufgerufen wird, wenn deb_print aufgerufen wird. deb_print erhält ein Argument und druckt es auf dem Bildschirm in einer echolist aus. Dabei wird die Frame-Nummer mit angegeben. Deb_log loggt das auch in die acklog.txt, deb_log kann zusätzlich seperat benutzt werden, allerdings nur mit einem char* Parameter. Für deb_print werden primär nur char* Argumente angenommen, also ein beliebiger String, z.B. deb_print("Hello World"), aber es gibt auch überladene Funktionen, die aus int, float, var, SRING* und double Parametern char's machen und an die "echte" deb_print Funktion senden. Für einige unerfahrene Leser sei dies nur kurz erklärt, da das Prinzip des sogenannten Function-Overloadings eher erfahrene C/Lite-C Benutzer anspricht und im gesamten Workshop nicht weiter verwendet wird.

Das Modul registriert automatisch das "debug"-Verzeichnis und erwartet dort eine Font-Datei, die dynamisch geladen wird, also nicht beim Startup der Engine. Das hat folgenden Grund: wenn man das Spiel final kompiliert und das Debug-Modul nicht herausgenommen hat, will man ja auch keine überflüssigen Dateien laden. Das gesamte Modul reagiert auf das COMPILE define. Wenn irgendwo im Code vorher #define COMPILE steht, dann werden die Funktionen zwar ausgeführt, aber ohne Effekt. Es wird also kein überflüssiger Code ausgeführt.

Das Verfahren mit dem Compile-Flag wollen wir später auch noch nutzen, daher schreiben wir das flag in eine "Switches" ("Schalter") Sektion in der Datei "rudi.c" und kommentieren es aus, weil wir ja noch in der Entwicklungsphase stecken. Diese Sektion markieren wir durch einen „// Switches“ Kommentar im Quellcode, um die Sektion später einfach wiederzufinden. Der Code sieht daher so aus:

```
//-----------------------------------------------------------------------
// switches
//-----------------------------------------------------------------------

   //#define COMPILE

//-----------------------------------------------------------------------
// pre-processor
//-----------------------------------------------------------------------

   //- acknex & liteC linkage ---------------------------------------
      #include <acknex.h>
      #include <default.c>
```

Oben habe ich gesagt, das wir in der finalen Version auf jeden Fall das Spiel im Vollbildmodus ausführen wollen. Mit der Einführung des COMPILE Defines kann man das ganz elegant lösen und in der "sys.h" das DEF_VIDEO_MODE Define umschreiben:

```
#ifndef COMPILE
   #define DEF_VIDEO_MODE   2   //window
#else
   #define DEF_VIDEO_MODE   1   //fullscreen
#endif
```

In diesem Fall wird, wenn COMPILE an ist, DEF_VIDEO_MODE mit 2 gleichgesetzt. Wenn COMPILE aus ist, dann wird DEF_VIDEO_MODE mit 1 gleichgesetzt.

Kapitel 4: Der Spieler und die Kamera

Bevor wir nun irgendwelche Levels irgendwie bauen oder irgendwelche Models in das Spielverzeichnis holen, müssen wir uns mit folgender Thematik befassen: Wie skalieren wir eigentlich unsere Spielwelt?

Die ewige Frage: wie groß ist ein Quant und warum das wichtig ist

In der A7 werden Entfernungen mit den so genannten Quants gemessen. Quants sind keine "echten" Maßeinheiten - sie haben keine Entsprechung in der realen Welt. Wir wollen aber eine virtuelle Welt erstellen und darin auch zwangsläufig mit irgendeinem Maß messen können. Das ist besonders dann wichtig, wenn Objekte unproportional zueinander scheinen: das weist darauf hin, dass ein Objekt nicht dem Maßsystem der virtuellen Welt entspricht. Wir müssen also herausfinden, wie groß 1 Quant, z.B. = 1 Meter, ist. Wenn wir das wissen, können wir Modelle, Texturen und Oberflächen daraufhin ausrichten. Es gibt verschiedene Theorien, wie lang ein Quant nun ist. Es gibt dazu auch recht interessante Diskussionen, aber das nützt nichts, wenn es noch mehr verwirrt. Fakt ist: wenn man eine Textur auf eine Block-Surface im WED legt, ohne die Textur zu skalieren, entspricht 1 Pixel der Textur = 1 Quant zum Quadrat. Wenn beispielsweise eine 128x128 px Textur = 1 Quadratmeter entspricht, legen wir indirekt fest, dass 128 Quants = 1 Meter entspricht. Aber da wir recht große Levels bauen werden, skalieren wir 1 Meter auf 64 quants herunter, da sonst wie in unserem Beispiel 500 Meter 64000 Quants entsprächen.

Wir können jetzt anhand dieser Maße immer sehen, ob unsere Levelgeometrie stimmt und wie sich Models in ihren Größen gegeneinander verhalten. Wenn Rudi im Model beispielsweise 120 Quants hoch ist, dann entspräche das etwa 2 Meter, ein wenig zu groß! Anstatt jetzt die Entitygröße in WED zu ändern, skalieren wir das Modell direkt im MED. Um eine Art "Quant-Lineal" zu haben, erzeugen wir im WED einen 64x64x64 Würfel, zentrieren ihn und speichern ihn über "File/Export/Export to MDL7 file" als "reference.mdl" im documents-Ordner ab. Wir können das Modell dann immer in WED oder MED laden, um Proportionen zu überprüfen.

Das Spielermodell: Rudi ist da!

Unser Spielheld heißt Rudi und so heißt auch das 3D-Modell, welches wir zur Darstellung unsere Protagonisten benutzen wollen: die Datei „rudi.mdl“ befindet sich im Ordner "work\game\levels". Dort werden wir alle Leveldateien und die in den Levels vorkommenden Modelle speichern. Rudi ist auch ein solches Modell, also befindet es sich dort.

Wenn man Rudi im MED öffnet, sieht man, dass er diverse Animationen, u.a. einen walkcycle hat, also eine Laufbewegung. In diesem Kapitel wollen wir alles Wesentliche zu seiner Fortbewegung programmieren. Zunächst muss ein Testlevel her, dass alle Dinge widerspiegelt, die Rudi können muss. Rudi soll auf einer Ebene mit konstanter Geschwindigkeit laufen. Da wir gesagt haben, dass Rudi dauernd in Bewegung ist, wird er auch nicht stehen bleiben. Wenn er also gegen ein Hinderniss läuft, knallt er voll dagegen (eine Wand zum Beispiel). Schrägen sollte er auch hochlaufen können und an Vorsprüngen hinunterfallen. Das war's bereits.

Das Testlevel

Wir wollen als Testlevel einen Level bestehend aus Blöcken bauen. Wir öffnen also zunächst den WED und speichern die Datei direkt ab. Die Datei soll "testlevel.wmp" heißen. Die späteren Levels bekommen echte Namen, aber soweit sind wir ja noch nicht. Die Datei legen wir im Ordner "work\game\levels" ab. Wir laden Rudi hinein und erstellen einen Block, der groß genug für unser Testlevel ist.

Wir können unsere WMP Dateien auf gemeinsame Textur-Ordner verweisen lassen. Dazu verweisen wir die WMP Datei mit dem Verzeichnis "textures" im Work-Verzeichnis, indem sich ein paar Testtexturen befinden ("rock.bmp", "earth.bmp" und "grass.bmp"). Um nun dem Level diesen Ordner hinzuzufügen, öffnen wir den

Textur Manager (siehe Manual), klicken auf „Add Folder“ und geben oben in der Adressleiste ".\..\..\textures" ein. Dann wird der von uns eben erstellte Ordner relativ zur WMP Datei hinzugefügt. Wenn wir nun neue Texturen in den Ordner kopieren, können wir über die WMP Datei direkt darauf zugreifen!

Am besten erstellen wir zunächst einen großen Block, auf dem Rudi laufen soll. Ich habe einen recht großen Block genommen mit den Ausmaßen 4096 x 4096 x 1024 quants, was in etwa 64 x 64 x 16 Metern entspricht. Da haben wir auf jeden Fall genug Platz zum Testen. Aufgrund der oben genannten Merkmale habe ich noch folgende Dinge in die Map eingebaut: eine Senke mit flachen Schrägen, eine Erhebung mit flachen und etwas steileren Hängen und einer Felskante, die senkrecht auf dem Boden steht, ein Block, stellvertretend für ein Hindernis. Von dem Block zur Erhebung geht eine Art Brücke mit kleinen Blöcken als Abgrenzung. Rudi darf über dieser Abgrenzung nicht hinweg gehen, außerdem ist die Brücke gerade so hoch, dass Rudi darunter durch passt. Sie können die drei Texturen verwenden, um ein wenig die einzelnen Teile des Testlevels hervorzuheben. Ich habe außerdem die Sonne eingeschaltet (unter map properties, siehe manual), um die räumliche Tiefenwahrnehmung bei den sehr wenigen Levelelementen zu unterstützen (dies werden wir beim späteren Level nicht machen, weil wir dort das Level aus Modellen zusammenbauen, die eigene Schatten haben). Haben wir den Level kompiliert, können wir ihn mit der Vorschau begutachten, indem wir auf "File\Preview Map" klicken.

Das Testlevel ist nun soweit fertig. Wir müssen nur noch dafür sorgen, dass es geladen wird. Dazu gehen wir in die Datei "game.c" und löschen die Zeile mit dem error-Befehl. Wir schreiben *level_load("testlevel.wmb");* hin, damit das Testlevel geladen wird. Damit die Engine aber auch das Level und das Modell von Rudi findet, müssen wir den Ordner "levels" noch zum Dateisystem hinzufügen. Das erledigen wir in einer init-Funktion des Spiels. Wir schreiben also eine Funktion *void game_init ()* , die das erledigt und starten sie, bevor wir das Level laden.

```
void game ()
{
   game_init();

   level_load("testlevel.wmb");
}

void game_init ()
{
   //add game content folders to engine file system
      add_folder("game\\levels"); //leveldata
}
```

Damit das klappt, muss der Prototyp der game_init -Funktion noch in die "game.h" hinzugefügt werden. Wenn wir nun das Spiel starten, wird das Level geladen. Wenn wir die Taste 0 (Null) drücken, können wir im Level herumfliegen und die recht karge Umgebung begutachten.

Das Lied von der Kamera und warum wir den Player dennoch zuerst brauchen

Bevor wir die Playerfunktion programmieren und testen können, brauchen wir eine vernünftige Kamera. Der Player initialisiert die Kamera, also müssen wir auch mit der Playerprogrammierung anfangen - aber nur ein bisschen! Wir konzentrieren uns erst vollständig auf die Kamera und dann auf den Player.

Der Player und die Kamera bekommen ihre eigenen Codedateien, die wir auch im "game" -Verzeichnis anlegen. Logischerweise heißen die Dateien "player.c/.h" und "cam.c/.h". Damit die Dateien auch geladen werden, setzen wir die entsprechenden includes in der Datei "rudi.c". Wir rücken die includes etwas ein, damit wir wissen, dass sie zum Spiel gehören:

```
//- game ------------------------------------------------------

   //paths for nested include files

      #define PRAGMA_PATH "sys"
      #define PRAGMA_PATH "game"
      #define PRAGMA_PATH "debug"
```

```
//debug

   #include "deb.c"     //debug

//headers
   #include "sys.h"     //framework
   #include "game.h"    //game
      #include "player.h"  //player
      #include "cam.h"     //camera

//codefiles
   #include "sys.c"     //framework
   #include "game.c"    //game
      #include "player.c"  //player
      #include "cam.c"     //camera
```

Bevor wir wild drauflos anfangen, irgendwie mit actions im Code rumzuhantieren, möchte ich gerne noch einen „indischen Seiltrick“ präsentieren: anstatt die Playerfunktion zu einer Action (Actions sind Funktionen im Programmcode, die vom WED erkannt werden. Diese können dann den Entities zugewiesen werden.) zu machen, erstellen wir eine *.wdl Datei, die action-Prototypen besitzt. Ja, Sie haben richtig gelesen: eine *.wdl-Datei. Das hat einen entscheidenden Vorteil: Anstatt den gesamten Spiel-Code zu parsen, lädt WED nur diese kleine Datei. Wir können beispielsweise auch für jedes Level eine eigene WDL schreiben, um bestimmte actions zu verstecken oder speziell anzubieten. Das gilt auch für alle anderen actions, die wir im Laufe des Workshops noch schreiben werden.

Wir erstellen nun eine Datei namens "actions.wdl", kopieren sie direkt in das Levels-Verzeichnis und weisen sie der "testlevel.wmp" zu. Der Inhalt der WDL-Datei ist ganz simpel:

```
action pl_rudi
{
   wait(1);
}
```

Anstatt der vollwertigen Funktion schreiben wir nur ein wait(1); hinein und fertig ist der action-Prototyp. Die action hat das Präfix "pl", was sinnbildlich für "player" steht. Alle unsere Player-Funktionen bekommen diesen Namenszusatz. Wir weisen Rudi diese action im WED-Eigenschaftsfenster zu und aktualisieren die zuvor kompilierte WMB Datei mit "update entities". Die engine sucht dann nach einer Funktion beim Laden des Levels, die "pl_rudi" heißt. Dabei ist es egal, welchen Rückgabewert sie hat, also können wir in die "player.c" ganz beruhigt Folgendes schreiben:

```
//------------------------------------------------------------------
// main player action
//------------------------------------------------------------------
void pl_rudi ()
{
   error("Rudi!");
}
```

Natürlich gehört der Prototyp der Funktion auch in die Headerdatei, damit es nicht zu Fehlern kommt. Wenn wir das Spiel starten, wird wie erwartet die Fehlermeldung mit der Nachricht "Rudi!" ausgegeben. Wir müssen die "actions.wdl" übrigens nicht inkludieren. Sie ist quasi eine Dummy-Datei. Sie dient nur dem schnellen Laden der actions und der möglicherweise speziell auf Levels zugeschnittenen Liste ausgesuchter Actions. Damit die Kamera, die Gegner usw. auch auf den Player über einen Entity-Zeiger zugreifen (Zeiger sind Referenzen auf existierende Objekte. Eine komplette Erklärung des Zeigerbegriffs finden Sie im manual oder in jedem guten C-Tutorial), fügen wir den gleich hinzu. Ein Ausschnitt aus der player.h, die Definition:

```
//------------------------------------------------------------------
// variables
//------------------------------------------------------------------

   ENTITY* Rudi;
```

Den Zeiger weisen wir dann in einer neuen Init-Funktion zu:

```
//------------------------------------------------------------------
// main player action
```

```
//---------------------------------------------------------------------
void pl_rudi ()
{
   pl_rudi_init();
}

//---------------------------------------------------------------------
// player initialization
//---------------------------------------------------------------------
void pl_rudi_init ()
{
   Rudi = my;
}
```

Die Kamera wird vom Player aus gesteuert. Sie wird jedes Frame geupdatet, also bietet es sich an, die Kamerafunktion in der Schleife des Spielers als Frame-Funktion aufzurufen. Wir schreiben Folgendes in die Playeraction:

```
void pl_rudi ()
{
   pl_rudi_init();

   while (1)
   {
      cam_update();
      wait(1);
   }
}
```

Diese Funktion muss natürlich existieren, also schreiben wir eine Dummyfunktion in die cam.c/.h:

```
//---------------------------------------------------------------------
// main camera action
//---------------------------------------------------------------------
void cam_update ()
{
   deb_print("camera!");
}
```

Der String "camera!" taucht in der Echolist auf, also wird die Kamerafunktion auch aufgerufen. Die Kamera wird in jedem Frame aktualisiert und wir können auf Rudi zugreifen. Wir können uns nun mit vollständiger Aufmerksamkeit der Kamera widmen.

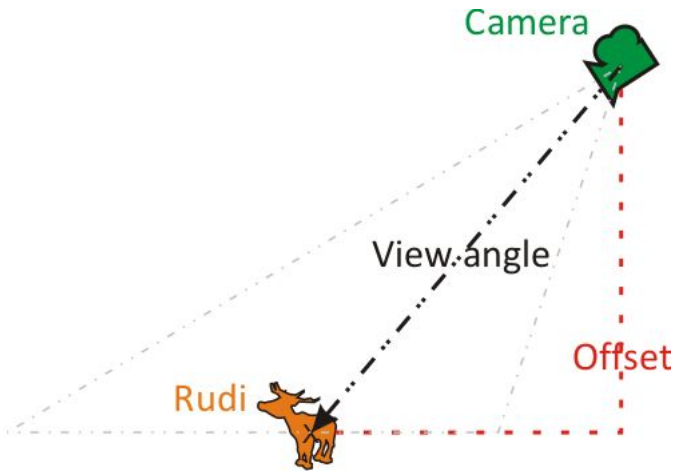
Die Kamera

Für die Kamera nutzen wir eine quasi-isometrische Ansicht. Wir gucken also von einer festen Position auf den Spieler und bewegen uns mit ihm mit. Sie brauchen dafür ein wenig Vektorrechnung, folglich auch Vektoren. In Lite-C gibt es dafür das VECTOR-struct und die entsprechenden Befehle. Wir benötigen zunächst ein paar temporäre Vektoren. Die können wir dann für temporäre Berechnungen - wie eben für die Kamera - dann im ganzen Code benutzen. Diese fügen wir im Framework in der Datei "sys.h" hinzu:

```
//---------------------------------------------------------------------
// global temporary variables
//---------------------------------------------------------------------

   VECTOR vecTemp;
   VECTOR vecTemp2;
   VECTOR vecTemp3;
   VECTOR vecTemp4;
```

Wichtig: Wir schreiben die Vektoren OHNE Sternchen, es handelt sich also nicht um Zeiger. Wir greifen dann mit Hilfe des Punkt-Operators auf diese Vektoren zu. Sie werden vom Lite-C Compiler automatisch initialisiert. Wenn wir später noch andere temporäre Variablen brauchen, fügen wir sie an dieser Stelle hinzu.

Wir wollen in der Funktion cam_update nun beschreiben, wie sich die Kamera verhalten soll. Die Kamera soll immer in einem festen Abstand und einem festen Winkel auf Rudi gucken – die fest Positionsverschiebung bezeichnet man auch im Englischen als „Offset“. Dazu addieren wir ein solches offset auf die Spielerposition, sodass an dieser Stelle dann die neue Position der Kamera eingestellt wird. Damit die Kamera auch auf den Spieler schaut, müssen wir die Richtung von der Kamera zu ihm berechnen und den Vektor in einen Winkel umrechnen. Diesen Winkel weisen wir dann der Kamera zu. Zusätzlich kann man dann noch den Kamerawinkel kontrollieren, um etwas mehr Überblick zu haben.



Das schreiben wir dann so:

```
void cam_update ()
{
   vec_set(vecTemp.x, vector(0, -800, 800));
   vec_add(vecTemp.x, Rudi->x);
   vec_set(camera.x, vecTemp.x);

   vec_diff(vecTemp.x, Rudi->x, camera.x);
   vec_to_angle(camera.pan, vecTemp.x);

   camera.arc = 80;
}
```

Wir wollen die Zahlenwerte so nicht im Code haben, sondern in den Header auslagern, damit die Werte von dem Code getrennt sind. Zudem sollten wir nicht den Fehler machen und die Werte als defines deklarieren; die Kamera sollte sich immer der aktuellen Spielsituation anpassen. Wenn wir zur Laufzeit solche fixen Werten ändern wollten, ginge das natürlich nicht. Also definieren wir die Werte in normalen Variablen und Vektoren in der "cam.h":

```
//-----------------------------------------------------------------
// defines and presets
//-----------------------------------------------------------------

   //- standard cam preset ---------------------------------------
   VECTOR*  cam_std_offset = {x = 0; y = -800; z = 800;}
   var      cam_std_arc = 80;
```

Im Gegensatz zu unseren temporären Vektoren in der "sys.h" definieren wir für unser offset einen VECTOR Zeiger mit initialisierten Werten (Es ist in diesem Fall deshalb ein Zeiger, weil wir ihn initialisieren und nicht uninitialisiert lassen). Das Präfix beider Werte lautet "cam_std" was für Kamera Standard(-Einstellung) steht. In der "cam.c" müssen wir die fest eingestellt – hardkodierten - Werte und die vector(..) -Anweisung natürlich durch diese Variablennamen ersetzen, damit sie übernommen werden. Wenn wir nun das Spiel starten, sehen wir zwar, dass die Kamera funktioniert, wenn wir aber mit der Taste 0 die Debugkamera anschalten, können wir nicht mehr durch das Level umherfliegen - das ist schlecht! Beim genauen Studium der inkludierten Datei "default.c" stellt man fest, dass die globale Variable "def_camera" != 0 ist, wenn die Debug Kamera an ist. Wir reagieren darauf und beenden vorzeitig die Kamerafunktion, wenn dies der Fall ist:

```
void cam_update ()
{
```

```
   if (def_camera != 0) {return;} //stop here, if debug camera is on

   //position
      vec_set(vecTemp.x, cam_std_offset);
      vec_add(vecTemp.x, Rudi->x);
      vec_set(camera.x, vecTemp.x);

   //look at player
      vec_diff(vecTemp.x, Rudi->x, camera.x);
      vec_to_angle(camera.pan, vecTemp.x);

   //camera angle
      camera.arc = cam_std_arc;
}
```

Wir können nun im Spiel die Taste 0 drücken und umherfliegen. Wenn wir wieder 0 drücken, wird wieder unsere eigene Kamerafunktion ausgeführt. Unsere Kamera ist jetzt eine sehr statische Variante, aber für die Player-Programmierung reicht es erstmal aus. Wenn wir uns mit Rudi bewegen können, fassen wir die Kamera nochmal an, damit sie etwas dynamischer wirkt.

Der Player

Wir haben bereits eine Init-Funktion namens pl_rudi_init geschrieben, da wollen wir beginnen und einige Dinge vorbereiten, bevor der Spieler mit dem Rudi rumlaufen darf. Wir definieren die Entity als „Rudi“, damit andere Funktionen auf ihn zugreifen können und lassen die Engine die Kollisionsbox (siehe manual) neu berechnen. Wir halbieren die Box, damit Rudi keine Probleme bei eventuell Übergängen, Brücken oder soetwas hat.

```
void pl_rudi_init ()
{
   Rudi = my;           //indicate that I am Rudi!

   c_setminmax(my);     //calculate real bounding box
   my.max_z *= 0.5;
}
```

Rudi ist nun initialisiert und bereit, loszulaufen. Doch wie stellen wir das an? Es gibt in Lite-C den c_move Befehl, der es uns erlaubt, eine absolute und relative Bewegungsrichtung anzugeben. Die Entity wird dann bewegt und eventuell an schrägen Flächen vorbeigeschoben, wenn sie dagegenläuft. Das wollen wir nutzen! Bevor Rudi den entscheidenen Bewegungsschritt ausführt, muss er aber auch auf die Benutzereingaben reagieren und sich gedreht haben. Außerdem müssen wir wissen, ob wir uns in der Luft befinden und dementsprechend nach unten bewegen, um die Gravitation zu simulieren. Viele Leute tendieren dazu, wirklich alles, was den Spieler betrifft, in die Hauptschleife des Spielers zu schreiben. Damit das ein wenig geordneter ist, lagern wir alle Schritte in entsprechende Funktionen aus.

```
while (1)
{
   cam_update();

   pl_turn();
   pl_gravity();
   pl_move();

   wait(1);
}
```

Wir erstellen dann weiter unten für jede dieser Funktionen Dummyversionen, in denen nur ein wait(1); steht, damit sie zumindest ausgeführt werden. Nachdem wir die Prototypen in der "player.h" hinzugefügt haben, widmen wir uns der Funktion pl_turn. Dort wollen wir 1.) herausfinden, was für eine Eingabe der Spieler macht, 2.) schließen wir dann auf die gewünschte Bewegungsrichtung und 3.) müssen wir Rudi auch noch in diese Richtung drehen.

Die Tastatureingabe und die Bewegungsrichtung

Als wir uns Gedanken über die Spielidee gemacht haben, haben wir gesagt, dass der Spieler über die Tastatur und ein Joypad steuerbar sein soll. Wenn wir jetzt fixe Abfragen der Tastatur mit fixen Abfragen des Joypads im Player-Code verbinden, ist das genauso unflexibel wie hardkodierte Zahlenwerte. Wir wollen nur herausfinden, ob der Spieler HOCH, RUNTER, LINKS und RECHTS drückt - unabhängig davon, ob er die Tastatur oder ein Joypad benutzt. Wir definieren also am besten diese "generellen" flags im Framework und lassen das Framework diese Auswertung ausführen. Wir fügen also in der "sys.h" eine neue Sektion für globale flags und Variablen hinzu und erstellen 4 simple flags, die die aktuelle Eingabe repräsentieren:

```
//-----------------------------------------------------------------------
// global flags and variables
//-----------------------------------------------------------------------

   //general button states
      int UP, DOWN, LEFT, RIGHT;
```

Damit diese Flags auch jederzeit aktuell sind, wird eine Frame-Funktion in der Schleife von sys_loop aufgerufen:

```
void sys_loop()
{
   while (1) {
      sys_keys_check();
      wait(1);
   }
}
```

Die Funktion sys_keys_check wird die Auswertung übernehmen. Wir erstellen diese void sys_keys_check() Funktion und werden über logische ODER Verknüpfungen die 4 Zustände berechnen:

```
void sys_keys_check ()
{
   UP    = ((key_cuu || key_w) || (joy_force.y > 0));
   DOWN  = ((key_cud || key_s) || (joy_force.y < 0));
   LEFT  = ((key_cul || key_a) || (joy_force.x < 0));
   RIGHT = ((key_cur || key_d) || (joy_force.x > 0));
}
```

Das scheint erst sehr verwirrend, aber diese Ausdrücke sind sehr logisch. Nehmen wir mal die Zeile für UP: die "key" -Ausdrücke fragen die Tastatur nach dem Zustand der Tasten "Pfeil-nach-oben" und "W" ab. "joy_force.y" gibt die vertikale Position des Analogsticks wieder. Ist also joy_force.y > 0, wird der Analogstick nach oben gedrückt. Wenn also MINDESTENS einer dieser Zustände eintritt, wird UP auf 1 geschaltet, ansonsten ist UP ausgeschaltet. Wenn wir jetzt auf UP reagieren, wissen wir NICHT, WIE der Benutzer das eingegeben hat, wir wissen aber, dass er es so wünscht. Die restlichen logischen Ausdrücke erschließen sich leicht.

Wir wollen nun in pl_turn diese 4 states auswerten, um die Richtung, in die der Spieler Rudi lenken will, zu bestimmen. Den Richtungsvektor können wir dann auch gleich dazu benutzen, um Rudi in die richtige Richtung zu drehen:

```
void pl_turn ()
{
   vec_set(vecTemp, vector(RIGHT - LEFT, UP - DOWN, 0));
   vec_to_angle(my.pan, vecTemp);
}
```

Wenn wir die entsprechenden Tasten drücken, dreht er sich sofort richtig - ausgezeichnet! Die Daten, die speichern, wohin sich unser Rudi bewegt, müssen irgendwo zwischengespeichert werden, damit die Funktion pl_move diese auswerten kann. Dafür legen wir ein Array von 3 Skills (Skills sind Werte, die lokal in einer Entity gespeichert werden kann – siehe manual) an. Wir setzen ein Skilldefine und nennen es:

```
//-----------------------------------------------------------------------
// defines
//-----------------------------------------------------------------------
```

```
#define moveDir   skill1   //2,3
```

Damit sich Rudi nicht schlagartig dreht, interpolieren wir die Bewegung. Wenn wir Positionen interpolieren wollen, können wir die Funktion vec_lerp nutzen. Für Winkel gibt es so etwas leider nicht, wir benötogen aber eine Art Interpolation für Winkel. Da dies eine eigene Hilfsfunktion sein wird, legen wir die Datei sysUtils.c/.h an (Utilities = dt. Werkzeuge) und fügen sie in die include-Hierarchie hinter dem Framework ein. In den Dateien steht:

sysUtils.h:

```
//-----------------------------------------------------------------------
// prototypes
//-----------------------------------------------------------------------

    VECTOR* ang_lerp (VECTOR* _result, VECTOR* _source, VECTOR* _dest, var _factor);
           VECTOR ANG_LERP_VEC; //private vector
```

sysUtils.c:

```
//-----------------------------------------------------------------------
// like vec_lerp, but interpolates angles correctly
//-----------------------------------------------------------------------
VECTOR* ang_lerp (VECTOR* _result, VECTOR* _source, VECTOR* _dest, var _factor)
{
    vec_set(_result, _source); //copy source into result
    vec_diff(ANG_LERP_VEC, _dest, _result); //difference vector

    ANG_LERP_VEC.x = ang(ANG_LERP_VEC.x); //adjust the angle validity
    ANG_LERP_VEC.y = ang(ANG_LERP_VEC.y);
    ANG_LERP_VEC.z = ang(ANG_LERP_VEC.z);

    vec_scale(ANG_LERP_VEC.x, _factor); //interpolate the rotation
    vec_add(_result, ANG_LERP_VEC); //apply interpolated angle

    return(_result); //return result angle
}
```

Die Funktionsweise von ang_lerp ist identisch mit der von vec_lerp. Der Winkel wird zu einem anderen auf dem kürzesten Weg interpoliert. Wir wollen das hier nicht vertiefen, da die Funktion für mathematisch Interessierte recht einfach zu verstehen ist. Alle weiteren allgemeinen Hilfsfunktionen schreiben wir hier hinein.

Wir können die Funktion nun zum Beispiel so benutzen:

```
void pl_turn ()
{
   //calculate movement direction
      vec_set(vecTemp, vector(RIGHT - LEFT, UP - DOWN, 0));

   //if we enter a valid direction, overwrite old movement vector
      if (vec_length(vecTemp) > 0) {
         vec_set(my.moveDir, vecTemp);
      }

     //interpolate angle
            vec_to_angle(vecTemp, my.moveDir);
            ang_lerp(vecTemp2, my.pan, vector(vecTemp.x, 0, 0), 0.4 * time_step);
            my.pan = vecTemp2.x; //set only pan!
}
```

Erst wenn wir eine Eingabe machen, wird der alte Bewegungsvektor überschrieben. Ansonsten interpolieren wir immer zum letzten eingetragenen Bewegungsvektor. In der ang_lerp -Anweisung verwenden wir deshalb ausschließlich vecTemp.x, weil wir nur den Drehwinkel interpolieren wollen. 0.4 * time_step ist der zeitkorrigierte Interpolationsfaktor. Wenn wir Richtung 1 erhöhen, ist die Interpolation härter und Rudi dreht sich schneller; gegen 0 ist die Drehung langsamer. Wir wollen jetzt diesen Zahlenwert von 0.4 in den Header auslagern und im Code ersetzen:

```
#define pl_turn_blendFac   0.4
```

Als letzte Tat in dieser Funktion normalisieren wir den Bewegungsvektor auf die Länge 1. Wenn wir z.B. Rudi

später doppelt so schnell machen wollen, könnten wir den Bewegungsvektor einfach mit 2 multiplizieren oder anders verarbeiten:

```
void pl_turn ()
{
    //calculate movement direction
           vec_set(vecTemp, vector(RIGHT - LEFT, UP - DOWN, 0));
           vec_normalize(vecTemp, 1);

    //(...)
```

Damit wären die Arbeiten an dieser Funktion beendet. Rudi kann sich nicht nur schön drehen, wir haben auch gleichzeitig einen Bewegungsvektor kalkuliert, den wir später verarbeiten können. Wir haben außerdem eine generelle Key-Binding-Methode kennengelernt und sogar noch eine tolle Winkel-Interpolation in unser Programm mit aufgenommen!

Die Gravitation - oder: vom Winde verweht

Eine weitere wichtige Information, die wir für die Bewegung brauchen, ist die Antwort auf die Frage, ob wir uns in der Luft befinden. Wenn ja, wie weit ist es bis zum Boden? Um beide Fliegen mit einer Klappe zu schlagen, erstellen wir ein Skilldefine, in das wir einfach die Höhe bis zum Boden einspeichern. Ist es = 0, stehen wir auf unseren Füßen (bzw. Hufe), ist es > 0, sind wir in der Luft und dieser Wert ist die Länge in Quants bis zum Boden. Ist der Wert < 0, stecken wir im Boden! Dann müssen wir schnellstens aus dem Boden raus mit Hilfe der absoluten Quantzahl, die im Skill gespeichert wird. In der "player.h" erstellen wir also folgendes Define:

```
#define height    skill4
```

In der Funktion pl_gravity werden wir einzig und allein diesen Wert berechnen. Dazu benutzen wir c_trace, um die Entfernung von der Entity bis zum Boden zu bestimmen.

```
void pl_gravity ()
{
   my.height = c_trace(my.x, vector(my.x, my.y, -2000), IGNORE_ME | IGNORE_PASSABLE) + my.min_z;

   deb_print(my.height);
}
```

Das erste Argument ist die Startposition. Wir nehmen den Origin (Modellursprung) und dann tracen wir von dort aus nach unten und ignorieren uns (IGNORE_ME) und alle passablen Gegenstände (IGNORE_PASSABLE). Die Länge des Tracestrahls wird zurückgegeben und ist in diesem Fall die Höhe über dem Boden. Wir rechnen dann die Strecke vom Origin bis zum unteren Ende der Boundingbox auf, damit wir direkt auf unseren Füßen stehen, wenn wir später die Gravitation anwenden. Zu Testzwecken geben wir diesen Wert aus. Im WED verschieben wir Rudi auf den Boden und führen den Code aus. Es wird 0 ausgegeben - das ist richtig! Dann schieben wir Rudi höher. Es wird uns ein Wert > 0 zurückgegeben. Das heißt, Rudi schwebt in der Luft und wir müssen ihn herunterbewegen. Dies wird aber nicht in dieser Funktion gemacht, sondern in der Funktion pl_move.

Die Bewegungsfunktion - Rudi das Ren(n)tier

Wir wollen in der Bewegungsfunktion nun Bewegung und Gravitation hintereinander ausführen. Dieser Code in der pl_move sollte das tun:

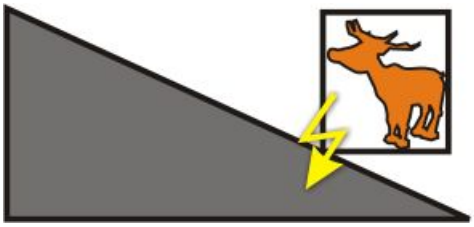
```
// if move vector is valid, move
    if (vec_length(my.moveDir) > 0) {

           //create relative motionvector
           vec_set(vecTemp, vector(vec_length(my.moveDir), 0, 0));      //base motion
           vec_scale(vecTemp, 48); //motion speed
           vec_scale(vecTemp, time_step); //time correction
```

```
            move_friction = 0; //no friction
            c_move(my, vecTemp, nullvector, IGNORE_PASSABLE | GLIDE); //move

        }

    // gravitation
        vec_set(vecTemp, vector(0, 0, -minv(my.height, 85 * time_step)));
        c_move(my, nullvector, vecTemp, IGNORE_PASSABLE | GLIDE);
```

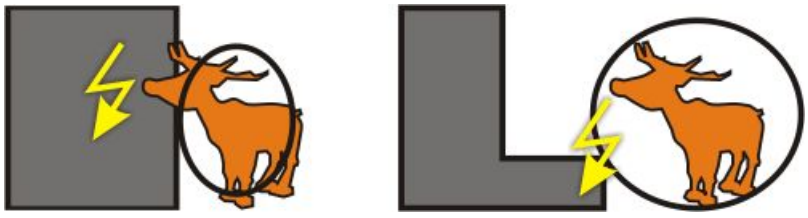
Wenn wir also einen Bewegungsvektor haben, bewegen wir uns. Dafür wird die Länge des Richtungsvektors genommen (der im Moment immer die Länge 1 hat) und mit der Laufgeschwindigkeit von Rudi multipliziert und dann zeitkorrigiert. „Zeitkorrigieren“ bedeutet in diesem Zusammenhang, dass wir darauf achten, dass Rudi nicht in jedem Frame immer genau die Distanz läuft, die er laufen soll, weil wir ja nicht wissen, mit welcher Bilderrate (also „frames per second“) das Spiel läuft. Es ist ganz einfach. Wenn sich Rudi mit mit 5 quants pro frame bewegt, dann ist er nach 100 frames 500 quants weitgelaufen. Allerdings kann es sein, dass auf dem einen PC das Spiel mit 30 fps und auf einem anderen mit 100 fps läuft. Während Rudi also auf dem einen Rechner eine Sekunde benötigt, um die 500 quants zu laufen, benötigt er gut 3 Sekunden auf dem anderen. Deshalb multiplizieren wir solche Angaben mit time_step, da time_step die Zeit von dem letzten Frame zum jetzigen in ticks wiedergibt (16 ticks = 1 Sekunde). Wenn wir das tun, dann wird ausgedrückt, das Rudi 5 quants pro tick läuft, was nun unabhängig von der framerate ist. Daher benötigt Rudi auf beiden Beispielrechnern die gleiche Zeit, bis er die 500 quants gelaufen ist – das nennt man Zeitkorrektur.

Daraufhin wird der realtive Richtungsvektor benutzt, um Rudi auf der Levelgeometrie zu bewegen und zwar mit möglichst wenig Reibung. Das Glide-Flag ist an, also sollte er auch an leichten Schrägen entlanggleiten. Bei der Gravitation fahren wir mit einer konstanten Geschwindigkeit herunter. Wenn sie größer wird als die tatsächliche Höhe, nutzen wir nur den Höhenwert.

Wenn wir mit Rudi ein wenig durchs Level laufen, stellen wir fest, dass wir umherlaufen, die Schrägen hochlaufen und an Abhängen runterfallen können, aber wenn wir beispielsweise auf den Schrägen plötzliche Drehungen machen, bleibt Rudi kurz stehen und fährt dann weiter - das war so nicht geplant. Der Grund ist, dass wir die Kollisionshülle mit c_setminmax auf die echten Ausmaße von Rudi gesetzt haben und nun eine "passende" und orientierte Kollisionsbox haben.



Wenn wir das nicht getan hätten, hätten wir eine Kugel als Hülle und würden in diesem Fall die Schrägen spielend erklimmen können. Der Haken an der Sache ist, das wir dann aber mit dem Kopf in der Wand stecken - ein Rentier ist nunmal alles andere als kugelförmig. Wenn wir die Kugelhülle vergrößern würden, dann hätten wir aber das Problem kollisionstechnisch vielleicht 2 oder 3 mal so breit wäre als sonst, was auch nicht gerade besonders toll ist.

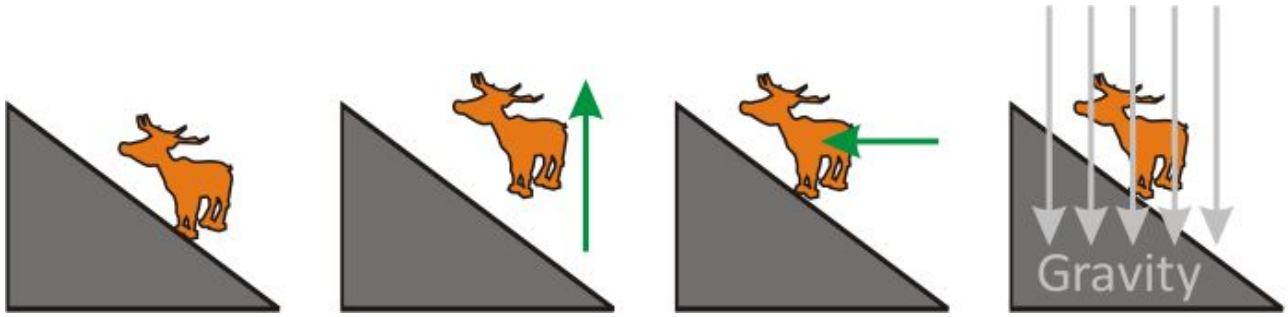


Zwar gibt es einige Variablen und Verfahren, die das Verhalten von c_move beeinflussen (dazu gehört unter anderem das Trennen von Bewegung und Gravitation), allerdings müssen wir hier tricksen, um zum gewünschten und butterweichen Ergebnis zu kommen. Das Problem ist, dass die Entity einen Quader als Kollisionserkennung besitzt und damit besonders an stark ansteigendenen Schrägen Probleme bereitet. Der Trick ist nun, dass wir die

Entity etwas anheben, die Entity vorwärts bewegen, wieder runterbewegen und dann die Gravitation ausführen.

```
        //(...)

        my.z += 10;                                                 //raise for a better margin
        c_move(my, vecTemp, nullvector, IGNORE_PASSABLE | GLIDE);   //move
        my.z -= 10;                                                 //revert margin raise
    }

// gravitation
    vec_set(vecTemp, vector(0, 0, -minv(my.height, pl_gravitation * time_step)));
    c_move(my, nullvector, vecTemp, IGNORE_PASSABLE | GLIDE);

    //(...)
```

Wir erhöhen erst Rudi um ein paar Quants, damit wir etwas Spielraum bekommen und machen dies dann wieder rückgängig, bevor der Gravitationscode ausgeführt wird. Wenn wir jetzt mir Rudi durch das Testlevel fahren, gibt's keine Probleme an Schrägen mehr.



Wir lagern dieses offset über ein define aus:

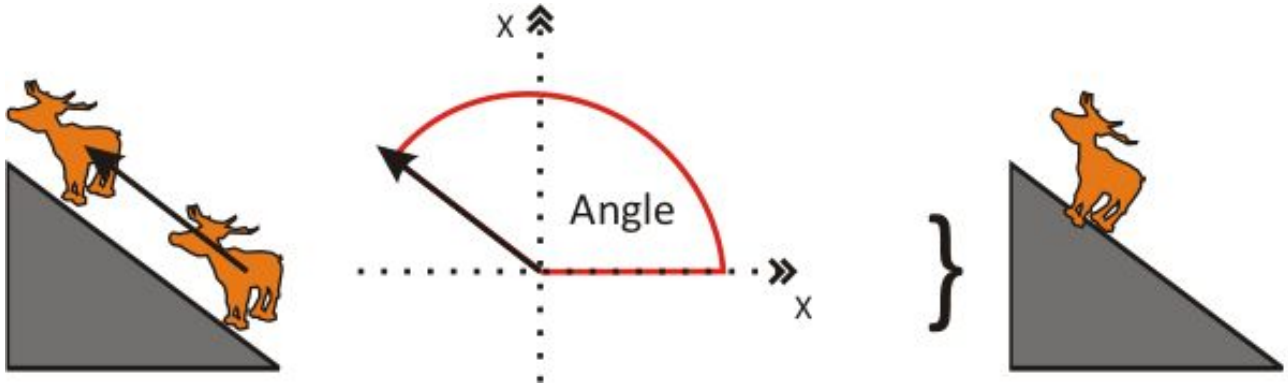
```
#define pl_move_marginOffset  10
```

Im Prinzip haben wir die grundsätzliche Spielerbewegung programmiert, wir können uns durch das ganze Testlevel bewegen. Wir wollen aber ein noch besseres Ergebnis!

Generische Neigungsanpassung an Oberflächen

Eine Sache, die noch komisch wirkt, ist die Tatsache, dass Rudi immer gerade steht. Wenn er aber an Schrägen entlanggeht oder irgendwo herunterspringt, wäre es schöner, wenn er sich der Oberflächenneigung oder dem Fall nach unten anpassen würde. Es gibt verschiedene Ansätze, wie man das lösen kann. Wenn man wie bei unserer Höhenberechnung mit einem Trace-Strahl nach unten scannt, wird auch der Wert der Normalen der Blockfläche unter uns gespeichert. Die Normale eine Oberfläche ist ein Richtungsvektor, der senkrecht von der Oberfläche wegzeigt. Wenn man eine Figur in die Richtung der Normalen auf die Oberfläche stellt, dann steht sie senkrecht darauf - genau das, was wir wollen. Also verlieren sich viele Leute in dem Irrglauben, dass dies die Lösung für unser Problem sei. Sie erkennen dabei aber nicht, dass sich die Figur sich um die Normale drehen und in allen Richtungen auf der Oberfläche stehen kann, wenn wir eine Figur in Richtung der Normalen hinstellen.
Wir wissen aber nicht, wie die Figur um die Normale herum gedreht sein muss, das ist völlig offen.

Eine von mir benutze Methode umgeht dieses Problem sehr elegant und ist auch völlig unabhängig davon, ob wir uns auf Blöcken bewegen, auf Models, Sprites oder Terrains. Nachdem wir uns bewegt haben, bilden wir den Richtungsvektor von der alten zur neuen Position, wandeln diesen in einen Winkel um und schreiben ihn in unseren eigenen Winkel. Die Entity passt sich dann immer der Oberfläche an.



Zunächst müssen wir die Position zwischenspeichern („loggen“). Das machen wir in einem Skill-Vektor und legen dafür ein define an:

```
#define logPos    skill5   //6,7
```

Das Loggen wird über eine Funktion geschehen, die pl_log heißt:

```
//---------------------------------------------------------------------
// logs some current player stats
//---------------------------------------------------------------------
void pl_log ()
{
   vec_set(my.logPos, my.x); //log position
}
```

Beim Start steht im log-Vektor der Nullvektor, deshalb müssen wir bei der Initialisierung einmal die Funktion pl_log aufrufen, damit Rudi mit korrigierten Log-Werten startet:

```
void pl_rudi_init ()
{
   Rudi = my;           //indicate that I am Rudi!

   c_setminmax(my);     //calculate real bounding box

   pl_log();            //reset logging
}
```

Das Interpolieren des Winkels geschieht in einer Funktion namens pl_interpolate:

```
//---------------------------------------------------------------------
// does some interpolation calculation to make the actor move and
// behave smmoothly
//---------------------------------------------------------------------
void pl_interpolate ()
{
   //INTERPOLATE MOTION ANGLE

      //interpolate only if we actually moved
      if (vec_length(my.moveDir) > 0) {

         vec_diff(vecTemp, my.x, my.logPos); //look from old pos to current position

         vec_set(vecTemp2, vecTemp); //copy; min. move dist will be checked on XY plane
         vecTemp2.z = 0;

         if (vec_length(vecTemp2) > 10 * time_step) { //min. XY plane movement

            vec_to_angle(vecTemp, vecTemp); //calc angle
            ang_lerp(my.pan, my.pan, vecTemp, 0.25 * time_step); //interpolate angle
         }
      }
}
```

Das Prinzip ist ganz einfach: Wenn wir uns bewegt haben, überprüfen wir, ob sich unsere Position im Vergleich zur

jetzigen in Bezug auf die XY -Ebene verändert hat. Tun wir dies nicht, kann es zu falschen Winkeln kommen, wenn wir uns nur in Richtung der Z -Achse bewegen oder wenn wir gerade gegen eine Wand laufen.

Nachdem wir die Prototypen in der "player.h" erstellt haben, tragen wir die beiden Funktionen in pl_rudi ein - und zwar die Interpolationsfunktion vor der log Funktion, weil sonst die aktuelle Position immer der geloggten Position entspräche. Wenn wir jetzt mit Rudi durch das Level laufen, passt er sich schön, sauber und vor allem butterweich der Levelumgebung an.

Zu guter Letzt lagern wir noch die Zahlenwerte in Konstanten aus:

```
#define pl_lerp_surfaceAng_fac         0.25
#define pl_lerp_surfaceAng_threshold   10
```

Eine dynamischere Kamera

Damit die Kamera nicht so starr wirkt, wollen wir sie ihre Position und ihren Winkel interpolieren lassen. Das erzeugt dann den Effekt einer schwingenden Kamera, was dynamischer wirkt. Dazu benutzen wir den vec_lerp -Befehl, um zwischen der alten und angepeilten Position zu interpolieren, nachdem wir die Zielkoordinaten der Kamera berechnet haben. Dasselbe Prinzip wenden wir bei der Winkelberechnung an:

```
void cam_update ()
{
   if (def_camera > 0) {return;} //stop here, if debug camera is on

   //position
      vec_set(vecTemp.x, cam_std_offset);
      vec_add(vecTemp.x, Rudi->x);

      //interpolation
      vec_lerp(camera.x, camera.x, vecTemp.x, 0.5 * time_step);

   //look at player
      vec_diff(vecTemp.x, Rudi->x, camera.x);
      vec_to_angle(vecTemp.x, vecTemp.x);

      //interpolation
      ang_lerp(camera.pan, camera.pan, vecTemp.x, 0.2 * time_step);
      camera.roll = 0;

   //camera angle
      camera.arc = cam_std_arc;
}
```

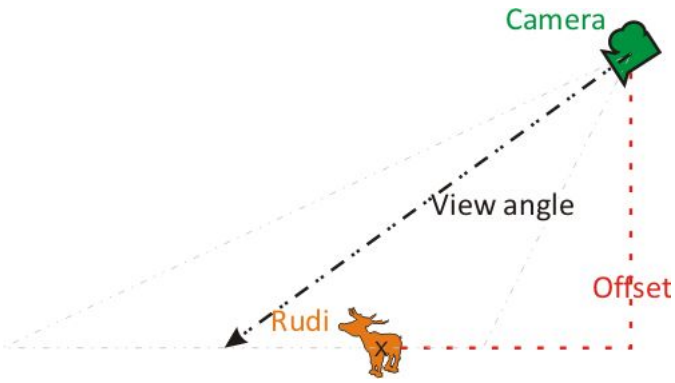
Bei der Winkelberechnung setzen wir den Rollwinkel der Kamera auf 0, damit die Kamera nicht schief steht. Die beiden Interpolationsfaktoren lagern wir wie folgt aus:

```
#define cam_interpolate_pos   0.2
#define cam_interpolate_ang   0.2
```

Wenn wir nun mit Rudi durch das Testlevel laufen, sieht die Kamera schon gleich viel besser aus, weil sie sich dynamisch anpasst. Außerdem erzielen wir auch andere Winkel, mit der wir von oben auf die Umgebung schauen, wodurch wir einen besseren 3D-Effekt erzeugt haben.

Dadurch, dass wir sowohl die Position als auch den Winkel interpolieren, "schleift" die Kamera nach. Ein weiteres Problem ist, dass wir nicht unbedingt dass sehen, was vor dem Spieler liegt, weil wir auf den Spieler gucken und ihn quasi zentrieren. Da wir aber relativ flott durch das Level galoppieren und aufpassen müssen, dass wir nicht gegen Hindernisse laufen, wäre es gut, wenn die Kamera sich in die Richtung drehen würde, in die wir laufen. Wenn das geschieht, hätten wir das erste Problem auch eliminiert.

Dazu berechnen wir einfach einen Punkt, der ein bisschen vor dem Spieler liegt und gucken dann dorthin.



Das sieht im Code dann so aus:

```
//angle

   //calc a relative look-at point in front of the player
      vec_set(vecTemp, vector(100,0,0));
      vec_rotate(vecTemp, Rudi->pan);
      vec_add(vecTemp, Rudi->x);

   //look at there
      vec_diff(vecTemp.x, vecTemp.x, camera.x);
      vec_to_angle(vecTemp.x, vecTemp.x);

   //interpolation
      ang_lerp(camera.pan, camera.pan, vecTemp.x, cam_interpolate_ang * time_step);
      camera.roll = 0;
```

In diesem Fall liegt der Look-At -Punkt 100 quants vor dem Spieler. Dadurch erhöht sich die Übersicht drastisch. Das Offset für den Look-At Punkt lagern wir als Vektor "cam_std_lookAt" aus und fügen den Vektor in den Code ein.

```
VECTOR* cam_std_lookAt = {x = 100; y = 0; z = 0;}
```

Der finale Touch - die Animation

Wir haben es im Prinzip geschafft: wir haben ein Testlevel, eine dynamische Kamera und einen Rudi, der sich echt gut durch das Level bewegen kann. Nur "schwebt" er noch durch das Level. Wir wollen ihm jetzt eine einfach Laufanimation geben, indem wir eine Funktion namens pl_animate schreiben und nach allen Bewegungsberechnungen vor der Interpolation platzieren. Wenn er nur steht, soll er eine „Idle“-Animation abspielen. Die Funktion sieht so aus:

```
void pl_animate ()
{
   //are we moving?
   if (vec_dist(my.logPos, my.x) > 2 * time_step) { //yes; This is the walk animation

      //increase animation counter and clamp it
      my.animCounter = (my.animCounter + 12 * time_step) % 100;
      ent_animate(my, "walk", my.animCounter, ANM_CYCLE); //animate!

   } else {

      //increase animation counter and clamp it 1t 100 percent
      my.animCounter = (my.animCounter + 3 * time_step) % 100;
      ent_animate(my, "idle", my.animCounter, ANM_CYCLE); //animate!

   }
}
```

Wir schauen zunächst, ob wir uns tatsächlich bewegt haben und berechnen dann die Distanz zwischen letztem und diesem Frame und müssen daher auch den Vergleichswert zeitkorrigieren. Wir nehmen hierfür einen

Schwellenwert von 2, damit es zu keinen Fehlern durch Ungenauigkeiten kommt. Sollten wir uns nicht bewegt haben wie beispielsweise zu Beginn des Spiels, dann stehen wir und spielen die idle-Animation ab. Wir haben hier einen Skilldefine als Animationszähler benutzt, um die Animation zu steuern. Zusätzlich dazu wollen wir die Variablen auch in den player.h auslagern.

```
//(...)

var   pl_anim_move_threshold = 2;

var   pl_anim_walk_speed = 12;
char* pl_anim_walk_frame = "walk";

var   pl_anim_idle_speed = 3;
char* pl_anim_idle_frame = "idle";
```

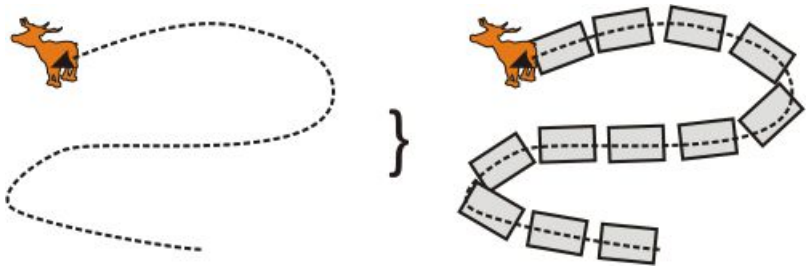
Kapitel 5: Das Schlittensystem

Wir wollen uns in diesem Kapitel mit dem Schlittensystem befassen. Vor der Umsetzung müssen wir uns Gedanken machen, was die Schlitten leisten sollen und wann was passiert.

Die TRACK Datenstruktur

Wir haben festgelegt, dass neue Schlitten an die Schlange gehängt werden, sobald Rudi einen Geschenkehaufen aufsammelt. Eine andere Eigenschaft der Schlittenschlange ist das bedingungslose Herfahren hinter Rudi. Bei vielen Snake-Klonen ist es irrtümlicherweise schon Methode geworden, die Glieder am jeweiligen Vorglied zu orientieren und dann in diese Richtung aufzurücken zu lassen. Das ist zwar einfach programmiert und sieht auch ganz witzig aus, nur entsteht dadurch ein schwerer Fehler. Wenn man nämlich eine Kurve fährt, **schneidet** die Schlange diese und fährt einen kürzeren Weg als man eigentlich mit Rudi gelaufen ist. Bei "freien" Arealen ist das auch gar nicht so wichtig, aber wir wollen eine Winterlandschaft mit vielen Objekten.

Deshalb müssen wir dafür sorgen, dass eine Art "Pfad" während des Spiels aufgezeichnet wird, an dem sich die Schlitten orientieren können. Wir werden auf die Schlitten auch keine Bewegung mit Kollisionsberechnung ausführen, da der Pfad statisch ist und aufgezeichnet wird. Zwar wird Rudi später mit den Schlitten kollidieren können, das werden wir aber nicht über die Kollisionserkennung der engine lösen. Aus den Anhalts- oder Kontrollpunkten kann man darauf schließen, wo der Schlitten sich befindet. Ziel ist, auch bei vielen Schlitten möglichst wenig Rechenleistung zu verbrauchen. Wir verwenden drei unterschiedlische Schlittenmodelle, die wir dann zufällig auswählen und hinzufügen.. die Dateien lauten „sledgeA.mdl“ bis „sledgeC.mdl“.

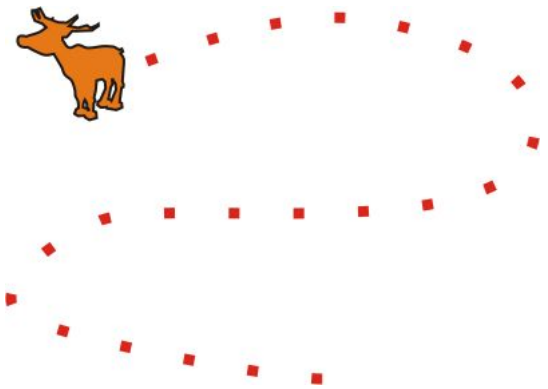


Die TRACK Datenstruktur

Wir wollen wir Rudis Lauf aufzeichnen, damit sich die Schlitten an diesem Pfad orientieren können. Dafür gibt es mehrere Lösungen und ich stelle nun eine recht einfache Möglichkeit vor.

Wenn wir in einem bestimmten Intervall Kontrollpunkte setzen, können sich die Schlitten den Pfad ableiten, an dem sie entlanglaufen. Wie die Schlitten sich daran ausrichten, behandeln wir später. Wir wollen zunächst eine geeignete Datenstruktur finden, die uns diesen Pfad in Form von Kontrollpunkten speichert. Damit wir nicht

unendlich viele Punkte speichern, können wir sagen, dass wir nur eine bestimmte Anzahl an Kontrollpunkten gleichzeitig vorliegen haben. Idealerweise muss die Länge der Kontrollpunkt-Kette länger sein als die mögliche Länge aller Schlitten – sonst kommt es zu Fehlern. Weil wir eine begrenzte Anzahl an Kontrollpunkten haben, müssen wir immer, wenn ein neuer Kontrollpunkt gesetzt werden soll, den letzten entfernen und vor den ersten Punkt als „neuen“ dort hinsetzen.

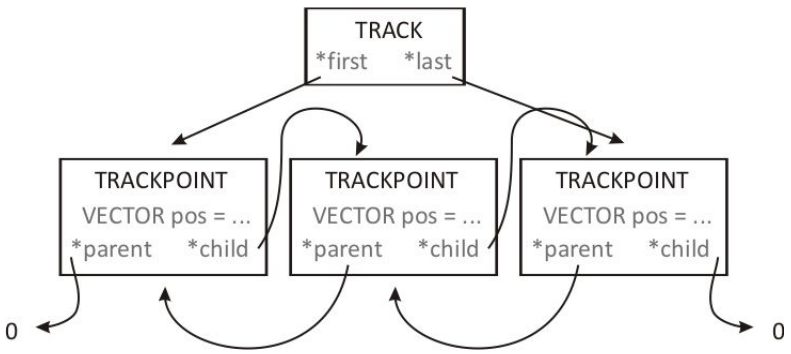


Für solche eine Anforderung bietet sich beispielsweise eine verkettete Liste an. Die Liste enthält all diese Kontrollpunkte und verknüpft den letzten Punkt mit dem Anfang der Liste beim Einfügen eines Kontrollpunktes. Weil man dabei nur mit Zeigern jongliert und die Elemente mit nur einer Position ganz wenig Daten speichern müssen, ist das einfach und schnell. Andere Verfahren, wie das Speichern und Verschieben der Positionen in einem Array ist auch mögöich, aber erstens ist das nicht dynamisch und zweitens viel langsamer.

Wir wollen eine zusammenhängende Datenstruktur dafür festlegen. Zunächst erzeugen wir eine Struktur namens "TRACK", die den Pfad festhält und die Liste verwaltet. Zweitens erstellen wir eine Struktur namens "TRACKPOINT", die den einzelnen Kontrollpunkt speichert. Dazu könnte man folgende struct -Definitionen benutzen:

```
typedef struct TRACKPOINT {
   VECTOR* pos;
   struct TRACKPOINT *parent, *child;
} TRACKPOINT;

typedef struct TRACK {
   TRACKPOINT *last;
   TRACKPOINT *first;
} TRACK;
```

Die Struktur TRACK besitzt nur 2 Zeiger, die auf den Kopf und das Ende der Kette weisen. Jedes Kettenglied ist vom Typ TRACKPOINT und speichert einen Positionsvektor und Zeiger auf das Vorgängerglied und das Nachfolgeglied ab. Die folgende Zeichnung veranschaulicht dies:



Dieses System ist unabhängig vom Spiel und könnte jede Form von 3D-Pfaden speichern. Wir wollen im "sys" -Ordner 2 Dateien namens track.c/.h erstellen und zusätzlich zum Kopf der Datei eine ausführlich dokumentierte Fassung dieser beiden Structs einfügen:

```
//-----------------------------------------------------------------
```

```
// TRACK RECORDING DATA STRUCTURE
//---------------------------------------------------------------------
// FIGURE:      last       first
//               |           |
// points: 0 <- (A) - (B) - (C) -> 0
//             child <-|-> parent
//
// time:   "older" ...   ... "newer"
//---------------------------------------------------------------------
// STORES A POSITION IN 3D SPACE & IS PART OF A CHAIN OF POINTS
//
// The parent point is a newer record point, whereas the child
// point is older than the current

   typedef struct TRACKPOINT {
      VECTOR* pos;
      struct TRACKPOINT *parent, *child;
   } TRACKPOINT;

// STORES A LINKED LIST OF TRACKPOINTS TO STORE A CHANGING PATH
//
// Stores a path which is recorded by a moving entity. The "first"
// trackpoint is at the recording entity, whereas the "last" entity
// is the oldest recorded trackpoint. If a new trackpoint is
// recorded, the position is written into the last trackpoint and
// this point will be moved in front of the first point to become
// the new "first" point, the second last point will be the new
// "last" point

   typedef struct TRACK {
      TRACKPOINT *last;
      TRACKPOINT *first;
   } TRACK;
```

In der track.c -Datei schreiben wir nun alle wichtigen Operationen zum **Erstellen** dieser Objekte, zum **Einfügen** von *neu erstellten* Trackpoints in den Track und dem **registrieren** eines *neuen* Trackpoints, wobei dann wie beschreiben der letzte Punkt der Neueste wird. Zusätzlich benötigen wir eine Operation, die ein Track-Objekt und alle darin enthaltenen Track-Points **löscht** und eine **Debug**-Operation, die uns den Track in der 3D-Szene anzeigt. Ich zeige und erkläre die Funktionen der Reihe nach:

```
TRACK* create_TRACK (VECTOR* initPos, int nrPoints)
{
   TRACK* temp = malloc(sizeof(TRACK));

   temp->last = 0;
   temp->first = 0;

   if (nrPoints > 0) {
      temp->first = create_TRACKPOINT(initPos);
      temp->last = temp->first;

      int i;
      for (i = 1; i < nrPoints; i++) {
         add_TRACK_point(temp);
      }
   }

   return(temp);
}
```

Die Funktion erzeugt ein neues Track-Objekt mit der angegebenen Anzahl der Punkte (nrPoints) und einem Initialisierungsvektor (initPos); idealerweise ist das die aktuelle Position der aufzeichnenden Entity. Es werden ein erster Punkt erzeugt und die first- und last- Zeiger gesetzt. Danach werden alle anderen Punkte ans Ende hinzugefügt. Das Erzeugen eines Punktes ist recht trivial, er wird alloziert und der übergebende Vektor wird hineingeschrieben. Der Punkt ist im erzeugten Zustand unverknüpft mit anderen Punkten.

```
TRACKPOINT* create_TRACKPOINT (VECTOR* initPos)
{
   //allocation
      TRACKPOINT* temp = malloc(sizeof(TRACKPOINT));
      temp->pos = malloc(sizeof(VECTOR));
```

```
   //initialization
      vec_set(temp->pos, initPos);
      temp->parent = 0;
      temp->child = 0;

   return(temp);
}
```

Das Hinzufügen eines Punktes zur Kette erzeugt einen Punkt und verknüpft ihn mit dem Ende der Kette. Dabei wird er mit dem Vektor des letztens Punktes initialisiert (ist dann interessant, wenn eine bereits vorhandene und schon laufende Kette erweitert wird):

```
void add_TRACK_point (TRACK* track)
{
   track->last->child = create_TRACKPOINT(track->last->pos);
   track->last->child->parent = track->last;
   track->last = track->last->child;
}
```

Die Funktion zum entfernen eines Track -Objektes ist:

```
void remove_TRACK (TRACK* track)
{
   if (track == 0) {return;}

   TRACKPOINT* point = track->first;
   TRACKPOINT* point_next;

   while (point != 0) {
      free(point->pos);
      point_next = point->child;
      free(point);
      point = point_next;
   }

   free(track);
   track = 0;
}
```

Erst wird der Speicher aller Punkte freigegeben, dann der Speicher für das Track-Objekt an sich.

Die Funktion, die einen neuen Punkt registriert, ist die eigentlich interessante Funktion. Sie nimmt den letzten, ältesten Punkt, überschreibt ihn mit seiner neuen Position und setzt ihn an den Anfang der Liste. Die Funktion muss die Punkte am Anfang neu verknüpfen, sodass der first-Zeiger auf den neuen last-Punkt zeigt und der neue last-Punkt auf den ursprünglich vorletzten Punkt zeigt. Des Weiteren muss die Funktion vorne den neuen Punkt sauber mit dem alten first-Punkt verknüpfen und die Enden mit Null-Zeigern austatten:

```
void reg_TRACK_point (TRACK* track, VECTOR* pos)
{
   //we need at least 2 points, which is indicated by
   //different last/first pointers:

   if (track->last != track->first) {

      // Set new position
         vec_set(track->last->pos, pos);

      // Push last point to the front of the chain and modify the pointers
      // so that the chain is repaired

         track->last->child = track->first;
         track->last->parent->child = 0;
         track->first->parent = track->last;
         track->first = track->last;
         track->last = track->last->parent;
         track->last->child = 0;
   }
}
```

Zu guter Letzt schreiben wir uns eine kleine Debugfunktion, die uns an jedem Track-Point einen gut sichtbaren Punkt malt. Diese Funktion soll nur einen frame lang gültig sein:

```
void debug_TRACK (TRACK* track)
{
   TRACKPOINT* point = track->first;

   //go through the chain and draw each point with a big red dot

      while (point != 0) {
         draw_point3d(point->pos, vector(0,0,255), 0, 8);
         point = point->child;
      }
}
```

Um nun Rudi seinen Lauf loggen zu lassen, fügen wir in der player.h einen Zeiger auf ein TRACK -Objekt hinzu:

```
//-----------------------------------------------------------------------
// logging
//-----------------------------------------------------------------------

     TRACK* trackRudi = 0;
```

Rudi wird jedes Mal, wenn er in seinem Level startet, dieses Track-Objekt initialisieren, falls es nicht schon existiert. Deshalb schreiben wir in die Initialisierungsfunktion pl_rudi_init:

```
//create track logging datastructure
if (trackRudi == 0) {
   trackRudi = create_TRACK(my.x, 20);
}
```

In diesem Fall wird ein TRACK -Objekt mit 20 möglichen Punkten erstellt. Das wird später im Spiel nicht reichen, aber zum ersten Ausprobieren reicht das. Das Loggen der Positionen in das TRACK -Objekt erledigen wir in der bereits vorhandenen Logging-Funktion pl_log in der "player.c". Das Logging soll, damit der Lauf möglichst gleichförmig rekonstruierbar ist, in gleichlangen Abständen erfolgen - beispielsweise alle 60 quants:

```
void pl_log ()
{
   //INTERNAL LOGGING
      vec_set(my.logPos, my.x); //log position

   //TRACK RECORDING

      // Log the position into the track determined by
      // equidistant intervals

      if (trackRudi != 0) {
         if (vec_dist(my.x, trackRudi->first->pos) > 60) {
            reg_TRACK_point(trackRudi, my.x);
         }
      }
}
```

Damit wir das Loggen auch sehen, können wir in der Schleife in pl_rudi vor dem wait(1); unsere Debugfunktion aufrufen:

```
      //(...)

      pl_animate();
      pl_interpolate();
      pl_log();

      debug_TRACK(trackRudi);

      wait(1);
   }
}
```

Wenn wir jetzt mit Rudi herumlaufen, sehen wir, dass eine Spur von roten Punkten seinen exakten Weg nachzeichnet.

Vorbereitung des Trackings für das Schlittensystem

Wir haben jetzt das Tracking testweise mit hypothetischen Werten ausprobiert. Bevor wir jedoch unseren ersten Schlitten dafür bauen, wollen wir das System noch etwas anpassen, bevor wir es einsetzen.

Ein Problem ist die Tatsache, dass wir die Kontrollpunkte immer an Rudis Origin gesetzt haben, nun „fliegen“ die Schlitten durch die Lüfte und gleiten nicht auf dem Boden. Anstatt mit Performance zehrenden c_trace -Aufrufen für die Schlitten anzufangen, nutzen wir lieber die Tatsache aus, dass Rudi immer korrekt auf dem Boden steht. Rudi setzt den Kontrollpunkt direkt auf die Höhe seiner Hufe! Wenn wir jetzt den Origin des Schlittens auf die Höhe der Schlittenkufen stellen (also dort wo der Boden berührt wird), steht der Schlitten immer richtig! Wir müssen dazu nur den Aufruf von reg_TRACK_point in der Funktion pl_log ändern. An diesem Zeitpunkt steht Rudi nämlich schon korrekt und wir nutzen die Höhe unserer Boundingbox aus, um den Punkt auf den Boden zu stellen:

```
reg_TRACK_point(trackRudi, vector(my.x, my.y, my.z + my.min_z));
```

Im Moment steht der Wert für die Punktabstände auf 60, was etwas weniger als die Hälfte einer Schlittenlänge entspricht. Das ist ein guter Abstand weil wir dann pro Schlitten eine doppelte Genauigkeit haben, also nehmen wir diesen Wert und lagern ihn als die Variable pl_track_logDist in die player.h aus. Des Weiteren wird in der Funktion pl_rudi_init im Aufruf von create_TRACK die Anzahl der Kontrollpunkte mit 20 angegeben. Bei ca. 180-200 quants Abstand entspricht dies einer Länge von etwa 6 Schlitten. Bei unserem Schlittensystem wollen wir mit maximal 20 Geschenkepaketen rechnen, was zu 210 Schlitten führen würde. Dafür bräuchten wir ca. 650 Kontrollpunkte. Damit wir etwas Spielraum nach hinten haben, erhöhen wir die Zahl auf 1000 Kontrollpunkte und lagern diese Zahl auch als Variable pl_track_points aus und ersetzen natürlich die fixen Zahlenwerte im Code damit:

```
//------------------------------------------------------------------------
// logging
//------------------------------------------------------------------------

   TRACK* trackRudi = 0;
   int   pl_track_logDist  = 60;
   int   pl_track_points   = 1000;
```

Zuletzt löschen wir das Trackobject und seine Kontrollpunkte. Da wir Rudis TRACK -Objekt nur maximal einmal erzeugen, reicht es, es beim Herunterfahren der Engine es zu löschen. Für solche Dinge haben wir den on_exit -Event bereits angefasst und im Framework eine eigene Funktion geschrieben, die für solche Dinge konzipiert ist, nämlich die Funktion sys_close in der Datei sys.c. Wir fügen einfach folgende Zeilen vor dem sys_exit(""); ein:

```
//removing Rudi's tracking object
   remove_TRACK(trackRudi);
```

Die Schlittenkaravane: eine weitere verkettete Liste

Nachdem wir nun ein Instrument in unserer Hand halten, um Rudis Lauf zu rekonstruieren, können wir uns endlich mit den Schlitten an sich befassen. Ähnlich unserer TRACK -Datenstruktur sind die Schlitten nichts Anderes als eine verkettete Liste - im wahrsten Sinn des Wortes. Die einzelnen Schlitten sind über Schnüre miteinander verbunden und haben einen Vorder- und einen Hintermann. Der vorderste Schlitten ist jedoch an Rudi festgeschnürt, weil der die Schlitten zieht. Im Umkehrschluss muss Rudi auch wissen, welcher der erste Schlitten ist, den er zieht, und wir müssen auch wissen, welcher der letzte ist, damit wir dort neue Schlitten einfügen können. Wir wollen all diese Daten über Entity -Zeiger festhalten, da Rudi und die Schlitten auch Entities sind. Die Zeiger speichert jede Entity selbst in ihren Skills.

Wir fügen in der player.h zunächst die folgenden Skilldefinitionen hinzu:

```
#define sledgeFirst  skill9
#define sledgeLast   skill10
```

Wir vereinbaren: ist my.sledgeLast = 0, dann hat Rudi keinen Schlitten. Hat er einen Schlitten, zeigt my.sledgeLast auf den letzten und my.sledgeFirst auf den ersten. Bei nur einem Schlitten zeigen beide Zeiger auf denselben

Schlitten.

Für die Schlitten erstellen wir 2 Dateien namens sledge.c/.h im "game" -Verzeichnis und inkludieren sie. In der sledge.h fügen wir zwei Skilldefinitionen ein, die einen Zeiger auf den Vordermann und einen auf den Hintermann speichern:

```
//-----------------------------------------------------------------
// defines
//-----------------------------------------------------------------

   #define sledgeFront  skill9
   #define sledgeBack   skill10
```

Wir vereinbaren für die Schlitten: Wenn der Zeiger auf den Vordermann = 0 ist, wird das eine Kettenreaktion nach hinten auslösen, die die Schlitten nacheinander zerstört, nach einem Crash oder wenn die Kette irgendwo getrennt wird. Dieses Verhalten werden wir später einprogrammieren. Ist der Vordermann ungleich 0, verhält sich der Schlitten normal. Wir werden später auch ein Band zwischen 2 Schlitten einrichten. Wenn der Hintermann gleich 0 ist, dann wird kein Band angezeigt.

Wir wollen nun eine Funktion schreiben, die einen Schlitten erstellt und mit Rudi oder bereits vorhandenen Schlitten verknüpft. Danach setzen wir uns mit der Frage auseinander, wie sich der Schlitten auf dem Pfad ausrichtet.

Zunächst erzeugen wir uns in der sledge.c eine Schlittenfunktion, die ähnlich wie Rudi erst eine Init-Funktion aufruft und dann seine Hauptschleife ausführt. In der Initialisierung soll der Schlitten passable gemacht werden und eine Zufallsskin erhalten, denn das mitgelieferte Schlittenmodell hat 3 Skins. Da der Schlitten auch kaputt gehen kann, wollen wir einen weiteren Skill definieren, der angibt, wann der Schlitten "tot" ist. Wir könnten dafür auch ein Flag benutzen, aber wir nehmen einen Skill, damit wir mehrere "Todesarten" realisieren können - steht der Wert auf 0, ist der Schlitten noch nicht kaputt. Den Skill definieren wir wie folgt:

```
#define sledgeDeath skill80
```

Wir lösen dieses Ereigniss erst später aus, bauen es aber als Wahrheitsbedingung für die Main-Loop des Schlittens ein.

In der sledge.c steht dann:

```
//-----------------------------------------------------------------
// main sledge action
//-----------------------------------------------------------------
void sl_main ()
{
   sl_init();

   while (my.sledgeDeath == 0) {

      wait(1);
   }
}

   //-----------------------------------------------------------------
   // sledge initialization
   //-----------------------------------------------------------------
   void sl_init ()
   {
      my.passable = on;    //make passable (faster!)

      //random skin
         my.skin = 1 + (int)(random(3));
   }
```

Dort, wo ich den todo - Kommentar geschrieben habe, werden wir später weiter fortfahren. Wenn man in seinen Code diverse Todo-Punkte aufschreibt, erleichtert das später die Weiterarbeit am Code.

Die Funktion, die einen Schlitten erzeugt und an Rudi, bzw. die restlichen Schlitten anfügt, heißt „sl_add“. Wenn Rudi keine Schlitten hat, bindet die Funktion einen neuen Schlitten an Rudi an, wenn bereits Schlitten existieren,

bindet sie sie an den letzten an. Wenn wir den Schlitten erzeugen, dann wählen wir aus 3 verschiedenen Modellen, damit die Kette nicht allzu gleich und damit langweilig aussieht. Die Schlitten heißen jeweils „sledgeA.mdl“ bis „sledgeC.mdl“. Dies können wir durch eine einfache String Operation leisten!

```
void sl_add ()
{
   // if there is no Rudi, there won't be sledges
   if (Rudi == NULL) {return;}

   // if a last sledge exists, add to sledge chain
   if (Rudi->sledgeLast) {

      char* buffer[32];
      sprintf(buffer, "sledge%c.mdl", 'A'+(int)(random(3)));
      entTemp[0] = ent_create(buffer, ((ENTITY*)Rudi->sledgeLast)->x, sl_main);

      entTemp[0]->sledgeFront = Rudi->sledgeLast;
      entTemp[1] = Rudi->sledgeLast;
      entTemp[1]->sledgeBack = entTemp[0];
      entTemp[0]->sledgeBack = 0;
      Rudi->sledgeLast = entTemp[0];

      //orientate like parent
      vec_set(entTemp[0]->pan, ((ENTITY*)entTemp[0]->sledgeFront)->pan);

   } else {

      // Rudi has no sledges, so this will be the first one

      char* buffer[32];
      sprintf(buffer, "sledge%c.mdl", 'A'+(int)(random(3)));
      entTemp[0] = ent_create(buffer, Rudi->x, sl_main);

      Rudi->sledgeFirst = entTemp[0];
      Rudi->sledgeLast = Rudi->sledgeFirst;
      entTemp[0]->sledgeFront = Rudi;
      entTemp[0]->sledgeBack = 0;

      //orientate like Rudi
      vec_set(entTemp[0]->pan, Rudi->pan);
   }
}
```

Damit wir diese Funktion und die Schlitten testen können, führen wir die Funktion aus, wenn wir die Space-Taste drücken. Wir können das eben schnell in Rudis Hauptschleife vor dem wait(1); einbauen:

```
if (key_space) {
   while (key_space) {wait(1);}
   sl_add();
}
```

Schlittenpositionierung

In der Hauptschleife wird der Schlitten in jedem Frame neu berechnen, wo er sich zu befinden hat, indem er den Lauf von Rudi und die Position des Vordermanns betrachtet. Dazu wird er jedes Mal die Suche initialisieren müssen, weil nicht klar ist, ob der Schlitten an Rudi oder an einem anderen angehängt ist. Ist die Initialisierungsfunktion erfolgreich - was bedeutet, dass der Schlitten richtig verknüpft ist - sucht er seine Position auf Rudis TRACK und bewegt sich dann, indem er die finale Position berechnet. Weil einige Daten auch im nächsten Frame eine Rolle spielen, müssen wir diese auch loggen. Die sl_main sieht dann so aus:

```
void sl_main ()
{
   sl_init(); //intialization

   while (my.sledgeDeath == 0) {

      // Initialize the tracking calculation and do
      // the processing if no error occurs
```

```
         if (sl_tracking_init()) {
            sl_tracking_search();   //search position on track
            sl_move();              //move sledge
         }

      sl_log(); //logging

      wait(1);
   }

   //todo: process death types
}
```

Die Suche nach dem richtigen Ort des Schlittens auf dem TRACK ist kein wirklich komplizierter Algorithmus, aber dennoch erklärenswert. Man startet an einem bestimmten Kontrollpunkt auf dem Track, der vom Vordermann belegt ist, geht dann solange die Kontrollpunkte weiter, bis der gegangene Weg den Mindestabstand zum Vordermann überschreitet. Das bedeutet, dass irgendwo zwischen dem Punkt und dem vorherigen Punkt der Ort des Schlittens sein muss. Dazu müssen wir auch die Entfernung festhalten, die wir bis einen Schritt vorher gegangen sind, um diesen Punkt durch Vektor-Interpolation zu bestimmen. Der Punkt, den wir dann erhalten, ist der approximierte Ort des Schlittens auf dem Pfad - denn der Pfad ist selber nur eine ungenaue Abbildung des Weges, den Rudi gegangen ist. Mit diesem Ort als Anhaltspunkt wird nachher der genaue Ort bestimmt.

Für diese Berechnung muss man einige Variablen definieren, die in jedem Schleifendurchlauf gebraucht werden. Da mehrere Funktionen innerhalb des Schlittens darauf zugreifen und wir noch viele freie Skills haben, werden wir diese benutzen, um die Parameter zu speichern. Wir definieren die folgenden Skills in der sledge.h:

```
#define sledgeCtrlPoint          skill11 //cast to: TRACKPOINT*
#define sledgeTargetPos          skill12 //13,14
#define sledgeParentPos          skill15 //16,17
#define sledgeDist               skill18
#define sledgePoint              skill19 //cast to: TRACKPOINT*
#define sledgeTrackLength        skill20
#define sledgeTrackLengthPrev    skill21
```

Die Skills beschreiben nacheinander

- den aktuellen Kontrollpunkt, den wir belegen (my.sledgeCtrlPoint, muss auf TRACKPOINT* gecasted werden),
- den Näherungspunkt auf dem Track,
- die Position des Vordermanns, an dem wir uns ausrichten,
- die Mindestdistanz zum Vordermann,
- den aktuellen verarbeiteteten TRACKPOINT,
- die aktuelle Entfernung, die wir beim Suchvorgang bereits auf dem Track zurückgelegt haben und die der vorigen Iteration.

Wir mussten diese Skills bereits einführen und den Algorithmus erklären, damit wir die Initialisierungsfunktion sl_tracking_init verstehen:

```
int sl_tracking_init ()
{
   //Get the parent and if it exists, start intialization.

   you = (ENTITY*)(my.sledgeFront);
   if (you) {

         if (you == Rudi) { //We are the first sledge

            vec_set(my.sledgeParentPos, vector(Rudi->x, Rudi->y, Rudi->z + Rudi->min_z));
            my.sledgeDist = sl_dist_toRudi;
            my.sledgePoint = (void*)(trackRudi->first);
            my.sledgeTrackLength = 0;

         } else {

            //We are somewhere in the chain.
            //Our parent is another sledge
```

```
            vec_set(my.sledgeParentPos, you.x);
            my.sledgeDist = sl_dist_toSledge;
            my.sledgePoint = you.sledgeCtrlPoint;

            //parent sledge is somewhere in front of us an not AT the point
            my.sledgeTrackLength = vec_dist(((TRACKPOINT*)my.sledgePoint)->pos, you.x);
         }

      my.sledgeTrackLengthPrev = my.sledgeTrackLength;

      return(1); //everything is fine
   } else {
      return(0); //error
   }
}
```

Es ist nämlich wichtig zu wissen, wo der Schlitten sich befindet. Wenn der Schlitten beispielsweise an Rudi befestigt ist, müssen wir einen anderen Abstand einhalten als zu einem Vorderschlitten. Außerdem starten wir bei der Suche bei dem ersten Punkt von Rudis Track. Im Gegensatz dazu fangen "normale" Schlitten bei dem Kontrollpunkt an, der von ihrem Vordermann eingenommen wird und starten bereits mit ein wenig Vorlauf, nämlich der Distanz des Vordermanns zum Kontrollpunkt). Es wurden 2 Variablen aus der player.h benutzt, die den Mindestabstand zu Rudi oder einem Schlitten festlegen:

```
var sl_dist_toRudi = 180;
var sl_dist_toSledge = 150;
```

Wurden diese Einstellungen getätigt, können wir den Algorithmus einprogrammieren, den wir oben beschrieben haben. Der Algorithmus basiert auf den Skills, sodass es ihm egal ist, wo wir festgebunden sind.

```
void sl_tracking_search ()
{
   // As long as we are too close to our parent,
   // move through the tracking chain

   while (my.sledgeTrackLength < my.sledgeDist) {

      //Log the distance from the previous iteration
      my.sledgeTrackLengthPrev = my.sledgeTrackLength;

      //We moved one step away, so we add the moved distance to the total moved distance
      my.sledgeTrackLength += vec_dist(((TRACKPOINT*)my.sledgePoint)->pos,my.sledgeParentPos);

      //Log current point so that my child can start tracking right here.
      //If NO child is available, we reached the end of the chain (exit)

      if (((TRACKPOINT*)my.sledgePoint)->child != 0) {
         my.sledgePoint = (void*)(((TRACKPOINT*)my.sledgePoint)->child);
      } else {
         break;
      }
   }

   //The current point is too far away and the parent point to narrow. We calculate
   //the right point between them now:

   //We only do this if we aren't the first point and if the difference of the
   //went track isn't to small (prevents division through zero later)

   if ((((TRACKPOINT*)my.sledgePoint)->parent != 0) &&            //not the first point
      ((my.sledgeTrackLength - my.sledgeTrackLengthPrev) > 5))    //prevents division error
   {
      //We interpolate between both points. The result is the right position which fits the
      //predestined minimum distance. The factor is calculated by the ratio between the
      //tracked distance from the parent and current point and the searched distance.

      vec_lerp(my.sledgeTargetPos,  ((TRACKPOINT*)my.sledgePoint)->parent->pos,
                                    ((TRACKPOINT*)my.sledgePoint)->pos,
                                    ((my.sledgeDist - my.sledgeTrackLengthPrev) /
                                    (my.sledgeTrackLength - my.sledgeTrackLengthPrev)));

   } else {
```

```
      //in the case that we are the first point, we directly set the position
      vec_set(my.sledgeTargetPos, ((TRACKPOINT*)my.sledgePoint)->pos);
   }
}
```

Die vec_lerp Anwiesung in der letzten Hälfte könnte verwirren: es wird einfach über eine Verhältnisgleichung auf den Interpolationsfaktor geschlossen. Nach dieser Funktion steht in my.sledgeTargetPos der angenäherte Ort des Schlittens. Die Funktion sl_move soll den Schlitten dann auf seine endgültige Position bringen und ihn an seinem Vordermann ausrichten:

```
void sl_move ()
{
   // Move to the constraint position
      vec_diff(vecTemp.x, my.sledgeTargetPos, my.sledgeParentPos);
      vec_normalize(vecTemp.x, my.sledgeDist);
      vec_add(vecTemp.x, my.sledgeParentPos);

      vec_lerp(my.x, my.x, vecTemp.x, 0.9 * time_step);

   // Look to my parent
      vec_diff(vecTemp.x, my.sledgeParentPos, my.x);
      vec_to_angle(vecTemp.x, vecTemp.x);
      ang_lerp(my.pan, my.pan, vecTemp.x, 0.5 * time_step);
      my.roll = 0;
}
```

Wir müssen in jedem Frame den Mindestabstand einhalten und wir wissen, wo in etwa der Schlitten ist. Dann stellen wir den Schlitten dort hin und bewegen ihn in Richtung des Vordermanns, sodass der Mindestabstand eingehalten wird. Das Bewegen hin zum Vordermann wird durch eine einfache Methode geleistet: wir nehmen den Richtungsvektor des Vordermanns zum angenäherten Punkt auf dem Track und bringen den Vektor auf die Länge des Mindestabstandes. Wenn wir jetzt noch die Position des Vordermanns addieren, haben wir die exakte Position des Schlittens berechnet. Die Interpolation wird von vec_lerp geleistet. Das Ausrichten an dem Vordermann ist auch relativ einfach geleistet, wie wir es in etwa schon bei der Kamera kennengelernt haben.

Die Logfunktion ist auch recht einfach. Sie loggt einfach den nachfolgenden Punkt des gerade benutzen TRACKPOINTs des Schlittens. An diesem Punkt startet dann der Nachfolger seine Suche:

```
void sl_log ()
{
   // Log control point
      my.sledgeCtrlPoint = (void*)(((TRACKPOINT*)my.sledgePoint)->child);
}
```

Wenn wir jetzt öfter die Space-Taste drücken, wird ein weiterer Schlitten an Rudi drangehängt, und sie werden ihm folgen.

Schlittenspawning

Wenn wir also oft Space drücken, erstellen wir schnell ganz viele Schlitten – und zwar gleichzeitig! Wir wollen die Schlitten zwar trotzdem schnell hintereinander erzeugen, aber eben nicht gleichzeitig. Wir wollen uns eine Funktion schreiben, die uns eine gewissen Anzahl an Schlitten auf diese Weise erzeugt. Die neue Funktion nennen wir sl_addNr und geben ihr einen Parameter, der angibt, wieviele Schlitten erzeugt werden sollen. Der Code sieht so aus:

```
//-----------------------------------------------------------------------
// SPAWNS SLEDGES BUT AFTER ANOTHER AT THE END OF THE SLEDGE CHAIN
//-----------------------------------------------------------------------
VECTOR sl_addNr_spawnPos; //static vector
void sl_addNr (int nr)
{
   // if there is no Rudi, there won't be sledges
   if (Rudi == NULL) {return;}

   int i;
```

```
    //spawn sledges after another..
    for (i = 0; i < nr; i++) {
       sl_add(); //spawn sledge
       wait(-0.2);
    }
}
```

Zunächst brechen wir ab, wenn es keinen Rudi gibt. Danach starten wir eine For-Schleife, die so oft durchlaufen wird, wie wir Schlitten spawnen wollen. In der wait Anweisung können wir die Geschwindigkeit des Spawnens bestimmen. Ich habe einen sehr kurze Spawn-Pause pro Schlitten eingestellt – Sie können natürlich eine längere Zeit einstellen.

Wir ändern in Rudis Testaufruf von sl_add() die Funktion in sl_addNr(5); und erzeugen damit immer 5 Schlitten bei einem Druck auf Space. Mittlerweile brauchen wir auch das Debugging des TRACKs nicht mehr, also entfernen auch gleich die Anweisung "debug_TRACK(trackRudi);" aus der Spielerfunktion.

Kapitel 6: Die Weihnachtsgeschenke

Am Anfang haben wir gesagt, das Spielprinzip von Rudi gleiche dem von Snake. Wie im Vorbild wird Rudi Futter in Form von Geschenkpaketen einsammeln und dadurch wird sein Schwanz in Form einer Schlittenkette immer länger. Im letzten Kapitel kamen wir auf die Idee, dass es recht lange dauern würde, bis Rudis Schlange eine entsprechende Größe erreicht hätte, wenn er immer nur einen Schlitten pro eingesammeltem Geschenkehaufen hinzu bekommen würde. Deshalb haben wir gesagt, dass er jedes Mal ein Paket mehr hinzu bekommt, also erst eins, dann zwei, dann drei, und so weiter.

Wir haben uns aber noch keine Gedanken gemacht, wie wir das Spawning der Pakete realisieren wollen. Das ist auch gar nicht so einfach, weil im Gegensatz zu den klassischen Snakespielen der Spieler hier nicht die gesamte Karte auf einmal sieht und daher eventuell auch nicht genau weiß, wo der nächste Futterpunkt, pardon, Geschenkehaufen liegt. Wenn wir also immer nur einen einzigen Paketehaufen auf der Karte haben, dann könnten wir durch einen Hinweis den Spieler zu diesem Paket leiten. Dafür eignet sich ein Pfeil, der immer in die Richtung des Pakets zeigt, sodass der Spieler in etwa weiß, wo er hin muss.

Das Paket wird also gespawned und wenn Rudi darüber läuft, werden Schlitten hinzugefügt. Damit wir wissen, wie viele Schlitten wir anhängen, müssen wir irgendwo speichern, wie viele Pakete Rudi bereits eingesammelt hat. Am besten speichern wir das in einem Skill von Rudi selbst. Wenn das Paket also eingesammelt worden ist, muss ein neues auftauchen – es stellt sich dann die Frage, wo genau das sein soll. Es dürfen nur diejenigen Pakethaufen auftauchen, die noch nicht eingesammelt worden sind.

Das Spiel sollte Spaß machen, überlegen Sie sich daher gut, wo die Pakete auftauchen sollen. Wenn der Spawnpunkt in der Nähe erzeugt wird, ist das erfreulicher für den Spieler. Ihnen ist es selbst überlassen, später einen härteren Spielmodus einzubauen, der die Pakete an unmöglichen Orten erzeugt und es dem Spieler nicht einfach macht. Wir wollen das Modell „packets.mdl“ für die Pakete benutzen und im Level verteilen.

Das Grundgerüst der Pakete

Die Pakete wollen wir in einer eigenen Datei behandeln. Wir erzeugen dazu 2 Dateien im "game" Ordner, nennen sie packets.c bzw. packets.h und inkludieren sie.

Wenn wir bestimmen wollen, welches Paket als nächstes an der Reihe ist, müssen wir auch wissen, wie viele Pakete im Moment überhaupt vorhanden sind und wieviele es zu Beginn des Levels überhaupt gab. Das ist auch noch für andere spielsteuernden Mechanismen interessant, wie für das Tor, das sich öffnen soll, wenn wir alle Geschenke eingesammelt haben. Dazu erzeugen wir zwei Variablen in der packets.h, die dies festhalten:

```
//---------------------------------------------------------------------
// variables
//---------------------------------------------------------------------

   int pk_count;                //currently available packets
   int pk_count_max;            //maximal amount of available packets
```

Unser Präfix für die Pakete lautet "pk_". Wir haben die Hauptfunktion beim kompletten Namen genannt, weil das später die Auswahl im WED erleichtert.

Damit sich das Paket im Spiel registrieren und initialisieren kann, erstellen wir eine Hauptfunktion und fügen eine Initialisierungsfunktion hinzu. Die Initialisierung schaltet das Paket auf passable und unsichtbar, da es erst aktiviert werden muss, darauf wartet das Paket in einer seperaten While-Schleife, bevor es seinen Code ausführen kann. Das sieht dann so aus:

```
//---------------------------------------------------------------------
// PACKET FUNCTION
//---------------------------------------------------------------------
void packet ()
{
   pk_init();

   //waiting for activation
   while (is(my, INVISIBLE)) {wait(1);}

   //main loop
   while (1) {
      my.pan += 5 * time_step;
      wait(1);
   }
}

   //---------------------------------------------------------------------
   // PACKET INITIALIZATION
   //---------------------------------------------------------------------
   void pk_init ()
   {
      //make me passable
         set(my, PASSABLE);

      //increase packet count
         pk_count++;
         pk_count_max++;

      //not visible until we are switched on
         set(my, INVISIBLE);
   }
```

Zum Testen öffnen wir nun den Testlevel im WED, ordnen um Rudi herum 5 Pakete an und weisen ihnen die action zu. Nach dem Kompilieren mit "update entities" setzen wir in Rudis Funktion provisorisch die Zeile

```
deb_print(pk_count);
```

ein und lassen somit die Anzahl der Pakete ausgeben. Zunächst sind die Pakete sind unsichtbar und uns wird eine 5 ausgegeben. Wir können den Debugeintrag nun wieder löschen.

Wenn wir später ein neues Level laden würden mit nochmals 5 Paketen, würden aber 10 Pakete dabei herauskommen, da zwar die Variable beim Start der Engine mit 0 initialisiert wird, wenn wir aber das Level wechseln, müssen wir diese Variable zurücksetzen. Wir werden später einige Dinge für die Levelverwaltung programmieren müssen, so etwas wie Variablen initialisieren und zurücksetzen gehört nun einmal zwangsläufig dazu. Aus diesem Grund eröffnen wir im "game" -Ordner zwei Dateien namens "levels.c" und "level.h", inkludieren sie und schreiben in die levels.c Datei eine Resetfunktion:

```
void lvl_reset ()
{
   //package count
      pk_count = 0;
      pk_count_max = 0;
}
```

Diese muss natürlich auch aufgerufen werden. In der game.c haben wir unseren provisorischen level_load -Befehl stehen. Wir fügen den Aufruf danach ein:

```
void game ()
{
   game_init();

   level_load("testlevel.wmb");
   lvl_reset();
}
```

Das Spawning der Pakete

Wir wollen es so machen: Wenn ein Paket eingesammelt worden ist, nimmt man aus der Menge aller übrigen Pakete zufällig eines heraus. Damit wir Zugriff auf die Pakete haben, nutzen wir ein Entity-Array und eine Zahl, die uns sagt, wieviel Pakete maximal vorhanden sein dürfen:

```
#define pk_max 30          //maximal amount of packets
ENTITY* pk_ents [pk_max];  //pointers to the packets
```

Damit sich das Paket in die Liste schreibt, ziehen wir Nutzen aus der Zahl der bisher registrierten Pakete und schreiben den My-Zeiger in das Array, damit wir später auf das Paket wieder Zugriff haben.

```
void pk_init ()
{
   //make me passable
      set(my, PASSABLE);

   //increase packet count
      pk_count++;
      pk_count_max++;

   //register packet
      pk_ents[pk_count - 1] = my;

   //not visible until we are switched on
      set(my, INVISIBLE);
}
```

Wenn das Paket sich später löschen möchte, muss es wissen, in welches Feld des Arrays es sich geschrieben hat. Wir speichern also die ID des Pakets. Dazu legen wir ein neues Skilldefine an:

```
#define ID skill9 //ID number
```

Wir ändern dann den entsprechenden Code in der pk_init:

```
//save ID
   my.ID = pk_count - 1;

//register packet
   pk_ents[my.ID] = my;
```

Damit können wir auch den Fall behandeln, dass ein Paket aus seiner Schleife aussteigt, aus der Registrierung entfernt und dann löscht. Dazu fügen wir einfach nach der While-Schleife in der packet -Funktion folgendes ein:

```
//de-register
   pk_ents[my.ID] = 0;
   pk_count--;

//remove entity
   ent_remove(my);
```

Die Entity nullt den Zeiger im Registrierungsarray um anzuzeigen, dass sie nicht mehr verfügbar ist, und entfernt sich dann selbst. Damit wir ein neues Paket spawnen lassen, müssen wir uns erst eine Funktion dafür schreiben, die das tut - wir nennen sie pk_spawn. Die Funktion wird zwei andere Funktionen aufrufen, pk_select und pk_activate, wobei pk_select ein Paket aussucht und den Entity-Zeiger darauf zurückliefert. Die Funktion pk_activate erhält diesen und aktiviert das Paket.

Die Funktion pk_spawn ist relativ einfach gehalten:

```
void pk_spawn ()
{
   you = pk_select();

   if (you) {

      pk_activate(you);

   } else {

      //no packets left

      error("no more packets left");
   }
}
```

Wenn eine Entity gefunden wurde, aktivieren wir sie. Die Funktion pk_spawn sieht so aus:

```
ENTITY* pk_select ()
{
   ENTITY *current;
   int randomNr = (int)(random(pk_count)); //take the n'th entity
   int i;

   //get through the list
   for (i = 0; i < pk_count_max; i++) {
      current = pk_ents[i];      //get entity
      if (current) {             //if entity is valid

         if (randomNr <= 0) {    //break if this is the n'th entity
            break;
         } else {
            randomNr--;          //we still have to search
         }
      }
   }

   // This is the selected entity.
   // If no one has been found, its 0!

      return(current);
}
```

Mit dem Entity-Zeiger current holen wir den Zeiger auf eine Entity in der Liste. Wir wissen durch den Wert in der Variable pk_count immer, wieviele Pakete noch zur Verfügung stehen. Wir können also eine Zufallszahl erzeugen, die angibt, welches Paket wir von den vorhandenen Paketen nehmen. Wir gehen danach das Entity -Array durch und suchen das vorher ausgewählte, n-te Paket, das einen validen Entityzeiger besitzt. Ist er = 0, wissen wir, dass das Paket nicht mehr vorhanden ist. Wenn wir dann ein Paket gefunden haben, wird der Zeiger darauf zurückgegeben.

In der Funktion pl_activate machen wir erstmal nichts Anderes, als das Paket sichtbar zu schalten:

```
void pk_activate (ENTITY* ent)
{
   reset(ent, INVISIBLE);
}
```

Damit im Level auch überhaupt das erste Mal ein Paket auftaucht, müssen wir zu Beginn des Levels ein pk_spawn(); ausführen. Um zu überprüfen, ob das notwendig ist, führen wir einen Zeiger ein, der auf das aktuelle Geschenk zeigt. Wenn dieser 0 ist, wird pk_spawn aufgerufen und der Zeiger wird gefüllt. Dazu muss pk_spawn aber den Entityzeiger zurückliefern.

Den Zeiger richten wir in der packets.h ein...

```
ENTITY* pk_current = 0;
```

...und setzen ihn beim Levelreset zurück:

```
void lvl_reset ()
{
   //package count
      pk_count = 0;
      pk_count_max = 0;

   //package pointer
      pk_current = 0;
}
```

Wir bauen pk_spawn um, damit der Entityzeiger zurückgeliefert werden kann:

```
ENTITY* pk_spawn ()
{
   ENTITY* newPacket = pk_select();

   if (newPacket) {
      pk_activate(newPacket);

   } else {

      //no packets left

      error("no more packets left");
   }

   return(newPacket);
}
```

Damit sich die Pakete beim Levelstart selbst initialisieren, fügen wir in der pk_init ganz am Ende folgenden Code hinzu:

```
wait(1); //let all other packets register themselves

//create a new packet, if there is no one
   if (pk_current == 0) {
      pk_current = pk_spawn();
   }
```

Wir warten einen Frame, damit sich alle Pakete registriert haben. Wenn dann der Zeiger auf das aktuelle Paket immer noch nicht gefüllt ist (also kein Paket aktiv ist), spawnen wir es. Wenn wir das Spiel jetzt starten, sehen wir genau einen Pakethaufen, der sich dreht und damit aktiviert ist - super!

Damit wir auch testen können, ob auch alle Pakete durchgeschaltet werden, fügen wir den folgenden Code in der While Schleife von packet hinzu:

```
//debug: deactivate packet with return key
if (key_enter) {
```

```
   while (key_enter) {wait(1);}
   break;
}
```

Wenn wir Enter drücken, wird das Paket künstlich deaktiviert und bricht seine main-loop ab. Damit das nächste Paket gespawned wird, fügen wir

```
//spawn new packet
   pk_current = pk_spawn();
```

nach dem de-registrieren und **vor** ent_remove ein. Wenn wir nun das Spiel starten und 5 Mal (für fünf Pakete) die Enter-Taste drücken, wird das Paket jedes Mal woanders gespawned - Super! Genau das wollten wir!

Pakete triggern

Bevor wir fortfahren, verteilen wir die Pakete etwas im Level, damit die nicht alle an einem Ort liegen. Danach entfernen wir den Debug-Code aus der Funktion packet und ersetzen den Eintrag durch den Aufruf einer Funktion pk_check und der Auswertung davon. pk_check soll überprüfen, ob Rudi das Paket berührt oder nicht und liefert eine 1 zurück, wenn Rudi das Paket berührt; ansonsten eine 0:

```
if (pk_check()) {
   break;
}
```

Wenn also die Funktion pk_check eine 1 zurückliefert, berührt Rudi das Paket. Es wird durch ein Break-Statement seine Hauptschleife verlassen und dadurch veranlassen, dass ein neues Paket erzeugt wird.

Die Funktion pk_check prüft zunächst, ob Rudi existiert, sonst kann er ja nicht die Entfernung berechnen, danach wird geschaut, ob sich Rudi zum Paket in einer gewissen Reichweite namens pk_collectRange befindet. Wenn diese unterschritten wird, berührt Rudi das Paket. Die Variable pk_collectRange ist im Header mit 100 quants initialisiert.

```
int pk_check ()
{
   if (Rudi) {
      return(vec_dist(Rudi->x, my.x) <= pk_collectRange);
   } else {
      return(0);
   }
}
```

Wenn wir nun jedes einzelne Paket anfahren, wird es getriggert. Daraufhin wird es entfernt und ein neues aktiviert. Wir können also den bisherigen Hilfscode (den mit der Entertaste) entfernen.

Schlitten hinzufügen

Wenn ein Paket sich de-registriert hat, können wir leicht über die Differenz aller anfangs registrierten Pakete und den aktuell noch zur Verfügung stehenden Pakete auf die Anzahl der gesammelten Pakete schließen - die Formel lautet: pk_count_max - pk_count. Wenn sich also ein Paket entfernt und ein neues spawned, dann können wir dort die Schlitten anhängen lassen.

Dazu entfernen wir den Debugcode in der Schleife in pl_rudi, der über die Spacetaste die Schlitten erzeugt. Danach fügen wir dann einfach die sl_addNr Funktion ein:

```
//(...)

//de-register
   pk_ents[my.ID] = 0;
   pk_count--;
```

```
   //add sledges
      sl_addNr(pk_count_max - pk_count);

   //spawn new packet
      pk_current = pk_spawn();

   //remove entity
      ent_remove(my);
```

Wenn wir das ausprobieren, verschwindet zwar das Paket, aber es werden keine Schlitten hinzugefügt...

Wir haben es dabei mit einem Timingproblem zu tun. Wir rufen die Funktion sl_addNr auf, die aber nicht sofort alle Schlitten erzeugt, sondern unter Umständen mehrere Frames lang aktiv ist. Da die aufrufende Entity, hier unser Paket, danach mit ent_remove entfernt wird und sich dabei alle von der my -Entity aufgerufenen Funktionen, das heißt auch die noch laufenden, terminieren, wird auch sl_addNr vorzeitig gestoppt. Das war so nicht geplant.

Wir können das Problem aber umgehen. Wenn eine Funktion nicht terminiert werden soll, wird innerhalb dieser Funkion der my -Zeiger auf 0 gesetzt. Das gilt auch rückwirkend für die aufrufende Entityfunktion, wenn wir nämlich dort eine ent_remove(my); -Anweisung ausführen, kommt es zu einem Fehler, weil wir bereits den my-Zeiger mit 0 überschrieben haben. Die Entity befindet sich dann noch im Speicher und wir können sie zwar sehen, haben aber jegliche Referenz auf sie verloren. Daher müssen wir den Zeiger kurzfristig zwischenspeichern. Wir nullen den my -Zeiger in der Funktion sl_addNr:

```
void sl_addNr (int nr)
{
   //keep on running if called from an entity
      my = 0;

   //(...)
```

In der Funktion packet speichern wir dann den my Pointer, bevor wir sl_addNr aufrufen und wenden den Hilfszeiger in ent_remove an:

```
   //(...)

   //save my pointer
      ENTITY* mySave = my;

   //de-register
      pk_ents[my.ID] = 0;
      pk_count--;

   //add sledges
      sl_addNr(pk_count_max - pk_count);

   //spawn new packet
      pk_current = pk_spawn();

   //remove entity
      ent_remove(mySave);
}
```

Wenn wir das Spiel spielen, wird das Paket entfernt und die Schlitten korrekt gespawnedt! Super!

Interludium: Schönheitskorrekturen

Es gibt noch einige Dinge, die noch eleganter gelöst werden können. Zunächst einmal schauen wir uns die Funktion packet an. Alles, was nach der While-Schleife folgt, gehört zum Abschluss der Funktion dazu. Wir gliedern den Code in eine eigene Funktion namens pk_remove aus.

Eine weitere Auffälligkeit ist die Startrichtung der Schlitten beim Erzeugen. Anscheinend sind sie immer nach rechts gedreht. Wie wäre es, wenn sie bereits am Vordermann ausgerichtet wären, wenn sie erzeugt werden? In der Funktion sl_add orientieren wir die neuen Schlitten in etwa so wie den Vordermann:

```
if (Rudi->sledgeLast) {

   //(...)

   //orientate like parent
   vec_set(entTemp[0]->pan, ((ENTITY*)entTemp[0]->sledgeFront)->pan);

} else {

   //(...)

   //orientate like Rudi
   vec_set(entTemp[0]->pan, Rudi->pan);
}
```

Die Pfeilanzeige

Wir wollten wir einen Pfeil einrichten, der in die ungefähre Richtung des nächsten Pakets anzeigt. Der Pfeil ist ein 3D-Modell, welches immer sichtbar sein soll, daher bietet sich eine View Entity an, die außerdem auflösungsunabhängig ist. Warum das wichtig ist, behandeln wir später in einem separaten Abschnitt. Die Datei des Pfeils lautet "arrow.mdl".

Der Pfeil wird den Paketen zugerechnet. Daher definieren wir in der packets.h folgenden Zeiger:

```
ENTITY* pk_arrow = 0;
```

Die Viewentity erstellen wir manuell und schreiben keinen Initialisierungsblock, weil die Datei für den Pfeil im Levels-Ordner liegt und wir zum Zeitpunkt der Initialisierung den Pfad noch nicht zum System hinzugefügt haben. Man kann zwar über eine WDL-Datei Pfade angeben, bevor das Spiel selbst kompiliert wird, ich halte das aber für eine sehr unsaubere Lösung. Deshalb erzeugen wir die Entity dynamisch.

Wir werden in der packets.c zwei Funktionen schreiben, pk_arrow_main und pk_arrow_init, wobei die main die init aufruft. pk_arrow_main wird bei der Spiel-Initialisierung einmalig aufgerufen, weil die Viewentity des Pfeils das gesamte Spiel über existiert. Daher schreiben wir in die Funktion game_init in der game.c:

```
void game_init ()
{
   //add game content folders to engine file system
      add_folder("game\\levels"); //leveldata

   //start packet arrow function
      pk_arrow_main();
}
```

Die pk_arrow_main initialisiert sich und führt dann in jedem Frame eine Berechnung durch, wo der Pakethaufen liegt:

```
//---------------------------------------------------------------------
// PACKET ARROW
// Points to the next packet if it is outside of the screen
//---------------------------------------------------------------------
void pk_arrow_main ()
{
   //creates and initializes the arrow
      pk_arrow_init();

   //Main loop
   while (1) {

      //ROTATION

         //Doesn't work if Rudi and the packet aren't there!
         if ((Rudi != 0) && (pk_current != 0)) {

            // Angle from Rudi to packet
```

```
                vec_diff(vecTemp.x, pk_current->x, Rudi->x);
                vec_to_angle(vecTemp, vecTemp);

            // Rotation interpolation. Uses a special angle transformation
            // to project from entity rotation to view entity rotation.
                    ang_lerp(pk_arrow->pan, pk_arrow->pan, vector(vecTemp.x + 90, 180, 0), 0.5 *
time_step);
            }

         wait(1);
      }
   }

      //---------------------------------------------------------------
      // CREATES & INITIALIZES THE ARROW
      //---------------------------------------------------------------
      void pk_arrow_init ()
      {
         //create arrow as view entity
            pk_arrow = ent_createlayer("arrow.mdl", 0, 1);

         //pose and orientate it
            vec_set(pk_arrow->x, vector(700, 0, -350));
            pk_arrow->tilt = 180; //see rotation code; init rotation
      }
```

In der Initialisierung wird die View Entity auf dem Bildschirm erzeugt und unten mittig positioniert. Verwirrend ist vielleicht die Tatsache, dass der Pfeil im tilt umgedreht wird. Dazu müssen wir uns den Code anschauen, der die Rotation berechnet. Zunächst wird nur etwas berechnet, wenn sowohl Rudi und das Paket vorhanden sind. Sonst führt die Berechnung zu Fehlern aufgrund von leeren oder falschen Zeigern. Als Anhaltspunkt wird der Winkel von Rudi zum Paket genommen. Wenn wir jetzt den Winkel in den Pfeil übertragen, dann zeigt er immer in eine völlig andere(nicht klar) Richtung. Durch Herausprobieren oder auch logisches Denken kriegt man heraus, dass wir um 90 Grad gedreht auf die Map schauen und dann, wenn wir den Pfeil ausrichten, dieser horizontal gespiegelt auf das Paket zeigt (deshalb der umgedrehte Tilt). Wir initialisieren den Tilt deshalb mit 180, weil wir sonst eine unwillkürliche Rotation erhalten, wenn sofort im ersten Frame ein Paket sichtbar ist und der Pfeil noch eine falsche Orientierung hat.

Der Pfeil ist nun immer sichtbar. Allerdings wollen wir nur einen Pfeil sehen, wenn kein Paket in unserem Blickfeld liegt und ihn ausblenden, wenn wir ein Paket sehen oder kein Paket mehr vorhanden ist. Dazu skalieren wir den Pfeil, sodass ein Popup-Effekt erzielt wird und die Aufmerksamkeit des Spielers auf sich zieht. Wir fügen also nach der Rotation einen Popup-Block ein:

```
   //POPUP

      //If packet is available and outside the screen -> popup
      if (pk_current != 0) {
         vec_set(vecTemp.x, pk_current->x);
         if (vec_to_screen(vecTemp.x, camera) == NULL) {
            vec_lerp(pk_arrow->scale_x, pk_arrow->scale_x, vector(1,1,1), 0.5 * time_step);
         } else {
            vec_lerp(pk_arrow->scale_x, pk_arrow->scale_x,
                     vector(0.05, 0.05, 0.05), 0.9 * time_step);
         }
      } else {
         vec_lerp(pk_arrow->scale_x, pk_arrow->scale_x,
                  vector(0.05, 0.05, 0.05), 0.9 * time_step);
      }
```

Ist das Paket sichtbar, wird der Pfeil blitzschnell klein skaliert, das Einblenden erfolgt langsamer. Wenn der Pfeil ganz klein ist, sieht man ihn noch, also fügen wir noch folgenden Block hinzu:

```
   //VISIBILITY

      //if the arrow is too small, clip it
      if (pk_arrow->scale_x <= 0.075) {
         set(pk_arrow, INVISIBLE);
      } else {
         reset(pk_arrow, INVISIBLE);
      }
```

Wenn wir jetzt mit Rudi durch das Testlevel fahren, poppt immer dann der Pfeil auf, wenn wir gerade kein Geschenk sehen. Als letzten Akt lagern wir all diese Variablen aus:

```
//Arrow

   ENTITY*  pk_arrow = 0;

   var      pk_arr_rotBlend = 0.5;

   var      pk_arr_popup_speedIn = 0.5;
   VECTOR*  pk_arr_popup_scaleIn = {x = 1; y = 1; z = 1;}

   var      pk_arr_popup_speedOut = 0.9;
   VECTOR*  pk_arr_popup_scaleOut = {x = 0.05; y = 0.05; z = 0.05;}

   var      pk_arr_popup_clip = 0.075;
```

Kapitel 7: Crashing, zerstörbare Schlitten und die ersten Effekte

Entityevents

Man kann für Entities sogenannte Events in Form von Funktionszeigern setzen. Wenn ein event ausgelöst wird, wird diese Eventfunktion automatisch aufgerufen. Solche Events sind vielfältiger Art, es kann eine Kollision mit einem Block oder einer Entity sein, man wird von einer andere Entity überrannt und dergleichen. Dies ist besonders dann nützlich, wenn man bestimmte Reaktionen auf bestimmte Kollisionssituationen behandeln will. Genau dies ist auch der Fall, wenn wir mit Rudi gegen eine Wand laufen, er soll in diesem Fall crashen, umfallen und alle Schlitten sollen kaputt gehen.

Dafür werden wir die Events EVENT_BLOCK und EVENT_ENTITY abfangen, die genau dann ausgelöst werden, wenn wir uns mit c_move gegen einen Block oder eine Entity bewegen. In diesen Fällen erhalten wir auch einen Normalen-Vektor, der uns die Richtung der aufgetroffenen Fläche wiedergibt. Anhand der vertikalen z-Komponente kann man dann herausfinden, ob man gerade gegen eine Wand gelaufen ist oder ob man nur einen Abhang hinauf- oder herabläuft. Da wir ständig mit dem c_move Befehl eine Gravitation nach unten ausüben, wird zwangsläufig in jedem Frame einer dieser Events aufgerufen.

In der Eventfunktion werden wir die Normale analysieren und schauen, ob wir gegen eine Wand laufen oder ob es doch nur eine Schräge war. Dementsprechend setzen wir einen Skill, der es uns erlaubt, die Schleife in der Hauptfunktion zu verlassen und je nach Wert darauf zu reagieren, in erster Linie mit dem Abspielen der Crash-Animation und dem Zerstören der Schlitten.

Einrichten des Eventsystems

Zunächst müssen wir Rudi für diese Events sensibilisieren. Wir müssen der Engine also mitteilen, dass bei bestimmten Events die zugewiesene Eventfunktion von Rudi aufgerufen wird. Im Standardzustand ist eine Entity für keinen Event-Typ sensibilisiert, sodass wir alles selbst aufsetzen müssen. Wir tun dies in der Funktion pl_rudi_init:

```
//set events for crashing
   my.emask |= (ENABLE_BLOCK | ENABLE_ENTITY);
   my.event = pl_event;
```

Die Variable my.event erhält immer nur einen Zeiger auf eine Funktion, deshalb darf man keine Parameter benutzen, Ausnahmen wären Funktionen, die einen Zeiger auf eine Eventfunktion wiedergeben, aber in der Regel setzt man direkt den Funktionszeiger. Wir müssen auch eine Eventfunktion aufsetzen namens pl_event:

```
void pl_event ()
{
   //CRASH: running against entities or blocks
   if ((event_type == EVENT_BLOCK) || (event_type == EVENT_ENTITY)) {

      //crash against orthogonal walls (with a bit threshold)
      if (normal.z <= 0.2) {
         my.playerDeath = 1; //standard death: crashing
      }
   }
}
```

Zunächst wird die Crash-Überprüfung nur für Block- und Entityevents durchgeführt. Weil wir keinen anderen Event definiert haben, könnte man das auch weglassen, aber man benötigt die Verzweigung dann, wenn man selbst noch andere Events definiert. Danach überprüfen wir die z-Komponente der Normalen. Wenn dieser Wert > 0 ist, stößt Rudi gegen eine horizontale Oberfläche, wie z.B. den Boden. Wenn der Wert = 0 ist, läuft er gegen eine rechtwinklige Wand (90° Winkel). Ist der Wert < 0, stößt er gegen die Decke, was im Spiel nicht der Fall sein wird. Nun, in diesem Fall habe ich auf kleiner gleich 0.2 überprüft, damit er auch gegen stark ansteigende Flächen stößt, ansonsten würde er diese auch hochlaufen.

Wenn also all dies zutrifft und Rudi gegen eine Wand rennt, wird hier ein Death-Skill auf 1 gesetzt, der wie folgt in der player.h definiert ist:

```
#define playerDeath  skill11
```

Ähnlich wie bei den Schlitten, wollen wir hier mehrere "Todes"-Arten abfangen. Wenn wir in der Hauptschleife in der Bedingung Folgendes schreiben:

```
while (my.playerDeath == 0)
{
   //(...)
}
```

...dann wird die Hauptschleife beendet. Wir können den Skillwert auswerten, um je nach Situation Rudi sterben zu lassen. Wir vereinbaren, dass der Wert 1 für den Standard-Crash steht. Alle anderen Werte behandeln andere Situationen.

Rudis Tod

Der Event wird im Falle eines Crashes die Hauptschleife beenden. Wir müssen desweiteren den Skillwert abfangen und auswerten. Dazu fügen wir nach der Hauptschleife den Aufruf der Funktion pl_death(); ein und schreiben eine Dummyfunktion:

```
void pl_death ()
{
   beep();
}
```

Wir probieren erstmal das ganze Zeug aus und galoppieren gegen eine Wand: es ertönt ein kurzer Beep-Sound. Alles hat geklappt! In der Funktion pl_death werden wir nun über einen switch-case-Baum den Skillwert analysieren und die Ausführung an eine andere Funktion delegieren, die sich um Rudi's Tod kümmert:

```
void pl_death ()
{
   switch (my.playerDeath) {
      case 1: pl_death_crash(); return;   //crashing (default)
      //add other death types here
   }
}
```

Bei einem normalen Crash wird dann die Funktion pl_death_crash ausgeführt. Diese Funktion muss dann die Crash-Animation von Rudi einmal abspielen:

```
void pl_death_crash ()
```

```
{
   var i;

   //play crash animation
   for (i = 0; i < 100; i += 7 * time_step) {
      ent_animate(my, "crash", i, 0); //crash
      wait(1);
   }
}
```

Wenn wir jetzt mit Rudi gegen eine Wand laufen, knallen wir dagegen und Rudi macht dann einen Flip nach hinten und bleibt auf dem Rücken liegen - super! Nun gibt es aber ein Problem: wenn wir mit Rudi gegen eine Wand fahren und dabei in diesem Moment kein Geschenk auf dem Bildschirm sehen, dann sieht man aber den Pfeil noch, was so nicht richtig ist (schließlich können wir das Paket nicht mehr einsammeln). Dazu müssen wir einfach in der Popup Sektion vom Pakete-Pfeil auf Rudi's Status und speziell seinem Death-Skill reagieren:

```
//POPUP

   if (Rudi) {

      //If packet is available and outside the screen -> popup
      if ((pk_current != 0) && (Rudi->playerDeath == 0)) {
         //(...)
      } else {
         //(...)
      }
   }
```

Abtrennen der Schlitten

Damit jeder Schlitten feststellen kann, ob Rudi in ihn hineinläuft, überprüfen wir dies in einer Funktion, die wir in sl_main aufrufen. Diese Funktion heißt sl_check und wird vor dem wait platziert:

```
   //(...)

   // Check if Rudi hits me
      sl_check();

   wait(1);
}
```

Der erste Schlitten hinter Rudi wird er nicht überlaufen können. Alle anderen Entities markieren wir nun mit einer eigenen Eigenschaft. Wir erstellen ein neues Skilldefine namens:

```
#define sledgeDestroyable        skill22
```

und weisen es allen Schlitten, die zerstört werden können, zu. Dafür müssen wir in der Funktion sl_add allen neuen angehängten Schlitten, die an einen anderen Schlitten – und nicht Rudi! - sensibilisieren.

```
if (Rudi->sledgeLast) {

   //(...)

   entTemp[0]->sledgeDestroyable = 1;

   //(...)

} else {
   //(...)
}
```

Nun können wir die Funktion testweise so schreiben:

```
void sl_check ()
{
   if (Rudi) {
      //only if we are destroyable
      if (my.sledgeDestroyable) {
```

```
            if (vec_dist(Rudi->x, my.x) < 80) { //triggerrange
               set(my, INVISIBLE); //debug: invisibility
            }
         }
      }
   }
```

Wenn wir ein zerstörbares Schlittenglied treffen, schalten wir es erstmal unsichtbar. Nach einem kurzen Test, stellen wir fest, dass es klappt. Wir wollen uns damit aber nicht begnügen - wir werden nun eine Funktion schreiben, die den Entity-Zeiger nimmt und die Restkette, beginnend mit dem getroffenen Schlitten, löscht. Die Funktion heißt sl_cut und wir setzen sie an der Stelle ein, wo wir vorher das INVISIBLE flag gesetzt haben. In der Zwischenzeit lagern wir den Schnittschwellenwert als "var sl_cutDist = 90;" aus:

```
if (vec_dist(Rudi->x, my.x) < sl_cutDist) {  //triggerrange
   sl_cut(my);
}
```

Die Funktion ist selber recht einfach gehalten: zunächst nimmt die Funktion den Vordermann des abzuschneidenden Schlittens und sagt Rudi, dass dies der neue Schwanz sei. Schließlich wollen wir auch wieder neue Schlitten an der abgeschnittenen Stelle anhängen können! Desweiteren schaltet die Funktion bei dem abgeschnittenen Schlitten das Death-Flag ein, was dem betroffenen Schlitten mitteilt, das er "gestorben" ist. Dies findet aber nur dann statt, wenn der übergebene Schlitten auch existiert - sonst erzeugen wir einen „empty pointer“ Fehler (das bedeutet, das wir auf den Inhalt des Zeigers zugreifen, obwohl der Inhalt nicht mehr existiert).

```
void sl_cut (ENTITY* sledge)
{
   if (sledge) {
      Rudi->sledgeLast = sledge->sledgeFront;
      sledge->sledgeDeath = 1;
   }
}
```

Die Hauptschleife des betroffenen Schlitten wird sich nun selbst terminieren, weil dort die Bedingung

```
while (my.sledgeDeath == 0) {
   //(...)
}
```

angegeben ist – und zutrifft! Danach soll der Schlitten sterben, in diesem Fall bedeutet das jediglich, dass wir die Entity entfernen. Wir werden dies aber u.a. mit zerplatzenden Paketen und anderen Spezialeffekten noch „schöner“ machen. Dies alles werden wir in der Funktion sl_death regeln, die nach der While Schleife aufgerufen wird:

```
      //(...)

      wait(1);
   }

   sl_death();
}
```

Die Funktion selber beinhaltet vorerst nur das Löschen der Entity:

```
void sl_death ()
{
   ent_remove(my);
}
```

Das Schöne ist: wenn wir nun durch unsere Schlitten fahren, zerstören und entfernen wir den berührten Schlitten, aber bereits der Schlitten, der vormals dahinter hang, wird sich erheblichst beschweren, weil er immer noch auf den ehemaligen Vordermann referenziert. Bevor wir diesen Fehler überhaupt zulassen, wollen wir diesen Fehler abfangen und dazu nutzen, diesen Schlitten genauso sterben zu lassen wie seinen Vordermann – der bereits entfernt wurde. Dieses Verhalten setzt dann automatisch eine Kettenreaktion nach hinten fort und führt schließlich dazu, dass alle restlichen Schlitten auch entfernt werden. Damit es nicht zu Fehlern kommt, fangen wir vorsichtshalber auch den Fall ab, dass der Vordermann noch existiert, dessen Deathflag aber schon angeschaltet ist. In diesem Fall sterben der Schlitten auch.

Diese Abfrage gehört auch in die sl_check Funktion. Damit wir aber nicht mehrere unterschiedliche Checks in einer Funktion haben, lagern wir den Berührungscheck für Rudi in die Funktion sl_check_cut aus und erzeugen für die Überprüfung des Vordermanns die Funktion sl_check_frontDeath. Diese Konstruktion sieht dann so aus:

```
void sl_check ()
{
   sl_check_cut();
   sl_check_frontDeath();
}

   void sl_check_cut ()
   {
      //(...)
   }

   void sl_check_frontDeath ()
   {
      if (my.sledgeDestroyable) {

         // Check if the front sledge isn't dead

         you = (ENTITY*)my.sledgeFront;
         if (you != 0) {
            if (you.sledgeDeath != 0) {

               //front sledge is dead -> so I die, too
               my.sledgeDeath = 1;

            }
         } else {

            // I have no front sledge, so we die instantly
               my.sledgeDeath = 1;
         }
      }
   }
```

Wenn wir nun das Spiel spielen, wird die Schlittenkette korrekt abgetrennt, wenn wir gegen einen Schlitten unserer Kette fahren. Obwohl wir uns schon damit befasst haben, Rudi gegen Wände knallen zu lassen, haben wir auch noch gar nicht die Reaktion der Schlitten in dieser Situation behandelt. Im Moment fahren sie einfach auf - und das war es! Natürlich könnte man sagen, dass das eigentlich schon reicht. Aber wäre es nicht witzig (oder anspornend, weil tragisch), wenn die Schlitten gleichzeitig auch alle kaputt gehen würden? Man könnte auch argumentieren, dass wenn Rudi stirbt, die Geschenke eh nicht zum Weihnachtsmann gelangen und daher ihre Zerstörung gerechtfertigt wären (die armen Kinder!) :-)

Damit dies geschieht, fügen wir in der Funktion pl_death vor dem switch-case-Baum folgenden Eintrag hinzu:

```
//cut sledges
   sl_cut((ENTITY*)(my.sledgeFirst));
```

Wenn Rudi dann gegen eine Wand fährt, werden alle seine Schlitten daraufhin zerstört.

Effekt: Schlitten zerstören

Wir wollen nun unseren ersten Spezialeffekt schreiben. Den haben wir auch bitter nötig, denn das "zerstören" der restlichen Schlitten sieht noch reichlich trist aus: die Schlitten werden einfach entfernt – das war's.

Bevor wir allerdings Effekte abspielen, müssen wir genauso wie für alle anderen Dinge dies vorbereiten. Effekte werden nämlich grundsätzlich dynamisch geladen und somit kann man die damit verbundenen Dateien zusammenfassen, z.B. in einem eigenen Ordner. Weil wir immer Spiel-spezifische Effekte schreiben, erstellen wir im "work\game" Ordner das Verzeichnis "effects".

Für den Schlitteneffekt habe ich 3 Sprite-Dateien vorbereitet, die wir als Spriteanimation benutzen werden. Es werden platzende Pakete dargestellt. In diesem Fall sind es 3 verschiedene Dateien, damit nicht jeder Effekt gleich

aussieht. Die Dateien schieben wir in den neuen effects Ordner. Sie lauten:

- effPacketExploA+4.tga
- effPacketExploB+4.tga
- effPacketExploC+4.tga

Effekt-Daten sollten immer eindeutig benennt werden, dazu gehört z.B. eine einheitliche Syntax oder zumindestens ein einheitliches Prefix, wie hier "eff" für "effect". Das "+4" vor der Dateiendung signalisiert eine Spriteanimation, die 4 Frames beinhaltet. Damit die engine die Dateien auch findet, müssen wir noch in der Funktion game_init in der game.c den folgenden Eintrag hinzufügen:

```
add_folder("game\\effects");       //effects data
```

Nun können wir munter auf diese Dateien zugreifen. Wir wollen nun, bevor der Schlitten per ent_remove in der Funkion sl_death gelöscht wird, eben diesen Effekt aufrufen. Den Effekt nennen wir sl_eff_exploPacket (man beachte das "eff" Kürzel!) und fügen am besten im hinteren Teil der sledge.c eine Sektion für Schlitteneffekte hinzu. Für generelle Spiel-Effekte werden wir noch extra Dateien anfertigen. Allerdings macht es Sinn, objektbezogene oder spezielle Effekt dort hinzuschreiben, wo sie auch sinvoll hingehören (z.B. Effekte die Rudi betreffen, in die player-Dateien).

Die Funktion sl_eff_exploPacket soll nun eine der 3 Dateien auswählen und erzeugen. Der Sprite soll eine Funktion zugewiesen bekommen, die das Verhalten der Spriteanimation steuert. Weil der Origin des Schlittens zur Zeit des Effekt aller Vorraussicht nach am Boden klebt (andernfalls würde der Schlitten halt durch die Luft fliegen), erhöhen wir die Z-Kooordinate der Erzeugung-Position um die Höhe des Models. Demnach sieht die Funktion so aus:

```
void sl_eff_exploPacket ()
{
   //choose from one of 3 sprite animations
      char entFile [32];
      sprintf(entFile, "effPacketExplo%c+8.tga", 'A' + (int)(random(3)));

   //create it: raise it by my height (because origin-z == ground)
      ent_create(entFile, vector(my.x, my.y, my.z + my.max_z), sl_eff_exploPacket_func);
}
```

Wie man dem Quellcode entnehmen kann, lautet die Funktion der Sprite-Entity sl_eff_exploPacket_func. Das "_func" signalisiert hier die tatsächlich ausführende Funktion des Effektes. Die Funktion, die man für den Effekt aufruft, sorgt sich normalerweise nur darum, die erforderlichen Parameter einzustellen und den Effekt zu organisieren, z.B. die Position des Effektes zu berechnen. Bei Partikel-Funktionen werden wir später dasselbe Muster wiederbenutzen.

Die sl_eff_exploPacket_func soll nun mehrere Dinge leisten, die sich jeweils nur auf einen Sprite beziehen. Zunächst einmal muss der Sprite initialisiert werden. Darunter fallen visuelle Parameter also auch Kollisionseigenschaften. Denn der Sprite muss durchlässig sein, damit Rudi nicht „dagegen“ läuft. Dann soll der Sprite seine Animation einmal mit einer gewissen Geschwindigkeit durchlaufen und sich danach selbst löschen. Der Code sieht folgendermaßen aus:

```
void sl_eff_exploPacket_func ()
{
   //init
      set(my, PASSABLE);   //so that Rudi doesnt crash against me
      my.roll = 1;         //enable facing

   //different explosion scales
      vec_scale(my.scale_x, 1.5 + random(1.5));

   //one animation cycle; my.skill1 = animation counter
      while (my.skill1 < 100) {
         ent_animate(my, "", my.skill1, 0);
         my.skill1 += 30 * time_step;
         wait(1);
      }

   //animation is over, remove sprite
```

```
        ent_remove(my);
}
```

Wir setzen hier den roll Winkel auf einen Wert ungleich 0 (während pan und tilt = 0 sind), damit der Sprite sich immer zur Kamera dreht (siehe Manual). Außerdem skalieren wir den Sprite zufällig etwas größer, damit die Sprites nicht alle gleichförmig aussehen, wenn viele Schlitten kaputt gehen. Desweiteren missbrauchen wir einen Skill als Counter, anstatt eine lokale Variable anzulegen - denn lokale Variablen verbrauchen Zeit und Speicher wenn sie erzeugt werden, wohingegen Skills immer automatisch und nicht „extra“ erzeugt werden.

Damit der Effekt auch aufgerufen wird, fügen wir den Aufruf der Funktion sl_eff_exploPacket(); vor das ent_remove(my); in der sl_death. Wenn wir jetzt mit Rudi durch einen Schlitten fahren, wird die hintere Kette abgetrennt und deren Pakete explodieren in einem schönen Effekt.

Der "Puff!"-Effekt

Nur leider sieht das noch recht dürftig aus und wirkt recht verloren. Eine Idee könnte sein, bei solchen Sachen einen kleinen Raucheffekt abzuspielen. Da Raucheffekte jetzt nicht sonderlich etwas mit Schlitten zutun haben, wollen wir nun daraus einen generellen Effekt zusammenstellen. Für den Effekt benutzen wir einen Sprite, dessen Datei "effPuff.tga" lautet. Dort drin ist ein locker-flockiger Rauchsprite (ganz nach dem Wortlaut "Puff!" - eben ein Puff-Sprite) den wir mit Hilfe einer Partikelfunktion erscheinen lassen wollen. Wir legen die Datei in den effects Ordner, indem sich auch die Paket-Explosions-Sprites befinden. Die generellen Effekte werden auch in einem eigenen Code-Modul behandelt, das wir in den Dateien effects.c und .h festhalten wollen. Diese speichern wir dann im "game" Ordner und inkludieren sie in der jeweiligen Sektion in der Datei "rudi.c".

Partikeleffekte benutzen bereits geladene Bitmapdateien für die Darstellung von Partikeln. Das heißt: wenn man in dem Effekt eine Bitmap angibt, muss sichergestellt sein, dass sie geladen ist. Man kann zum Beispiel Bitmaps bereits beim Engine-Startup laden, indem man eine statische Definion der Bitmap durchführt:

```
BMAP* eff_puff_bmap = "effPuff.tga";
```

Allerdings würde die Engine meckern, weil sie die Datei nicht findet. Schließlich werden die Content-Verzeichnisse erst später zur Laufzeit hinzugefügt. Um das Problem zu umgehen, wollen wir uns eine Hilfsfunktion schreiben, die eine Bitmap checkt, ob sie geladen ist oder nicht - und wenn sie nicht geladen ist, wird die Bitmap dynamisch erzeugt. Die Funktion nennen wir bmapCheck und schreiben sie in die Datei "sysUtil.c". Als Parameter erhält sie den Bitmap-Zeiger, der überprüft werden soll und eine Zeichenkette, die den Dateinamen angibt.

```
BMAP* bmapCheck(BMAP* source, char* file)
{
   if (source == 0) {
      source = bmap_create(file);
   }

   return(source);
}
```

Nachdem der Bitmapzeiger überprüft und (eventuell) gefüllt wurde, wird der Zeiger zurückgegeben - diesen Mechanismus nutzen wir, um dann die Bitmap in Zuweisungen zuzuweisen. Wir haben zwar noch nicht die Effektfunktion geschrieben, können aber schonmal den Sprite in der Effects-Headerdatei angeben:

```
//-----------------------------------------------------------------
// ressources
//-----------------------------------------------------------------

   //PUFF EFFECT

      BMAP* eff_puff_bmap;
         char* eff_puff_file = "effPuff.tga";
```

Für das Erzeugen eines Partikeleffektes gibt es nun in der Engine einen eigenen Befehl, der sich "effect" nennt. Ihm übergibt man die initialisierende Partikelfunktion, die Anzahl der Partikel, die man erzeugen will, den Ort an dem man den Effekt emittieren will und einen Geschwindigkeitsvektor, mit dem die Partikel zu Beginn initialisiert werden (z.B. für Wind, Wasserdruck oder sowas). Wir werden den Puff-Effekt „eff_puff“ nennen. Der Rumpf der Funktion sieht so aus:

```
void eff_puff (PARTICLE *p)
{
   //(...)
}
```

Der Parameter ist vom Typ PARTICLE: die effect Funktion erzeugt halt die Partikel und ruft für jeden Partikel die Startfunktion auf und übergibt die Partikelreferenz, damit die Startfunktion den Partikel auch initialisieren kann. Zu Beginn definiert man in der Regel ganz einfache Sachen, wie die Bitmap, die Skalierung, die Transparenz, etc.:

```
void eff_puff (PARTICLE *p)
{
   p.bmap = bmapCheck(eff_puff_bmap, eff_puff_file);
   p.size = 32 + random(64);
   p.alpha = 25+random(25);
   p.lifespan = 100;
   set(p, TRANSLUCENT | MOVE);

   //(...)
```

In diesem Fall wenden wir auch gleich schon unsere bmapCheck Funktion an. Desweiteren skalieren wir den Sprite zufällig und tun das auch für die Alphatransparenz. Lifespan gibt die Anzahl der Ticks wieder, die der Partikel existiert, in diesem Fall 100 Ticks. Desweiteren schalten wir das Alphatransparenz-Flag an und auch das MOVE Flag, damit sich der Sprite bewegen kann. Im nächsten Schritt kann man z.B. die Position verschieben:

```
//displace start position

   vec_set(p.skill_x, vector(16 + random(48),0,0));
   vec_rotate(p.skill_x, vector(random(360), random(90), 0));
   vec_add(p.x, p.skill_x);
```

In diesem Fall missbrauchen wir den skill_x Vektor, um temporäre Berechnungen auszuführen. Damit der Partikel auch etwas "wabert" - und sich also in der Luft bewegen kann -, müssen wir seine Geschwindigkeitswerte (in diesem Fall zufällig) initialisieren:

```
//velocity

   vec_set(p.vel_x, vector(5 + random(8),0,0));
   vec_rotate(p.vel_x, vector(random(360), random(90), 0));
```

Damit ist der Partikel eigentlich schon vorbereitet. Fehlt nur noch die Eventfunktion des Partikels. Die Eventfunktion wird für jeden Partikel automatisch aufgerufen. Die Funktion, die wir festlegen, ist eine engine-interne Framefunktion, also dürfen wir keine wait-Anweisungen benutzen. Die Eventfunktion dient dem Verändern des Partikels während seiner Lebenszeit. Die Eventfunktion für unseren Puffsprite ist recht simpel:

```
void eff_puff_func (PARTICLE* p)
{
   p.alpha -= 5 * time_step;           //fade out
   p.size += 2 * time_step;            //grow
   p.lifespan = (p.alpha > 0) * 100;   //survive if visible
}
```

Der Partikel blendet langsam aus, wird etwas größer und bleibt solange am Leben, wie der Alphawert noch nicht 0 erreicht hat. Wir müssen die Funktion nur noch in den Partikel eintragen, indem wir in der Funktion eff_puff am Schluss den folgenden Eintrag hinzufügen:

```
p.event = eff_puff_func;
```

Damit ist der Effekt vorbereitet. Wir können nun z.B. in der Funktion sl_eff_exploPacket am Ende den Partikeleffekt einfach aufrufen:

```
//some smoke puffs
   effect(eff_puff, 2, vector(my.x, my.y, my.z + my.max_z * 0.5), nullvector);
```

Wenn wir nun in unsere eigene Schlange fahren, zerplatzen die Schlitten nun mit einem ansehnlichen, dennoch nicht zu aufgesetztem Effekt. Dies ist ein Screenshot aus dem fertigen Spiel, dass diesen Effekt schön wiedergibt:

Der SparkMagic - Effekt und automatisierte Emittertechniken

Wir wollen jetzt vorerst einen letzten Effekt programmieren, der dann im Zusammenspiel mit den anderen Effekten und dem Gameplay die Stimmung noch etwas mehr auflockern soll. Dieser Effekt wird "magisch" sein und dann abgespielt werden, wenn wie durch Geisterhand die Schlitten angehängt werden – oder auch wenn Rudi Pakete einsammelt. Wir benutzen einen weiteren Sprite, der "effSparkMagic.tga" heißt.

Die Idee ist, dass wenn ein neuer Schlitten erzeugt wird, an der Außenhülle des Schlittens magische Funken austreten und wegfliegen - weil die Erzeugung des Schlittens "ohne alles" ziemlich langweilig aussieht! Wir müssen bei der Erzeugung mit der effect Funktion immer angeben, wo die Partikel erzeugt werden. Um dann den Partikel zu verschieben, haben wir die Koordinaten bisher in der Starterfunktion verändert. Das die Funken an der "Außenhülle" erzeugt werden sollen, lässt uns aber auf folgenden Umstand schließen: da das Modell des Schlittens aus Vertexen besteht, müssen wir einfach ein paar dieser Vertices nehmen und an deren Stellen den Partikel erzeugen - somit werden sie "quasi" an der Außenhülle erzeugt!

An diesem Beispiel kann man recht schnell sehen, dass man bestimmte Muster gut abtstrahieren kann. Für Partikeleffekte kann man dann z.B. eigene Erzeuger-Funktionen schreiben, die dann nach einem bestimmten Muster und bestimmten Parametern einen oder mehrere Partikeleffekte startet - wie in diesem Beispiel das "Erstellen eines Partikeleffekts an der Außenhülle einer Entity". Dafür schreiben wir uns in der effects.c folgende Spawn-Funktion:

```
void eff_spawnEnt_verts (ENTITY* ent, int nr, void* fnc)
{
   int i;
   for (i = 0; i < nr; i++) {
      vec_for_vertex(vecTemp.x, ent, random(ent_vertices(ent)));
      effect(fnc, 1, vecTemp, nullvector);
   }
}
```

Die Funktion nimmt eine Entity (für die Hülle), die Anzahl der Partikel und einen void* Pointer. Dieser Zeiger ist der Zeiger auf eine Funktion - in diesem Fall die Partikel-Startfunktion. In der Spawn-Funktion selbst wird so oft wie Partikel erzeugt werden sollen, ein zufälliger Vertex genommen und an dessen Stelle der Effekt mit der angegebenen Funktion gestartet. Jetzt müssen wir nur noch unseren Funken-Effekt schreiben:

```
void eff_sparkMagic(PARTICLE* p)
{
   p.bmap = bmapCheck(eff_sparkMagic_bmap, eff_sparkMagic_file);
   p.size = 16 + random(8);

   //velocity

      vec_set(p.vel_x, vector(5 + random(8),0,0));
      vec_rotate(p.vel_x, vector(random(360), random(90), 0));

   my.alpha = 100;

   set(p, TRANSLUCENT | BRIGHT | MOVE);
```

```
   p.lifespan = 50;
   p.event = eff_sparkMagic_func;
}

   //- particle function ------------------------------------------
   void eff_sparkMagic_func (PARTICLE* p)
   {
      p.alpha -= 5 * time_step;                 //fade out
      p.size = maxv(8, p.size - 1 * time_step); //shrink, at least 1
      p.lifespan = (p.alpha > 0) * 100;         //survive if visible
   }
```

Die nötigen Ressourcen sind in der effects.h wie folgt definiert:

```
//SPARK MAGIC EFFECT

   BMAP* eff_sparkMagic_bmap;
      char* eff_sparkMagic_file = "effSparkMagic.tga";
```

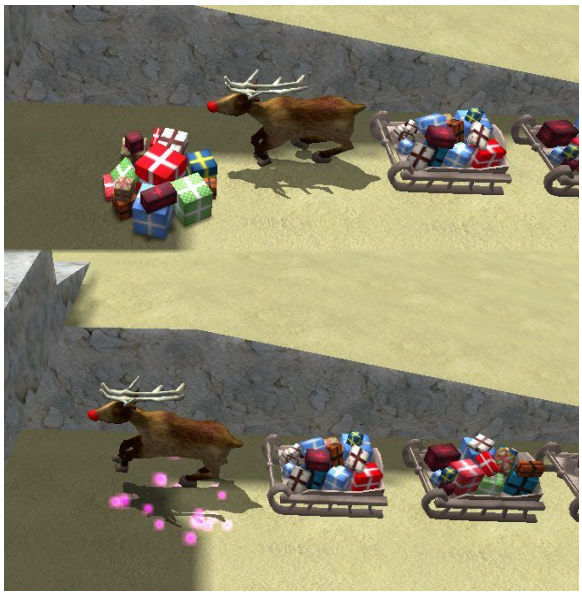
Damit ist der Effekt einsatzbereit. Wenn nun also ein Schlitten erstellt wird, führt er automatisch die Initialisierungsfunktion sl_init aus. Dort können wir den Effekt platzieren und rufen ihn über die neue Hüllen-Emitter Funktion auf:

```
void sl_init ()
{
   //(...)

   //effect: spark
      eff_spawnEnt_verts(my, 6, eff_sparkMagic);
}
```

Wir erzeugen nun an der Stelle, an der der Schlitten erzeugt wird, ein paar Funken. Wenn wir das im Spiel testen funktioniert es und sieht echt stimmig aus! Um dem ganzen die Krone aufzusetzen, wenden wir denselben Effekt an, wenn Rudi durch einen Pakete-Haufen fährt. Dort müssen wir es genau andersherum machen: dort wird das Paket in der Funktion pk_remove entfernt. Dort fügen wir ganz zu Beginn den Aufruf ein:

```
void pk_remove()
{
   //effect: spark
      eff_spawnEnt_verts(my, 20, eff_sparkMagic);

   //save my pointer
      ENTITY* mySave = my;

   //(...)
```

Wir haben jetzt im Prinzip einen super-einfachen Effekt geschrieben und doch so viel gewonnen: es sieht alles gleich viel stimmiger aus!



Zusammenfassung

Wir haben uns noch einmal mit den Schlitten und der Schlittenkette auseinandergesetzt. Nun sind alle wichtigen Dinge der Schlitten von Anfang bis Ende erfasst: wir erzeugen Schlitten, sie bewegen sich und dann sterben sie. Rudi kann gegen Wände crashen und „stirbt“ dann. Dies ist sogar die Umsetzung einer unserer wichtigsten Spielregeln: der Spieler kann „verlieren“ und muss das Level erneut spielen, weil er z.B. „unachtsam“ war. Je komplizierter die Levels sind (oder je nachdem wie wir die Pakete platzieren), desto herausfordernder wird das Spiel – und das macht den Spaß des Spiels aus!

Zudem haben wir bereits ein paar generelle und spezielle Spezialeffekte programmiert. Wir haben Partikeleffekte geschrieben und auch Spriteanimationen benutzt und schon einen ganz interessanten Einblick gewonnen, was man und vor allem wie man mit einfachen Mitteln ein Spiel bereichern kann.

Im Nächsten Abschnitt programmieren wird das Spielziel (das Ausgangstor) und das Punktesystem des Spiels. Damit sind dann auch die primären Arbeiten erledigt und wir haben – zumindestens in unserem Testlevel – das komplette Gameplay eingebaut. Alles was danach kommt dient der Abwechslung, der Austattung und der Herausforderung.

Kapitel 8: Der Levelabschluss

Das Tor

Wir haben in unseren Überlegungen für das Spiel eine Art Tor vorgesehen, dass den Zugang zum nächsten Level gewährt, wenn der Spieler alle Pakete eingesammelt hat. Dazu wollen wir ein animiertes Modell nehmen. Dem Spiel ist ein solches Model in 2 Modeldateien beigelegt: das Tor ist in der Datei "levelgate.mdl" modelliert. Die passende Umgebung dazu (die nicht animiert ist!) findet sich in der Datei "levelgateEnvr.mdl" (wobei das "envr" für engl. "environment" steht). Wenn beide Modelle dieselbe Position im Raum haben, passen sie genau ineinander. Das Tor ist mit 2 Bones animiert, um die Dateigröße kleinzuhalten. Die Animationssequenz lautet "gate" und zeigt nur, wie sich das Tor öffnet. Die beiden Modelle sind deshalb gesplittet, weil das Umgebungs-Modell die ganze Zeit kollisionsfähig sein soll und das Tor nur dann, wenn es geschlossen ist.

Das Tor und seine Umgebung werden eigene Actions erhalten. Da wir bisher noch keine generellen Objekte benötigt haben, werden wir eine Datei für so etwas anlegen. Die Datei nennen wir "objects.c" und "objects.h". In die objects.c schreiben wir einen trivialen Code, der beide Objekte initialisieren soll:

```
//-----------------------------------------------------------------------
// LEVELGATE
// Opens, when no packets are left to collect.
//-----------------------------------------------------------------------
void obj_gate ()
{
   // Initialization
      set(my, POLYGON);          //enable collision
      reset(my, DYNAMIC);        //not dynamic
}

   //- environment model ------------------------------------------
   void obj_gateEnvr ()
   {
      set(my, POLYGON);
      reset(my, DYNAMIC);
   }
```

Beide Objekte sollen polygonale Erkennung besitzen und nicht dynamisch sein. Ist das DYNAMIC Flag nicht gesetzt, ist die Entity statisch, was bedeutet, sie verändert sich nicht zur Laufzeit. Statische Entities verbrauchen weniger Speicher- und CPU-Ressourcen als dynamische Entities - und genau aus diesem Grund verwenden wir dies für das Gate und das Environment Modell.

Damit wir den Entities im WED auch eine Action zuweisen können, fügen wir in die actions.wdl die folgenden Einträge ein:

```
action obj_gate
{
   wait(1);
}

   action obj_gateEnvr
   {
      wait(1);
   }
```

Wir laden dann im WED beide Modelle und weisen ihnen die Actions zu. Stelle sicher, dass beide Modelle die gleiche Position haben. Zur Not gibst Du in der zweiten Entity die Positionsdaten direkt im Eigenschaftsfenster ein.

Damit das Gate nun wartet, bis alle Geschenke eingesammelt sind, fügen wir in der Funktion obj_gate nach der Initialisierung eine pausierende While-Schleife ein, die genau diesen Zustand beobachtet:

```
// Wait until last packet has been collected
   while (pk_count > 0) {
      wait(1);
   }
```

Ist der Zustand eingetreten, hört die Schleife auf und wir können fortfahren. Das Tor soll sich dann in diesem Fall öffnen, in dem die Animation einmal abgespielt wird. Außerdem stellen wir die Entity auf passable, damit wir nicht gegen eine "unsichtbare Wand" crashen, wenn wir durch das Tor fahren (denn die Kollisionshülle wird nicht geupdatet, wenn eine Entity animiert wird).

```
// Open gate (animation)
   set(my, PASSABLE);
   var i;
   for (i = 0; i < 100; i += 5 * time_step) {
      ent_animate(my, "gate", i, 0);
      wait(1);
   }
```

Das Tor bleibt danach auch auf passable stehen. Das Problem ist: wenn wir die Kollision dann wieder anschalten wollen, müssten wir mit c_updatehull die Kollisionshülle für den aktuellen Animationsframe neu berechnen. Dies ist langsam und führt zu einem merklichen Ruckler im Spiel. Desweiteren neben dem Tor große Steine und zwei überdimensionierte Zuckerstangen - wenn der Spieler es wirklich nicht schaffen sollte, durch das Tor zu fahren, dann knallt er halt dagegen und gut ist. Deshalb lassen wir die Flügeltüren des Tores auf passable stehen. In diesem Fall ziehen wir die schnellere als die akkuratere Lösung vor.

Weil das Tor animiert wurde, ist das DYNAMIC Flag wieder aktiv. Wir schalten es dann schlussendlich aus:

```
// It won't change anymore, so we disable dynamic
   reset(my, DYNAMIC);
```

Zuletzt lagern wir noch ein paar Konstanten in den Header aus:

```
int obj_gate_threshold = 0;            //with x packets left the gate is opened
char* obj_gate_openAnim_str = "gate";  //animation string
var obj_gate_openAnim_speed = 5;       //animation speed
```

"obj_gate_threshold" gibt an, wieviele Pakete noch übrig sein müssen, damit sich das Tor öffnet. Wenn die Variable also z.B. den Wert 3 hat, wird das Tor vorzeitig geöffnet, wenn nur noch 3 Pakete zu sammeln wären. Diese Variable wird in

```
// Wait until last packet has been collected
   while (pk_count > obj_gate_threshold) {
      wait(1);
   }
```

eingesetzt. Wir lassen sie erstmal auf 0 stehen. Die restlichen beiden Variablen werden in den Animationscode eingesetzt.

Das Tor ist nun einsatzfähig: wir haben im Moment nur ein paar Pakete im Level platziert, deshalb ist es ein einfaches, eben alle Pakete einzusammeln. Daraufhin sieht man das Tor sich öffnen (wenn Du gerade das Tor nicht im Blickfeld hast, freeze das Spiel mit F8 und gehe mit der Debug-Kamera (Taste 0) dort hin) und man kan hindurch fahren. Wenn es noch nicht offen ist, fährt man dagegen und Rudi crashed.

Die Lebens- und Punkteanzeige

Damit der Spieler immer weiß, wieviele Leben er noch zur Verfügung und wieviele Punkte in Form von Lebkuchenherzen er gesammelt hat, wollen wir 2 Grafiken und 2 Zahlenanzeigen auf dem Bildschirm anzeigen. Dazu sind diesem sind dem Spiel 3 spezielle Dateien beigelegt: eine Zahlen-Font Datei namens "hudFntNr.dds", ein Symbol für die Herzen ("hudHeart.DDS") und ein Symbol für die Leben ("hudLifes.dds"). Für die Anzeige werden wir 2 neue Code-Dateien anlegen: "hud.c" und "hud.h". Außerdem werden wir im game Ordner einen weiteren Unterordner namens "hud" eröffnen, indem die ganzen HUD-spezifischen Dateien gespeichert werden - dort werden wir auch sofort die 3 oben beschriebenen Dateien hinspeichern.

Damit die Anzeigen auch etwas anzeigen können, müssen wir desweiteren 2 Variablen einführen, die die Anzahl der Leben und der Herzen festhält. Wir wollen dies in der Datei game.h tun:

```
var lifes;
var points;
```

Wir brauchen für diese Anzeige 3 Dinge: 2 Panels, die die beiden Symbolgrafiken und die Zahlen als Digits anzeigen und einen Font, der dafür geladen ist. Wir werden die Panels und den Font dynamisch zur Laufzeit erzeugen, weil die Pfade zu den Dateien, wie bereits bekannt, erst nach engine-Start registriert werden. Als erstes wollen wir den Font laden. Dazu definieren wir eine Font-Referenz in der Datei hud.h:

```
FONT* hud_fntNr;
```

Wir wollen den Font erzeugen und dann für das HUD benutzen. Dazu bietet sich eine Initialisierungsfunktion an, die nur einmal aufgerufen wird. Sie lautet:

```
void hud_init ()
{
   // Create fonts
      hud_fntNr = font_create("hudFntNr.dds");
}
```

Damit die Funktion auch aufgerufen wird, schreiben wir den Aufruf in den Aufruf der game_init in der Datei game.c. Die Dateien, die wir für das HUD brauchen, befinden sich im Ordner "hud". Diesen müssen wir auch noch registrieren:

```
void game_init ()
{
   //add game content folders to engine file system
      add_folder("game\\levels");   //level data
      add_folder("game\\effects");  //effects data
      add_folder("game\\hud");   //effects data

   //(...)

   //initialize HUD
      hud_init();
}
```

Wenn also das Spiel startet, wird automatisch der Font für die Ziffernanzeigen geladen und erzeugt.

Panels werden dynamisch mit pan_create. Zusätzlich dazu werden wir die Positionen der Panels bestimmen und sie anschalten müssen. Weil das typische Operationen sind, die man für jedes dynamisch erzeugte Panel durchführen muss, verallgemeinern wir dies zunächst in einer neuen Utility-Funktion namens pan_createEx und speichern sie in der Datei sysUtil.c:

```
PANEL* pan_createEx (char* content, var layer, char* bitmap, int posX, int posY, long flags)
{
   PANEL* tmpPnl = pan_create(content, layer);

   if (bitmap != NULL) {
      tmpPnl->bmap = bmap_create(bitmap);
      tmpPnl->size_x = bmap_width(tmpPnl->bmap);
      tmpPnl->size_y = bmap_height(tmpPnl->bmap);
   }

   tmpPnl->pos_x = posX;
   tmpPnl->pos_y = posY;

   set(tmpPnl, flags);

   return(tmpPnl);
}
```

Das Panel wird mit einem content String und einem Layer erstellt (der content String dient z.B. dazu, Digits oder Buttons anzugeben). Daraufhin wird eine Bitmap geladen und das Panel positioniert. Wenn allerdings der Parametert bitmap == NULL ist, dann wird der bmap_create nicht ausgeführt. Dieses Verhalten benötigen wir z.B. in der Gestaltung des Menüs, wenn wir Panels mit buttons laden wollen, aber keine Panel_grafik selber brauchen. Angegebene Flags werden im Anschluss gesetzt. Die Funktion liefert am Ende den Zeiger auf das neu erstellte Panel zurück.

Nun können wir in der Funktion hud_init innerhalb von 2 Zeilen die beiden Panels erzeugen:

```
// Create panels
   hud_heart = pan_createEx("digits(-131, 64, 5, hud_fntNr, 1, points);",
                            998, "hudHeart.dds", 845, 604, 0);

   hud_lifes = pan_createEx("digits(97, 126, 1, hud_fntNr, 1, lifes);",
                            998, "hudLifes.dds", 20, 542, 0);
```

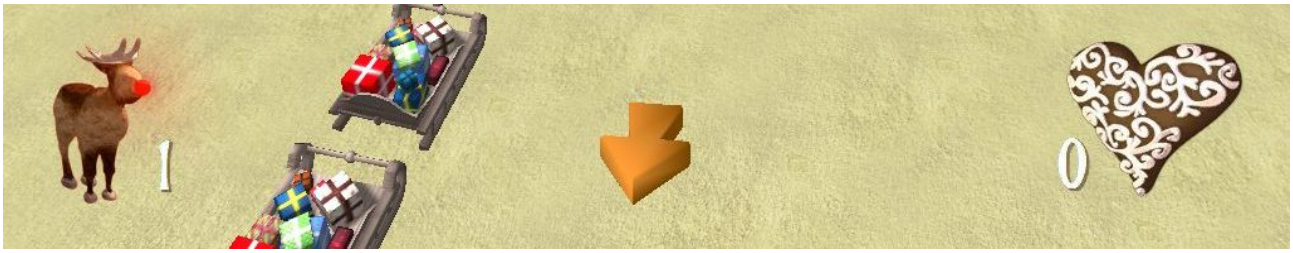
Hier passieren viele interessante Sachen. Zunächst wird den Panels ein content string gegeben, die sagen, dass eine Digitanzeige auf dem Panel erstellt werden soll. Die Digits werden auf dem Panel positioniert und der vorhin erzeugte Font zugewiesen. Die Digits nehmen die anzuzeigende Zahl aus den Variablen points und lifes. Das Panel selbst hat eine Grafik, die angegeben ist und wird an einer bestimmten Stelle positioniert. Im Anschluss werden die Panels sichtbar gemacht.

Damit die Lebensanzeige bei Spielstart zurückgestellt wird, wollen wir dies in die Initialisierungsfunktion von Rudi packen: denn wenn ein Level geladen wird, die Lebensanzeige auf 0 steht, dann ist klar, das wir das Spiel

insgesamt komplett neu gestartet haben. Wir schreiben also in Rudi's Initialisierungsfunktion:

```
// Reset lifes if they are =0
   if (lifes <= 0) {
      lifes = lifes_resetNr;
      points = 0;
   }
```

Die Variable lifes_resetNr ist = 5 und wurde in die game.h ausgelagert. Bei Spielstart hat Rudi sofort 5 Leben und 0 gesammelte Herzen.



Das Level beenden

Nachdem wir nun das Tor öffnen lassen, stellen wir fest, das wir ganz einfach dort hindurch fahren können, ohne das etwas passiert. In dem Moment, wenn wir dort hindurch fahren, soll Rudi anhalten und das Level soll beendet werden. Dafür basteln wir uns einen Trigger, der dies feststellt. Diesem Kapitel ist ein Modell beigelegt, welches den Namen "levelgateTrig.mdl" trägt. Dieses 3D Modell ist eine einfache Scheibe. Wir wollen Sie genau hinter dem Gate platzieren - der Radius der Scheibe ist der Radius des Triggers. Die Funktion des Triggers ist auch ganz einfach gestrickt: wir warten bis Rudi ankommt und teilen ihm dann mit, dass er das Level beendet hat: dazu werden wir seinen Death-Skill anstatt auf 1 (Tod durch Crash) auf 2 setzen und ihn dann darüber stoppen.

```
//- levelgate trigger ---------------------------------------------
void obj_gateTrig ()
{
   set(my, PASSABLE | INVISIBLE); //no collision

   while (!Rudi) {wait(1);} //pointer validation

   while (vec_dist(my.x, Rudi->x) > my.max_x) {wait(1);} //wait until Rudi comes along

   // Deactivate Rudi. Death = 2 means: level finish
      Rudi->playerDeath = 2;
}
```

Damit wir der Entity die Funktion auch zuweisen können, fügen wir die Funktion wie die anderen Actions in die actions.wdl ein und weisen sie dann der Trigger-Scheibe im WED zu.

Rudi wird nun seine While-Schleife beenden, weil sein Death-Skill nunmehr nicht mehr auf 0 steht und führt die Funktion pl_death aus. In dieser Funktion wurden die restlichen Schlitten abgetrennt und dann über einen Switch-Case Tree bei der 1 die Funktion pl_death_crash ausgeführt. Bei der 2 wollen wir die Schlitten aber nicht abtrennen, also verschieben wir den Aufruf von sl_cut in die Funktion pl_death_crash und führen den Aufruf vor allen anderen Codeteilen aus. Im Switch-Case Tree fügen wir dann den Fall für die 2 ein und rufen dafür eine Funktion auf, die wir pl_death_levelend ein. Witzigerweise stirbt Rudi nicht, aber wir bezeichnen das Aufhören seines "normalen" Verhaltens (nämlich rumrennen und Geschenke sammeln) als "Tod". Die Funktion pl_death sieht demnach so aus:

```
void pl_death ()
{
   switch (my.playerDeath) {

      case 1:  pl_death_crash();    //crashing (default)
               return;
```

```
      case 2:  pl_death_levelend(); //levelend
               return;

      //add other death types here
   }
}
```

Wenn Rudi also das Level beendet, wird seine Hauptschleife verlassen und pl_death_levelend ausgeführt. Rudi wird nun nur noch veranlassen, dass die Schlitten, die er noch hat, in Lebkuchenherzen (für die Punktewertung) verwandelt werden. Dafür werden wir uns eine Funktion schreiben, die am Kopf der Schlange anfängt und sich halb-rekursiv durch die Schlange "bewegt" und jeden Schlitten veranlässt, sich aufzulösen und in die Punktewertung einzufließen. Diese Funktion ist den Schlitten zugeteilt un dlautet sl_getHearts. Ihr wird der Entitypointer des ersten Schlittens zugewiesen:

```
void pl_death_levelend ()
{
   //We need at least one sledge
   if (my.sledgeFirst != 0) {

      //Get hearts for sledges
      sl_getHearts((ENTITY*)(my.sledgeFirst));
   }
}
```

Die Funktion legen wir in der sledges.c an. Zu Testzwecken schreiben wir sie erstmal so:

```
void sl_getHearts (ENTITY* sledge)
{
   deb_print("Trigger!");
}
```

Wir beenden das Level und fahren mit Rudi durch das Tor. Wenn alles gut geht, wird uns die Debuganzeige das Wort "Trigger!" ausdrucken - alles läuft gut und so, wie wir es wollten!

Wie man aus Schlitten Lebkuchenherzen machen

Genau wie wir Rudi über den Death-Skill einen bestimmten Steuerungscode gegeben haben, werden wir dies auch bei den Schlitten machen. Allerdings haben wir beim Programmieren des Schlitten-Todes nur generell darauf reagiert, dass der Death-Skill des Schlittens ungleich 0 wird. Wir lagern also den Code der sl_death Funktion in eine eigene Funktion aus namens sl_death_crash aus. Das Verhalten des Schlittens wird in der Funktion sl_death_heart festgehalten. Wir sagen, dass sich der Hintermann sich auch in ein Herz verwandeln soll und der aktuelle Schlitten gelöscht wird. Das sieht dann so aus:

```
void sl_death ()
{
   switch (my.sledgeDeath) {
      case 1:  sl_death_crash(); return;  //crash
      case 2:  sl_death_heart(); return;  //levelend -> hearts
   }
}

   void sl_death_crash ()
   {
      sl_eff_exploPacket();
      ent_remove(my);
   }

   void sl_death_heart ()
   {
      if (my.sledgeBack != 0) {
         sl_getHearts((ENTITY*)(my.sledgeBack));
      }

      ent_remove(my);
   }
```

In der Funktion sl_death_heart wird wieder die Funktion sl_getHearts benutzt. Das eine Funktion eine andere aufruft, damit diese die erste wieder aufruft und so weiter (also immer abwechselnd) nennt man indirekt rekursiv. Wir könnten das auch echt rekursiv schreiben (sl_getHearts würde sich dann immer selbst aufrufen) aber es erscheint intuiver, dass der Schlitten diese Nachricht erhält und weiterschickt, als wenn dies "über" seinem Kopf geschieht. Aber man kann es, wie gesagt auch anders machen. Dem geneigten Leser wird dies eine einfache Übung sein.

Wir haben die Funktion sl_getHearts selber noch nicht geschrieben (naja, zumindestens nicht wirklich. Wir werden die debug-Anweisung jetzt ersetzen). Sie tut nichts anderes, als den Death-Skill eines Schlittens zu setzen:

```
void sl_getHearts (ENTITY* sledge)
{
   //If starting sledge is available
   if (sledge) {
      sledge->sledgeDeath = 2; //initialize chain reaction
   }
}
```

Wenn wir jetzt mit Rudi alle Pakete einsammeln und durch das Tor laufen, bleibt er stehen und die Schlitten werden alle gelöscht. Super!.. naja fast: eigentlich interessieren uns ja zwei Sachen: einmal primär, das uns Herzen gutgeschrieben werden und zweitens, dass die Schlitten sich in einem tollen Effekt auflösen.

Someone is watching you

Damit die Punkte, die wir durch die Schlitten erhalten werden, nicht stümperhaft auf die points Variable aufaddiert werden, wollen wir uns eine Methode ausdenken, die die Zahl eher "hochschnellen" lässt, als plötzlich den Wert zu ändern. Das kommt dann auch besser rüber, wenn wir dies später z.B. mit Soundeffekten vertonen werden. Dazu denken wir uns folgendes Konstrukt aus: die Variable points enthält den echten Punktestand. Wenn wir also z.B. 10 Punkte addieren wollen - und dies soll nicht mit einem Schlag passieren - müssen wir mitteilen, das wir 10 Punkte addieren wollen und die werden dann nacheinander addiert. Dazu führen wir die Variable points_toAdd ein, die wir direkt hinter points in der game.h platzieren. Dort geben wir an, wieviele Punkte zu addieren sind.

Wir werden nun während des Spiels eine Funktion nebenbei laufen lassen, die diese Variable überwacht und dann dementsprechend reagiert. Die Funktion heißt game_watchPoints und wird in der game.c geschrieben:

```
void game_watchPoints ()
{
   while (1) {

      //Add points, if requested
      if (points_toAdd > 0) {

         points++;
         points_toAdd--;

         wait(-0.025);
      }

      wait(1);
   }
}
```

Jedes Mal, wenn in points_toAdd eine Punktzahl steht, die hinzugefügt werden soll, wird points um 1 erhöht und points_toAdd um 1 erniedrigt. Dies geschieht aber in einem festen Intervall: ein negatives Argument in einer wait Anweisung gibt einen Sekundenwert wieder. In diesem Fall einen recht kurzen. Die Funktion werden wir von Rudi aus starten, damit sie aufhört, wenn das Level gewechselt wird:

```
void pl_rudi_init ()
{
   //(...)

   //call game watchers
```

```
        game_watchPoints();
}
```

Wir können nun in die Funktion sl_death_heart Punkte übergeben, bevor der Schlitten gelöscht wird:

```
//add points
   points_toAdd += 10;
```

Damit die 10 nicht hardgecodet ist, lagen wir sie aus. Wir legen in der game.h eine Auflistung aller zu vergebenden Punkte an:

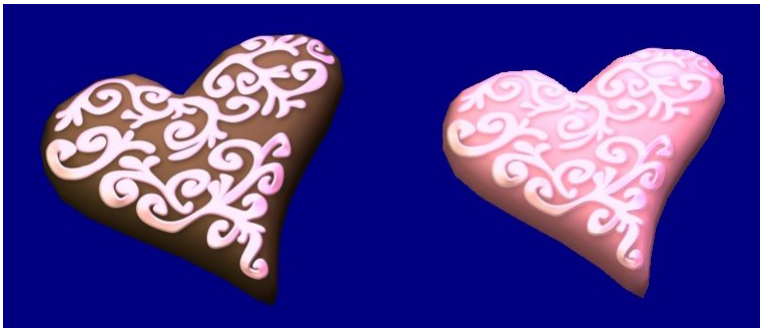
```
// Point table
   #define POINTS_SLEDGE   10
```

und ersetzen die 10 durch POINTS_SLEDGE in der Funktion sl_death_heart.

Der Herzeffekt der Schlitten

Wir haben bereits einige Effekte geschrieben. Obwohl ich ein eigenes Kapitel für einige echt schöne Effekte vorgesehen habe (dort werden dann ganz viele verschieden Techniken für Effekte vorgestellt), schreiben wir hin und wieder schon einen, um das jeweilige Kapitel abzurunden. Dazu gehört auch in diesem Fall der Effekt, den wir platzieren wollen, wenn ein Schlitten zu einem Herz wird. Zusätzlich zu einem weiteren Partikeleffekt wollen wir auch ein Herz aufploppen lassen. Wir wollen als erstes das aufploppende 3D Modell des Herzens zur Gemüte führen.

Dem Spiel sind zwei Modelle eines Herzens beigelegt, wobei sich beide Modelle in ihren Texturen unterscheiden. Das "heart.mdl" Modell ist das Herz, welches auch auf dem Bildschirm zu sehen ist. Die Variante, "heartB.mdl" hat einen rosa Zuckerüberzug und werden wir für diesen Effekt benutzen.



Die Effektfunktion, die wir in der effects.c Datei dafür schreiben werden, wird einen Positionsvektor übernehmen und dort das Herz erstellen, welches seine eigene Funktion haben wird. Die Effektfunktion sieht so aus:

```
void eff_sledgeHeart3D (VECTOR* pos)
{
   ent_create("heartB.mdl", pos, eff_sledgeHeart3D_func);
}
```

Die dazugehörige Entity-Funktion eff_sledgeHeart3D_func wird das Modell auf passable stellen und ein wenig drehen und dann wieder entfernen. Die Funktion ist relativ einfach gehalten und wird deshalb nicht gesondert erläutert:

```
void eff_sledgeHeart3D_func ()
{
   set(my, PASSABLE);

   my.skill1 = 1.75 * 16;        //time
   my.skill2 = 15 + random(15);  //pan speed
   my.skill3 = 16;               //z-raise speed
   my.skill4 = random(360);      //seed (scaling)
```

```
   if (my.pan == 0) {
      my.pan = random(360);
   }

   while (my.skill1 > 0) {

      my.pan += my.skill2 * time_step;

      my.z += my.skill3 * time_step;
      my.skill3 *= 0.8 * time_step;

      my.skill1 -= time_step;

      my.scale_y = my.scale_z = my.scale_x = 1 + sin((total_ticks + my.skill4) * 8) * 0.25;

      wait(1);
   }

   ent_remove(my);
}
```

Wir können nun den Effekt in die Funktion sl_death_heart einbauen. Dazu berechnen wir die Stelle, an der das Herz erscheinen soll und erzeugen es dann. Damit wir nicht für jeden Schlitten ein Herz Modell erzeugen (es sind nur einige wenige in der Kamera sichtbar, schränken wir die Schlitten ein, indem wir die Distanz zu Rudi prüfen. 800 Quants als erlaubter Radius sollten reichen:

```
   //(...)

   if (vec_dist(my.x, Rudi->x) < 800) {

      vec_set(vecTemp.x, vector(0, 0, 145));
      vec_rotate(vecTemp.x, my.pan);
      vec_add(vecTemp.x, my.x);

      eff_sledgeHeart3D(vecTemp.x);
   }

   //add points
      points_toAdd += POINTS_SLEDGE;

   ent_remove(my);
}
```

Damit ist der Herzeffekt zwar "vorhanden" aber noch irgendwie unspektakulär. Wir könnten unseren eff_sparkMagic - Effekt am Anfang des Herzens (wenn es erscheint) und am Ende abspielen:

```
void eff_sledgeHeart3D_func ()
{
   set(my, PASSABLE);

   //effect: spark
      eff_spawnEnt_verts(my, 12, eff_sparkMagic);

   //(...)

   //effect: spark
      eff_spawnEnt_verts(my, 12, eff_sparkMagic);

   ent_remove(my);
}
```

Damit haben wir den Effekt des Herzens zwar schön gemacht, aber der Schlitten muss auch noch verschwinden. Um jetzt nicht immer denselben Effekt zu benutzen, schreiben wir uns einen neuen Effekt, der etwas anders aussieht als der eff_sparkMagic - Effekt. Wir nennen ihn eff_sparkBright, weil er genauso funkeln soll, wie der Magic-Effekt, aber heller und leuchtender aussehen soll. Er wird eine Mischung des Puff und des Magic Effektes sein. Dem Kapitel ist der Sprite effSparkBright.tga beigelegt, den wir dafür benutzen wollen:

effect.h:

```
BMAP* eff_sparkBright_bmap;
   char* eff_sparkBright_file = "effSparkBright.tga";
```

effect.c:

```
void eff_sparkBright (PARTICLE* p)
{
   p.bmap = bmapCheck(eff_sparkBright_bmap, eff_sparkBright_file);
   p.size = 16 + random(8);

   //velocity

      vec_set(p.vel_x, vector(10 + random(10),0,0));
      vec_rotate(p.vel_x, vector(random(360), random(180), 0));

   my.alpha = 100;

   set(p, TRANSLUCENT | BRIGHT | MOVE);

   p.lifespan = 50;
   p.event = eff_sparkBright_func;
}

   void eff_sparkBright_func (PARTICLE* p)
   {
      p.alpha -= 10 * time_step;                //fade out
      p.size += 2 * time_step; //grow
      p.lifespan = (p.alpha > 0) * 100;         //survive if visible
   }
```

Wir können jetzt z.B. in der Funktion sl_death_heart folgenden Aufruf tätigen, bevor wir den Herz-Effekt aufrufen:

```
eff_spawnEnt_verts(my, 10, eff_sparkBright);
```

Damit lösen sich alle Schlitten in einem schönen Glitzer auf.

Das Tor will auch funkeln!

Von den Effekten her haben wir das Tor bisher sträflichst vernachlässigt. Wir wollen aus dem neuen Funkel-Effekt Profit schlagen und das Tor auch funkeln lassen! Damit der Effekt nicht ganz so häufig, sondern nur hin und wieder aufgerufen wird, basteln wir uns eine Funktion, die in den Entityskills des Tors einen Effektecounter laufen lässt und dann in einem Intervall diesen Effekt abspielt:

```
void obj_gate_sparkle ()
{
   if (my.skill1 <= 0) {
      eff_spawnEnt_verts(my, 2, eff_sparkBright);
      my.skill1 += 16;
   } else {
      my.skill1 -= time_step;
   }
}
```

Wenn der Skill1 - unser Zähler - abgelaufen ist, wird der Effekt abgespielt und danach der Zähler zurückgestellt. Der Zähler wird mit time_step heruntergezählt, was bedeutet, das wir in Ticks zählen. 16 Ticks entsprechen genau einer Sekunde. Damit das Tor funkelt, müssen wir die Funktion immer aufrufen, solange es auf Rudi wartet:

```
while (pk_count > obj_gate_threshold) {

   // Sparkle effect
      obj_gate_sparkle();

   wait(1);
}
```

Nachdem es geöffnet wurde, können wir nach dem reset(my, DYNAMIC); eine weitere While-Schleife einbauen, die das Tor weiter funkeln lässt:

```
while (1) {

   // Sparkle effect
      obj_gate_sparkle();

   wait(1);
}
```

Weil das Öffnen des Tores etwas besonderes ist, passt der Magic-Effekt ganz gut dazu. Wenn das Tor dann final offen ist, kann man dann jede Menge des Funkeneffekts abspielen. Die Funktion sieht nunmehr so aus:

```
void obj_gate ()
{
   // Initialization
      set(my, POLYGON);          //enable collision
      reset(my, DYNAMIC);        //not dynamic

   // Wait until last packet has been collected
      while (pk_count > obj_gate_threshold) {

         // Sparkle effect
            obj_gate_sparkle();

         wait(1);
      }

   // Effect
      eff_spawnEnt_verts(my, 64, eff_sparkMagic);

   // Open gate (animation)
      set(my, PASSABLE);
      var i;
      for (i = 0; i < 100; i += obj_gate_openAnim_speed * time_step) {
         ent_animate(my, obj_gate_openAnim_str, i, 0);
         wait(1);
      }

   // Effect
      eff_spawnEnt_verts(my, 64, eff_sparkBright);

   // It won't change anymore, so we disable dynamic
      reset(my, DYNAMIC);

   while (1) {

      // Sparkle effect
         obj_gate_sparkle();

      wait(1);
   }

}
```

Das Tor sieht nun auch ganz toll aus, wenn es geöffnet wird! - Super!

Was das Tor kann, können die Schlitten schon längst!

Wir haben das mit dem Tor wunderschön hingekriegt. Lassen wir die Schlitten doch auch funkeln! Wie für das Tor basteln wir uns auch eine Funkel-Funktion für die Schlitten:

```
void sl_sparkle ()
{
   if (my.skill1 <= 0) {
      eff_spawnEnt_verts(my, 1, eff_sparkBright);
      my.skill1 += 16 + random(32);
   } else {
```

```
      my.skill1 -= time_step;
   }
}
```

und rufen sie in der Hauptschleife der sl_main vor dem wait(1); auf. Hier wurde das Intervall höher gesetzt und außerdem wird es zufällig in die Länge gezogen. Der Grund liegt auf der Hand: es kann sein das wir ganz viele Schlitten besitzen und in diesem Fall funkelt die ganze Kette wie wild. Außerdem kann es leicht passieren, dass ein "Muster" in den Effektaufrufen zu sehen ist, wenn alle Schlitten im gleichen Intervall den Effekt abspielen. Deswegen streuen wir das Intervall ein wenig.

Zusätzlich ergibt sich ein positiver Nebeneffekt: dadurch, dass der Effektcounter beim Start des Schlittens bereits auf 0 steht, wird auch beim Erzeugen des Schlittens bereits ein Funkeln abgespielt.

Eine noch dynamischere Kamera

Im Zuge dieses Kapitel wollen wir die Kamera nun ein wenig ausbauen. Bisher haben wir die Kamera mit festen Werten gefüttert. Diese hat dann immer schön auf dieser Basis ihre Position usw. berechnet und hat sich dann ausgerichtet. Als wir die Kamera geschrieben haben, habe ich bereits erwähnt, das wir die Werte in Variablen festhalten wollen, damit wir eventuell auf dynamischere Änderungen reagieren wollen. Wie z.B. würden wir es lösen wollen, dass eine spezielle Situation andere Kamerawinkel und -offset Werte benutzen will? Wir wissen, dass die aktuelle Kamerafunktion bereits auf Variablen ausgerichtet ist, also kommt es nur noch darauf an, welche Daten nun wirklich benutzt werden. Wir wollen z.B. erreichen, dass, wenn Rudi stirbt, die Kamera etwas heranzoomed oder wenn er durch das Tor fährt, die Kamera etwas anders steht. Um das leisten zu können, müssen wir die Kamerafunktion und die -variablen verändern.

Wir wollen zunächst nur das offset und den Kamerawinkel steuerbar machen. Dazu führen wir zwei Variablen ein, in denen die Daten stehen, die die Kamera verarbeiten wird (in der cam.h platziert):

```
//- variables ------------------------------------------------------
var      cam_arc_target;
VECTOR   cam_offset_target;
```

In der Funktion cam_update würde man nun die Verarbeitung des Kamerawinkels so ändern, dass sich die Funktion den aktuellen Kamerawinkel anschaut, berechnet, wieviel sie den Winkel verändern müsste, um den Ziel-Winkel zu erreichen und interpoliert dann. Wir ersetzen die bisherige Zeile mit dieser:

```
//camera angle
   camera.arc += (cam_arc_target - camera.arc) * 0.1 * time_step;
```

Weil wir ja eigentlich von unseren Standardwerten ausgehen und solche speziellen Kameraeinstellungen weniger häufig auftreten, müssen wir nach jedem Kamera-Update die Target-Werte zurückstellen, damit - falls im nächsten Frame die speziellen Werte nicht mehr gefordert werden - sich die Kamera zu ihren Standardwerten zurückstellen kann. Wir fügen die folgenden Zeilen ganz am Ende der Funktion ein:

```
//Revert to default values
   cam_setArc(cam_std_arc);
```

Wobei die Funktion cam_setArc auch ganz einfach gehalten ist:

```
void cam_setArc (var arc)
{
   cam_arc_target = arc;
}
```

Dasselbe Verfahren benutzen wir für den offset Vektor: statt den cam_std_offset Vektor benutzen wir den cam_offset_target Vektor als Grundlage und ersetzen ihn in den ersten Zeilen der Kamera:

```
void cam_update ()
{
   if (def_camera > 0) {return;} //stop here, if debug camera is on
```

```
   //position
      vec_set(vecTemp.x, cam_offset_target);

   //(...)
```

und stellen den Vektor am Ende wieder zurück:

```
//Revert to default values
   cam_setOffset(cam_std_offset);
   cam_setArc(cam_std_arc);
```

Die Funktion cam_setOffset lautet:

```
void cam_setOffset (VECTOR* offset)
{
   vec_set(cam_offset_target, offset);
}
```

Wir können jetzt ganz einfach in speziellen Situationen die Kamera beeinflussen, was das Spielgeschehen noch dynamischer macht. Gut geeignet ist z.B. Rudi's Tod durch einen Crash. Wir definieren uns eine Funktion, die die Werte setzt und verwenden sie:

```
void pl_death_crash ()
{
   //cut sledges
      sl_cut((ENTITY*)(my.sledgeFirst));

   var i;

   //play crash animation
   for (i = 0; i < 100; i += 7 * time_step) {
      pl_death_crash_cam ();
      ent_animate(my, "crash", i, 0); //crash
      wait(1);
   }

   while (1) {
      pl_death_crash_cam ();
      wait(1);
   }
}

   void pl_death_crash_cam ()
   {
      cam_setOffset(cam_rudiDeath_offset);
      cam_setArc(cam_rudiDeath_arc);

      cam_update();
   }
```

Die Werte, die in pl_death_crash_cam übergeben werden, sind in der cam.h definiert:

```
//- rudi death preset ----------------------------------------------
var      cam_rudiDeath_arc = 70;
VECTOR*  cam_rudiDeath_offset = {x = 0; y = -500; z = 600;}
```

Wenn Rudi nun also crashedt, fährt die Kamera ganz automatisch in eine andere Stellung - toll!

Das gleiche können wir z.B. für das Tor machen. Weil das Tor jedoch sequenziell geschrieben wurde, lassen wir eine Co-Routine (eine sogenannte nebenläufige Funktion - sie läuft gleichzeitig mit der aufrufenden Funktion, wird aber eigenständig abgearbeitet und daher unabhängig vom Verlauf der aufrufenden Funktion) mitlaufen. Diese überwacht die Distanz zu Rudi und schaltet dann ab einem bestimmten Radius um:

```
void obj_gateCam ()
{
   while (!Rudi) {wait(1);}

   while (my) {
```

```
      if (vec_dist(my.x, Rudi->x) < 750) {
         cam_setOffset(cam_gate_offset);
         cam_setArc(cam_gate_arc);
      }

      wait(1);
   }
}
```

Die Werte sind so definiert:

```
//- gate cam preset ------------------------------------------------
var      cam_gate_arc = 90;
VECTOR*  cam_gate_offset = {x = -50; y = -700; z = 700;}
```

Die Funktion muss auch gestartet werden - dies tun wir einmalig am Anfang der Gate-Funktion:

```
void obj_gate ()
{
   obj_gateCam();

   //(...)
```

Die Kamera bringt das Tor nun etwas besser zur Geltung, wenn Rudi in seiner Nähe ist.

Leben verlieren und neu starten

Zum Abschluss wollen wir noch eben die Formalität umsetzen, das Rudi auch in echt die Leben verliert. Ähnlich wie bei den Punkten werden wir dafür eine überwachende Funktion in der game.c schreiben:

```
void game_watchLifes ()
{
   int original_lifes = lifes; //save the current lifes to recognize changes

   while (1) {
      if (lifes != original_lifes) {      //Recognize difference
         if (lifes >= 0) {                //we are still alive..
            if (lifes < original_lifes) { //lost one life
               deb_print("lifes -1");
            } else {
               deb_print("lifes +1");     //one new life
            }

            original_lifes = lifes;       //reset checking

         } else {                         //Rudi is dead
            lifes = 0;
            deb_print("dead");
         }
      }

      wait(1);
   }
}
```

Sobald sich die Leben ändern, merkt die Funktion das. Zur Zeit werden nur Debug-Nachrichten ausgedruckt. Wir werden hier später z.B. Soundeffekte platzieren. Die "echten" Leben ziehen wir in der Funktion pl_death_crash ab: nachdem die crash-Animation abgespielt wurde, erniedrigen wir einfach die lifes Variable:

```
//decrease lifes
   lifes--;
```

und schon verliert Rudi auch sein Leben. Wir müssen die Funktion genau wie die andere überwachende Funktion in die Initialisierung von Rudi schreiben:

```
void pl_rudi_init ()
{
   //(...)

   //call game watchers
      game_watchPoints();
      game_watchLifes();
}
```

Rudi hat zu Beginn ein paar Leben und verliert er eins, soll er im Level neu starten, also an der Stelle, bei der er das letzte Mal begonnen hat. Die Pakete müssen auch komplett neu initialisiert werden: egal ob Rudi ein paar, alle oder keine Geschenke gesammelt hat – es müssen alle wiederhergestellt werden. Das selbe Prinzip wenden wir auf das Tor auch an, denn es kann ja sein, dass Rudi alle Geschenke eingesammelt hat und dann kaputt geht - wobei das Tor schon offen ist. Wenn wir es nicht neu starten, dann würde es nach dem Neustart weiterhin offen stehen.

Das kann man auf verschiedene Arten angehen. Es liegt eigentlich auf der Hand, das wir das Level einfach neu starten, indem wir es neu laden. In der Zeichenkette level_name steht die aktuelle Leveldatei, also würde es theoretisch reichen, alle benötigten globalen Variablen zu resetten und dann level_load(level_name); durchzuführen. Im Prinzip ist das auch durchaus OK so, allerdings hat dies einen Nachteil: zunächst gibt die engine erst alle verbrauchten ressourcen frei und läd dann die Leveldatei und ihre ressourcen erneut. Bei unserer aktuellen Leveldatei ist das nicht viel, aber bei komplexen Leveldateien mit vielen Texturen, Modellen usw. können die Ladezeiten sehr leicht mehrere Sekunden betragen – für einen schnellen Restart des Levels ist das undenkbar! Wenn der Spieler nämlich startet und beispielsweise sofort gegen ein Hinderniss rennt, dann hat er vielleicht 10 Sekunden gespielt, muss aber viel länger auf den level_load warten.

Eine Alternative ist es, das Level einfach so zu belassen und nur ausgewählte Entities zurückzusetzen – wie in unserem Fall Rudi, das Tor und die Geschenke. Dazu müssen wir für jede Entity ein paar Wiederherstellungsinformationen speichern, wie z.B. die ursprüngliche Position, die Rotation und die Funktion, mit der die Entity gestartet werden soll. Diese Informationen reichen uns aus, um in unserem Spiel die Entities zurückzusetzen und da das nur ganz wenig Daten sind, wollen wir es so lösen:

Soll eine Entity wiederherstellbar sein, so ruft sie eine Funktion auf, die alle nötigen Informationen speichert. Das betrifft die Position, die Rotation und die Entity-Funktion. Diese Daten werden in Entity-Skills gespeichert. Wenn Rudi stirbt und noch genug Leben hat, wird er dann folgendes tun: er wird die Level-Variablen (Paketanzahl, etc.) zurücksetzen und dann über ein globales Flag signalisieren, dass sich die Entities wiederherstellen sollen. Die Entities sind diesbezüglich so geschrieben, dass sie auf dieses Flag reagieren: wenn also das Flag angeschaltet ist, rufen sie eine Funktion auf, die aus den zuvor gespeicherten Daten die Entity wiederherstellt. Dazu wird die Entity mit diesen Parametern geklont. Die nun wiederhergestellte „alte“ Entity wird dann entfernt. Dies geschieht in einem Frame – wenn dieser vorbei ist, stellt Rudi das Flag wieder aus und stellt sich dann schlussendlich selbt wieder her. Damit wäre der Wiederherstellungsprozess durchgeführt!

Die Wiederherstellung werden wir als Funktionen in der Funktionssammlung „sys_utils.c“ aufsetzen. Die Daten werden in Skills festgehalten und damit wir den Quellcode diesbezüglich besser verstehen, setzen wir Skill-Defines auf, die angeben wo was gespeichert wird. Wir speichern nacheinander die Position, die Rotation und den Funktionszeiger („ev“ für „event“) in den folgenden Skills:

```
#define resetData_pos   skill90  //91,92
#define resetData_rot   skill93  //94,95
#define resetData_ev    skill96
```

Wir können nun eine Funktion schreiben, die diese Daten speichert:

```
void reset_backupMe (ENTITY* myEnt, void* resetAction)
{
   if (myEnt) {
      vec_set(myEnt->resetData_pos, myEnt->x);
      vec_set(myEnt->resetData_rot, myEnt->pan);
      myEnt->resetData_ev = (void*)(resetAction);
   }
}
```

Rudt die myEnt diese Funktion auf, werden ihre Daten gesichert. Analog dazu können wir eine Funktion schreiben,

die von einer Entity aufgerufen wird, um sich wiederherzustellen:

```
void reset_restoreMe (ENTITY* myEnt)
{
   if (myEnt) {
      you = ent_create(myEnt->type, myEnt->resetData_pos, (void*)(myEnt->resetData_ev));
      vec_set(you.pan, myEnt->resetData_rot);
      ent_remove(myEnt);
   }
}
```

Die Entity wird geklont und mit den Wiederherstullungsdaten gefüttert. Das globale Flag, was wir zur Signalisierung benutzen wollen, definieren wir in der „sys_utils.h“ so:

```
var reset_global;
```

Als erstes wollen wir Rudi mit der Wiederherstellung instrumentieren. In seiner Initialisierungsfunktion sichern wir seine Daten:

```
void pl_rudi_init ()
{
   // Create reset data
      reset_backupMe(my, pl_rudi);

   //(...)
```

Die Wiederherstellung findet dann in der Funktion pl_death_crash statt, weil diese ja ausgeführt wird, wenn er ein Hinderniss rammt. Nachdem ihm ein Leben abgezogen worden ist, warten wir kurz und starten dann den Reset:

```
//(...)

wait(-1);

// Reset level

   reset_global = 1;       // Global reset indication
   lvl_reset();            // Reset level values
   wait(1);                // Wait for the resetted ent's
   reset_global = 0;       // Stop resetting
   reset_restoreMe(my);    // Reset player
```

Erst wird das Flag angeschaltet, dann wird die Funktion lvl_reset ausgeführt (womit die Variablen zurückgestellt werden). Anschließen warten wir einen frame, sodass alle Entities sich wiederherstellen. Danach wird das Flag wieder ausgeschaltet und Rudi stellt sich selbst wieder her.

Als nächstes stellen wir das Tor wieder her. In der Initialisierung stellen wir die Daten genauso wie bei Rudi sicher:

```
void obj_gate ()
{
   // Initialization
      set(my, POLYGON);          //enable collision
      reset(my, DYNAMIC);        //not dynamic

   // Create reset data
      reset_backupMe(my, obj_gate);
```

In der darauffolgenden Schleife wartet das Tor darauf, dass das letzte Paket eingesammelt wird. Dort setzen wir in die Schleife die Abfrage und die erste Wiederherstellung ein:

```
//(...)

// Wait until last packet has been collected
   while (pk_count > obj_gate_threshold) {

      // (...)

      if (reset_global) {
         reset_restoreMe(my);
```

```
            return;
         }

         wait(1);
      }
```

Das gleiche bei der Öffnung des Tores:

```
   // Open gate (animation)
      set(my, PASSABLE);
      var i;
      for (i = 0; i < 100; i += obj_gate_openAnim_speed * time_step) {
         ent_animate(my, obj_gate_openAnim_str, i, 0);

         if (reset_global) {
            reset_restoreMe(my);
            return;
         }

         wait(1);
      }
```

und auch in der letzten Schleife, in der das Tor nur offen steht und glitzert:

```
   while (1) {

      // Sparkle effect
         obj_gate_sparkle();

      if (reset_global) {
         reset_restoreMe(my);
         return;
      }

      wait(1);
   }
```

Als letztes stehen die Pakete an. Dort müssen wir nun besonders aufpassen, da die Initialisierung der Pakete und die Registrierung genau getimed ist. Während nach der Erstellung der Paket-Entities beim level_load genau nach einem Frame die Pakete gezählt waren, haben wir bei der Wiederherstellung genau einen Frame mehr, den wir warten müssen, bevor wir das Paket registrieren (dies ergibt sich aus dem Umstand, das wir die Entities manuell initialisieren und eine andere Startreihenfolge der Actions vorgeben). Desweiteren gibt es in der Hauptfunktion der Pakete genau zwei Fälle, wann ein Paket wiederhergestellt werden kann: während es darauf wartet aktiviert zu werden oder während es aktiv ist und auf das Einsammeln wartet. Wenn wir die Funktion packet diesbezüglich anpassen, sieht die Funktion nun so aus:

```
   void packet ()
   {
      // Reset assertion
         wait(1);

      // Initialization
         pk_init();

      // Create backup data
         reset_backupMe(my, packet);

      // Waiting for activation
         while (is(my, INVISIBLE)) {

            // We were invisible and inactive -> reset
               if (reset_global) {
                  reset_restoreMe(my);
                  return;
               }

            wait(1);
         }

      // Main loop - waiting for being collected
```

```
      while (!reset_global) {

         if (pk_check()) {
            break;
         }

         wait(1);
      }

   // If we were active and Rudi died -> reset
      if (reset_global) {
         reset_restoreMe(my);
         return;
      }

   // We "die"
      pk_remove();
}
```

Jetzt haben wir ein spezielles Problem: vorher haben wir in der Funktion pk_remove die Entity über eine externe Funktion auf my = 0 gestellt und haben dann über einen Trick die Entity dazu gebracht sich dann doch noch selbst zu löschen. Allerdings dürfen wir die Entity nicht löschen, wenn wir sie wiederherstellen wollen, da die Entity die Informationen dafür speichert. Also müssen wir pk_remove umschreiben, sodass das Paket doch nicht gelöscht wird und nur darauf wartet, wiederhergestellt zu werden.

Wir lösen das, indem wir die Zeilen unseres ehemaligen Tricks wahlweise löschen oder auskommentieren:

```
void pk_remove()
{
   //(...)

   //save my pointer
      //ENTITY* mySave = my;

   //(...)

   //remove entity
      //ent_remove(mySave);
}
```

und in der Funktion sl_addNr das „my = 0“ – Schalten genauso auskommentieren. Nun existiert das Paket weiter, was allerdings schlecht ist, denn wir haben es ja gesammelt! Deshalb machen wir es unsichtbar und warten, bis das Signal zur Wiederherstellung kommt:

```
void pk_remove()
{
   //(...)

   // Stay alive to be reset-enabled

      set(my, INVISIBLE);

      while (1) {

         // We are collected -> reset
            if (reset_global) {
               reset_restoreMe(my);
            }

         wait(1);
      }
}
```

Nun sind Rudi, das Tor und die Pakete wiederherstellbar und der Level-Reset wird ordnungsgemäß durchgeführt! Wenn Rudi keine Leben mehr hat, folgt der Game Over Bildschirm, aber den schreiben wir erst später.

Kapitel 9: Das Menü

Die Verpackung unseres Spiels

Bisher haben wir das Spiel immer gestartet und sind direkt im Level angefangen. Das ist für das Testing ganz nett, aber das fertige Spiel muss auch eine Art "Drumherum" bieten. Damit ist in erster Linie das Menü gemeint. Von dort aus kann der Spieler das Spiel starten, beenden und konfigurieren. Vor dem Menü ist in der Regel auch eine Reihe von Splashscreens geschaltet. Manchmal werden dort auch Splash-Videos angezeigt. Diese Screens und Videos sind manchmal aus rechtlichen Gründen zu platzieren (z.B. wenn man das Engine-Logo anzeigen muss), weisen aber meistens darauf hin, wer das Spiel gemacht hat usw. Manchmal wird ein Video vor dem Menü gezeigt, das wie eine Art Intro oder Teaser des Spiels selbst gestaltet ist. Danach gelangt der Spieler in das Menü.

Hat der Spieler das Spiel geschafft und komplett durchgespielt, gehört auch ein Abschluss zum Repertoire eines Spiels. In aller Regel gibt es dann einen Abspann wie bei einem Kinofilm, bei dem alle beteiligten Personen an der Entwicklung des Spieles aufgelistet werden. Um den Spiel eine persönliche Note zu geben, gibt es dann nach dem Abspann eine Einblendung eines Dankeschön-Bildschirms mit einem netten "Thanks for playing!" und einer netten Grafik, mit der sich die Entwickler bedanken, dass der Spieler ihr Spiel gespielt haben. Danach startet das Spiel neu oder geht zurück ins Menü. Es kann auch sein, dass der Spieler dass Spiel nicht schafft indem er alle Leben verliert. Dann zeigt man ein Game Over Screen an, der dann bei Tastendruck wieder ins Menü zurückführt.

Während man spielt, gibt es außerdem die Möglichkeit, das Spiel zu pausieren und es zu beenden. Entweder kann der Spieler es komplett beenden oder nur abbrechen und in das Menü zurückkehren. Dafür wird es ein kleines Menü ("ingame Menü") geben, dass auftaucht, wenn man die ESC Taste drückt. Wenn der Spieler die Taste P oder die Pause Taste auf der Tastatur drückt, wird das Spiel (genauso wie beim ingame Menü) angehalten und ein Pausesymbol taucht auf. Beim Druck auf die ESC, P oder Pause - Taste kehrt der Spieler dann wieder ins Spiel zurück.

Wir wollen uns zunächst mit dem Anfang des Spiels, also der Splashscreen-Sequenz und dem Hauptmenü befassen.

Laden der Splashscreen-Daten

Im Ordner game/splash befinden sich 3 Grafiken „splashA.tga“ bis „splashC.tga“. Sie sollen in dieser Reihenfolge abgespielt werden. Auf den Splash screens stehen informative Dinge, wie z.B. die verwendete engine (falls dies angezeigt werden muss) und weitere Infos, wie z.B. wer das Spiel gemacht hat, usw. Die Splash-Screen Sequenz wollen wir in einem eigenen Modul programmieren und legen daher im "game" Verzeichnis die Dateien splash.c und splash.h an und inkludieren die beiden neuen Dateien.

Bevor wir damit anfangen, die Screens irgendwie anzuzeigen, müssen wir dafür sorgen, dass sie geladen sind. Daher müssen wir natürlich auch schauen, wie wir die Daten verfügbar machen. Die Splashscreens werden aneinandergereiht dargestellt, daher bietet es sich folgende Datenstruktur an:

```
//-----------------------------------------------------------------------
// defines
//-----------------------------------------------------------------------

   #define SPLASH_SCREENS_CNT 3

//-----------------------------------------------------------------------
// variables
//-----------------------------------------------------------------------

   BMAP* splash_bmaps[SPLASH_SCREENS_CNT];
   PANEL* splashPanel;
```

Zunächst definieren wir mit SPLASH_SCREENS_CNT die Anzahl der Screens. Dann definieren wir uns ein Array von Bitmap-Zeigern, in das wir später die Grafiken laden werden. Die Größe des Arrays - und damit die Anzahl die Anzahl der Bilder - wird durch SPLASH_SCREENS_CNT angegeben, also in diesem Fall 3 Bilder. Wir können dann

später ganz einfach durch das Array "laufen" und immer den aktuellen Splashscreen holen, um alle hintereinander anzuzeigen. Wir werden später sehen, warum das so praktisch ist.

Wir brauchen außerdem nur ein Panel dafür. Es wäre Quatsch, für alle Splashscreens jeweils ein Panel zu erzeugen (außer man möchte mit speziellen Übergangseffekten arbeiten) - deshalb werden wir nur ein Panel benötigen (splashPanel).

Damit die Ressourcen auch geladen werden, schreiben wir uns in der splash.c eine Initialisierungsfunktion, die die Daten läd. Wir haben mit dem define SPLASH_SCREENS_CNT eine echt tolle Möglichkeit, halbwegs dynamisch eine unbegrenzte Zahl an Splashscreens zu laden. Das Muster der Dateinamen ist auch recht einfach gestrickt: dem Wort "splash" folgt ein Buchstabe, beginnend bei "A", danach "B" etc. mit der aschließenden Dateierweiterung. Wir können das in einem schönen iterativen Algorithmus einbetten, der sich jedes Mal den Dateinamen generiert, anstatt statische Dateinamen zu benutzen:

```
void splash_init ()
{
   // Load bitmap ressources
      int i; char buffer [128];
      for (i=0; i < SPLASH_SCREENS_CNT; i++) {
         sprintf(buffer, "splash%c.tga", 'A'+i);
         splash_bmaps[i] = bmap_create(buffer);
      }

   // Create panel for the splashes
      splashPanel = pan_create("", 999);
}
```

Die for-Schleife geht alle Zahlen von 0 bis 2 durch und erzeugt sich mit sprintf einen String, der den Dateinamen angibt. das %c ist ein Formatierungszeichen, das angibt, das im String dort ein char-zeichen eingesetzt wird. Das 'A'+i ist dieser Buchstabe: es wird das A genommen und mit + i wird im Alphabet weitergesprungen. Bie i=0 ist das dann immer noch 'A', bei i = 1 ist das aber schon ein 'B', usw. Der Hintergrund ist der, dass man ASCII Zeichen auch als Zahlen interpretieren kann. Die Großbuchstaben stehen im ASCII Code alle hintereinander und man kann Zeichen auch mit Zahloperationen benutzen. Das entspricht z.B. der Zahl 65. Wenn wir darauf jetzt eine 1 addieren, wird daraus eine 66. Da dieser Wert als Char-Zeichen in den String eingefügt wird, wird diese Zahl wieder in ein Zeichen umgewandelt. Die 66 entspricht dann in diesem Beispiel dem 'B'.

Wenn wir also SPLASH_SCREENS_CNT z.B. mit 6 definieren würden, würde die letzte Datei "splashF.tga" lauten. Das interessante ist aber die Art und Weise wie der Code darauf getrimmt ist. Immer dann wenn man Dateien algorithmisch laden kann, sollte man dies auch tun, weil man dadurch den Code vereinfach und dynamisch anpassbar machen kann!

Im zweiten Schritt wird hier ein Panel erzeugt, aber mehr passiert auch nicht. Das wollen wir jetzt ändern! Wir wollen eine einfach Funktion aufrufen, die uns die Splashscreens anzeigt. Einfacherweise wird diese die Splashscreens auch laden. Die Funktion lautet vorerst so:

```
void splash_show ()
{
   splash_init ();   //load all ressources
}
```

Danach würde der Code kommen, der das Abspielverhalten der Screens steuert (das machen wir gleich erst). In der game.c in der Funktion game steht bisher der loading-Code für das erste Level. Das haben wir dort nur provisorisch geschrieben, wir ersetzen das jetzt durch das Aufrufen der Splashscreens.

```
void game ()
{
   // Game initialization
      game_init();

   // Splashscreens
      splash_show();
}
```

Die Splashscreen-Sequenz

Wir werden jetzt die Funktion splash_show erweitern, sodass uns die Splashscreens auch angezeigt werden. Als erstes fügen wir ein wait(3); nach splash_init(); ein. Diese Dreifachpufferung (englisch: triple buffering) beschreibt ein Konzept in der Computergrafik, bei dem bei gleichzeitiger Verwendung von VSync (vertikale Synchronisation) und Doppelpufferung (double buffering) auftretenden Nachteile während des Bildaufbaus kompensiert werden. In unserem Beispiel bedeutet dies, dass wenn die engine gestartet wird und die Bitmaps geladen werden, wir dies abwarten und dann die Splashscreens anzeigen. Ohne diesen Einschub von extra Frames wäre die Pause (das wait(-3); im weiteren Verlauf des Codes) vom ersten zum zweiten Splashscreen kürzer, weil die Ladezeit von den 3 Sekunden abgezogen werden würde (kann man ganz einfach testen indem man das wait(3); entfernt).

Als nächstes folgt dann schon das Anzeigen der Screens. Dies wird wieder algorithmisch gelöst, indem wir wieder mit einer for-Schleife über die Splashscreens "laufen":

```
void splash_show ()
{
   splash_init ();   //load all ressources

   wait(3);          //tripple buffering

   // Show splashscreens

      int i;
      for (i=0; i < SPLASH_SCREENS_CNT; i++) {

         set(splashPanel, VISIBLE);
         splashPanel->bmap = splash_bmaps[i];

         wait(-3);
      }
}
```

Es wird jedes Mal das Panel sichtbar geschaltet (zumindestens für das erste Mal) und danach wird die Bitmap gesetzt. Die Laufvariable i wird immer um 1 erhöht ("inkrementiert"), sodass wir mit i in der Splashscreen-Sequenz immer parallel die entsprechende Bitmap zur Verfügung haben. Darauf folgt die Pause bis zum nächsten Screen, in diesem Fall sind es 3 Sekunden. Damit wir nicht daran gebunden sind, jedem Splashscreen eine fixe Anzeigedauer zu geben, definieren wir uns ein Array, dass diese Zeiten speichert:

```
var splash_screen_time [] = {4, 5, 6}; // Seconds
```

Auf dass wir dann anstatt mit der wait(-3); Anweisung mit

```
wait(-splash_screen_time[i]);
```

darauf zugreifen. Nachdem also die Splashscreens angezeigt wurden, wollen wir die Ressourcen wieder entfernen, um nicht unnötig Speicherplatz zu belegen. Wir erzeugen uns also eine Funktion namens splash_free, die den benutzen speichern wieder freigibt:

```
void splash_free ()
{
   // Make panel and current bitmap invisible
      reset(splashPanel, VISIBLE);

   // Remove splashscreens
      int i;
      for (i=0; i < SPLASH_SCREENS_CNT; i++) {
         bmap_remove(splash_bmaps[i]);
      }

   // Remove Panel
      pan_remove(splashPanel);
}
```

Erst wird das Panel weggeblendet. Das ist wichtig, weil wenn eine Bitmap noch angezeigt und dann entfernt wird,

es zu einem schweren Fehler kommt. Dann entfernen wir die Bitmaps, indem wir das Array durchlaufen und bmap_remove auf jede Bitmap anwenden. Schlussendlich wird auch noch das Panel entfernt. Damit dies getan wird, fügen wir den Aufruf von splash_free analog zu splash_init ganz am Ende der splash_show Funktion ein.

Ein Teaser-Video

Nachdem die Splashsscreens durchlaufen sind, wollen wir nun ein kurzes Video, was wie ein "Trailer" wirkt, anzeigen.

Videos kann mit den media-Befehlen abspielen. Videos und Musikdateien, die mit den media-Befehlen abgespielt werden, können auf dem Bildschirm, auf einer Textur oder einem Panel wiedergegeben werden. Die Dateien werden gestreamed, was bedeutet, dass die Datei nicht komplett in den Speicher geladen wird, sondern immer nur ein kleines Stück davon im Speicher ist. Dies ist sehr speicherschonend und ideal für sehr große Medien-Dateien. Alle Dateitypen, die mit dem Windows-Media-Player abgespielt werden können, können auch mit den media-Anweisungen abgespielt werden, wie z.B. AVI oder MPG Dateien. Es gibt aber auch Einschränkungen: erstens, muss der PC, auf dem die Datei abgespielt wird, dies auch können. Wenn man z.B. ein Video mit dem allerneuesten Video Codec (z.B. DivX 6.7, XVid 1.1.3, WMV 10) abspielt und der PC des Spielers diese nicht installiert hat, kann das Video NICHT abgespielt werden! Ein weiteres Problem ist, dass gestreamedte Dateien nicht in Ressourcen-Dateien eingebettet werden können. Das 3D Gamestudio kann WRS Ressourcen handhaben (erstellen aber nur mit der Pro-Version) und daraus Levels, Sounds, Texturen, Modelle usw. lesen, aber keine Dateien daraus streamen. Daher müssen diese Dateien "offen" im Verzeichnis des Spiels herumliegen.

Diesem Kapitel ist ein Video beigelegt: "splashRudi.wmv". Es ist im Windows Media Video - Format gespeichert. Dazu seien ein paar Sachen gesagt: das WMV Format ist ein Videoformat, das von Microsoft im Zuge der Entwicklung des Windows Media Players entwickelt wurde und hat daher eine sehr breite Verbreitung. WMV Dateien komprimieren recht stark, bieten eine vergleichsweise gute Grafikqualität und sind für das Streaming besonders gut geeignet. Eine andere Frage bei solchen Sachen ist auch die Frage, ob man einen bestimmten Video Codec überhaupt "so" benutzen darf. Vor allem ist diese Frage kritisch bei kommerziellen Spielen. Das WMV Format darf man frei verwenden. Andere Videoalternativen sind Ur-Alt Codecs wie z.B. MPEG-2, die zwar eine ausreichende Videoqualität bieten bei vergleichsweise hohem Speicherplatzverbrauch.

Wir werden im work Ordner ein neues Verzeichnis erstellen, das wir "media" nennen und speichern dort die Videodatei ab. In diesen Media-Ordner werden wir alle Dateien ablegen, die wir im Spiel streamen werden.

Wir können mit media_play eine Datei abspielen. Die Syntax des Befehls sieht so aus:

```
media_play(STRING* name, BMAP* target, var volume);
```

"name" gibt den relativen Pfad zur Datei und "volume" die Abspiellautstärke an. "target" gibt das "Ziel" der Datei an. Wie bereits gesagt, kann man Videos auch auf Texturen abspielen (z.B. ein Video auf einem Überwachnungsbildschirm oder sowas). Wenn man ein Video abspielt und NULL übergibt, wird das Video im fullscreen Modus abgespielt. Dies funktioniert nicht bei allen codecs und ist daher eine unsicher Option. Zur Sicherheit benutzt man ein bitmap-Target, das z.B. in einem Panel eingebettet ist (= das Video wird auf einem panel abgespielt). Dazu definieren wir in der splash.h eine Bitmapreferenz:

```
BMAP* splash_video;
```

und initialisieren diese in der init Funktion:

```
void splash_init ()
{
   //(...)

   // Create video target
      splash_video = bmap_createblack(640, 480, 24);

   //(...)
}
```

Die Bitmap ist 640x480 Pixel groß, weil das Video so groß ist. Wir können das Video nun abspielen, indem wir nach den Splashscreens (und vor splash_free!!!) das Panel für das Video vorbereiten und dann das Video darauf abspielen:

```
// Show Video

   // Prepare target bitmap
      splashPanel->bmap = splash_video;

      // Scale to fit fullscreen
         splashPanel->scale_x = screen_size.x / bmap_width(splash_video);
         splashPanel->scale_y = screen_size.y / bmap_height(splash_video);

      set(splashPanel, FILTER); //no unfiltered playback

   // Play video and wait until it's done

      var videoHandle = media_play(splash_video_file, splashPanel->bmap, 100);
      while (media_playing(videoHandle)) {wait(1);}
```

Zunächst weisen wir dem Panel die Video-Bitmap zu. Daraufhin skalieren wir das Panel, sodass es den gesamten Bildschirm abdeckt. Weil das Panel skaliert wird, kann das Vide pixelig aussehen. Deshalb stellen wir das filtering des Panels an. Danach spielen wir das Video ab und speichern ein sogenanntes handle dafür ab. Mit diesem handle können wir das Abspielverhalten des Videos steuern. Wir warten dann mit einer While-Schleife das Ende des Videos ab, denn wenn ein media-Stream nicht mehr aktiv ist, liefert media_playing eine 0 zurück, ansonsten bei Aktivität eine 1. Das bedeutet, dass die While-Schleife eben solange die Funktion anhält, bis das Video vorbei ist. Das Argument splash_video_file in media_play gibt den Pfad zur Datei an und ist so in der splash.h definiert:

```
char* splash_video_file = "media\\splashRudi.wmv";
```

Media-Dateien werden ausgehend vom Spielverzeichnis gesucht. mit "media\\" steigen wir in den Media-Ordner ein, haben die Datei dann über "plashRudi.wmv" gefunden und spielen sie ab.

Damit das erzeugte Bitmap-Target später keinen Speicher belegt, entfernen wir es in der Funktion splash_free:

```
// Remove video target
   bmap_remove(splash_video);
```

Die Splash-Sequenz überwachen und steuern

Es gibt nun zwei störende Dinge an dem bisheringen Code, die nicht sofort auffallen. Zum einen kann der Spieler die Splash-Sequenz nicht abbrechen. Er muss sich alle Splashscreens und das Video komplett angucken - das ist sehr schlecht! Zum anderen wissen wir intern nicht, wann die Splashsequenz vorbei ist. Zwar wissen wir das innerhalb der splash_show Funktion, aber wir wissen es nicht außerhalb. Wir wollen nämlich nach den Splashscreens das Menü aufrufen und wollen das nicht in der Funktion splash_show machen, weil dies eine unabhängig funktionierende Funktion bleiben soll.

Eine Funktionsüberwachung kann man auf mehrere Arten lösen. Wir wollen eine globale Flag Variable dafür benutzen. Wir führen in der splash.h eine Variable ein:

```
var splash_ready;
```

diese stellen wir am Anfang auf 0 zurück und wenn wir fertig sind, stellen wir sie auf 1, um auszudrücken, das die Splashsequenz zuende ist:

```
void splash_show ()
{
   splash_init ();   //load all ressources
   wait(3);          //tripple buffering

   splash_ready = 0; //reset global flag

   //(...)
```

```
   splash_free();    //remove ressources
   splash_ready = 1; //finished. Activate association flag
}
```

Wir können in der Funktion game nun eine wartende While-Schleife einfügen:

```
void game ()
{
   // Game initialization
      game_init();

   // Splashscreens
      splash_show();
      while (!splash_ready) {wait(1);}

   deb_print("finished!");
}
```

Das Ausrufezeichen vor splash_ready ist ein Bit-Operator, der den Wert umdreht: wenn also splash_ready auf 0 steht (also die Splashscreens noch laufen), steht dann da 1 und solange der Wert in der Schleifenbedingung steht, läuft die Schleife. Wenn splash_ready auf 1 steht (Splashscreens sind zuende), steht in der Bedingung 0 und die Schleife bricht ab. Anschließend drucken wir "finished!" aus, um zu überprüfen, ob wir die Splashscreen-Sequenz auch wirklich richtig überwacht haben. Nach einem Test stellt sich heraus, dass alles stimmt und wir die Sequenz richtig überprüft haben.

Für das Abbrechen der Bildfolge und des Videos müssen wir uns beide Dinge getrennt anschauen und überlegen, wie wir die Tasteneingabe am besten abfangen. Fangen wir zunächst mit der Bildfolge an.

Das Problem steckt im Detail: wir benutzen die wait(-3); Anweisung, um 3 Sekunden zu warten und dann das nächste Bild anzuzeigen. Wir würden jetzt gerne in diesen 3 Sekunden die Tasteneingabe abfangen, aber das geht ja nicht, weil die Funktion bei diesem wait(-3); eingefroren wird. Daher können wir nichts abfangen. Wir müssten schon in jedem frame checken, ob der Spieler abbrechen will. Dafür bauen wir das wait(-3); in eine frame-basierende Warteschleife um:

```
int i;
var timer;
for (i=0; i < SPLASH_SCREENS_CNT; i++) {

   set(splashPanel, VISIBLE);
   splashPanel->bmap = splash_bmaps[i];

   // Wait for next splashscreen
      timer = splash_screen_time[i] * 16;
      while (timer > 0) {
         timer -= time_step;
         wait(1);
      }
}
```

Nach jedem Bild wird ein timer initialisiert, der die Anzahl der Ticks beinhaltet, die man warten soll. Ticks ist eine Zeiteinheit in der engine und 16 Ticks entsprechen einer Sekunde (deshalb multiplizieren wir den Wert mit 16). In einer While-Schleife zählen wir den timer dann mit time_step runter. Die Schleife bleibt solange aktiv - und hält damit die umhüllende For-Schleife an - wie der timer größer 0 bleibt. Wir haben das ehemalige wait(-3); nun frame-basiert umgesetzt.

In diese wartende Schleife bauen wir jetzt den Abfang-Code ein:

```
// Skipping
if (key_any) {                //user wants to skip
   while (key_any) {wait(1);} //wait until key is released
   i = SPLASH_SCREENS_CNT;    //skip splashscreens
   break;                     //stop now
}
```

Hierbei wird in die Verzweigung eingestiegen, wenn der Spieler irgendeine Taste drückt. Dann wird gewartet, bis

er die Taste wieder loslässt. Dieses Verhalten ist ganz wichtig! Denn wenn wir beim Video wieder mit key_any eine Eingabe einfangen und der Spieler die Taste noch gedrückt hält, wird das Video auch automatisch übersprungen - deshalb müssen wir warten. Danach wird i so gesetzt, dass die For-Schleife auf jeden Fall abbricht, danach brechen wir mit break die wartende While-Schleife ab. Wenn der Spieler also eine Taste drückt, überspringt er die Splashscreens und schaut sich dann direkt das Video an.

Beim Abspielen des Videos haben wir bereits eine While-Schleife, die wartet, bis das Video fertig ist. Dort bauen wir den Code auf ähnliche Weise ein:

```
var videoHandle = media_play(splash_video_file, splashPanel->bmap, 100);
while (media_playing(videoHandle)) {

   // Skipping
   if (key_any) {                //user wants to skip
      while (key_any) {wait(1);} //wait until key is released
      media_stop(videoHandle);   //stop video
      break;                     //stop now
   }

   wait(1);
}
```

Hier wird auch wieder die Tasteneingabe abgefragt. Wurde die Eingabe getätigt wird zuächst das Video mit media_stop abgebrochen. Mit break verlassen wir die umhüllende While-Schleife. Wenn wir das Video nicht abbrechen würden, dann würde es einen schweren Fehler geben, weil wir in der Funktion splash_free das Bitmap-Target des Videos entfernen.

Damit ist die Splashsequenz komplett programmiert!

Das Spielmenü - eine Frage des Designs

Jedes Spiel hat ein Spielmenü. Der Spieler kann dort einen Spielmodus anwählen und starten, das Spiel einstellen und es auch beenden. Das Design eines Spielmenüs beschäftigt in großen Spieleproduktionen eigens dafür abgestellte Mitarbeiter. Diese kennen sich mit Grafikdesign und Softwareergonmie aus. Schließlich soll das Menü grafisch anspruchsvoll sein, nicht langweilen oder den Spieler vergraulen und dennoch alle nötigen Informationen und Navigationsschritte so einfach und schnell wie möglich anbieten. Es gibt zum Teil Spiele, da ist ein einziger Programmierer während der ganzen Produktion damit beschäftigt, nur das Spielmenü zu bauen! Zudem treffen viele Designs und Herangehensweisen aufeinander: mache ich ein konventionelles 2D Menü? Arbeite ich mit tollen Übergangseffekten? Mache ich das Menü nicht vielleicht doch in 3D? Denke ich mir eine ganz neue Weise aus, das Menü aufzubauen oder kupfere ich irgendwo ab? Wie aufwendig soll ich die Gestaltung machen und wieviele Menübildschirme soll ich bauen? Welche Funktionen bietet das Menü und seine Untermenüs? Und so weiter und so fort.

Wir entwickeln hier nur ein kleines Spiel und wollen daher auch kein monströses Menü bauen. Zudem würde dann auch die Ladezeit steigen, wenn wir viel mehr Daten laden als nötig (für eine 3D Szene zum Beispiel). Wir wollen ein simples 2D Menü mit einer Hintergrundgrafik mit dem Spiellogo und den Navigationspunkten. Der User soll das Spiel starten, das Spiel konfigurieren und das Spiel verlassen können. Zusätzlich kann der User auch auf eine Schaltfläche klicken, um sich die Credits anzuschauen. Das Menü soll hinterher so aussehen:

Für das Menü existiert im game-Ordner ein eigener Unterordner namens "menu": dort speichern wir alle Daten, die mit dem Menü in Verbindung stehen. Dort befinden sich alle Grafiken, die wir für unser Menü und alle anderen (Unter-)Menüs verwenden werden. Natürlich ist es dem Leser (wie bei allen anderen Grafiken/Modellen auch) freigestellt, eigene Sachen einzubauen - die vorgefertigten Sachen erleichtern aber die Entwicklung eines Menü-Designs und man kann sich hier jetzt völlig auf die restlichen - und damit das Spiel auch was wird - wichtigeren Sachen konzentrieren. Die Dateien für das primäre Hauptmenü sind:

menu_background.tga	*der Hintergrund*
menu_cursor.tga	*der Mauszeiger, den wir benutzen werden*

Die button-Grafiken haben ein _n und _t Anhang im Dateinamen. Dabei steht _n für "normal" und _t für "touched". Die "touched" Grafiken sollen angezeigt werden, wenn der Spieler mit dem Mauszeiger auf einem solchen Button verweilt.

menu_startgame_n(_t).tga	*Button zum Starten des Spiels*
menu_config_n(_t).tga	*Button für die Konfiguration*
menu_credits_n(_t).tga	*Button zum Anschauen der Credits*
menu_exit_n(_t).tga	*Button zum Verlassen des Spiels*

Das Menü soll dann mal so aussehen:

Für alle Menüs werden wir die Dateien menu.c und menu.h erzeugen und inkludieren. Wir werden alle Bitmaps permanent in den Speicher laden, weil im Gegensatz zu den Splashscreens das Menü beliebig oft zu sehen ist. Für die einzelnen Panels und die cursor-Bitmap erzeugen wir in der menu.h die folgenden Referenzen:

```
PANEL* menu_background;
PANEL* menu_startGame;
PANEL* menu_config;
PANEL* menu_credits;
PANEL* menu_exit;

BMAP* menu_cursorBmap;
```

Wir wollen uns eine Init-Funktion schreiben, die die Daten läd. Diese Funktion wird die Bitmaps laden und alle Panels erzeugen. Die Panels, auf die man klicken soll, erhalten einen button, der auf die geladenen Grafiken referenziert. Die button erhalten zunächst keine Funktion zugewiesen, das machen wir später. Wir benutzen u.a. die pan_createEx Funktion, um mit möglichst wenig Aufwand die panels zu erzeugen:

```
void menu_init ()
{
   // GENERAL

      menu_cursorBmap = bmap_create("menu_cursor.tga");

   // MAIN MENU

      menu_background = pan_createEx("", 100, "menu_background.tga", 0, 0, 0);

      menu_startGame = pan_createEx("button(0,0, menu_startGame_n.tga, menu_startGame_n.tga,
                       menu_startGame_t.tga, menu_main_startGame, NULL, NULL);",
                       101, NULL, 730, 403, 0);
```

```
	menu_config = pan_createEx("button(0,0, menu_config_n.tga, menu_config_n.tga,
	              menu_config_t.tga, NULL, NULL, NULL);", 101, NULL, 658, 452, 0);

	menu_credits = pan_createEx("button(0,0, menu_credits_n.tga, menu_credits_n.tga,
	               menu_credits_t.tga, NULL, NULL, NULL);", 101, NULL, 27, 713, 0);

	menu_exit = pan_createEx("button(0,0, menu_exit_n.tga, menu_exit_n.tga, menu_exit_t.tga,
	            menu_main_exit, NULL, NULL);", 101, NULL, 737, 561, 0);
}
```

Die ganzen Pixelwerte resultieren aus dem Designprozess des Menüs, der hier nicht behandelt worden ist (s.o.). Wir setzen hier für die Panels kein VISIBLE flag, weil dies das Menü in eigenen Unterfunktionen machen soll, je nachdem was das Menü (oder Untermenüs) für Grafiken braucht). Wir geben außerdem in dem content-String den direkten Dateinamen der Grafik an, sodass wir nicht extra die Grafiken mühsam in irgendwelche Bitmapreferenzen laden müssen. Wir brauchen auch keine Angst haben, das Grafiken doppelt im Speicher lagern: die engine überprüft beim Laden einer Grafik, ob sie nicht schon irgendwo anders verwendet wird. Wenn dies nicht der Fall ist, wird sie erst geladen.

Damit das Menü auch initialisiert wird, betten wir es in eine Art Menü-Hauptfunktion ein:

```
void menu_main ()
{
   menu_init();
}
```

und rufen diese in der Funktion game nach den Splashscreens auf:

```
void game ()
{
   // Game initialization
      game_init();

   // Splashscreens
      splash_show();
      while (!splash_ready) {wait(1);}

   // Menu
      menu_main();
}
```

Das Hauptmenü

Das Hauptmenü hat - wie bereits gesagt - verschiedene Zustände. Einmal kann es das "richtige" Hauptmenü sein, dann kann es das Konfigurationsmenü des Hauptmenüs sein, dann kann es passieren, dass ein ingame Menü angezeigt werden soll. Außerdem wollen wir den Thanks for playing! und den Game Over Bildschirms auch als Bestandteil des Menüs ansehen. Deshalb ist es gut, erstmal diese verschiedenen Arten festzuhalten und jeden Modus durch eine Zahl zu kennzeichnen. Dazu definieren wir in der menu.h erstmal nur für das eigentliche Hauptmenü:

```
var menu_mode;

   #define MENU_MODE_NONE     0
   #define MENU_MODE_MAIN     1
```

Die anderen Modi werden später noch hinzugefügt.

Nun haben wir ja bereits die Funktion menu_main geschrieben, die bereits das Menü initialisiert. Wir könnten dort nun die Modus-Nummer auswerten und dementsprechend eine Frame-Funktion aufrufen, die das aktuelle "Menü" anzeigt und kontrolliert. Dazu können wir einen switch-case-tree in die Funktion schreiben:

```
void menu_main ()
{
   menu_init();
```

```
   while (1) {

      switch (menu_mode) {
         case MENU_MODE_NONE: break;
         case MENU_MODE_MAIN: menu_main_show(); break;
      }

      wait(1);
   }
}
```

Wenn kein eindeutiger Modus angegeben ist, wird nichts ausgeführt. Wenn das Hauptmenü gefordert ist, wird die Funktion menu_main_show ausgeführt. Damit zu Beginn dies auch getan wird (also nach den Splashscreens), könnten wir in der Initialisierung dies so einstellen:

```
void menu_init ()
{
   //(...)

   menu_mode = MENU_MODE_MAIN;
}
```

Welche Aufgabe hat nun die Funktion menu_main_show? Im Prinzip muss die Funktion nur dafür sorgen, dass alle unnötigen Grafiken des Menüs ausgeblendet werden und alle wichtige Grafiken angezeigt werden. Wir brauchen die Maus, also muss auch die Maus angezeigt werden. Außerdem wird das HUD angezeigt, also müssen wir das auch ausblenden. Das wäre eigentlich schon alles! Die Funktion sieht so aus:

```
void menu_main_show ()
{
   hud_hide();             //hide HUD
   menu_hide_all();        //hide all elements

   menu_main_elements();   //show selected elements

   menu_cursor();          //draw cursor
}
```

Witzigerweise sind alle Aufgaben in andere Funktionen gekapselt. Der Grund dafür liegt auf der Hand: wenn wir andere Modi als Funktionen definieren, werden sehr wahrscheinlich oft Funktionsaufrufe wiederholen - also wäre es nur redundant alle Panel-Anweisungen usw. immer explizit hinzuschreiben. Wir gehen die Funktionen nun nacheinander durch.

Die Funktion hud_hide schreiben wir natürlich nicht in die Menü-Dateien, sondern in die Datei hud.c:

```
void hud_hide ()
{
   reset(hud_heart, VISIBLE);
   reset(hud_lifes, VISIBLE);
}
```

Damit wir später das HUD auch wieder anschalten können, schreiben wir jetzt schonmal eine entsprechende Funktion:

```
void hud_show ()
{
   set(hud_heart, VISIBLE);
   set(hud_lifes, VISIBLE);
}
```

Die Funktion menu_hide_all (jetzt bewegen wir uns wieder in der Datei menu.c) ist so gedacht, dass sie ALLE Menüelemente (wirklich alle!) ausschaltet. Das Prinzip ist nämlich so, das wir zunächst alles ausstellen und dann nur ein paar wieder anschalten. Dazu gehört auch, die Maus auszuschalten! Denn wenn wir ein Menü haben, was keine Maus erfordert (und wir hatten die gerade eben noch an), dann müssen wir die natürlich ausschalten:

```
void menu_hide_all ()
{
   reset(menu_background, VISIBLE);
```

```
   reset(menu_startGame, VISIBLE);
   reset(menu_config, VISIBLE);
   reset(menu_credits, VISIBLE);
   reset(menu_exit, VISIBLE);

   mouse_mode = 0;
}
```

Die Funktion menu_main_elements ist jetzt speziell für die Funktion menu_main_show konzipiert: sie zeigt nur die Elemente an, die wir brauchen:

```
void menu_main_elements ()
{
   set(menu_background, VISIBLE);
   set(menu_startGame, VISIBLE);
   set(menu_config, VISIBLE);
   set(menu_credits, VISIBLE);
   set(menu_exit, VISIBLE);
}
```

Witzigerweise sind das genau dieselben Elemente, die wir auch in menu_hide_all ausschalten - aber das wird sich im Verlauf der Menüentwicklung noch ändern ;)

Die Funktion menu_cursor soll den Mauszeiger zeichnen. Das geht ganz einfach: wir weisen einfach die geladene Bitmap zu und stellen den mouse_mode auf 4. Das bewirkt, dass der Mauszeiger in der engine mit der Bewegung der Maus automatisch synchronisiert wird:

```
void menu_cursor ()
{
   mouse_map = menu_cursorBmap;
   mouse_mode = 4;
}
```

Wenn wir nun das Spiel starten, gelangen wir in das Hauptmenü und wenn alles geklappt hat, werden auch die alternativen Bitmaps angezeigt, wenn wir mit dem Zeiger über den Buttons sind - super!

Das Spiel starten und beenden

Buttons können bis zu 3 Funktionszeiger für Interaktionen ausführen. Unter anderem gilt das für einen Klick auf einen Button - was für uns im Moment am interessantesten ist. Wir wollen als erste Menüfunktionalität die Verbindung zum eigentlichen Spiel herstellen und das Spiel beendbar machen. Die Funktion zum beenden ist super simpel:

```
void menu_main_exit ()
{
   sys_exit("");
}
```

Die Funktion zum starten des Spiels muss etwas mehr leisten: zunächst muss das Menü ausgeblendet werden, das Menü überhaupt ausgeschaltet werden und das Level geladen werden. Den Level-Reset dürfen wir auch nicht vergessen!

```
void menu_main_startGame ()
{
   menu_hide_all();
   menu_mode = MENU_MODE_NONE;
   level_load("testlevel.wmb");
   lvl_reset();
}
```

Wir haben vorhin das HUD ausgeschaltet, also müssen wir es im Level-Reset wieder anschalten:

```
void lvl_reset ()
{
```

```
   //package count
      pk_count = 0;
      pk_count_max = 0;

   //package pointer
      pk_current = 0;

   //turn on HUD
      hud_show();
}
```

Als letzten Schritt müssen wir die Funktionen noch an die Buttons binden:

```
void menu_init ()
{
   //(...)

   menu_startGame = pan_createEx("button(0,0, menu_startGame_n.tga, menu_startGame_n.tga,
                    menu_startGame_t.tga, menu_main_startGame, NULL, NULL);",
                    101, NULL, 730, 403, 0);

   //(...)

   menu_exit = pan_createEx("button(0,0, menu_exit_n.tga, menu_exit_n.tga, menu_exit_t.tga,
               menu_main_exit, NULL, NULL);", 101, NULL, 737, 561, 0);

   //(...)
}
```

Wenn wir jetzt im Menü auf "start game" klicken, starten wir das Spiel und wenn wir auf "exit" klicken, wird das Spiel beendet.

Auflösungsunabhängigkeit

Wir werden im Konfigurationsmenü u.a. die Spielauflösung ändern können. Das ist zwar alles schön und gut, aber es gibt da ein Problem: wenn wir eine geringere Auflösung wählen, sehen wir nur noch einen kleineren Teil des Bildschirms und wenn wir eine größere Auflösung wählen ist das Menü kleiner als der Bildschirm. Wir wollen das auch gleich mal ausprobieren: wir gehen in die Datei "sys.c" und kommentieren in der sys_keys_init die Zeile on_f5 = sys_key_null; aus.

Wenn wir jetzt im Menü z.B. F5 drücken, ändert sich die Auflösung, aber - wie beschrieben - ist das Menü nun "kaputt", bzw. wird nicht richtig dargestellt. Nun, es gibt mehrere Möglichkeiten, dies zu umgehen. Die erste Möglichkeit ist, dass wir dem Spieler nicht anbieten, die Auflösung zu ändern. Die Lösung ist recht einfach gestrickt, aber auch ziemlich blöd. Die zweite Lösung ist, das wir für jede Auflösung ein eigenes Interface speichern - also sodass es für jede Grafik mehrere Dateien gibt, die für bestimmte Auflösungen gedacht ist. Damit haben wir zwar in jeder Auflösung eine optimale Darstellungsqualität, aber damit steigt der Festplattenspeicherverbrauch mehr als sprunghaft.

Ein andere Lösungsansatz ist es, alle Grafiken als View-Entities zu definieren und darzustellen. Das ist eigentlich ganz trickreich, weil die engine dann das Rendering kontrolliert und die Grafik immer automatisch auflösungsunabhängig ist. Nachteil an der Geschichte: Sehr viele Viewentities senken die Performance. Problematisch wird das Ganze, wenn man Texte und Digits ausgibt. Dann kann man diese Lösung auch vergessen. Ein weiterer Ansatz ist, die Panels , Texte und so weiter selber manuell zu skalieren und immer an die Auflösung anzupassen. Das ist etwas aufwändiger, weil man dann immer selbst die Daten des Panels umrechnen muss. Da Programmierer aber schlau sind und nicht gerne Code schreiben, der sich ewig und oft wiederholt, kapseln wir das in schöne Funktionen.

Die Idee ist, das wir davon ausgehen, dass das Interface für eine bestimmte Auflösung konzipiert ist. Wir gehen hier von einer Referenz-Auflösung von 1024 x 768 Pixeln aus (das ist auch die Auflösung, womit wir die Engine initialisieren). Darauf basierend können wir mithilfe der aktuellen Auflösung den Faktor ausrechnen, womit wir z.B. die Panels skalieren müssen. Wenn wir uns also in der Auflösung 640x480 befinden, dann müssten wir ein Panel auf der X-Achse 640 : 1024 = 0,625 mal skalieren und auf der Y-Achse auch 0,625 mal. Zwar ist das hier der

gleiche Faktor, aber wenn wir die Auflösung 2048x1024 mit 1024x768 vergleichen, dann müssten wir auf der X-Achse mit 2048 : 1024 = 2 skalieren und auf der Y-Achse mit 1024 : 768 = 1,333. Daher betrachten wir beide Richtungen. Um jetzt also einen Wert (z.B. die Position) zu berechnen, multiplizieren den Wert (z.B. 100 Pixel) mit diesem Faktor.

Bevor wir das in Programmcode umsetzen, eröffnen wir dafür im "sys"-Ordner die Dateien "ri.c" und "ri.h" und inkludieren sie in der "rudi.c". "RI" steht hier für "resolution independent". Wir können eine Funktion schreiben, die uns einen Wert auf der Basis eines Referenz-Maßes und dem Ziel Maß umrechnet:

```
var ri (var in, var reference, var currentMeasure)
{
   if (in != 0) {
      return((currentMeasure / reference) * in);
   } else {
      return(1);
   }
}
```

Wir nennen die Funktion der Einfachkeit einfach "ri". Der Parameter ist der zu umrechnende Wert, reference in diesem Fall z.B. die Referenz-Auflösung und currentMeasure das Maß der aktuellen Auflösung.

Damit wir "einfachere" Werkzeuge haben, um einen Wert auf der X-Achse und einen auf der Y-Achse umzurechnen, schreiben wir uns einfachere Funktionen, sogenannte "convenience functions":

```
//- convenience functions -----------------------------------------
var ri_x (var in) {return(ri(in, RI_REFRES_X, screen_size.x));}
var ri_y (var in) {return(ri(in, RI_REFRES_Y, screen_size.y));}
```

Hierbei wird das aktuelle Maß direkt durch die Breite/Höhe der aktuellen Auflösung ersetzt. RI_REFRES_X und -_Y geben hierbei die Referenz_Auflösung an. Die beiden Werte sind in der ri.h so definiert:

```
#define RI_REFRES_X  1024
#define RI_REFRES_Y  768
```

Sollte man für eine andere Auflösung designen, muss das hier geändert werden. Wir können nun ganz einfach für die X-Achse einen Wert umrechnen, indem wir ri_x aufrufen und direkt den Wert angeben. Das gleiche gilt für ri_y.

Um jetzt ein Panel auflösungsunabhängig anzuzeigen, müssen wir die Position angeben, wo das Panel in der Auflösung 1024x768 wäre. Wir müssen das ausdrücklich irgendo festhalten, weil wir die Position und die Skalierung des Panels ändern. Wenn wir das also einmal gemacht haben, sind die Positionsdaten und die Skalierung überschrieben! Wir schreiben uns nun also eine Funktion, die ein Panel genau positioniert und skaliert, wie wir uns das vorstellen:

```
void ri_refreshPanel (PANEL* panel, VECTOR* pos)
{
   if (panel) {
      panel->pos_x = ri_x(pos->x);
      panel->pos_y = ri_y(pos->y);
      panel->scale_x = ri_x(1);
      panel->scale_y = ri_y(1);
      set(panel, FILTER);
   }
}
```

Es wird ein Panel genommen und die Position, bzw. die Skalierung ausgerechnet. Der Parameter pos ist ein Vektor, der in seiner x und y Komponente die eigentliche Position des Panels in der Referenzauflösung hätte. Wenn nun aber die Auflösung geändert wird, sind alle Panels wieder falsch (skaliert und positioniert). Deshalb muss man schauen, dass die Panels immer bei einer Änderung der Auflösung geändert wird oder dass sie dauernd geupdated werden. Das FILTER flag wird gesetzt, damit man keine Aliasing-Fehler beim Skalieren sieht.

Dazu können wir uns eine Funktion schreiben, die für das Menü immer alle Update refreshedt:

```
void menu_refresh_all ()
{
   ri_refreshPanel (menu_logo, vector(77, 84, 0));
}
```

Hier wird jetzt probeweise das Logo immer richtig skaliert. Wenn man nun im Hauptmenü F5 drückt und damit die Auflösung ändert, wird das Logo immer richtig an die Auflösung angepasst. Wir müssen das nun für alle weiteren Panels machen:

```
void menu_refresh_all ()
{
   // MAIN MENU
      ri_refreshPanel(menu_background, vector(0,0,0));
      ri_refreshPanel(menu_startGame, vector(730, 403, 0));
      ri_refreshPanel(menu_config, vector(658, 452, 0));
      ri_refreshPanel(menu_credits, vector(27, 713, 0));
      ri_refreshPanel(menu_exit, vector(737, 561, 0));
}
```

Drückt man nun im Hauptmenü abermals öfters F5 sieht man, wie das Hauptmenü immer richtig an die Auflösung angepasst wird - Super! Damit aber die Zahlen nicht so im Code rumstehen, ersetzen wir das durch vorinitialisierte Vektoren:

menu.h:

```
//-----------------------------------------------------------------------
// layout
//-----------------------------------------------------------------------

   //MAIN MENU

      VECTOR* menu_background_pos   = {x = 0;   y = 0;   z = 0;}
      VECTOR* menu_startGame_pos    = {x = 103; y = 430; z = 0;}
      VECTOR* menu_config_pos       = {x = 131; y = 588; z = 0;}
      VECTOR* menu_credits_pos      = {x = 132; y = 645; z = 0;}
      VECTOR* menu_exit_pos         = {x = 143; y = 702; z = 0;}
```

menu.c:

```
void menu_refresh_all ()
{
   // MAIN MENU
      ri_refreshPanel(menu_background, menu_background_pos);
      ri_refreshPanel(menu_startGame, menu_startGame_pos);
      ri_refreshPanel(menu_config, menu_config_pos);
      ri_refreshPanel(menu_credits, menu_credits_pos);
      ri_refreshPanel(menu_exit, menu_exit_pos);
}
```

Ein auflösungsunabhängiges HUD

Das HUD ist von dem Auflösungsproblem auch betroffen: wir müssen es anpassen! Bisher rufen wir nur einmal die Funktion hud_init auf. Wir ändern das nun so um, wie wir es bei dem Menü auch gemacht haben. Wir schreiben uns eine main-Funktion, die das HUD initialisiert und in einer Schleife die HUD Elemente dann refreshedt. Für das refreshen benutzen wir dann wieder Vektoren.

game.c:

```
void game_init ()
{
   //(...)

   //initialize HUD
      hud_main();
}
```

hud.c:

```
void hud_main ()
{
   hud_init();

   while (1) {
      hud_refresh();
      wait(1);
   }
}

void hud_refresh ()
{
   ri_refreshPanel(hud_lifes, hud_lifes_pos);
   ri_refreshPanel(hud_heart, hud_heart_pos);
}
```

hud.h:

```
VECTOR* hud_heart_pos   = {x = 846; y = 603; z = 0;}
VECTOR* hud_lifes_pos   = {x = 15;  y = 552; z = 0;}
```

Jetzt passt sich das HUD genauso wie das Menü der Auflösung an - toll!

Damit wir die Auflösung nur noch über das Menü ändern können, entfernen wir in der Funktion sys_keys_init wieder den Kommentar vor der Zeile mit der event-Zuweisung für die Taste F5.

Das Konfigurationsmenü

Der Spieler soll im Konfigurationsmenü die Detailstufe, die Auflösung und die Lautstärke einstellen können. Dazu bedarf es mehrerer Dialogelemente, die auf in einem eigenen Dialog dargestellt werden sollen: zunächst brauchen wir ein Hintergrundbild und die Menütitelzeile der Konfiguration, sowie eine "OK" Schaltfläche, um zum Hauptmenü zurückzukehren.

Die Gestaltung von Dialogen ist nicht trivial. Es erweist sich durchaus als schwer, einen sowohl komfortablen als auch effizienten und wohl durchstruktierten Dialog zu entwerfen. Wir haben zudem den Drang, bei unserem kleinem Spiel alles so einfach wie möglich zu halten. Wir machen es so: wir werden auf dem Bildschirm alle verfügbaren Detailstufen (niedrig, mittel und hoch) und Auflösungen (640x480, 800x600, 1024x768 und 2048x1024) anzeigen und die aktuell aktive Auswahl hervorheben. Klickt der Benutzer auf eine nicht aktive Auswahl, wird diese aktiviert. Im Fall einer geänderten Bildschirmauflösung wird dann sofort die Auflösung gewechselt. Für die Musik- Soundeffektlautstärke wird es zwei Schieberegler geben, die man zwischen 0 und 100% bewegen kann. Wenn wir später dem Menü noch Musik hinzufügen, dann wird es so sein, dass der Nutzer beim Bewegen des Musikreglers die Auswirkung direkt hören wird.

Das Konfigurationsmenü sieht hinterher so aus:

Ressourcen und andere Vorbereitungen

Im Menü Ordner sind viele Dateien, die wir für das Konfigurationsmenü benutzen wollen. Sie alle beginnen mit dem Kürzel "menu_cfg", um sie eindeutig zu unterscheiden. Bei Button-Grafiken weisen die Suffixe vor der Dateiendung "_n" und "_t" - wie bereits bekannt - auf den normalen und berührten (engl: touched) Zustand des Buttons hin. Einige Grafiken enden mit "_u" und "_c". Dies steht für "unchecked" und "checked", wobei dies den inaktiven, bzw. aktiven Zustand des Knopfes meint. Wir wollen nämlich einen sogenannten Radiobutton benutzen, um die Detail und Auflösungszustände zu steuern. Dabei ist es so, dass wenn ein Radiobutton einer Gruppe (alle anderen Radiobuttons eines Panels) aktiviert wird, dass alle anderen deaktiviert werden - sodass immer nur einer aktiv ist.

Wir werden für das Konfigurationsmenü folgende Panels benötigen:

```
PANEL* menu_cfg_background;
PANEL* menu_cfg_detail;
PANEL* menu_cfg_detailLabel;
PANEL* menu_cfg_res;
PANEL* menu_cfg_resLabel;
PANEL* menu_cfg_volume;
PANEL* menu_cfg_OK;
```

Die "label" Panels sind quasi "Hinweis"-Panels, die anzeigen, dass man hier z.B. die Details und dort die Auflösung einstellen kann. Das Panel menu_cfg_volume fasst die beiden Schieberegler zusammen. Schieberegler benötigen eine Grafik, die sie für einen Knopf am Schieberegler benutzen. Die Bitmap, die wir dafür nutzen, lautet:

```
BMAP* menu_cfg_dot;
```

Damit wir das Menü auch im Code ansteuern können, fügen wir es für die menu_mode Variable als define mit dem Wert 2 hinzu:

```
var menu_mode;

   #define MENU_MODE_NONE     0
   #define MENU_MODE_MAIN     1
   #define MENU_MODE_CFG      2
```

Für die mitgelieferten Grafiken gibt es auch schon festgelegte Werte, wo sich die einzelnen Panels auf dem Bildschirm befinden. Wenn Sie ihr eigenes Menü bauen wollen, müssen Sie die Werte natürlich anpassen:

```
//CONFIG

   VECTOR* menu_cfg_bg_pos          = {x = 0;   y = 0;   z = 0;}
   VECTOR* menu_cfg_detailLabel_pos = {x = 287; y = 210; z = 0;}
   VECTOR* menu_cfg_detail_pos      = {x = 429; y = 210; z = 0;}
   VECTOR* menu_cfg_resLabel_pos    = {x = 61;  y = 330; z = 0;}
   VECTOR* menu_cfg_res_pos         = {x = 185; y = 315; z = 0;}
   VECTOR* menu_cfg_vol_pos         = {x = 112; y = 451; z = 0;}
   VECTOR* menu_cfg_OK_pos          = {x = 483; y = 562; z = 0;}
```

Die Konfiguration aufrufbar machen

Als erstes wollen wir die Anbindung im Hauptmenü herstellen. Wir weisen dem Button "Configuration" eine Funktion zu, die den Menümode umstellt, sodass im nächsten Frame das Konfigurationsmenü angezeigt wird:

```
void menu_main_config ()
{
   menu_mode = MENU_MODE_CFG;
}
```

Damit die Funktion aufgerufen wird, müssen wir in der Funktion menu_init in der Button-Definition des Panels menu_config den entsprechenden Funktionszeiger im OnClick-Event setzen:

```
menu_config = pan_createEx("button(0,0, menu_config_n.tga, menu_config_n.tga, menu_config_t.tga,
                            menu_main_config, NULL, NULL);", 101, NULL, 0, 0, 0);
```

Damit das Menü auch aufgerufen wird, wenn der Mode gesetzt ist, müssen wir es in die menu_main Funktion im Switch-Case-Tree verfügbar machen:

```
switch (menu_mode) {
   case MENU_MODE_NONE: break;
   case MENU_MODE_MAIN: menu_main_show(); break;
   case MENU_MODE_CFG:  menu_cfg_show(); break;
}
```

Wir benutzen für das Menü also die Funktion menu_cfg_show. Die Funktion ist ähnlich gestrickt wie die anderen Menüfunktionen:

```
void menu_cfg_show ()
{
   hud_hide();             //hide HUD
   menu_hide_all();        //hide all elements
   menu_cfg_elements();    //show selected elements
   menu_cursor();          //draw cursor
}
```

Die Funktion menu_cfg_elements ist neu: sie wird die Menüelemente anzeigen. Wir werden einfach alle beteiligten Panels anschalten:

```
void menu_cfg_elements ()
{
   set(menu_cfg_background, VISIBLE);
   set(menu_cfg_detailLabel, VISIBLE);
   set(menu_cfg_detail, VISIBLE);
   set(menu_cfg_resLabel, VISIBLE);
   set(menu_cfg_res, VISIBLE);
   set(menu_cfg_volume, VISIBLE);
   set(menu_cfg_OK, VISIBLE);
}
```

Damit die Elemente auch wieder automatisch ausgeschaltet werden, fügen wir sie der Funktion menu_hide_all hinzu:

```
void menu_hide_all ()
{
   //(...)

   reset(menu_cfg_background, VISIBLE);
   reset(menu_cfg_detailLabel, VISIBLE);
   reset(menu_cfg_detail, VISIBLE);
   reset(menu_cfg_resLabel, VISIBLE);
   reset(menu_cfg_res, VISIBLE);
   reset(menu_cfg_volume, VISIBLE);
   reset(menu_cfg_OK, VISIBLE);

}
```

Damit die Elemente wie die anderen Elemente der anderen Menüs auch auflösungsunabhängig dargestellt wird, fügen wir die Panels der Funktion menu_refresh_all hinzu:

```
void menu_refresh_all ()
{
   //...
   // CONFIG
      ri_refreshPanel(menu_cfg_background, menu_cfg_bg_pos);
      ri_refreshPanel(menu_cfg_detailLabel, menu_cfg_detailLabel_pos);
      ri_refreshPanel(menu_cfg_detail, menu_cfg_detail_pos);
      ri_refreshPanel(menu_cfg_resLabel, menu_cfg_resLabel_pos);
      ri_refreshPanel(menu_cfg_res, menu_cfg_res_pos);
      ri_refreshPanel(menu_cfg_volume, menu_cfg_vol_pos);
      ri_refreshPanel(menu_cfg_OK, menu_cfg_OK_pos);
}
```

Die Initialisierung des Menüs

Nun ist das Menü aufrufbar, allerdings sehen wir noch nichts, weil wir in der Funktion menu_init die Panels und Grafiken noch nicht geladen, bzw. erzeugt haben. Als erstes laden wir die Hintergrundgrafik und die Titelzeile:

```
menu_cfg_background = pan_createEx("", 200, "menu_cfg_bg.tga", 0, 0, 0);
```

Das war ja noch einfach. Der OK Button am unteren Ende des Bildschirms ist auch recht einfach gestrickt:

```
menu_cfg_OK = pan_createEx("button(0,0, menu_cfg_OK_n.tga, menu_cfg_OK_n.tga, menu_cfg_OK_t.tga,
                            NULL, NULL, NULL);", 201, NULL, 0, 0, 0);
```

Außerdem können wir auch noch recht einfach die labels für die Details und die Auflösung erzeugen:

```
menu_cfg_detailLabel = pan_createEx("", 201, "menu_cfg_detailLabel.tga", 0, 0, 0);
menu_cfg_resLabel = pan_createEx("", 201, "menu_cgf_resLabel.tga", 0, 0, 0);
```

Für die Zeile mit den 3 Detailstufen benötigen wir in der content-Definition beim Aufruf von pan_createEx 3 Radiobutton Definitionen. Ein Radiobutton ist folgendermaßen definiert:

```
button_radio(x, y, bmapOn, bmapOff, bmapOver, functionClick, functionLeave, functionOver);
```

wobei bmapOn den aktivierten und bmapOff den deaktivierten Zustand meint. Wir können daher nun das Panel menu_cfg_detail mit 3 Radiobuttons wie folgt erzeugen (die Pixelangaben sind wieder bereits vorgegeben und müssen bei eigenen Grafiken angepasst werden):

```
menu_cfg_detail = pan_createEx("button_radio(0, 0, menu_cfg_detailLow_c.tga, menu_cfg_detailLow_u.tga,
                                menu_cfg_detailLow_u.tga, NULL, NULL, NULL);

                                button_radio(93, 0, menu_cfg_detailMed_c.tga, menu_cfg_detailMed_u.tga,
                                menu_cfg_detailMed_u.tga, NULL, NULL, NULL);

                                button_radio(246, 0, menu_cfg_detailHigh_c.tga, menu_cfg_detailHigh_u.tga,
                                menu_cfg_detailHigh_u.tga, NULL, NULL, NULL);", 201, NULL, 0, 0, 0)
```

Den übergebenen char* Parameter darf man ruhig in mehrere Zeilen schreiben. Die gleiche Definition können wir für die Auflösung machen:

```
menu_cfg_res = pan_createEx("button_radio(0, 0, menu_cfg_res640_c.tga, menu_cfg_res640_u.tga,
                             menu_cfg_res640_u.tga, NULL, NULL, NULL);

                             button_radio(192, 10, menu_cfg_res800_c.tga, menu_cfg_res800_u.tga,
                             menu_cfg_res800_u.tga, NULL, NULL, NULL);

                             button_radio(385, 0, menu_cfg_res1024_c.tga, menu_cfg_res1024_u.tga,
                             menu_cfg_res1024_u.tga, NULL, NULL, NULL);

                             button_radio(588, 0, menu_cfg_res2048_c.tga, menu_cfg_res2048_u.tga,
                             menu_cfg_res2048_u.tga, NULL, NULL, NULL);", 201, NULL, 0, 0, 0);
```

Bevor wir nun die Schieberegler erzeugen, müssen wir die Grafik laden, die wir für den Reglerknopf benutzen:

```
menu_cfg_dot = bmap_create("menu_cfg_volDot.tga");
```

Danach können wir die Schieberegler erzeugen. Die engine unterstützt horizontale und vertikale Schiebregler, die folgende Definition haben:

```
hslider (x, y, len, bmap, min, max, var);
```

Mit x, y, und len gibt man die Position und die Bitmap des KNopfes an. Mit min und max kann man den Wertebereich der Variable var - die wir regeln - angeben. Wir wollen die Variablen, in denen wir die Lautstärke zentral in der game.h festlegen:

```
var volMusic;
var volSfx;
```

Wir erstellen nun ein Panel mit einem Hintergrund und platzieren darauf zwei Schieberegler für die Lautstärke der Musik und der Soundeffekte:

```
menu_cfg_volume = pan_createEx("hslider(216, 16, 188, menu_cfg_dot, 0, 100, volMusic);
                                hslider(604, 16, 188, menu_cfg_dot, 0, 100, volSfx);",
                                201, "menu_cfg_volBars.tga", 0, 0, 0);
```

Damit hätten wir das Menü vorerst komplett eingefügt: wenn wir das Spiel starten, sollte bei einem Klick auf "Configuration" das Menü aufpoppen. Wenn wir auf die Radio-Buttons klicken, sollte auch nur immer jeweils einer aktiv sein.

Die Funktionalität sicherstellen

Doch leider kann man mit dem Menü bisher nicht viel anfangen. Zunächst wollen wir den OK Knopf verbessern: wenn der User darauf klickt, soll er wieder zum Hauptmenü gelangen. Dazu müssen wir - ähnlich wie bei dem Knopf im Hauptmenü, der zum Konfigurationsbildschirm führt - einfach nur den Menümodus ändern:

```
void menu_cfg_main ()
{
   menu_mode = MENU_MODE_MAIN;
}
```

Der Funktionszeiger der Funktion menu_cfg_main muss dann nur noch im Button gesetzt werden:

```
menu_cfg_OK = pan_createEx("button(0,0, menu_cfg_OK_n.tga, menu_cfg_OK_n.tga, menu_cfg_OK_t.tga,
                            menu_cfg_main, NULL, NULL);", 201, NULL, 0, 0, 0);
```

Bei einem Klick auf "OK" verlassen wir das Konfigurationsmenü und landen nun wieder im Hauptmenü.

Wenn wir ins Konfigurationsmenü wechseln, fällt einem direkt ins Auge, dass die Radio-Buttons inaktiv sind. Wir müssen dafür sorgen, dass die Buttons direkt richtig angeschaltet sind. Die Schieberegler sind zwar an die Variablen volMusic und volSfx gebunden, jedoch stehen die beiden Variablen bei engine-Start auf 0. Die Auflösung können wir direkt auslesen, aber die Detailstufe haben wir noch nicht definiert. Damit wir also die Buttons usw. einstellen können, definieren wir uns vorher noch in der game.h eine Variable für die Detailstufe und erzeugen uns ein paar defines, die für die einzelnen auswählbaren Detailstufen stehen:

```
var detail;

   #define DETAIL_LOW      0
   #define DETAIL_MEDIUM   1
   #define DETAIL_HIGH     2
```

Dadurch, dass wir den Datentyp var für detail benutzen, können wir den Detailgrad später gut in arithmetischen Ausdrücken benutzen. Das werden wir später dann noch näher iM Detail betrachten, wenn wir den Detailgrad für das Spiel einbauen.

Damit die Sound-Werte und der Detailgrad bei Spielstart voreingestellt sind, erzeugen wir bei der game_init Funktion eine neue Funktion, die diese generellen Spiel-Werte einstellt:

```
void game_init_variables ()
{
   volMusic = 80;
   volSfx = 95;

   detail = DETAIL_HIGH;

   fps_max = 60;
}
```

und rufen die Funktion game_init_variables dann in der game_init auf:

```
void game_init ()
{
   //(...)

   // General game variables
      game_init_variables();

}
```

Jetzt stehen die Schieberegler zumindestens schonmal auf den richtigen Positionen. Das gilt aber nicht für die Radio-Buttons - dies wollen wir durch eine Funktion lösen, die die Auflösung und die Variable detail ausliest und die Aktivitätsmerkmale der Radiobuttons daraufhin steuert. Man kann mit der Funktion button_state den Aktivitätsmodus eines Radiobuttons (gilt auch für Togglebuttons!) setzen. Wir schreiben uns die Funktion menu_cfg_setValues, die den Detailgrad auswertet und dann dementsprechend den entsprechenden Detail-Radiobutton setzt:

```
void menu_cfg_setValues ()
{
   //detail
      switch (detail) {
         case DETAIL_LOW:     button_state(menu_cfg_detail, 1, 1); break;
         case DETAIL_MEDIUM:  button_state(menu_cfg_detail, 2, 1); break;
         case DETAIL_HIGH:    button_state(menu_cfg_detail, 3, 1); break;
      }
}
```

Die Syntax von button_state sieht so aus:

```
button_state(PANEL*,var num,var state);
```

Dass das Panel angegeben werden muss, ist ersichtlich. Num gibt den button an, den man anspricht, wobei man sich da auf die Reihenfolge der Buttons in der Definition bezieht - der erste Button in der Definition hat die Nummer 1, usw. State gibt den Aktviitätsstatus an. Wenn wir bei einem Radio- oder Togglebutton 1 angeben, setzt der Button seinen state auf "checked". Bei Radiobuttons werden dann alle anderen Buttons ausgeschaltet, bei Togglebuttons bleiben alle anderen Buttons des Panels unberührt.

Über den Switch-Case-Tree werten wir den Detailmodus aus und reagieren dann darauf. Wir können die Funktion nun in die Funktion menu_cfg_show einbetten:

```
void menu_cfg_show ()
{
   hud_hide();             //hide HUD
   menu_hide_all();        //hide all elements
   menu_cfg_setValues();   //set all configurable values
   menu_cfg_elements();    //show selected elements
   menu_cursor();          //draw cursor
}
```

Wenn wir das Konfigurationsmenü nun starten, ist der Detailgrad richtig aktiviert. Bei der Auflösung werden wir ähnlich verfahren und folgenden Use-Case-Tree aufsetzen:

```
//videomode
   switch (video_mode) {
      case 6: button_state(menu_cfg_res, 1, 1); break;
      case 7: button_state(menu_cfg_res, 2, 1); break;
      case 8: button_state(menu_cfg_res, 3, 1); break;
      case 9: button_state(menu_cfg_res, 4, 1); break;
   }
```

Nun stimmt auch die Anzeige der korrekten Auflösung - fehlt nur noch die Reaktion der Buttons auf den Mausklick! Man kann mehreren Buttons ein und diesselbe Funktion zuweisen und über eine Variable, die die Button-Nummer angibt, auf unterschiedliche Button-Klicks reagieren. Dies wollen wir ausnutzen! Die Detailbuttons sind z.B. mit 1, 2 und 3 nummeriert, wobei die Detailgrade mit 0, 1 und 2 nummeriert sind. Wenn wir die Buttonnummer haben, ziehen wir einfach mit eins ab und haben dann den richtigen Detailgrad. Das kann man mit folgender Buttonfunktion lösen:

```
void menu_cfg_setDetail (var buttonNr)
{
   detail = buttonNr - 1;
}
```

Die engine übergibt automatisch als ersten Parameter an Funktionen immer eine Variable, die die Button-Nummer enthält. Wir können jetzt durch eine einfach Rechnung den Wert für die Variable detail berechnen. Wenn wir "menu_cfg_setDetail" in die Buttondefinitionen als OnClick-Event schreiben, dann wird der Radio-Button gewechselt, wenn wir auf einen deaktivierten Detail-Radiobutton klicken:

```
menu_cfg_detail = pan_createEx("button_radio(0, 0, menu_cfg_detailLow_c.tga, menu_cfg_detailLow_u.tga,
                                menu_cfg_detailLow_u.tga, menu_cfg_setDetail, NULL, NULL);

                                button_radio(93, 0, menu_cfg_detailMed_c.tga, menu_cfg_detailMed_u.tga,
                                menu_cfg_detailMed_u.tga, menu_cfg_setDetail, NULL, NULL);

                                button_radio(246, 0, menu_cfg_detailHigh_c.tga, menu_cfg_detailHigh_u.tga,
                                menu_cfg_detailHigh_u.tga, menu_cfg_setDetail, NULL, NULL);",

                                201, NULL, 0, 0, 0);
```

Dasselbe Verhalten werden wir für die Auflösung ausnutzen. Auf den Klick des Radiobuttons wird sich die Auflösung von selbst anpassen:

```
void menu_cfg_setRes (var buttonNr)
{
   video_switch(5 + buttonNr, 0, 0);
}
```

Wir müssen die Funktion menu_cfg_setRes nur noch in die Paneldefinition einbauen:

```
menu_cfg_res = pan_createEx("button_radio(0, 0, menu_cfg_res640_c.tga, menu_cfg_res640_u.tga,
                             menu_cfg_res640_u.tga, menu_cfg_setRes, NULL, NULL);

                             button_radio(192, 10, menu_cfg_res800_c.tga, menu_cfg_res800_u.tga,
                             menu_cfg_res800_u.tga, menu_cfg_setRes, NULL, NULL);

                             button_radio(385, 0, menu_cfg_res1024_c.tga, menu_cfg_res1024_u.tga,
                             menu_cfg_res1024_u.tga, menu_cfg_setRes, NULL, NULL);

                             button_radio(588, 0, menu_cfg_res2048_c.tga, menu_cfg_res2048_u.tga,
                             menu_cfg_res2048_u.tga, menu_cfg_setRes, NULL, NULL);", 201, NULL, 0, 0, 0);
```

Damit ist der Konfigurationsbildschirm nun voll funktionsfähig!

Der Gameover-Bildschirm

Nun, Rudi hat mehrere Leben - aber irgendwann hat er auch genug davon verbraucht und hat keine Leben mehr - dann ist er tot - oder vielmehr: das Spiel ist dann vorbei! Wir haben festgelegt, dass dann ein klassischer Gameover-Bildschirm erscheint. Dort hat der Spieler dann die Möglichkeit, in das Hauptmenü zurückzukehren, das Spiel komplett zu beenden oder das Spiel neu zu starten (im ersten Level aber dann!). Dazu müssen wir einen weiteren Dialog erzeugen und anzeigbar machen. Im Menüordner sind einige Grafiken dafür vorhanden, die mit "menu_gameover" anfangen. Das sind einmal ein Hintegrundbild mit dem "GAME OVER"-Logo und 3 Buttons für die oben beschriebenen Möglichkeiten, fortzufahren. Wir ordnen den Gameover-Dialog auch dem Menü zu, also bewegen wir uns weiterhin in den Dateien "menu.c" und "menu.h".

Wir definieren zunächst einen weiteren Modus für den Gameover-Dialog:

```
var menu_mode;

   #define MENU_MODE_NONE     0
   #define MENU_MODE_MAIN     1
   #define MENU_MODE_CFG      2
   #define MENU_MODE_GAMEOVER 3
```

Damit der Mode auch erkannt wird, fügen wir ihn zum Switch-Case-Tree der Funktion menu_main hinzu:

```
switch (menu_mode) {
   case MENU_MODE_NONE:       break;
   case MENU_MODE_MAIN:       menu_main_show(); break;
   case MENU_MODE_CFG:        menu_cfg_show(); break;
   case MENU_MODE_GAMEOVER:   menu_gameover_show(); break;
}
```

Wie bereits definiert, nutzen wir die Funktion menu_gameover_show zum Anzeigen des Dialogs. Diese Funktion sieht wie folgt aus:

```
void menu_gameover_show ()
{
   hud_hide();                //hide HUD
   menu_hide_all();           //hide all elements
   menu_gameover_elements();  //show selected elements
   menu_cursor();             //draw cursor
}
```

Die Funktion menu_gameover_elements zeigt die Elemente des Gameover-Bildschirms an. Bevor wir uns der Funktion zuwenden, wollen wir die Elemente aber erstmal definieren. Wir benötigen für alle Elemente die folgenden Elemente:

```
// GAMEOVER

   PANEL* menu_gameover_background;
   PANEL* menu_gameover_retry;
   PANEL* menu_gameover_back;
   PANEL* menu_gameover_exit;
```

wobei wir diese vorgefertigten Positionsdaten definieren und später dann auch benutzen:

```
// GAMEOVER

   VECTOR* menu_gameover_background_pos   = {x = 0;   y = 0;   z = 0;}
   VECTOR* menu_gameover_retry_pos        = {x = 273; y = 488; z = 0;}
   VECTOR* menu_gameover_back_pos         = {x = 487; y = 513; z = 0;}
   VECTOR* menu_gameover_exit_pos         = {x = 716; y = 543; z = 0;}
```

Damit die Panels auch automatisch ausgeblendet werden können, fügen wir sie der Funktion menu_hide_all hinzu:

```
void menu_hide_all ()
{
   //(...)

   reset(menu_gameover_background, VISIBLE);
   reset(menu_gameover_retry, VISIBLE);
   reset(menu_gameover_back, VISIBLE);
   reset(menu_gameover_exit, VISIBLE);

   //(...)
}
```

Die Erfassung für die auflösungsunabhängige Darstellung müssen wir auch noch hinzufügen:

```
void menu_refresh_all ()
{
   //(...)

   // GAMEOVER

      ri_refreshPanel(menu_gameover_background, menu_gameover_background_pos);
      ri_refreshPanel(menu_gameover_retry, menu_gameover_retry_pos);
      ri_refreshPanel(menu_gameover_back, menu_gameover_back_pos);
      ri_refreshPanel(menu_gameover_exit, menu_gameover_exit_pos);
}
```

Jetzt können wir die Funktion menu_gameover_elements schreiben, die uns die wesentlichen Elemente des Gameover-Bildschirms anzeigt:

```
void menu_gameover_elements ()
{
   set(menu_gameover_background, VISIBLE);
   set(menu_gameover_retry, VISIBLE);
   set(menu_gameover_back, VISIBLE);
   set(menu_gameover_exit, VISIBLE);
}
```

Jetzt kann man schonmal den Gameover-Dialog anzeigen. Nur leider wird das noch nirgends getan! Wir müssen dem Spiel erstmal beibringen, dass bei zuwenig Leben dies auch getan wird.

Den Gameoverbildschirm anzeigen

Wir haben bereits den Fall programmiert, das Rudi gegen eine Mauer o.ä. crashedt und dann tot auf dem Rücken liegt und dann darauf das Level neustartet. Wir können nun ganz bequem an dieser Stelle einen Check einfügen, ob wir noch genügend Leben haben – wenn ja, wird weiterhin das Level neu gestartet und wenn nicht, zeigen wir den Game Over Bildschirm. Wir haben am Ende der Funktion pl_death_crash eine While-Schleife, die ein Kamera-Preset via pl_death_crash_cam aufruft. Wir bauen also die While-Schleife erstmal um, sodass wir nur eine gewissen Anzahl von Sekunden Rudi nach seinem Crash sehen:

```
void pl_death_crash ()
{
   //(...)

   // Decrease lifes
      lifes--;

   var waitRestart = 3 * 16;
   while (waitRestart > 0) {
      pl_death_crash_cam ();
      waitRestart -= time_step;
      wait(1);
   }
}
```

Die While-Schleife wird dann genau 3 Sekunden ausgeführt, dann wird sie beendet. Die 3 kann man in der player.h auch auslagern:

```
var   pl_death_stayTime = 3; //seconds
```

Wir können danach prüfen, ob die Anzahl der Leben in der Variable lifes noch > 0 ist oder eben nicht. Im Falle, das wir keine Leben mehr haben, aktivieren wir den Gameover-Bildschirm, andernfalls führen wir den Neustart des Levels durch. Wir fügen also nach der While-Schleife folgende Abfrage ein:

```
if (lifes == 0) {

   menu_mode = MENU_MODE_GAMEOVER;

   //wait one frame to establish the Gameover screen
   wait(1);

   level_load("");

} else {

   // Reset level

      //(...)
}
```

Dazu sei gesagt, dass mit level_load(""); wir den Levelspeicher leeren und ein leeres Level laden. Wir warten

deshalb einen Frame, weil die Schleife in menu_main erst einen Frame nach setzen des Modus darauf reagiert und den Screen anschaltet. Andernfalls würden wir einen Frame lang "nichts" sehen, weil der level_load(""); Befehl noch in der gleichen Befehlssequenz - und damit vor der Darstellung des Gameover-Bildschirms - ausgeführt werden würde.

Wenn wir also nun fünf mal hintereinander gegen die Wand laufen, wird anstatt neuzustarten der Game Over Bildschirm angezeigt - Super!

Die Buttonfunktionen des Gameover-Bildschirms

Damit der Gameover-Bildschirm auch funktioniert, müssen wir eben noch die zugehörigen Funktionen erzeugen. Beim EXIT Knopf reicht es, die engine herunterzufahren:

```
void menu_gameover_quitGame ()
{
   sys_exit("");
}
```

In der Buttondefinition tragen wir die Funktion dann ein:

```
menu_gameover_exit = pan_createEx("button(0,0, menu_gameover_exit_n.tga, menu_gameover_exit_n.tga,
                                   menu_gameover_exit_t.tga, menu_gameover_quitGame, NULL, NULL);",
                                   101, NULL, 0, 0, 0);
```

Um vom Gameover-Bildschirm zum Hauptmenü zu wechseln, müssen wir nur den Modus entsprechend setzen:

```
void menu_gameover_main ()
{
   menu_mode = MENU_MODE_MAIN;
}
```

und setzen die Funktion auch im Button:

```
menu_gameover_back = pan_createEx("button(0,0, menu_gameover_back_n.tga, menu_gameover_back_n.tga,
                                   menu_gameover_back_t.tga, menu_gameover_main, NULL, NULL);",
                                   101, NULL, 0, 0, 0);
```

Der Button, der das Spiel neu startet, führt im Prinzip diesselben Funktionsaufrufe auf, wie wir sie in der Funktion menu_main_startGame bereits definiert hatten. Weil wir später noch einen Levelloader bauen werden, werden wir das in eine eigene Funktion auslagern, was jetzt in der menu_main_startGame steht. Wir nennen die Funktion game_start und platzieren sie in der "game.c":

```
void game_start ()
{
   menu_hide_all();
   menu_mode = MENU_MODE_NONE;
   level_load("testlevel.wmb");
   lvl_reset();
}
```

Natürlich ersetzen wir jetzt in der Funktion menu_main_startGame den vorhandenen Code durch den Aufruf von game_start:

```
void menu_main_startGame ()
{
   game_start(); //start game
}
```

Die Funktion für den Knopf im Gameover-Bildschirm lautet analog:

```
void menu_gameover_restartGame ()
{
   game_start(); //restart game
}
```

Wir könnten theoretisch auch diesselbe Button-Funktion nehmen, aber wir haben durch eigene Funktionen mehr Möglichkeiten spezielle Dinge abzufangen usw., wie wir später noch sehen werden. Die Funktion menu_gameover_restartGame muss noch in den Button eingepflegt werden:

```
menu_gameover_retry = pan_createEx("button(0,0, menu_gameover_retry_n.tga, menu_gameover_retry_n.tga,
                                    menu_gameover_retry_t.tga, menu_gameover_restartGame, NULL, NULL);",
                                    101, NULL, 0, 0, 0);
```

Wenn wir jetzt das Spiel starten, gegen eine Wand crashen, wird der Dialog aufgerufen. Beim testen aller 3 Knöpfe testen wir, dass alles wunderbarst funktioniert – toll!

Das Ingame-Menü

Der Spieler soll, wenn er während des Spiels die Escape-Taste drückt, ein kleines Menü aufrufen können - ein sogenanntes ingame- oder game-Menü. Es soll einen Knopf zum Fortfahren besitzen, einen Knopf, um zur Konfiguration zu gelangen und einen Knopf, um das laufende Spiel zu beenden und in das Hauptmenü zurückzukehren. Während das ingame-Menü und/oder das vom ingame-Menü erreichbare Konfigurationsmenü sichtbar sind, ist der Bildschirm etwas ausgedunkelt und das Spiel ist angehalten. Fährt man fort, wird das Spiel fortgesetzt. Damit wir das Rad nicht neu erfinden müssen, werden wir das Konfigurationsmenü des Hauptmenüs hier weiterverwenden. Wir werden es dennoch nicht "schluderig" wie man manchmal ist, kopieren und neu einfügen, sondern so anpassen, dass es mit dem ingame-Menü harmoniert: der OK Knopf muss im Falle des ingame-Menüs zurück zum Ingame-Menü führen und die HUD-Elemente dürfen nicht ausgeblendet.

Bevor wir allerdings anfangen können, die Funktionalität des Menüs in Angriff zu nehmen, wollen wir die Ressourcen laden und alle nötigen Sachen einstellen, damit das Menü in das System eingegliedert ist.

Das Ingame-Menü sieht im Spiel hinterher so aus:

Als erstes fügen wir einen neuen Menü-Modus namens MENU_MODE_GAME hinzu:

```
var menu_mode;

   #define MENU_MODE_NONE     0
   #define MENU_MODE_MAIN     1
   #define MENU_MODE_CFG      2
   #define MENU_MODE_GAMEOVER 3
   #define MENU_MODE_GAME     4
```

und fügen ihn in den Switch-Case-Tree in der Funktion menu_main hinzu:

```
switch (menu_mode) {
   case MENU_MODE_NONE:       break;
   case MENU_MODE_MAIN:       menu_main_show(); break;
   case MENU_MODE_CFG:        menu_cfg_show(); break;
   case MENU_MODE_GAMEOVER:   menu_gameover_show(); break;
   case MENU_MODE_GAME:       menu_game_show(); break;
}
```

Die zuständige Funktion zur Darstellung des Menüs lautet "menu_game_show", die wir so formulieren:

```
void menu_game_show ()
{
   menu_hide_all();           //hide all elements
   menu_game_elements();   //show selected elements
   menu_cursor();             //draw cursor
}
```

Auffällig ist, dass wir das HUD nicht ausblenden. Die Funktion menu_game_elements zeigt uns die benötigten Panels an - bevor die Funktion formulieren, wollen wir uns erst um die Panels kümmern. Wir benötigen vier Panels:

```
// INGAME MENU
   PANEL* menu_game_background;
   PANEL* menu_game_continue;
   PANEL* menu_game_config;
   PANEL* menu_game_back;
```

Dazu gehören die folgenden Layout-Koordinaten:

```
// INGAME MENU

   VECTOR* menu_game_background_pos = {x = 0;   y = 0;   z = 0;}
   VECTOR* menu_game_continue_pos   = {x = 409; y = 204; z = 0;}
   VECTOR* menu_game_cfg_pos        = {x = 344; y = 337; z = 0;}
   VECTOR* menu_game_back_pos       = {x = 371; y = 470; z = 0;}
```

Wir tragen dann in die Funktion menu_hide_all die neuen Panels ein:

```
void menu_hide_all ()
{
   //(...)

   // INGAME MENU
      reset(menu_game_background, VISIBLE);
      reset(menu_game_continue, VISIBLE);
      reset(menu_game_config, VISIBLE);
      reset(menu_game_back, VISIBLE);

   //(...)
}
```

und fügen die Panels auch der Funktion menu_refresh_all hinzu, damit die Panels auch immer schön auflösungsunabhängig gerendert werden:

```
void menu_refresh_all ()
{
   //(...)

   // INGAME MENU
      ri_refreshPanel(menu_game_background, menu_game_background_pos);
      ri_refreshPanel(menu_game_continue, menu_game_continue_pos);
      ri_refreshPanel(menu_game_config, menu_game_cfg_pos);
      ri_refreshPanel(menu_game_back, menu_game_back_pos);

   //(...)
}
```

Wir sind jetzt in der Lage, menu_game_elements zu formulieren: wir müssen einfach alle neuen Panels anschalten!

```
void menu_game_elements ()
{
   set(menu_game_background, VISIBLE);
   set(menu_game_continue, VISIBLE);
   set(menu_game_config, VISIBLE);
   set(menu_game_back, VISIBLE);
}
```

Bevor die Panels aber dargestellt werden können, müssen wir sie auch erzeugen und ihnen die Bitmaps zuweisen. Dies tun wir wie bei allen anderen Panels auch in der Funktion menu_init - wir erzeugen ein Panel mit einer normalen Bitmap und 3 Panels mit einem simplen Button:

```
void menu_init ()
{
   //(...)

   // INGAME MENU

      menu_game_background = pan_createEx("", 200, "menu_game_bg.tga", 0, 0, 0);

      menu_game_continue   = pan_createEx("button(0,0, menu_game_continue_n.tga,
                             menu_game_continue_n.tga, menu_game_continue_t.tga, NULL, NULL,
                             NULL);", 201, NULL, 0, 0, 0);

      menu_game_config     = pan_createEx("button(0,0, menu_game_cfg_n.tga, menu_game_cfg_n.tga,
                             menu_game_cfg_t.tga, NULL, NULL, NULL);", 201, NULL, 0, 0, 0);

      menu_game_back       = pan_createEx("button(0,0, menu_game_back_n.tga, menu_game_back_n.tga,
                             menu_game_back_t.tga, NULL, NULL, NULL);", 201, NULL, 0, 0, 0);

   //(...)
}
```

Das Menü aufrufbar machen

Wir müssen nun zusehen, wie wir das Menü aufgerufen bekommen. Die Prämisse ist, dass wir das ingame-Menü nur dann aufrufen können, wenn wir im Spiel sind. Ein Indikator, dass wir das auch sind, ist die Tatsache, das Rudi lebt. Wir könnten also erstmal schauen, ob der Zeiger auf Rudi gefüllt ist. Eine weitere Bedingung ist, dass kein anderes Menü bereits offen ist. Dies können wir ganz einfach über die menu_mode Variable feststellen. Das Menü sollte über die Escape-taste aktiviert werden, also müssen wir diese auch überprüfen. Wenn all dies überprüft ist und auch zutrifft, können wir das ingame-Menü einschalten, indem wir den menu_mode auf MENU_MODE_GAME stellen. Das Spiel würde aber dann trotzdem weiterlaufen - das ist nicht gut! Um das zu vermeiden frieren wir das Spiel über die Variable freeze_mode ein, indem wir diese Variable auf 1 stellen. Für all diese Abfragen erstellen wir uns folgende Funktion:

```
void menu_game_open ()
{
   if ((Rudi != NULL) && (menu_mode == MENU_MODE_NONE)) {
      if (key_esc) {
         freeze_mode = 1;
         menu_mode = MENU_MODE_GAME;
      }
   }
}
```

Die Funktion heißt "menu_game_open", weil wir davon ausgehen, dass der Spieler das Menü öffnet. Die Funktion menu_main läuft andauernd und überwacht die angeforderten Menü-Modi. Deshalb platzieren wir dort den Aufruf von menu_game_open:

```
void menu_main ()
{
   menu_init();

   while (1) {

      // Refresh panels
         menu_refresh_all();

      // Make a check for an ingame-menu call
         menu_game_open();

      //(...)

      wait(1);
   }
}
```

Wenn wir jetzt die ESC-Taste während des Spiels drücken, wird das ingame-Menü angezeigt – Super!

Die Funktionalität des ingame-Menüs

Der erste Button soll das Spiel ganz einfach fortsetzen. Dafür müssen wir das Menü einfach ausstellen, den freeze_mode wieder auf 0 setzen (damit es "weitergeht") und alle Menüelemente ausblenden. Dies erreichen wir durch folgende Funktion:

```
void menu_game_continueGame ()
{
   menu_mode = MENU_MODE_NONE;
   menu_hide_all();
   freeze_mode = 0;
}
```

Der Funktionszeiger von menu_game_continueGame muss nun noch in den "Continue"-Knopf eingebettet werden:

```
menu_game_continue = pan_createEx("button(0,0, menu_game_continue_n.tga, menu_game_continue_n.tga,
                     menu_game_continue_t.tga, menu_game_continueGame, NULL, NULL);",
                     201, NULL, 0, 0, 0);
```

Damit wir das Spiel auch wieder mit der Escape-Taste verlassen können, müssen wir ein wenig tricksen – es ist nämlich so: wenn wir mit ESC das Menü aufrufen, dann kann es sein, dass es dann aktiv ist und wie die ESC Taste immer noch gedrückt halten. In diesem Fall würde – wenn es schon eingebaut wäre, das Menü wieder schließen. Bevor wir also nun das Fortfahren mit der ESC Taste einbauen, wollen wir kurz schauen, wie wir das erwartete Problem lösen können. Zunächst richten wir eine Variable namens

```
var menu_game_keyEsc;
```

ein, die Aussage darüber geben soll, ob die Taste noch gedrückt wird oder nicht. In menu_game_open kann man dann beim Aufruf die Variable anschalten und beim Loslassen von ESC dann wieder ausschalten:

```
//...

if (key_esc) {
   freeze_mode = 1;
   menu_mode = MENU_MODE_GAME;

   menu_game_keyEsc = 1;
   while (key_esc) {wait(1);}
   menu_game_keyEsc = 0;
}
```

Wir können dann in menu_game_show die Abfrage nach ESC einbauen:

```
void menu_game_show ()
{
   //...
```

```
   if ((key_esc)&&(menu_game_keyEsc == 0)) {
      menu_game_continueGame();
   }
}
```

Jetzt können wir im ingame Menü auch das Menü mit ESC verlassen, allerdings haben wir den Nebeneffekt, den wir im Menü jetzt umgangen haben, nun auch im Spiel wieder. Deshalb fügen wir in menu_game_open eine Abfrage ein, ob die ESC Taste gedrückt wurde und frei ist, ein. Die Verzweigung in menu_game_show ändern wir dann so ab:

```
if ((key_esc)&&(menu_game_keyEsc == 0)) {
   menu_game_keyEsc = 1;
   menu_game_continueGame();
   while (key_esc) {wait(1);}
   menu_game_keyEsc = 0;
}
```

und ändern die If-Abfrage in menu_game_open ab:

```
void menu_game_open ()
{
   if (((Rudi != NULL) && (menu_mode == MENU_MODE_NONE)) &&
      (menu_game_keyEsc == 0))
   {
      //...
   }
}
```

und das Öffnen/Schließen mit ESC funktioniert nun einwandfrei!

Für die Anzeige der Konfigurations wollen wir - wie bereits erwähnt - auf den Konfigurationsbildschirm des Hauptmenüs zurückgreifen. Dazu reicht es im Prinzip, einfach nur den Modus für den Konfigurationsbildschirm zu setzen:

```
void menu_game_openConfig ()
{
   menu_mode = MENU_MODE_CFG;
}
```

und die Funktion in den Button reinzuschreiben:

```
menu_game_config = pan_createEx("button(0,0, menu_game_cfg_n.tga, menu_game_cfg_n.tga,
                   menu_game_cfg_t.tga, menu_game_openConfig, NULL, NULL);", 201, NULL, 0, 0, 0);
```

und tatsächlich rufen wir auch das Menü auf, wenn wir im ingame-Menü auf den Konfiguration-Button drücken. Allerdings stört es, dass der OK Knopf nicht wieder ins Game-Menü führt. Wir überprüfen mithilfe von Rudis Entity-Zeiger bereits, ob wir im Spiel sind, sodass wir dasselbe Verfahren auf den Konfigurationsbildschirm anwenden können.

Wir wollen eine Überprüfung, ob Rudi existiert, nutzen, um eine Verzweigung in der Funktion für den OK-Knopf zu bauen, die entscheidet, wohin der Konfigurationsschirm zurückführt: wenn Rudi existiert, dann leitet der OK-Knopf den Dialog zurück ins ingame-Menü. Dazu müssen wir die Funktion menu_cfg_main anpassen:

```
void menu_cfg_main ()
{
   if (!Rudi) {
      menu_mode = MENU_MODE_MAIN;
   } else {
      menu_mode = MENU_MODE_GAME;
   }
}
```

Uns fällt außerdem auch auf, dass die HUD-Elemente (die Herz- und die Lebensanzeige) ausgeblendet sind. Das macht natürlich keinen Sinn, also packen wir das auch in eine Verzweigung. Dazu müssen wir die Funktion menu_cfg_show anpassen:

```
void menu_cfg_show ()
{
   if (!Rudi) { //started from main menu
      hud_hide(); //hide HUD
   }

   menu_hide_all();        //hide all elements
   menu_cfg_setValues();   //set all configurable values
   menu_cfg_elements();    //show selected elements
   menu_cursor();          //draw cursor
}
```

Jetzt kann man aus dem ingame- und Hauptmenü aus den Konfigurationsbildschirm aufrufen, der sich dann dementsprechend anpasst und auch in das richtige Menü wieder zurückführt.

Damit das ingame-Menü auch wieder in das Hauptmenü führen kann, müssen wir lediglich nur den Menü-Modus umschalten. Ähnlich wie bei dem Gameover-Bildschirm leeren wir aber vorher noch das Level und heben dann noch den freeze_mode auf. Wir schreiben uns dafür die Funktion menu_game_mainmenu:

```
void menu_game_mainmenu ()
{
   menu_mode = MENU_MODE_MAIN;
   wait(1);
   level_load("");
   freeze_mode = 0;
}
```

und tragen die Funktion in den Button ein:

```
menu_game_back = pan_createEx("button(0,0, menu_game_back_n.tga, menu_game_back_n.tga,
                 menu_game_back_t.tga, menu_game_mainmenu, NULL, NULL);", 201, NULL, 0, 0, 0);
```

Nun können wir ganz einfach auch in das Hauptmenü zurückkehren!

Probleme

Wenn wir jetzt das Spiel starten und absichtlich sterben, in den Gameover-Bildschirm gelangen und dann in das Hauptmenü zurückkehren - oder - das Spiel über das ingame-Menü in das Hauptmenü beenden, stellen wir fest, dass, wenn wir das Konfigurationsmenü aufrufen, es so reagiert, als hätten wir es aus dem ingame-Menü heraus aufgerufen (anderer Hintergrund und wir sehen das ingame-Menü, wenn wir auf OK klicken). Aber wieso ist das so? Das hat doch bisher ganz gut geklappt!?!?

Das Problem ist, dass beim Start der engine der Zeiger noch = NULL ist und dann beim Start des ersten Levels gefüllt wird - soweit ist dass ja auch richtig. Wird das Level entfernt wird dann zwar die Entity von Rudi entfernt, der Zeiger zeigt aber immer noch auf einen Speicherbereich (auch wenn sich dahinter keine Entity mehr verbirgt). Deshalb ist der Zeiger immer noch ungleich NULL, sodass dass Konfigurationsmenü "denkt", dass es aus dem ingame-Menü heraus aufgerufen wurde, auch wenn der Benutzer sich zur Zeit im Hauptmenü befindet. Das kann man z.B. lösen, indem das Hauptmenü immer den Zeiger auf Rudi auf NULL setzt:

```
void menu_main_show ()
{
   Rudi = NULL;

   hud_hide();             //hide HUD
   menu_hide_all();        //hide all elements
   menu_main_elements();   //show selected elements
   menu_cursor();          //draw cursor
}
```

Damit wäre das Problem auch behoben. Aber an diesem Beispiel sieht man bestimmte Nebeneffekte wirken, die man so vorher nicht bedacht hat. Deshalb ist es gerade bei der Dialogführung in Anwendungen (und dazu gehören Spiele genauso) wichtig, sowas ausführlichst zu testen und frühzeitig das Problem zu erkennen.

Kapitel 10: Story und Credits

Jedes Spiel hat eine Geschichte zu bieten

In diesem Kapitel schreiben wir den Story-Bildschirm und die Credits. Zwar gehören die Credits zum Schluss des Spiels (und wir haben immer noch kein erstes Level), aber beide Dinge gehören halt zu einem Spiel, um es "rund" zu machen. Erst wenn wir sichergestellt haben, dass unser kleines Spiel "an sich" vollständig ist, wollen wir das erste Level erstellen und uns dann darauf konzentrieren.

Wenn der Spieler im Hauptmenü das Spiel startet, dann soll das erste Level gestartet werden. Viele Spiele, die eine Geschichte erzählen oder die auf irgendeiner Handlung oder Begebenheit basieren, erzählen diese, bevor es richtig losgeht - damit der Spieler weiß, worum es geht und was er machen muss. Anstatt also den Spieler direkt ins kalte Wasser zu werfen, wollen wir daa alles auch kurz erzählen, denn unser Spiel hat auch eine kleine Geschichte zu erzählen - Rudi rennt nämlich nicht ohne Grund durch die Gegend, um Geschenke zu sammeln. Zwar ist das Spielprinzip relativ leicht "intuitiv" erlernbar - die meisten Leute haben schon einmal Snake gespielt oder finden leicht heraus, dass man Geschenke sammeln muss - dennoch ist es einfach schöner die Story eben kurz anzureißen.

Man kann nun dafür verschiedene Methode benutzen. Früher war es üblich, für Zwischensequenzen und Intros sogenannte FMV's (engl.: Full Motion Video) zu benutzen. Mittlerweile sind die Anforderungen an Rechner höher und die Grafikqualität ist mittlerweile auch so hoch, dass man solche Sequenzen in Echtzeit mit der 3D-Engine darstellen kann. Allerdings ist es auch aufwendig, solche Szenen zu realisieren und optisch ansprechend zu gestalten. Je "kleiner" der Umfang ist, desto weniger lohnt es sich, sowas opulent aufzubereiten - zumindestens in unserem Fall ist es eher unnötig ein großartiges Intro zu produzieren (wobei nicht gesagt ist, dass dies nicht hübsch sei - entsprechend der Projektgröße ist die darauf verwendete Zeit nur unverhältnismäßig groß). Wir wollen eine etwas kleiner skalierte Herangehensweise bevorzugen.

Die Idee ist, dass wir die Geschichte auf einem Bildschirm darstellen. Es werden ein paar Bilder eingeblendet, die die Story umreißen und darstellen - im Zusammenspiel mit Untertiteln (für Dialoge z.B.). Die Bilder - ähnlich angeordnet wie in einem Comic blenden sanft hintereinander ein. Wurde das letzte Bild eingeblendet, wird der Spieler aufgefordert, eine Taste zu drücken, um fortzufahren (er kann dennoch den Bildschirm jederzeit mit einem Tastendruck überspringen) - daraufhin wird das erste Level geladen.

Der Storybildschirm enthält dementsprechend ein paar Bilder, die die Geschichte knapp zusammenfassen. Erst wird gezeigt, dass die Elfen die Pakete verloren haben, dann kriegt der Weihnachtsmann das mit und weiß nur, dass er es so nicht schaffen wir. Rudi bietet sich an und darauf sagt im der Weihnachtsmann, dass er am besten am Nordpol anfangen sollte (womit wir auch eine perfekte Anbindung zum ersten Level haben).

Dies soll später (nach Erscheinen aller Bilder) so aussehen:

Den Story-Bildschirm bereitstellen

Im Grunde genommen ist der Story-Bildschirm nichts anderes als ein normales Menü - welches allerdings selbstständig abläuft und nacheinander die Panels mit der Geschichte einblendet. Wir werden im Folgenden den Story-Bildschirm genauso einbauen wie die bisherigen Menüs, nur dass wir ihn etwas anders programmieren werden.

Zunächst richten wir einen neuen Modus für die Story in der menu.h ein:

```
var menu_mode;

   //(...)

   #define MENU_MODE_STORY    5
```

Der Bildschirm wird einen Hintergrund und die Story-Panels beinhalten. Deshalb fügen wir folgende Panels und deren Layoutinformation der menu.h hinzu:

```
PANEL* menu_story_background;
PANEL* menu_story_panelA;
PANEL* menu_story_panelB;
PANEL* menu_story_panelC;
PANEL* menu_story_panelD;

// (...)

VECTOR* menu_story_background_pos   = {x = 0;   y = 0;   z = 0;}
VECTOR* menu_story_panelA_pos       = {x = 5;   y = 9;   z = 0;}
VECTOR* menu_story_panelB_pos       = {x = 479; y = 30;  z = 0;}
VECTOR* menu_story_panelC_pos       = {x = 25;  y = 312; z = 0;}
VECTOR* menu_story_panelD_pos       = {x = 481; y = 374; z = 0;}
```

Bevor wir die Anzeige des Menüs programmieren, wollen wir erstmal alle Panels erzeugen und initialisieren. Die Grafiken dafür beginnen mit "menu_story" und werden im Menü-Ordner angelegt. Wir fügen folgende Einträge in die Funktion menu_init ein:

```
// STORY

   menu_story_background = pan_createEx("", 100, "menu_story_bg.tga", 0, 0, 0);
   menu_story_panelA = pan_createEx("", 101, "menu_story_picA.tga", 0, 0, 0);
   menu_story_panelB = pan_createEx("", 102, "menu_story_picB.tga", 0, 0, 0);
   menu_story_panelC = pan_createEx("", 103, "menu_story_picC.tga", 0, 0, 0);
   menu_story_panelD = pan_createEx("", 104, "menu_story_picD.tga", 0, 0, 0);
```

Jetzt haben wir auch Zugriff auf die Panels. In der Funktion menu_main rufen wir je nach angeschalteten Menümodus die jeweilige Anzeigefunktion des Menüs auf. Da wir einen neuen Modus hinzugefügt haben, müssen wir den zunächst einarbeiten:

```
void menu_main ()
{
   //(...)

   while (1) {

      //(...)

      switch (menu_mode) {

         //(...)

         case MENU_MODE_STORY:      menu_story_show();      break;
      }

      wait(1);
   }
}
```

Die Funktion, die menu_main dann aufrufen wird, lautet menu_story_show. Diese soll alle anderen Elemente ausstellen und nur die Elemente anzeigen, die für den Story-Bildschirm relevant sind. Im Gegensatz zu den anderen Menü-Bildschirmen soll aber keine Maus angezeigt werden. Die Funktion sieht so aus:

```
void menu_story_show ()
{
   menu_hide_all();        //hide all elements
   menu_story_elements();  //show selected elements

   menu_story_run();       //run story-script
}
```

In dieser Funktion sticht die Funktion menu_story_run heraus. Sie soll das Skript enthalten, die dafür sorgt, dass die Panels alle in einer zeitlichen Reihenfolge auftauchen. Außerdem soll die Funktion dafür sorgen, dass der Spieler den Story-Bildschirm überspringen darf. Bevor wir dort weitermachen, wollen wir den Story-Bildschirm erst einmal in das Menü integrieren und damit erreichbar machen.

Die Funktion menu_main_startGame hat bisher dafür gesorgt, dass mit dem Klick auf "START GAME" das erste Level geladen wird. Das wollen wir jetzt nicht, sondern den Story-Bildschirm laden. Dazu entfernen wir den Aufruf von game_start und fügen die Zuweisung des neuen Menümodus hinzu:

```
void menu_main_startGame ()
{
   menu_mode = MENU_MODE_STORY;
}
```

Das sollte vorerst reichen. Menu_main ruft dann korrekterweise den Story-Bildschirm auf.

Die Funktion menu_story_show ruft unter anderem die Funktion menu_story_elements auf, die die Elemente des Bildschirms anschalten soll. Diese sieht wie folgt aus:

```
void menu_story_elements ()
{
   // Background
      set(menu_story_background, VISIBLE);

   // Story panels
      set(menu_story_panelA, VISIBLE | TRANSLUCENT);
      set(menu_story_panelB, VISIBLE | TRANSLUCENT);
      set(menu_story_panelC, VISIBLE | TRANSLUCENT);
      set(menu_story_panelD, VISIBLE | TRANSLUCENT);
}
```

Wir schalten die Story-Panels nicht nur an, sondern auch auf TRANSLUCENT, indem wir beide mit dem | -Operator verknüpfen. Dies ist notwendig, weil wir die Panels einfaden wollen. Um die Transparenz eines Panels zu steuern, muss man das TRANSLUCENT flag gesetzt haben, bevor man die Durchsichtigkeit über den Alphawert des Panels angeben kann.

Damit die Panels auch alle ausgeschaltet werden können, müssen wir sie der Funktion menu_hide_all hinzufügen:

```
void menu_hide_all ()
{
   // (...)

   // STORY

      reset(menu_story_background, VISIBLE);
      reset(menu_story_panelA, VISIBLE);
      reset(menu_story_panelB, VISIBLE);
      reset(menu_story_panelC, VISIBLE);
      reset(menu_story_panelD, VISIBLE);

   // (...)
}
```

Das gleiche gilt für die auflösungsunabhängige Darstellung des Story-Bildschirms. Wir fügen die Panels und ihre

Layoutdaten in die Funktion menu_refresh_all ein:

```
void menu_refresh_all ()
{
   // (...)

   // STORY
      ri_refreshPanel(menu_story_background, menu_story_background_pos);
      ri_refreshPanel(menu_story_panelA, menu_story_panelA_pos);
      ri_refreshPanel(menu_story_panelB, menu_story_panelB_pos);
      ri_refreshPanel(menu_story_panelC, menu_story_panelC_pos);
      ri_refreshPanel(menu_story_panelD, menu_story_panelD_pos);

   // (...)
}
```

Wie wir die Geschichte erzählen

Wir wollen die Panels mit der Geschichte einzeln nach einer bestimmten Zeit einfaden lassen - und dies in einer bestimmten Reihenfolge. Um dies zu realisieren, brauchen wir eine Zeitvariable, die wie eine Art Timer funktioniert. Sie soll immer festhalten, wie "lang" die Story "schon geht", damit wir dann herauszufinden können, wann wir welches Panel einblenden können. Dafür definieren wir uns erstmal eine Variable namens

```
var menu_story_time;
```

in der "menu.h". Man könnte das Timer-Verhalten dann in der Funktion menu_story_run so ausdrücken:

```
void menu_story_run ()
{
   // Timer
      menu_story_time += time_step;
}
```

Wir addieren einfach immer die aktuelle Zeit hinzu. Das Problem ist, dass wir nicht sicher wissen, dass diese Variable beim Start der Story schon auf 0 stand oder nicht. Beim ersten Aufruf ist dies sicher der Fall. Aber wenn der Spieler aufgrund eines Gameover's das Spiel neu starten will, dann steht die Variable auf dem Zeitpunkt, an dem wir das letzte Mal den Story-Bildschirm verlassen haben und das Spiel begonnen haben. Deshalb fügen wir eine Initialisierungsfunktion hinzu:

```
void menu_story_init ()
{
   menu_story_time = 0;
}
```

Diese Funktion rufen wir auf, wenn wir den Storybildschirm starten:

```
void menu_main_startGame ()
{
   menu_story_init();
   menu_mode = MENU_MODE_STORY;
}
```

Das gleiche Problem für den Timer besteht auf für die Panels: weil wir sie einfaden, werden sie nach Beendigung der Story voll sichtbar bleiben. Deshalb müssen wir bei der Initialisierung den Alpha-Wert explizit auf 0 zurückstellen:

```
void menu_story_init ()
{
   menu_story_time = 0;

   menu_story_panelA->alpha = 0;
   menu_story_panelB->alpha = 0;
   menu_story_panelC->alpha = 0;
   menu_story_panelD->alpha = 0;
}
```

Um jetzt ein Panel nach einer bestimmten Zeit einfaden zu lassen, genügt es, den Timer mit einer Referenzzeit zu vergleichen um dann - falls wir über dieser Zeit liegen - das Panel einfaden zu können. Das kann man z.B. in eine If-Verzweigung schreiben, die dann so aussehen würde:

```
if (menu_story_time > 3 * 16) {
   menu_story_panelA->alpha += 5 * time_step;
}
```

Der Ausdruck 3 * 16 gibt 3 Sekunden an (16 Ticks = eine Sekunde, 3 mal 16 Ticks = 3 Sekunden) - die Verzweigung wird also ausgeführt, wenn der Timer über diesem Zeitpunkt liegt. In diesem Fall wird das Panel A mit 5 Prozent zusätzlicher Sichtbarkeit pro Tick eingeblendet. Bei einem Panel mag diese Verzweigung noch gehen, aber bei mehreren bläht sich der Code immer mehr auf. Wir können dieses Konstrukt auch in eine Zeile packen und zwar so:

```
menu_story_panelA->alpha += (menu_story_time > 3 * 16) * 5 * time_step;
```

Wir addieren also immer etwas auf die Alpha des Panels. Nur die Frage ist: nur was? Wir haben den Ausdruck aus der Verzweigung nun in die Berechnung der Addition eingefügt, was im ersten Moment komisch aussieht. Es ist nun aber so: jeder Wahrheitswert wird in Lite-C implizit als Integerwert aufgefasst - und zwar als 1 oder 0. Wir haben den Wahrheitswert nun in eine Multiplikation eingebaut, die bekanntlich komplett = 0 wird, wenn nur ein Element des Terms = 0 ist. Wenn wir also noch nicht die Zeitschwelle von 3 Sekunden überschritten haben, wir der Wahrheitswert gleich Null und wir addieren gar nichts auf die Alpha des Panels. Andernfalls wird der ursprüngliche Wert mit 1 multipliziert - was wiederum der Wert ist. Ziemlich praktisch das Ganze!

Wir können dies nun für alle Panels so zusammenbauen, indem wir die Bedingung, wann sie jeweils einfaden, in den Einblende-Code an sich einfügen. Anstatt nun feste Werte zu nehmen, werden wir wieder die Werte wie bisher auslagern:

menu.c:

```
void menu_story_run ()
{
   menu_story_time += time_step;

   menu_story_panelA->alpha += (menu_story_time > menu_story_panelA_start * 16) *
                               menu_story_panelA_speed * time_step;

   menu_story_panelB->alpha += (menu_story_time > menu_story_panelB_start * 16) *
                               menu_story_panelB_speed * time_step;

   menu_story_panelC->alpha += (menu_story_time > menu_story_panelC_start * 16) *
                               menu_story_panelC_speed * time_step;

   menu_story_panelD->alpha += (menu_story_time > menu_story_panelD_start * 16) *
                               menu_story_panelD_speed * time_step;
}
```

menu.h:

```
var menu_story_panelA_start = 1; //start time in seconds
   var menu_story_panelA_speed = 1.5; //fade speed in percent per tick

var menu_story_panelB_start = 6;
   var menu_story_panelB_speed = 1.5;

var menu_story_panelC_start = 10;
   var menu_story_panelC_speed = 1.25;

var menu_story_panelD_start = 14;
   var menu_story_panelD_speed = 1;
```

Wenn man nun im Hauptmenü auf START GAME klickt, sieht man den Storybildschirm und die einfadenden Panels. Die Werte können natürlich beliebig verändert werden, wenn es Unstimmgkeiten mit der durchschnittlichen Lesegeschwindigkeit des Spielers gibt oder soetwas - dafür wurden die Werte der Panels ausgelagert!

Den Story-Bildschirm beenden

Das Einzige, was stört, ist die Tatsache, dass das Spiel nicht startet und wir noch nicht die Story-Sequenz überspringen können. Wir fügen eine Abfrage der Tasten in die Funktion menu_story_run ein, die das regelt und dann selber das Spiel startet:

```
void menu_story_run ()
{
   // (...)

   // Cancel story and start game when anykey is pressed
      if (key_any) {
         game_start(); //start game
      }
}
```

Wenn wir nun im Story-Bildschirm eine Taste drücken, sollte das Spiel starten. Also probieren wir das mal aus. Es kann jetzt passieren, das beim Ausprobieren, wenn wir auf START GAME klicken, wir kurz den Hintergrund des Story-Bildschirms sehen und dann sofort das Level startet. Aber wieso?!

Das Problem ist, das Klicks der Maus auch als Tastenanschläge gewertet werden. Wir werden uns jetzt eine Hiflskonstruktion bauen, mit der wir dieses Problem umgehen. Die Idee ist, dass wenn der Spieler auf START GAME klickt, die Maus, bzw. jede Taste solange gesperrt bleibt, bis keine Taste mehr gedrückt wird - was u.U. sein kann, auch wenn die Story schon läuft. Dazu nutzen wir die Funktion menu_story_init aus, die nur einmal aufgerufen wird. Wir werden dort eine Variable ausstellen und solange warten, bis keine Taste mehr gedrückt ist. Dann erst schalten wir diese Variable an. Erst wenn sie an ist, erlauben wir dem normalen Story-Code, zu checken, ob eine Taste gedrückt wurde - wir wissen nämlich dann, dass auf jeden Fall die zuvor gedrückte Taste losgelassen worden ist.

Die Variable, die dies signalisiert, nennen wir

```
var menu_story_anykey_ready;
```

In der Funktion menu_story_init schreiben wir dann:

```
void menu_story_init ()
{
   // (...)

   // Wait until the key has been released
      menu_story_anykey_ready = 0;
      while (key_any) {wait(1);}
      menu_story_anykey_ready = 1;
}
```

Solange also irgendeine Taste (in diesem Fall die linke Maustaste z.B.) gedrückt bleibt, bleibt menu_story_anykey_ready auf 0 stehen. Dies können wir nun in der Funktion menu_story_run ausnutzen:

```
void menu_story_run ()
{
   // (...)

   // Cancel story and start game when anykey is pressed
      if ((menu_story_anykey_ready) && (key_any)) {
         game_start(); //start game
      }
}
```

Übersetzt heißt das: erst wenn der Spieler die Tastatur wieder freigegeben hat und genau dann eine Taste drückt, starten wir das Spiel. Was aber passiert nun, wenn der Benutzer keine Taste drückt und sich die Story komplett anschaut? Nunja, die Story wird ewig weiterlaufen. Ungeschulte PC Nutzer oder -spieler wüssten vielleicht nicht, dass sie eine Taste zu drücken haben, also bauen wir eine automatische Abschaltung hinzu. Wir lassen das Intro dann einfach ein paar Sekunden nach dem letzten Panel beenden.

Dazu führen wir einen neuen Wert ein, der diese Zeitgrenze in Sekunden ausdrückt:

```
var menu_story_end = 20;   // after x seconds we start the game
```

und arbeiten ihn in unsere bisherige Abbruchbedingung ein:

```
void menu_story_run ()
{
   // Cancel story and start game when anykey is pressed
   // After a certain time, we automatically start the game

      if (((menu_story_anykey_ready) && (key_any)) || (menu_story_time > menu_story_end * 16)) {
         game_start(); //start game
      }
}
```

Wir verknüpfen die Bedingungen korrekterweise mit einer ODER Verknüpfung, weil jeweils nur einer dieser beiden Zustände eintreten braucht, um den Story-Bildschirm abzubrechen.

Damit ist der Storybildschirm vollständig!

Der Abspann

Wenn man sich im Kino einen Film anschaut, sieht man meistens am Ende des Films die sogenannten Credits: eine Auflistung aller Schauspieler und aller Personen, die an dem Film beteiligt waren. Genau dies ist auch bei Computerspielen üblich - denn niemand entwickelt ein Spiel gänzlich allein (auch wenn manche dass so behaupten). Auch wenn es nur Danksagungen sind, sollte man auf die Menschen Rücksicht nehmen und sie zumindestens kurz erwähnen. Bei großen Spieleproduktionen ist so eine Liste irre lang, weil heutzutage meistens ein paar Dutzend Menschen an einem Spiel arbeiten. Bei Amateuren ist es dennoch meistens so, dass einige Leute Aufgaben in mehreren Bereichen übernehmen (dies war auch der Fall bei der Erstellung dieses Spiels). Es kann leicht lächerlich enden, wenn man dann für jeden Aufgabenbereich seinen eigenen Namen hinschreibt, aber sobald man mit mehreren Menschen zutun hat, freuen die sich auch, genannt zu werden.

In manchen Fällen ist dies sogar ein Muss, wenn man z.B. Content wie z.B. Musik, Texturen oder Modelle von Anderen benutzt: die Erwähnung in den Credits kann die "Bezahlung" dafür sein, dass man den Content benutzen darf. Lizenzmodelle wie die Creative Commons Lizenzen beinhalten z.B. immer die Erwähnung des Autors bei Benutzung.

Die Credits umfassen in aller Regel nochmal den Titel des Spiels und dann alle Beteiligten. Wie man diese Liste unterteilt ist komplett frei.. man kann einfach alle Beteiligten auflisten, meistens jedoch werden die Mitarbeiter in Kategorien eingeteilt oder mit ihrem Titel genannt. Meistens berücksichtigt man generell Kategorien, die das Management, das Spieldesign, die Programmierung, das Testing, die Grafik, die Vertonung u.ä. berücksichtigen. Meistens teilt man die Kategorien in Unterkategorien auf und teilt dann nochmal in unterschiedliche Mitarbeiterstäbe auf. Diese Kategorisierung kann man quasi beliebig fortführen - wird ein Spiel z.B. synchronisiert, wird dies auch nochmal in einer seperaten Sektion aufgeführt. Je "kleiner" das Projekt, desto kleiner ist die Liste und andersherum. Es ist wichtig, die Liste genau und fair zu handlen, sodass sich niemand auf den Schlips getreten fühlt und dass alle eventuell zu berücksichtigbaren Lizenzen wiederfinden.

Wir wollen für unser Spiel auch solche Credits bauen und in das Spiel integrieren. Die Credits sollen einmal über das Menü erreichbar sein und automatisch abgespielt werden, wenn der Spieler das Spiel durchgespielt hat (analog zum Kinofilm). In erster Linie dienen die Credits in diesem Fall der Auflistung aller Leute, die geholfen haben, das Spiel und diesen Workshop zu realisieren, aber es ist durchaus auch für Andere interessant.

Ich habe sehr lange darüber nachgedacht, wie ich die Credits am besten löse - und dass so einfach, elegant und flexibel wie möglich. Bevor ich meine Entscheidung präsentiere, möchte ich alle Alternativen aufzählen und die Vor- und Nachteile betrachten. Bei den meistens Problemstellungen gibt es immer mehrere Optionen, wie man das Problem lösen kann - und mit den Credits haben wir ein klassisches Beispiel!

Zunächst einmal muss man festhalten, dass die Credits - in aller Regel - nur ein sehr langer Text sind, der angezeigt

wird. Meistens in Form einer Art "Schriftrolle", die sich von unten nach oben bewegt (wie bei einem klassischen Kinofilm halt). Texte kann man in der A7 prima - wie soll es auch anders sein - mit TEXT* Objekten darstellen. Die Objekte nehmen einen oder mehrere Strings und stellen die Strings mit einem Font auf dem Bildschirm dar. Wenn wir das so machen würden, könnten wir z.B. einen schönen Bitmapfont bauen und an unser Text-Objekt übergeben. Wir könnten dann auch - falls sich die Credits irgendwie ändern - einfach die String ohne großen Aufwand verändern. Der Nachteil ist, dass Bitmapfont sogenannten Monospace-Fonts sind: jedes Zeichen hat eine feste Breite. Wenn wir in den Credits mehrere Schriftzüge hätten (etwas kleiner/größer, fett/normal/kursiv, andere Farben..) dann müssten wir mehrere Fonts einrichten. Die Programmierung wäre dann komplizierter, weil wir dann mehrere TEXT* Objekte bauen müssten usw.

Eine andere Idee ist, das Ganze bildbasiert zu realisieren. Anstatt also den Text in Strings zu packen, erzeugen wir in einem geeigneten Bildprogramm einmal die komplette Creditsliste auf einem sehr großen Bild und schneiden dann hinterher Teilbilder heraus (z.B. einzelne Kategorien). Diese Teile werden dann wieder auf dem Bildschirm zusammengesetzt, indem wir Panels damit füllen (kompliziert!) oder die Bilder als Sprites in einer Leveldatei anordnen und dann mit der Kamera entlangfahren (einfacher!). Wir hätten somit eine natürlichere Textdarstellung als mit Monospace-Bitmapfont und bei der Gestaltung hätten wir freie Hand was die Font-Auswahl angeht und wie der Text formatiert wird. Der Nachteil liegt auf der Hand: der Speicherverbrauch steigt sprunghaft, weil wir einige Bilder in den Speicher laden müssen anstatt einen Text darzustellen. Außerdem muss man jedes Mal, wenn sich was an der Credits-Liste ändert, den text in dem Zeichenprogramm ändern, das Teilbild erneut exportieren und in der Szene neu arrangieren.

Ein Spagat zwischen den beiden Methoden ist die Benutzung einer Proportionalen Schriftart als Font für TEXT* Objekte. Dies ist möglich in der A7 und bietet im Vergleich zu den Bitmapfonts eine natürlichere Zusammensetzung der einzelnen Zeichen und reagiert genauso dynamisch auf die Strings wie die erste Variante. Allerdings haben sogenannte TrueType-Fonts auch Nachteile: ersteinmal sind sie viel langsamer, was allerdings zu vernachlässigen wäre. Ein großes Problem ist, dass so ein TrueType-Font auf dem System des Benutzers installiert sein muss, sodass nur Standard-Fonts wie z.B. Arial oder Times New Roman in Frage kämen - andernfalls müsste man bei der Installation dafür sorgen, dass der Font bereits auf dem System existiert. Die Programmierung könnte noch komplizierter sein als bei Bitmapfonts und TrueType-Fonts sind nur unifarben und nicht frei gestaltbar.

TrueType-Fonts haben zur Zeit noch zuviele Mankos, sodass die Möglichkeit ausscheidet. Allerdings sehen Monospace-Fonts wirklich nicht sehr gut aus - aus diesem Grund sind bisher alle Schaltflächen usw. auch mit Bildern erstellt als mit Texten. Der größte Nachteil der bildbasierten Lösung ist in der Tat der vermeintlich hohe Speicherverbrauch. Tatsächlich ist das auch so, wenn wir mit BMP oder TGA Dateien arbeiten würden. Das finale Spiel wird aber überwiegend mit DDS Grafiken arbeiten - womit die Bilder komprimiert sind. Der Speicherverbrauch sinkt drastisch und daher ist dies auch kein schwerwiegendes Argument mehr.

Wir werden also das Ganze bild-basiert lösen.

Die Ur-Fassung der Credits erstellen

Die Creditsliste in Textform sollte man spätestens gegen Ende der Entwicklung vorliegen haben, sodass man - aufgrund unserer bildbasierenden Lösung - nur wenige Male die ursprüngliche Grafik erstellen und dann auseinanderschneiden muss. In einem geeigneten Grafikprogramm kann man dann - basierend auf einer Breite von 1024 Pixeln als Referenzauflösung! - einen Text erstellen, so wie die Creditsliste dann aussehen soll. Man tippt einfach den ganzen Text ab und sorgt dafür, dass alles auf der Breite von 1024 Pixeln zur Geltung kommt.

Hat man das geschafft, wird man nicht ganz so wahnsinnig sein und versuchen diese Grafik (die um einige Male länger als die Breite sein wird!) komplett in den Speicher zu laden. Vielmehr muss man nun dafür sorgen, dass aus allen wesentlichen Sektionen jede Kategorie mit den Beteiligten herausgeschnitten wird. Wenn man z.B. durch das Textobjekt eine Alphamaske erzeugt, kann man einfach die Maske editieren um den gewünschten Textteil zu markieren und dann in ein eigenes Bilddokument auszulagern. Der Credits-Text ist in der Regel zentriert, sodass der ausgeschnittene Text auch horizontal zentriert sein sollte.

Texturdaten, die geladen werden, sollten in der Regel wohlgeformt sein: dies bedeutet, dass die Breite und die Höhe der Grafik einer Zweierpotenz (..., 64, 128, 256, 512, 1024,...) entsprechen muss (Panel-bitmaps dürfen anders geformt sein!) Das bedeutet, dass das Bild, dass den ausgeschnittenen Text beinhaltet, auf ein solches Maß

erweitert werden soll (aber nicht durch resampling - dann wird nämlich der Text auch gestrechedt!). Wenn Sie nicht wissen, wie das geht, informieren Sie sich in der Anleitung Ihres Programms oder suchen Sie nach entsprechenden Tutorials. Jedes nicht-triviale Grafikprogramm unterstützt diese Vorgänge - einschließlich der Arbeit mit Alphamasken. Wenn Ihr Text immer noch eine Alphamaske hat, so können Sie diese z.B. beibehalten, wenn Sie die Credits z.B. vor einer Hintergrundgrafik abspielen (was wir nicht tun werden und daher auch nicht behandeln).

Die Credits erhalten im Game-Ordner ein eigenes Verzeichnis namens "Credits". Dort können wir nun die Teilgrafiken abspeichern. Wir geben den Grafiken ein Prefix "cred_" für "Credits" und schreiben dahinter ein Kürzel. Ich habe Meine Sektionen mit Buchstaben getrennt und dahinter eine Zahl für die x-te Grafik gegeben, z.B. "cred_B8" für die achte Grafik der zweiten Sektion.

Der Ablauf des Abspanns

Der Ablauf ist genau wie die Gestaltung und der Umfang der Liste völlig frei gestaltbar. Ich habe mich für eine klassische Weise entschieden, die sogar einigermaßen schick aussieht. Die Reihenfolge sieht wie folgt aus:

1) Der Hintergrund ist schwarz und das Spiellogo wird groß eingeblendet. Eine eigene Credits-Musik wird gestartet.

2) Wenn das Logo eingeblendet wurde, bleibt es kurz sichtbar. Danach startet der eigentliche Vorgang des Creditsliste.

3) Die ganzen Creditselemente (inkl. dem Logo) sind in einem Level als Sprites untereinander horizontal zentriert angeordnet. Die Kamera bewegt sich nach unten und fährt somit an allen Elementen vorbei.

4) Die Elemente werden im Mittelpunkt des Bildschirms voll sichtbar sein, während sie - wenn sie von unten kommen oder nach oben verschwinden - leicht transparent werden. Dies erhöht die Aufmerksamkeit des Zuschauers.

5) Die Creditsliste fährt vollständig aus dem Bildschirm heraus, sodass am Ende alles Schwarz ist.

6) Dann blendet ein "thanks for playing!"-Symbol ein, als Dank des Entwickler an den Spieler, dass er das Spiel durchgespielt hat.

7) Das Symbol fadet wieder aus und man kehrt zurück zum Menü.

Die Creditsliste von unserem Team enthält 3 seperate Sektionen: das Logo, die Game Credits und die Workshop Credits. Bei Ihnen kann das ganz anders sein! Deshalb sind die Credits-Grafiken mit A, B und C untergliedert. Das "thanks for playing!"-Symbol erhält den Dateinamen "cred_tfp". Wir erstellen mit dem Leveleditor nun eine WMP Datei im Ordner "Credits" und nennen sie "credits.wmp". Damit man die einzelnen Sprites besser anordnen kann, habe ich mir die gesamte Creditsgrafik einmal als niedrigauflösende Grafik als Sprite ins Level geholt und mit der 3D Ansicht die Sprites so angeordnet, dass es genau gepasst hat. Es ist wichtig, dass die Sprites alle auf der Y- und X-Achse die Position 0 haben, damit sie von der Kamera hinterher genau gleich entfernt sind und zentriert sind. Um den Ablauf jetzt zu realisieren, müssen wir uns einige Entity-Actions schreiben. Damit diese Actions ind er WMP Datei verfügbar sind, erstellen wir im Credits-Ordner eine Datei namens "actions.wdl", in der wir dann die verfügbaren Entity-Funktionen dann als Action-Prototypen verfügbar machen werden (wie wir es bereits im Level-Ordner getan haben).

Die Programmierung der Credits

Als erstes legen wir die Dateien "credits.c" und "credits.h" im game-Ordner an und inkludieren sie in der Datei "rudi.c" nach dem bereits mehrfach durchgeführten Prinzip. Damit wir das Level hinterher auch laden können, müssen wir den Ordner zum Dateisystem hinzufügen und machen dies in der "game.c" in der Funktion game_init:

```
void game_init ()
{
   // (...)

      add_folder("game\\credits");

   // (...)
}
```

Wir wollen als erstes die Credits so programmieren, sodass sie einwandfrei aus dem Menü heraus funktionieren. Dort haben wir bereits den Knopf dafür vorgesehen, allerdings noch nicht programmiert. Dafür werden wir analog zu den anderen Button-Events die Funktion menu_main_credits vorsehen, die dies erledigt. Da die Credits aber auch von Rudi heraus aufgerufen wird (nach dem letzten Level), werden wir in einer seperaten Funktion festhalten, wie die Credits gestartet werden, und zwar in der Funktion cr_load, die wir dann auch in der Funktion menu_main_credits ausführen werden:

menu.c:

```
void menu_main_credits ()
{
   cr_load();
}
```

credits.c:

```
void cr_load ()
{
   menu_mode = MENU_MODE_NONE;
   menu_hide_all();
   hud_hide();
   level_load("credits.wmb");
}
```

Wir schalten in der cr_load den Menü-Modus aus und lassen das Menü und evtl. offene HUD-Elemente verschwinden. Daraufhin laden wir das Level. Die Funktion menu_main_credits muss nun nur noch von dem Credits-Knopf ausgeführt werden. Dazu betten wir den Funktionszeiger in die Button-Definition in der Funktion menu_init ein:

```
menu_credits = pan_createEx("button(0,0, menu_credits_n.tga, menu_credits_n.tga, menu_credits_t.tga,
                             menu_main_credits, NULL, NULL);", 101, NULL, 0, 0, 0);
```

Damit werden die Credits nun gestartet. Viel mehr als dass das Level gestartet wird, passiert nicht. Dafür sorgen wir jetzt!

Als erstes wollen wir eine Dummy-Entity setzen, die die Kamera positioniert (mit dem Blick auf das "RUDI"-Logo) und die Credits startet. Ich habe mir einen kleinen Würfel im MED gebastelt und im Creditsordner als "dummy.mdl" gespeichert, den ich dafür benutzen will. Den Platzieren wir im Leveleditor nun dort, wo die Kamera startet. Wir schreiben für die Dummy-Entity folgende Funktion in der credits.c:

```
void cr_camStart ()
{
   // Initialization

      set(my, INVISIBLE);                 // Dummyentity should be invisible
      camera.arc = cr_cam_arc;            // Reset camera arc

      vec_set(sky_color, cr_sky_color);   // Solid sky

      vec_set(camera.x, my.x);            // Place camera
      vec_set(camera.pan, my.pan);
}
```

Damit wird die Dummy-Entity ausgeblendet und die Kamera eingestellt: sie wird platziert und der Winkel wird zurückgesetzt. Da wir ein ansonsten leeres Level haben, wird uns aller Vorraussicht nach ein unpassendes blau entgegenscheinen. In sky_color wird die Hintergrund eines Levels angegeben, falls das Level keinen Skycube

besitzt. Die Konstanten sind in der credits.h wie folgt definiert:

```
int      cr_cam_arc = 60;
VECTOR*  cr_sky_color = {x = 1; y = 1; z = 1;}
```

wobei cr_sky_color nicht RGB = 0,0,0 ist, weil dann der Hintergrund wieder blau wäre (RGB = 0,0,0 bedeutet, dass der Hintergrund nicht gefärbt wird). Wir erstellen in der actions.wdl den Prototypen

```
action cr_camStart   {wait(1);}
```

und weisen die Funktion der Dummy-Entity zu. Alle folgenen Funktionen werden auch Entity-Actions sein, die wir für die Credits schreiben werden - sie werden auch alle extra als Action Prototype in der actions.wdl gelistet werden müssen!

Wenn wir die Credits nun starten, ist die Kamera schonmal richtig positioniert und man sieht das Logo!

Keep on rolling!

Als nächstes müssen wir das Verhalten der Credits einprogrammieren. Zunächst kümmern wir uns um das Logo. Wir erstellen eine Funktion für das Logo namens cr_logo - und weisen die Action dann dem Sprite im WED zu. Es soll erst einmal einfaden und kurz warten, bevor es losgeht. Die Funktion sieht so aus:

```
void cr_logo ()
{
   // Initialization

      set(my, TRANSLUCENT);      // Alpha enabled
      my.alpha = 0;              // Invisible
      my.material = mat_cr_elem;

      wait(1);

      my.z = camera.z;

   // Wait a bit before I fade in
      wait(-cr_logo_waitStart);

   // Fade in
      while (my.alpha < 100) {
         my.alpha += cr_logo_fadeSpeed * time_step;
         wait(1);
      }

   // Stay a bit
      wait(-cr_logo_stay);
}
```

Zunächst wird es initialisiert, indem Transparenz angeschaltet wird und das Logo unsichtbar gemacht wird durch einen Sichtbarkeitswert von alpha = 0. Wir weisen dem Logo auch ein Material namens mat_cr_elem zu, dass in der credits.h definiert ist:

```
MATERIAL* mat_cr_elem = {

   ambient_red    = 250;
   ambient_green  = 250;
   ambient_blue   = 250;

   diffuse_red    = 0;
   diffuse_green  = 0;
   diffuse_blue   = 0;

   specular_red   = 0;
   specular_green = 0;
   specular_blue  = 0;

   emissive_red   = 32;
```

```
      emissive_green = 32;
      emissive_blue  = 32;

      albedo   = 50;
      power = 0;
   }
```

Weil wir für das Logo einen Sprite benutzen, wird es auch schattiert wie ganz normale Entities, wodurch der Sprite etwas dunkler erscheint. Mit diesem Material schalten wir das shading aus und die Grafik wird korrekt beleuchtet. Dann zentrieren wir das Logo sodass die Kamera genau draufschaut. Daraufhin warten wir kurz und dann faden wir das Logo ein, bis es vollständig sichtbar ist. Danach ist ein weiteres Wait eingebaut, dass die Ausführung kurz pausiert. Danach soll das Rollen der Credits beginne (dazu gleich mehr). Die Konstanten sind wie folgt ausgelagert:

```
   var cr_logo_waitStart = 1;       // in secs
   var cr_logo_fadeSpeed = 5;
   var cr_logo_stay = 1;            // in secs
```

Wir wollen den Start des Rollens in der Funktion cr_camStart realisieren. Damit die Funktion weiß, wann das Logo soweit ist, wollen wir eine globale Flag-Variable namens cr_logoReady benutzen, die in diesem Fall auf 1 gesetzt wird. Das Logo setzt die dann einfach:

credits.h:

```
      var cr_logoReady;
```

credits.c:

```
      //(...)

      // Stay a bit
         wait(-cr_logo_stay);

      // OK, lets start rolling down
         cr_logoReady = 1;
   }
```

und die Funktion cr_camStart wartet darauf. Wenn das Flag an ist, wird das Rollen gestartet, was supereinfach realisiert wird, indem die Kamera konstant nach unten bewegt wird:

```
      // (...)

      // Wait until logo is ready
         while (!cr_logoReady) {wait(1);}

      // Move down
         while (1) {
            camera.z -= cr_cam_rollSpeed * time_step;
            wait(1);
         }
   }
```

wobei die Roll-Geschwindigkeit so definiert ist:

```
   var cr_cam_rollSpeed = 2.5;
```

Die Elemente sollen etwas ein- und ausfaden, wenn sie sich am Bildschirmrand befinden. Außerdem sollen sie solange unsichtbar bleiben, wie das Rollen noch nicht angefangen hat. Die Funktion, die wir cr_element nennen und allen Elementen im WED zuweisen, ist relativ simpel gestrickt - bis auf das Fading:

credits.c:

```
      void cr_element ()
      {
         // Initialization

            set(my, TRANSLUCENT);         // Alpha enabled
```

```
      my.alpha = 0;                 // Invisible
      my.material = mat_cr_elem;

      wait(1);

   // Wait until logo has been shown
      while (!cr_logoReady) {wait(1);}

   // Fade according to the cam position
      while (1) {

         // Relative alpha depending on position to cam
            my.skill1 = (1 - (minv(abs(my.z - camera.z), cr_elem_fadeDist) /
                        cr_elem_fadeDist)) * 100;

         // Absolute difference alpha
            my.skill2 = my.skill1 - my.alpha;

         // Interpolate from current to new alpha by 25%
            my.alpha += cr_elem_fadeFactor * time_step * my.skill2;

         wait(1);
      }
}
```

credits.h:

```
var cr_elem_fadeDist = 350;
var cr_elem_fadeFactor = 0.25;
```

Nachdem es ähnlich wie das Logo initialisiert wurde, wartet es - wie die Kamera-Entity - darauf, das es losgeht. In der Schleife wird dann nur der aktuelle Alphawert berechnet. Wir berechnen den aktuellen Alphawert und interpolieren dann vom aktuellen zu dem gewünschten. Das brauchen wir deshalb, damit beim Start nach dem Logo darunter liegende Sprites (die vielleicht schon sichtbar sind), nicht plötzlich aufpoppen.

Die Berechnung funktioniert in 3 Schritten und zwar so: zunächst legen wir einmal fest, dass wenn der Sprite auf der Höhe der Kamera ist, voll sichtbar (alpha = 100) ist. Ist er außerhalb einer bestimmten Distanz, ist der Sprite unsichtbar (alpha = 0). Dieser Wert steht in cr_elem_fadeDist. Zunächst berechnen wir die Distanz zur Kamerahöhe. Liegen wir außerhalb des Radius, sagen wir, dass wir genau auf dem Rand des Radius liegen. Also liegen wir immer zwischen 0 und dem Radius wenn wir rechnen:

```
minv(abs(my.z - camera.z), cr_elem_fadeDist)
```

Wenn wir diesen Wert durch den Radius teilen (und damit in Relation setzen), erhalten wir einen Wert zwischen 0 und 1, wobei 1 genau auf dem Rand wäre (also außerhalb) und 0 genau auf Kamerahöhe wäre. Wenn wir diesen Wert von 1 abziehen, drehen wir das um: also kommt 1 heraus wenn wir genau auf Kamerahöhe liegen und zum Rand hin wird es eine 0. Wenn wir den Wert mit 100 multiplizieren erhalten wir den Alphawert, der 100% beträgt, wenn wir auf Kamerahöhe liegen und entsprechend 0 auf dem Rand des Radius oder weiter weg. Im Skill1 wird als gespeichert, wie transparent der Sprite zu sein hätte.

In der Zeile

```
my.skill2 = my.skill1 - my.alpha;
```

rechnen wir aus, um wieviel sich der aktuelle Alphawert verändern müsste, um auf diesen Wert zu kommen und speichern diese Differenz in Skill2. Um jetzt den Alphawert zu interpolieren, skalieren wir den Differenzwert mit einem kleinen Faktor und korrigieren ihn über die Zeit mit time_step. Damit ist das fading beschrieben!

Damit das Logo genaus wie die Elemente ausfadet, müssen wir cr_element einmal nach dem Start des Rollens aufrufen. Weil cr_element den Alpha-Wert intial auf 0 stellt, das Logo aber bereits auf 100 steht und nur ausfaden soll, müssen wir das eben nachkorrigieren:

```
// OK, lets start rolling down
   cr_logoReady = 1;
```

```
   // Fading behaviour like common elements
      cr_element();
      my.alpha = 100; //overwrite alpha
}
```

Damit haben wir ein schönes Fading der Elemente und des Logos - super!

Der Stopper und das Dankeschön

Damit die Kamera nach dem letzten Element auch stehen bleibt (und das Dankeschön anzeigt), müssen wir einer Art "Show-Stopper" programmieren. Am Einfachsten machen wir es, indem wir einfach eine Höhe angeben, die den Stop verursachen soll. Damit wir das nicht per Hand eintippen müssen, erzeugen wir uns eine Dummy-Entity, die wir im WED dafür rumschieben können. Wir wollen eine Variable definieren, die dann diesen Höhenwert festhält, namens cr_stop_z. Die Funktion für die Dummy-Entity nennen wir cr_stop - die wir einer Entity (-> "dummy.mdl") im WED natürlich auch zuweisen. Die Entity muss ein bißchen unter dem letzten Element platziert werden, sodass das letzte Element noch aus dem Bildschirm rausrollen kann.

Die Funktion ist simpel und lautet:

```
void cr_stop ()
{
   cr_stop_z = my.z;    // Set show-stopper height
   ent_remove(my);      // Remove dummy entity
}
```

Damit die Kamera auch aufhört zu rollen, müssen wie die Bedingung der Roll-Schleife abändern:

```
while (camera.z > cr_stop_z) {
   //(...)
}
```

Wenn die Credits abgerollt sind, soll als Dankeschön ein "thanks for playing!"-Symbol auftauchen. Ähnlich wie die Kamera auf das Logo wartet, lassen wir das Symbol auf die Kamera warten. Wir legen dazu das Logo unter dem letzten Element an und schreiben eine Funktion namens cr_tfp (wobei das Kürzel tfp für "thanks for playing!" steht) und weisen sie zu. Das Logo soll unsichtbar sein, warten bis die Kamera gestoppt wurde und sich dann vertikal zentriert präsentieren, indem es einfadet.

credits.c:

```
void cr_tfp ()
{
   set(my, TRANSLUCENT);
   my.alpha = 0;

   // Wait one frame so that the camera is placed
      wait(1);

   // Wait until camera has stopped
      while (camera.z > cr_stop_z) {wait(1);}

   // Center the image vertically
      my.z = camera.z;

   // Fade in
      while (my.alpha < 100) {
         my.alpha += cr_tfp_fadeInSpeed * time_step;
         wait(1);
      }
}
```

credits.h:

```
var cr_tfp_fadeInSpeed = 2;
```

Danach soll das Symbol kurz warten und danach ausfaden. Wenn es aufgefadet ist, soll es eine ganz kurze Verzögerung geben, bevor man wieder zum Hauptmenü dirigiert wird. Dazu erweitert man die Funktion wie folgt:

credits.c:

```
      // (...)

      // Stay a bit
         wait(-cr_tfp_stay);

      // Fade out
         while (my.alpha > 0) {
            my.alpha -= cr_tfp_fadeOutSpeed * time_step;
            wait(1);
         }

      // Wait a bit
         wait(-cr_tfp_waitAfter);

      // Go back to main menu
         menu_main();
   }
```

credits.h:

```
   var cr_tfp_stay = 4;             // in secs
   var cr_tfp_fadeOutSpeed = 1.25;
   var cr_tfp_waitAfter = 1;        // in secs
```

Damit wären die Credits fertig! .. naja, fast. Die Vertonung mit der Hintergrundmusik machen wir später, aber was noch fehlt - und wichtig ist - ist dass die Credits aufgerufen werden, wenn das Spiel zuende ist.

Abspann nach dem Spielende

Wenn der Spieler das Spiel durchgespielt hat, soll er die Credits als Abschluss des Spiel sehen. Dazu müssen wir erst einmal wissen, dass wir das letzte Level gespielt haben. Um das rauszufinden, müssen wir wissen, wieviele Levels das Spiel hat und in welchem wir uns gerade aufhalten. Dazu definieren wir ins in der Datei "levels.h" folgende Variablen:

```
   int lvl_current;     // Current level
   int lvl_count = 1;   // Number of levels
```

lvl_count gibt die Anzahl der Levels wieder (im Moment ja nur eins) und lvl_current wird zur Laufzeit Auskunft darüber geben, in welchem Level wir uns gerade aufhalten. Bei Spielstart des ersten Levels muss demnach die Variable auf 1 gesetzt werden. Wenn wir das Level dann wechseln würden, würde man die Variable erhöhen. Wenn lvl_current allerdings gleich der Anzahl der Levels ist, wissen wir, dass wir das letzte Level gespielt haben. Zunächst stellen wir die Variable zurück, wenn wir das Spiel starten und fügen folgende Zeile in game_start ("game.c") ein:

```
   void game_start ()
   {
      //(...)

      lvl_current = 1;

      //(...)
   }
```

Wenn der Spieler das Level beendet, zählen wir die Schlitten zusammen und dann steht der Spieler - bisher - nur so rum. Wir wollen dort eine Fallunterscheidung einbauen, die auf die Levelnummer reagiert und dann entsprechend zu den Credits führt, wenn wir das letzte Level erreicht haben. Diese Entscheidung wird in der Funktion pl_death_levelend getroffen. Dort haben wir eine while-Schleife stehen, die Rudi animiert - wir können diese Schleife erst dann abbrechen, wenn alle Herzen gezählt worden sind, um dann fortzufahren. Dazu schauen

wir uns die Variable points_toAdd an, die immer Auskunft darüber gibt, wieviele Punkte noch zum Punktestand addiert werden müssen. Wir bauen die Schleife deshalb wie folgt um:

```
//(...)

wait(1);

while (1) {

   if (points_toAdd <= 0) {
      break;
   }

   pl_animate();
   wait(1);
}
```

Erst warten wir einen Frame, damit points_toAdd auf jeden Fall gefüllt ist, bevor die Variable abfragen. Wenn points_toAdd dann = 0 wird, brechen wir die Schleife ab. Danach gestalten wir die Abfrage wie folgt:

```
// This was the last level -> Credits
   if (lvl_current == lvl_count) {
      cr_load();
   }
```

Wenn wir das letzte Level gespielt haben, werden die Credits geladen. Allerdings haben wir folgendes "Problem": sobald der letzte Punkt addiert wurde, springen wir sofort in die Credits - das ist viel zu hastig und einfach unschön gemacht. Rudi sollte in jedem Fall noch etwas rumstehen, bevor der Spieler fortfährt (zum nächsten Level oder eben zu den Credits). Wir bauen einen behelfsmäßigen Zähler ein, der runterzählt, wenn wir soweit sind und die Schleife zum Abbruch zwingt, wenn er = 0 ist:

player.c:

```
   //(...)

   wait(1);

   my.skill1 = pl_death_levelWaitTime * 16;
   while (my.skill1 > 0) {

      if (points_toAdd <= 0) {
         my.skill1 -= time_step;
      }

      pl_animate();
      wait(1);
   }

      // This was the last level -> Credits
      if (lvl_current == lvl_count) {
         cr_load();
      }
}
```

player.h:

```
var pl_death_levelWaitTime = 2.5; //seconds
```

Damit beendet der Spieler in die Credits, wenn er das letzte Level gespielt hat. Weil wir in diesem Workshop nur ein Level produzieren, ist es sowohl das erste als auch letzte, aber wenn wir noch weitere Levels entwickeln, dann können wir an der Fallunterscheidung den Aufruf der Ladefunktion des nächsten Levels aufrufen.

Damit sind die Credits vollständig programmiert!

Kapitel 11: Leveldesign

Was ist Leveldesign?

In großen Teams beansprucht das Leveldesign – also die Gestaltung der Spielewelt – mehrere Leute. Es fallen viele Aufgaben an, die einerseits nur die Herstellung der Grafik und der Darstellung betrifft, aber auch viele Dinge, die programmiert werden müssen, damit sich der Spieler in der Umgebung so verhalten kann, wie er es erwartet. Der „Leveldesigner“ muss also nicht nur ein künstlerisches Geschick haben, sondern auch ein technisches Know-How besitzen. Es ist ziemlich schwierig den Begriff „Leveldesign“ oder die Rolle des „Leveldesigners“ auf einen Punkt zu bringen. Fakt ist, dass man ein gewisses Händchen oder auch Talent haben muss, um interessante Levels hoher Qualität herzustellen. Doch was sind die Faktoren die gutes Leveldesign ausmachen?

Es gibt einige Dinge, die förderlich sind, wie z.B. die kontrollierte Freiheit des Spielers. Wir wollen für Rudi eine relativ weitläufige Spielwelt erzeugen, denn wir wissen, dass er eine relativ lange Schlange hinter sich herzieht. Durch verschieden gestaltete Bereiche mit unterschiedlichen Geschicklichkeitsstufen (z.B. wieviele mögliche Ausgänge gibt es für den Spieler, wenn er sich nicht selbst schneiden will? Wieviele große/kleine/schwer spielbare Hindernisse gibt es?) können wir den Spieler die Illusion geben, sich „selbst“ aus Gefahrensituationen zu retten, wobei wir durch das Leveldesign die Möglichkeiten limitieren.

Wichtig ist auch, dass wir – da wir ein nicht linear schwieriger werdendes Geschicklichkeitsspiel entwickeln – das Spiel durch verschiedene Spielegschwindigkeiten auflockern oder temporeicher machen. Zwar bewegt sich Rudi immer gleichschnell, aber durch die Schlange als „selbsterzeugtes“ Hinderniss wird der Spieler im Laufe eines Levels immer nervöser. Wenn dass ganze Level nur kompliziert und schwierig aufgebaut ist, ist der Spieler die ganze Zeit nervös, was in Frust enden kann, wenn er das Level nicht oder nur sehr schwer schaffen kann. Bauen wir jedoch einige weiter gefasste Areale ein, so wird die „gefühlte“ Geschwindigkeit heruntergesetzt - der Spieler kann mal aufatmen.

Die Orte, an denen wir die Pakete platzieren, entscheiden mit darüber, wie schwer das Paket zu erreichen ist. Wir können damit aber auch den Spieler durch das Level navigieren. Wenn wir z.B. die Pakete nur an uninteressanten Stellen platzieren, wird es dem Spieler schnell langweilig. Je nachdem wie das Level gebaut ist, können wir dadurch den Spieler auch rätseln lassen frei nach dem Motto „wie komme ich dahin?“. Verfügt das Level über mehrere Orte und Szenen, die nicht alle gleich aussehen und kontrastiv sind (im Aufbau, Design etc.), so wird der Spieler regelmäßig mit neuen Eindrücken konfrontiert, was die ganze map viel interessanter macht und zu einer Entdeckungsreise einläd.

Wichtig ist auch der sogenannte und vielfach „gehypedte“ Immersionsgrad. Dieser komplizierte Begriff beschreibt das „Eintauchen“ des Spielers in die Spielewelt. Neben der Anordnung der Szenen und Orte und dem Spieldesign sorgen auch visuelle Effekte, der Sound & die Musik, Charaktere (sog. NPCs, „non playable characters“), selbstablaufende Vorgänge und Aktionen und die Reaktion der Spielewelt für ein Gefühl für das Level. Würde unser Level nur aus einer ebenen Fläche mit einer Schneetextur bestehen, dann wäre das sicherlich der Worstcase, da alles langweilig ist und es gar nichts zu sehen ist und wir uns gar nicht in einer winterlichen Landschaft „zu Hause“ fühlen.

Unser erstes Level: der Nordpol!

Santa schickt Rudi erstmal in die Nachbarschaft, um dort die ersten Geschenke einzusammeln und dafür Lebkuchen abzustauben. Santa wohnt am Nordpol und über einen Pfad gelangt man dorthin. Ich habe mir überlegt, dass das Level mehrere locations haben soll, durch die sich das Level auszeichnet. Zunächst ist das Level so gestaltet, dass es im Norden von Eiswasser abgeschlossen sein soll (mit herumtreibenden Eisschollen und Eisbergen). Der Ausgang des Levels befindet sich im Norden und der ganze Norden, Westen und Osten des Levels soll durch Tannen und einem leichtem Gebirge abgegrenzt sein. Der Spieler soll im Osten auf einem Pfad starten, der von Santa's Dorf aus in das Level führt (erkennbar durch Schilder, den Elfen, die im Nachwinken und wildwachsenden Zuckerstangen ;)

Am Nordpol gibt es bekanntermaßen Eskimos, die auch in unserem Level nicht fehlen dürfen. Mit Iglus, Lagerfeuern und natürlich Eskimos, die mit Fischnetzen Fische fangen, wird das Level durch ein kleines Eskimodorf und einzelnen Iglus besiedelt sein. Es gibt auch einen kleinen Eisteich, indem Fische herumschwimmen sollen.

Desweiteren habe ich mich dafür entschieden, Pinguine in das Level einzubauen. Ungeachtet der Tatsache, dass es am Nordpol keine Pinguine gibt (sondern nur am Südpol!), sind Pinguine knuffiger als die meisten Eisbären und werden also in unserem Level auftauchen. Die Pinguine sollen ein Art erhöhte Ebene mit einem Kliff kriegen, auf dem sie sich tummeln, Eier ausbrüten und mit ihrem Nachwuchs spielen.

Als zusätzliches Bonbon wollen wir einen kleinen Gag in das Level einbauen: es soll einen abgetrennten Bereich geben, auf der es eine Baustelle geben soll. Halt! - seit wann gibt es Baustellen auf dem Nordpol? Nunja, es soll etwas ganz Besonderes sein: es soll eine Baustelle eines Eishotels sein, nämlich dem „Santa's Wellness Icehotel“! Durch die Baustelle können wir eine Art verwinkeltes Labyrinth bauen, das dem Spieler besonders herausfordern wird – außerdem mag ich Eishotels ;)

Wie bauen wir unser Level?

Mit dem 3D Gamestudio wird ein Leveleditor mitgeliefert, den wir auch schon ansatzweise betrachtet haben: den WED (Kürzel für „World Editor“). Der WED basiert auf einer konstruktiven Art, Levels zu bauen, indem der Leveldesigner mit Hilfe von Blöcken (oder auch „blocks“ oder „brushes“) die Spielumgebung baut. Um z.B. einen ganz simplen Gang zu realisieren, würde man einen Block für den Boden, einen für die Decke und 2 für die Seitenwände benutzen und dann texturieren (wobei man jede einzelne Fläche eines Blockes auswählen und mit einer individuellen Textur versehen kann). Man kann Blocks verzerren und deformieren, aber dies auch nur relativ eingeschränkt. Levels aus Blocks sind besonders geeignet für Spiele, die innerhalb Gebäuden spielen. Mithilfe der sogenannten BSP-tree – Technik kann die engine dann immer genau berechnen welche Teile des Gebäudes sichtbar ist und hat dadurch eine unheimlich starke Leistung.

Allerdings sind Blocklevels für Außenlevel überhaupt nicht geeignet. Dafür gibt es mehrere Gründe: während man vielleicht Straßen und „rechtwinklige“ Städte sehr gut mit Blöcken hinkriegen würde, sind „organische“ Außenwelten (ohne rechte Winkel und frei deformierten Flächen) nicht mit Blöcken in annehmbarer Qualität und – das ist ganz wichtig – Performance zu produzieren. Für Außenlevel sind sogenannte „Terrains“ sehr gut geeignet. Man kann über eine Höhenkarte (engl. „heightmap“) eine Landschaft erstellen, mit der man in Null-Komma-Nix riesige Landschaften, Täler und Berge erzeugen kann. Mit einer ansprechenden Textur und eventuell unterstützenden Shader kann man riesige und vor allem wunderschöne Welten bauen. Terrains kann man durch Graustufenbilder erzeugen, die dann als Höhendaten interpretiert werden. Der mitgelieferte MED kann solche Bilder einlesen und zu Terraindateien konvertieren.

Obwohl wir in unserem Spiel in einem Außenlevel herumlaufen, Berge haben und auf unterschiedlichen Höhenniveaus spielen, sind Terrains für uns jedoch nicht geeignet. Mit Terrains kann man z.B. keine Klippen erzeugen und die Ausrichtung der Vertexe ist immer nur rechteckig, was zu ungewünschten Verzerrungen führt, wenn wir diagonale Kanten oder soetwas in unserem Level haben. Wenn von einer Eisfläche eine Scholle abbricht, haben wir jedoch scharfe Kanten und sowas könnten wir durch ein Terrain nicht ausdrücken!

Wir wollen stattdessen ein Model als Levelgeometrie benutzen. Über eine polygonale Kollisionserkennung können wir Rudi dazu bringen, sich genauso in dem Level zu bewegen, als wenn wir ein Level aus Blocks (wie in unseren Testlevel) oder ein Terrain-Level nehmen würden. Dadurch dass wir das Level als ein Modell produzieren, haben wir viel mehr gestalterische Freiheiten, weil die Topologie und die Struktur des Levels nur davon abhängt, wie wir es wollen und was wir vorhaben. Das ist gut und das wollen wir auch, also machen wir das so. Wenn Sie natürlich ihre Levels anders gestalten wollen, ist das Ihnen überlassen. Im Folgenden gehe ich aber den Weg, den ich selber bei der Erzeugung und Gestaltung des Levels genommen habe.

In wenigen Schritten zum Level in 3D

Es gibt viele Wege, ein Level zu designen und es kommt darauf an, für welches Spiel man ein Level designed,

wieviel Know-How & Erfahrung man hat, welche Werkzeuge einem zur Verfügung stehen und was in welchem Fall die beste Methode ist, das Leveldesign anzugehen.

Der erste Schritt zu einem erfolgreichen Level sollte eine Auflistung aller Schlüsselelemente des Levels sein, so wie wir das bereits getan haben: wir haben am Anfang beschlossen, dass das Spiel in einer Winterlandschaft angesiedelt ist und nun haben wir uns auch schon einige konkrete Elemente und Akteure ausgedacht, die wir ins Level stecken wollen und wie das Level – in etwa! - aufgeteilt sein wird.

Der nächste Schritt ist in der Regel eine ungefähre Skizze zu malen, um herauszufinden, wie das Level am besten strukturiert sein wird. Die Skizze muss nicht perfekt sein, denn das Level wird hinterher garantiert nicht 100%ig der Skizze entsprechen, aber es ist sehr wichtig einmal eine grobe Vorstellung davon zu haben, wie man das Level aufbauen will. Am besten ist es, Skizzen per Hand auf einem Blatt Papier anzufertigen. Dabei sollte das Blatt nicht allzugroß sein, da man dann meistens eher dazu neigt, zuviele Details unterzubringen oder überflüssige Kommentare zu schreiben. Je kleiner der Arbeitsbereich ist, desto mehr muss das Gehirn abstrahieren, um den Kern des Designs zu treffen – schließlich geht es nicht darum, bereits alle Grashalme aufzumalen, sondern die Geometrie des Levels in groben Zügen zu erfassen.

Manchmal hat man auch Denkblockaden, wenn man sich darauf einschießt „jetzt das ultimative Leveldesign“ zu entwerfen. Ich habe beispielsweise morgens um 7 das Level auf meinem Notizblock auf dem Weg zur Uni zusammengekritzelt, als ich auf den Zug warten musste. Manchmal kommen einem eben Gedanken und Inspiration immer dann, wenn man gerade nicht daran denken muss.

Hat man die Skizze fertig (oder eventuelle mehrere), sollte man etwas Zeit aufwenden und den Gedanken „ruhen“ zu lassen und das Level überdenken, unter Umständen auch versuchen, das Level anders zu strukturieren oder zu überlegen „ob das so sinnvoll ist, wie es ist“. In meiner ersten Skizze hatte ich z.B. das Problem, dass ich nicht genau wusste, wie ich die Levelgrenzen gestalten soll und außerdem erschien es mir langweilig, dass ich eine relativ große Fläche habe, auf der die Pinguine rumlaufen und gleich nebenan das Eskimodorf liegt. So habe ich dann entschieden ein Plateu einzubauen, auf dem die Pinguine brüten.

Wenn alles soweit ist, sollte man einen ersten Prototypen für das Level bauen, indem die Struktur des Levels (im Folgenden u.a. auch „environment“ genannt) im Groben umgesetzt wird, sodass man Designfehler frühzeitig erkennt. Für die Modellierung kann man auf zahlreiche kostenlose Tools zurückgreifen. Im Notfall kann man auch auf den mitgelieferten Modelleditor „MED“ zurückgreifen, allerdings ist das vielleicht die umständlichste Variante – andere Programme eignen sich besser dazu (siehe Toolsliste am Anfang des Workshops). Das Level wurde von mir in Cinema 4D auf erstellt, texturiert und schattiert.

Ein vielfach unterschätztes Problem beim Leveldesign ist auch die Skalierung des Levels. Während wir bereits ein Umrechnungsmaß für Meter und Quants festgelegt haben, muss man nun zusehen, dass das Level auch proportional ist und die Maße stimmen. Ich habe das so gemacht, indem ich geschaut habe, ob das Testlevel in seinen Ausmaß für das reicht, was ich mir vorgestellt habe. Das Level habe ich rechteckig im Verhältnis 2:1 geplant, sodass ich das Testlevel solange skaliert habe, bis es mir beim Laufen mit Rudi breit und tief genug erschien. Während der Modellierung des Levels habe ich dann immer geguckt, ob die Höhen und Längen stimmig sind. Ich habe öfters auch eine unfertige Version genommen, sodass ich das Level „live“ im Spiel betrachten kann. In einer ersten Fassung war das Level auf der Z-Achse zu groß skaliert, sodass ich es erst flacher machen musste, bevor es gestimmt hat.

Wie bereits angekündigt, gehe ich hier nicht auf die Erstellung der Levelgeometrie oder die Erstellung der Gegenstände, die im Level als Dekoration platziert werden, ein – dies ist eine Wissenschaft für sich und würde den Rahmen dieses Workshops sprengen. Wer nicht soviel Erfahrung hat, kann die Leveldaten benutzen und darauf aufbauen oder selber versuchen, ein Level zu designen.

Das Nordpol-Level – ein großer Haufen Polygone!

Das Nordpol-Level ist eine einzige Modelldatei („envNorthpole.mdl“) und hat knapp 4000 Polygone. Wenn man das mit anderen Modellen vergleicht, ist das quasi lächerlich – bei heutigen modernen Spielen kann ein kompletter Charakter soviele Polygone haben. Für unsere Pläne reicht das voll und ganz – ich habe versucht, eine

möglichst gleichmäßige Polygonendichte zu erzeugen bei entsprechender Qualität. Ein weiterer Punkt ist die Tatsache, dass wir für das environment als 3D Modell auch eine polygonale Kollision einschalten wollen. Je „unkomplizierter“ das Modell ist, desto besser ist das für Performance.

Wenn man sich das Modell im MED Modeleditor anschaut, wird so mancher verwundert feststellen, dass das Level bereits einen Schatten hat und leicht unscharf wirkt. Ich habe für die Texturierung und die Schattierung eine große 1024x1024 Textur genommen, die auf das gesamte Level gemapped wird und sowohl die Textur und den Schattenwurf des Levels beinhaltet. Dieses Verfahren nennt man „texture baking“ und steht für das Verschmelzen der Textur und der Schatten in einer Textur. Der Nachteil ist, dass wir anstatt einer gekachelten Textur und einem damit detaillierten Modell eine Modell mit einer recht „matschigen“ Textur haben. Zumindestens ist die Textur nicht so scharf als wenn wir eine Schneetextur darauf kacheln würden!

Dieses Problem lösen wir mit einem Trick: wir nehmen die Textur des Modells und werden darüber eine Detail-Textur kacheln, die wir mit der originalen Textur kombinieren. Das funktioniert aber auch nur deshalb, weil sowieso alles mit Schnee bedeckt ist, sonst könnten wir diesen Trick nicht anwenden. Das werden wir später mithilfe eines einfachen shaders realisieren.

Wir erzeugen uns nun eine neue Datei namens „northpole.wmp“, die wir im „levels“ Ordner abspeichern und laden da das Modell rein. Beim Exportieren von 3D Modellen kann es immer sein, dass sie in einer anderen Skalierung gespeichert werden, also größer oder kleiner als benötigt. Wir importieren die Modelldatei in den WED, nehmen das Rudi-Modell und skalieren das Level erstmal so, dass es proportional zu Rudi ist. Im Anschluss speichern wir die Datei erstmal ab. Bevor wir das Level integrieren können, wollen wir erstmal ein paar Funktionen und Actions bereitlegen, die wir u.a. dem Levelmodell zuweisen können.

Alle Funktionen und Actions, die das environment direkt oder indirekt betreffen (z.B. Pflanzen und Bäume, Wasser oder Felsen auf dem environment) sollte man in einem neuen Modul erfassen. Wir erstellen hierfür die Dateien „environment.c“ und „environment.h“ und inkludieren sie in der „rudi.c“. Wir schreiben uns dann eine Funktion für das environment, die wir als action-prototype in die actions.wdl schreiben und dann dem environment zuweisen:

```
void env_ground ()
{
   set(my, POLYGON);
   reset(my, DYNAMIC);
}
```

Beim environment wird dann die polygonale Kollision eingeschaltet und das Model auf „nicht-dynamisch“ gesetzt, was ein wenig Performance spart. Wenn wir jetzt Rudi im Level platzieren, ihm seine Action zuweisen und das Level updaten, sind wir bereits in der Lage, im Level herumzulaufen. Damit wir das Level auch laden, müssen wir in der Funktion game_start statt „testlevel.wmb“ nun „northpole.wmb“ in die level_load Anweisung einfügen:

```
void game_start ()
{
   menu_hide_all();
   menu_mode = MENU_MODE_NONE;

   preload_mode = 2;
   max_entities = 5000;
   level_load("northpole.wmb");

   lvl_reset();
}
```

Wir setzen erstmal das Entity-Limit weit hoch (weil wir später mit den Environmentdetails und Rudis Schlittenschlange viele Entities haben – wir wollen nur auf Nummer Sicher gehen!). Dann haben wir außerdem noch vor dem level_load eine Variable namens preload_mode auf 2 gesetzt – was ist das? Nun, wenn wir ein Level starten, dann erzeugt die engine zwar alle entities und macht sie verfügbar, allerdings werden die Texturdaten nur dann zum ersten Mal in den Speicher geladen, wenn die Entity zum ersten Mal sichtbar ist. Zwar werden wir wohl Rudi und das environment direkt sehen, aber das wird nicht für alle entities gelten. Wenn wir nämlich viele entities haben, die jeweils eine Textur besitzen, dann kann es zu ungewünschten Rucklern kommen. Um dies gleich im Keim zu ersticken, setzen wir den preload_mode auf 2, was bedeutet, dass die engine beim level_load direkt für alle entities sofort den Videospeicher mit den erforderlichen Daten füllt, sodass keine Ruckler entstehen

können.

Wenn wir jetzt das Spiel und das Level starten, können wir mit Rudi auf dem Nordpol rumlaufen – Super!

Das Wasser – unser erster Shader!

Shader sind kleine Programme, die besondere Effekte implementieren, indem sie das Renderverhalten der 3D engine bei Texturen, den Modellen oder dem ganzen Bildschirminhalt manipulieren. Man kann mit shadern so ziemlich jeden trivialen als auch anspruchsvollen Effekt implementieren. Schlagwörter wie „bloom“, „HDR“, „bump/normal oder parallax mapping“, „depth of field“, „nightvision“, etc. sind auf den Siegeszug von shadern in der heutigen Computergrafik zurückzuführen. Gerade durch die gestiegenen Leistungsmerkmale heutiger PCs sind Effekte in Echtzeit möglich, wie man sie nur aus vorgerenderten Filmen aus dem Kino kennt.

Wir wollen für das Wasser, was unser environment umgibt, einen shader benutzen und zwar einen sogenannten „ocean shader“, der die Umgebung reflektiert und spiegelnde Wellen hat. Wir könnten dafür eine Echtzeitspiegelung implementieren (es wird die tatsächliche Umgebung reflektiert – wie bei einem echten Spiegel!), was allerdings dazu führen würde, dass die Performance stark in den Keller geht. Anstattdessen benutzen wir als Spiegelungquelle einen skycube (den wir danach auch um das Level herum anzeigen werden). Ein skycube ist eine Textur, die auf einen Würfel gemapped wird. Wenn man sich in diesem Würfel befindet und sich umschaut, sieht es so aus, als ob man in die Ferne gucken würde – der Vorteil ist, dass wir in die skycube-Textur bereits Wolken und weit entfernte Berge und sowas einbauen können.

Wir besitzen zwar schon eine Datei „effects.c“, aber dort drin wollen wir weiterhin standardmäßige Partikel und Spriteeffekte programmieren, während wir in einem eigenen shader-Modul die shader implementieren und den Zugriff, bzw. die Aktivierung derer zur Verfügung stellen. Dazu legen wir im Ordner game die Dateien „shader.c“ und „shader.h“ an und inkludieren sie in der Datei „rudi.c“.

Zwar könnten wir die shader Dateien auch in einer System-Shader Bibliothek erfassen, aber früher oder später wird die Sammlung der shader sowieso spielspezifisch werden (das muss nicht sein, zeigt aber die Erfahrung) – daher belassen wir die Datei im Spielordner.

Shader werden in der Regel in *.fx Dateien zusammengefasst. Der Code, der darin steht ist der eigentliche Shader-Code, der von der engine interpretiert und ausgeführt wird. Der Code für unseren ocean-shader steht z.B. in der Datei ocean.fx, die wir im Ordner effects speichern. Allerdings können wir so ohne weiteres nicht auf den Shader zugreifen. Dazu sind Materials da: sie beschreiben einmal die Beleuchtung und die Schattierung von Standard-Modellen, allerdings können sie auch über das effect-Attribut einen shader laden. Wir erstellen in der „shader.h“ nun ein Material für unseren shader, der den dann auch gleich läd:

```
MATERIAL* mtl_ocean = {
   flags = tangent;
   effect = "ocean.fx";
}
```

Das wir das „tangent“-Flag angeschaltet haben, hat was mit dem shader selber zutun – dies wollen wir nicht vertiefen. Über das effect-Attribut wird der shader in das Material geladen. Der shader erwartet nun folgendes: im Skin1 der Entity (also in dem Model, auf dass der shader angewendet wird, soll sich eine Normalmap angeben, die die Reflektion des skycubes steuert – darum kümmern wir uns gleich. Im sogenannten Material-Skin2 soll dann der skycube gespeichert sein, den der shader benutzt. Ein Material-Skin ist sowas wie der einer Entity, nur dass er für das ganze Material gilt – also alle Entities, die einen shader und das dazugehörige material benutzen, würden davon betroffen sein. Hört sich kompliziert an, ist es aber nicht.

Was wichtig ist, ist dass wir in diesen material-skin nun den skycube laden müssen. Um es vorzugreifen: wir werden später noch weitere Materials benutzen, die einen skycube benutzen – anstatt nun in jedem Material dieselbe Datei zu laden und zu einer cubemap zu konvertieren erzeugen wir diese ein einziges Mal.

Die cubemap speichern wir in einem globalen BMAP* Zeiger und erstellen die cubemap in einer startup-Funktion, die bei engine-Start gestartet wird.

shader.h:

```
BMAP* fx_cube_sky_northpole;
char* FX_CUBE_SKY_NORTHPOLE   = "cubeSkyNorthpole+6.tga";
```

shader.c:

```
void shader_startup ()
{
   fx_cube_sky_northpole = bmap_create(FX_CUBE_SKY_NORTHPOLE);
   bmap_to_cubemap(fx_cube_sky_northpole);
}
```

Die Skycubedatei – deren Name in FX_CUBE_SKY_NORTHPOLE gespeichert ist – besitzt das Kürzel „+6“, was der engine mitteilt, dass diese Datei für skycubes gedacht ist (ein skycube besitzt 6 Seiten, die alle quadratisch sind). Interessanterweise findet die engine die Datei – das liegt daran, dass erst die Main-Funktion (und indirekt damit über Umwege auch die Funktion, die uns alle Pfade registriert, aufgerufen wird. Dann kommen alle startup-Funktionen dran – somit haben wir kein Problem. Der Befehl bmap_to_cubemap erzeugt aus einer Bitmap – sofern das möglich ist – eine cubemap.

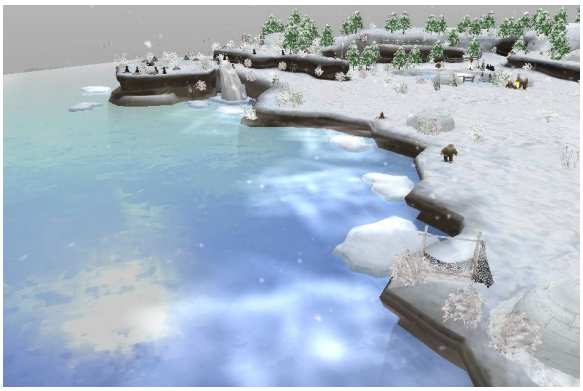
Für den shader habe ich mir nun folgendes ausgedacht: eventuell wollen wir in späteren Level denselben shader anwenden, nur mit einer andere cubemap. Daher sollte der shader über einen BMAP* Parameter eine cubemap zugewiesen bekommen (deshalb machen wir das so!). Um also auf der my-Entity den ocean-shader mit einer speziellen cubemap zu realisieren, kann die my-Entity die Funktion fx_ocean aufrufen:

```
void fx_ocean (BMAP* cube)
{
   my.material = mtl_ocean;
   mtl_ocean_init(cube);
}

   void mtl_ocean_init (BMAP* cube)
   {
      mtl_ocean->skin2 = cube;
   }
```

Der cube wird dann automatisch dem shader zugewiesen. Im Nordpol-Level wollen wir den entsprechenden Skycube realisieren, den wir speziell im Nordpol-Level anzeigen. Daher schreiben wir uns in der „environement.c“ eine eigene Funktion für das Eiswasser:

```
void env_iceWater ()
{
   fx_ocean(fx_cube_sky_northpole);

   set(my, TRANSLUCENT);
   reset(my, DYNAMIC);
   my.alpha = 25;
}
```

Dabei wird der shader mit fx_ocean aktiviert und das Wasser halbtransparent gemacht.



Wollen wir nochmal auf den Skin1 mit der Normal-Map zurückkommen. Sogenannte Normalmaps speichern in ihrer Textur mithilfe der Farbdaten Vektoren pro Pixel, die bestimmen, in welche Richtung des Sonnenlicht reflektiert wird (für das „normal“ Normalmapping). Bei unserem ocean-shader wird damit die Reflektion des skycubes verzerrt, sodass der Effekt einer reflektierenden Wasseroberfläche entsteht. Die Modelldatei, die für uns die Wasseroberfläche realisiert, heißt „envNorthpoleOcean.mdl“. Die Datei ist vom Namen her an das environment angelehnt, weil das Modell speziell an die Form des environments angepasst ist. Im Skin1 ist bereits eine Normalmap für die Wellen gespeichert – Sie können aber auch eine andere laden! Hinweise, wie Sie Normalmaps erstellen, finden Sie am Anfang des Workshops. Wir laden das Modell nun in das Level hinein, skalieren & platzieren es passend und weisen die Funktion env_iceWater zu – das Wasser sieht richtig toll aus, super!

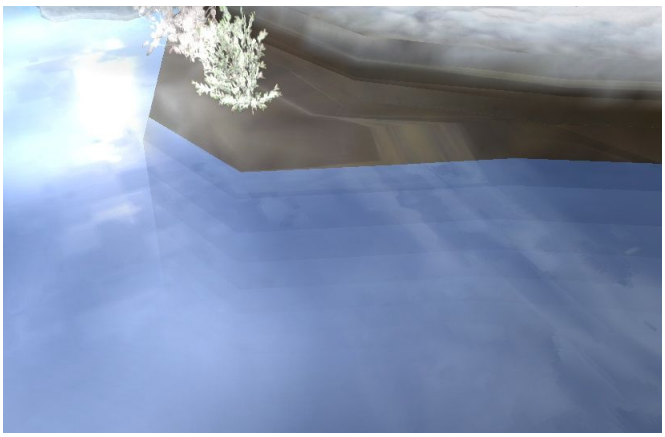
Allerdings kann es ja sein, dass Sie mit der Darstellung des Wasser nicht zufrieden sind: die Anzahl der Normalmap-Kacheln, die Geschwindigkeit der Wasserbewegung und die Wellenstärke kann, bzw. muss man direkt in der Shader-Datei ändern. Die beistehenden Kommentare in der Datei sind sehr leicht zu verstehen (also wo man was wie ändern muss).

Nur haben wir jetzt ein Problem: wenn wir auf das Environment und das Wasser schauen, so sehen wir durch das Wasser hindurch das Environment, wie es aufhört. Normalerweise würde man die Eiskante in der Tiefe des Wasser ausblenden sehen, aber sowas haben wir nicht. Damit wir mit relativ wenig Aufwand und Kosten (= Performance, die wir brauchen würde) diesen Effekt erreichen, habe ich folgenden Trick benutzt: ich habe eine Reihe von Quads (2 Polygone, die ein Rechteck bilden) in kleinen Abständen übereinander gestapelt und jedem Quad eine tiefblaue Farbe gegeben. Jedes Quad ist außerdem stark transparent. Da wir aber viele dieser Quads haben, kann man ab einer gewissen Schicht nicht mehr weiter schauen. Wenn wir ein Model in so einen Ebenen-“Cluster“ hineinschieben, sieht es so aus, als ob das Model „wegfadet“.

Genauso habe ich es mit dem Environment gemacht: das Model habe ich so skaliert, dass die ganze Küstenlinie abgedeckt war und die Kante des Environment-Models nicht mehr sichtbar war. Dem Model für die „Wassertiefe“ habe ich die Funktion gegeben:

```
void env_waterDepth ()
{
   set(my, PASSABLE);
   reset(my, DYNAMIC);
}
```

In Kombination mit dem Shader auf der Wasseroberfläche haben wir nun ein recht annehmbares, hübsches Wasser erzeugt – toll!



Die Skybox

Leider kann die engine die skybox, die um das Level herum gelegt wird, nicht aus cubemaps heraus eine Skybox erzeugen. Ähnlich wie bei den cubemaps wollen wir die Skies, die wir benutzen – und auch wiederbenutzen – separat erzeugen. Für den sky des Nordpols erstellen wir einen leeren Entity-Zeiger in der „environment.h“:

```
ENTITY* sky_northpole;
```

und erzeugen den Skycube vorläufig in der „environment.c“:

```
void env_skycubes_startup ()
{
   sky_northpole = ent_createlayer(FX_CUBE_SKY_NORTHPOLE, SKY | CUBE, 10);
}
```

Damit wir nun einen sky, den wir vorher schon geladen haben, aktivieren könen, schreiben wir uns eine Funktion, die das übernimmt und als Parameter die sky-Entity annimmt:

```
void env_loadSky (ENTITY* sky)
{
   sky_cube_level = sky;
}
```

sky_cube_level ist der globale engine-interne Zeiger auf den aktuell aktiven skycube. Damit wir den skycube nun im Nordpol-Level aktivieren, könnten wir das programmiertechnisch in irgendeiner speziellen Nordpol-Initialisierungs-Funktion schreiben. Das ist aber schlecht, weil wir dann die Level-spezifischen Einstellungen aus der eigentlichen Level-Datei in den Programmcode einbetten. Die Idee ist, eine Art Dummy-Entity in einem Level zu platzieren, die eine Action hat, die das erledigt. Wir können z.B. eine solche Funktion für den Nordpol-Skycube so schreiben:

```
void env_sky_northpole ()
{
   env_loadSky(sky_northpole);
   ent_remove(my);
}
```

Die Dummy-Entity weist den Skycube zu und macht sich dann vom Acker, indem sie sich löscht. Als Dummy-Entity habe ich mir eine große Kugel gebastelt, die ich in der Levelmap außerhalb des eigentlichen Levels platziert habe. Wenn ich nun z.B. einen anderen Himmel – einen Sternenhimmel zum Beispiel – haben will, würde ich mir die entsprechende Funktion schreiben und einfach im Level die Action der Dummy-Entity ändern. Wenn ich den Himmel woanders wiederverwenden will, dann mache ich dass dann genauso – alles sehr einfach, komfortabel und ohne statische Programmierung - super!

Das environment erhält seine Detail-Textur

Wie bereits erwähnt, wollen wir das environment mit einer detaillierteren Textur überziehen, damit wir weiterhin den vorberechneten Schattenwurf haben und trotzdem nicht auf die Details und Vorzüge einer hübschen Schneetextur verzichten müssen. Der shader hierfür lautet „detail.fx“ und diesen laden ganz einfach ohne eine komplizierte Initialisierung, wie wir sie bei dem ocean-shader machen mussten:

shader.c:

```
void fx_detail ()
{
   my.material = mtl_detail;
}
```

shader.h:

```
MATERIAL* mtl_detail = {
   effect = "detailModel.fx";
}
```

Wenn von einer Entity fx_detail nun aufgerufen wird, wird eine Detailmap über das Model gelegt. Dieser Shader funktioniert so, dass als Detailmap die Skin2 der Entity genommen wird. Deshalb müssen wir in das Model des

environments eine zusätzliche Textur – in diesem Falle eine kachelbare Schneetextur – laden, sodass sie über das environement gelegt werden kann (diese ist bereits von mir in die environment eingebettet worden!). Wir müssen nur noch den Aufruf von fx_detail(); in die Funktion env_ground einbauen:

```
void env_ground ()
{
   fx_detail();
   set(my, POLYGON);
   reset(my, DYNAMIC);
}
```

Der Grad der Kachelung (also wie oft die Detail-Textur über das environment gelegt wird) wird im shader bestimmt. Wenn man die Datei öffnet, findet man eine Transformationsmatrix im unteren Teil des shaders, die in jedem Element eine 0 stehen hat, bis auf die Positionen (0,0) und (1,1) – diese beiden Werte die da stehen geben an, wie oft die Detailtextur gekachelt wird. Je höher dieser Wert, desto öfter wir die Textur gekachelt.

Wie baut man ein Level für 3D Gamestudio?

Wir haben jetzt in den letzten Schritten das Level per Hand im WED zusammengesetzt. Es war ja auch nicht viel zutun: wir haben das Environment, das Wasser und Rudi platziert – das wars. Wenn man sich nun vorstellt, wie das Level später aussehen soll, mit all seinen Details, Objekten und Akteuren und man dann vor dem geistigen Auge realisiert, wieviele Objekte das sein werden (wenn man ein nicht allzu spärliches Level bauen will), dann kann einem direkt schlecht werden, weil man jedes Objekt einzeln einfügen, einstellen und platzieren muss – naja, zumindestens ist das die Aufgabe des Leveldesigners, deshalb dürfen die Programmierer unter uns ein wenig aufatmen.

Der WED ist das primäre Werkzeug, Levels für die A7 engine zu erzeugen, bzw. zusammenzustellen. Mit der 3D Vorschau, die ein gewisses WYSIWYG-Gefühl („what you see is what you get“) erzeugt, ist man schon ganz gut in der Lage Levels zu designen und zusammenzustellen. Das Level, was ich im Rahmen der Entwicklung von Rudi erstellt habe, wurde allerdings mit einem anderen Tool namens „GameEdit“ erstellt. Das Programm wird demnächst in das 3D Gamestudio integriert und auch separat erhältlich sein (Anmerkung: als ich diesen Teil des Workshops geschrieben habe, war die Entwicklung des Tools kurz vor der Fertigstellung). Das Tool ist in ein Projekt integrierbar und erlaubt dann die Modifikation eines Levels aus dem Spiel heraus. Wenn der Leveldesigner das Spiel zum Beispiel spielt und dann feststellt dass das Level an einer Stelle kahl aussieht, kann er direkt das Tool aufrufen, dort ein paar Modelle platzieren, speichern und weiterspielen – und all das während der Laufzeit des Spiels!

Es gibt auch einige andere Entwicklung in diese Richtung, darunter auch das Tool „IceX“. Allerdings erleichtern, bzw. beschleunigen diese Tools nur die Erstellung des Levels – im Prinzip kann man alles weitere auch im WED machen.

Um also im WED das Level zu gestalten, hilft es vielleicht auch, eine etwas detailliertere Skizze des Levels als Hintergrundbild zu laden oder freie Bleistiftskizzen zu malen, um die Anordnung der Objekte oder die Dichte der Gräser oder soetwas vorher festzulegen.

Spezifische Objekte des Environments

Egal, für welche Methode Sie sich jetzt für das Erstellen des Levels entscheiden, wollen wir nun dazu übergehen, Funktionen für die ganzen Objekte im Spiels schreiben.

Als erstes beginnen wir mit ganz simplen Steinen als Hindernisse. So einen Stein („rockA.mdl“ und „rockB.mdl“) bekommt z.B. folgende Funktion:

```
void env_rock ()
{
   set(my, POLYGON);
   reset(my, DYNAMIC);
}
```

Wir benutzen hier die polygonale Erkennung, weil Steine eben nicht „kastenförmig“ aufgebaut sind, wie es sich für die normale Boundinbox gehören würde.

Desweiteren wollen wir das Level mit Bäumen und Pflanzen bereichern – und das nicht zu knapp! Pflanzen sollen keine Kollsionserkennung haben – Rudi darf durch sie hindurchlaufen. Gegen Bäume sollte er aber kollidieren können. Damit wir für Bäume, die nicht erreichbar für sind, keine kostenintensive Kollisionserkennung durchführen wollen, erstellen wir eine seperate Funktion für Bäume mit und ohne Kollsionserkennung.

Wir werden im ersten Level nur zugeschneite Tannen verwenden („pinetree.mdl“). Dieses Model hat einen binären Alphakanal für die Blätter und Zweige der Tanne. Um diesen anzuschalten, müssen wie dem Baum das OVERLAY flag zuweisen. Es werden dann auch keine Z-Sorting Fehler angezeigt.

Da die meisten Bäume in unserem Level außerhalb von Rudis Reichweite sein werden, ist die Variante „Baum mit Kollision“ ein Spezialfall. Deshalb wird die normale Baumfunktion keine Kollisionserkennung erhalten:

```
void env_tree ()
{
   set(my, PASSABLE | OVERLAY);
   reset(my, DYNAMIC);
}
```

Für den Baum mit Kollisionserkennung wollen wir anstatt polygonaler Kollisionerkennung eine modifizierte Boundingbox benutzen: wir ermitteln erst die tatsächliche Größe und skalieren die daraus abgeleitete Boundingbox kleiner, sodass sie nur noch in etwa den Stamm umschließt: Rudi wird nämlich nicht durch die Berührung der Blätter/Zweige sterben, sondern wenn er mit Höchstgeschwindigkeit gegen das Holz rast ;)

Die Funktion für kollisionsaktivierte Bäume lautet so:

```
void env_treeColl ()
{
   fx_alpha();

   c_setminmax(my);

   //collision

      //small box
         vec_scale(my.min_x, 0.2);
         vec_scale(my.max_x, 0.2);

      //but no free space above and beneath
         my.min_z *= 10;
         my.max_z *= 10;

   reset(my, DYNAMIC);
}
```

Die Methode von unterschiedlichen Funktionsvarianten sollten Sie immer für alle Objekte durchführen, die unterschiedliche – Performance beeinflussenden! - Anforderungen haben. Wäre es andersherum und die meisten Bäume wäre Kollisionsobjekte, würde ich als Sonderfall die Funktion „env_treePass“ schreiben (wobei „pass“ für „passable“ steht).

Wir haben auch ein Baumstumpfmodell („treestump.mdl“) eingefügt, dass einem Eskimo mit dem Drang zum Förster „gehören soll“. Wir weisen dem Modell eine einfache Funktion zu:

```
void env_stump ()
{
   set(my, PASSABLE);
   reset(my, DYNAMIC);
}

   void env_stumpColl ()
   {
      c_setminmax(my);

      vec_scale(my.min_x, 0.75);
      vec_scale(my.max_x, 0.75);

      reset(my, DYNAMIC);
   }
```

Wir wollen allgemeine Gräser, Büsche und Pflanzen gerne unter einen Hut bringen, da sie die gleichen Eigenschaften haben – bzw. haben sollen: sie haben keine Kollisionerkennung, sollen ein wenig im Wind flattern (zumindestens sehe ich das so – wären sie statisch sähe das nur langweilig aus) und sollen schnell gerendert werden. Das schnelle Rendern wird uns später die genaue Kalibrierung des A7 internen ABT Renderes bescheren und den Nutzen, den wir aus dem ausgeschalteten DYNAMIC flag ziehen. Den Rest erledigen wir durch eine simple Funktion und einen Grass-Shader.

Es ist nämlich so, dass wenn wir Büsche usw. mit einer Wind-Animation versehen und ganz viele Büsche dann während des Spiels animieren, dass dies gewaltig an der Leistung zieht. Das gleiche gilt, wenn wir die Modelle

dauernd skalieren usw. Um einfach das Model etwas zu verzerren, sodass es aussieht, als ob durch den Busch/das Grass Wind weht, können wir einen einfachen Vertexshader benutzen, der die Koordinaten des Büschels nach einem schwingenden Muster verzerrt.

Der Shader ist in der Datei „wavingGrass.fx“ hinterlegt und wird über folgendes Material eingebunden:

```
MATERIAL* mtl_wavingGrass = {
   effect = "wavingGrass.fx";
   flags = TANGENT;
}
```

Das TANGENT flag hat uns hier nicht zu stören. Das Material wird über folgende einfache Funktion implementiert:

```
void fx_wavingGrass ()
{
   my.material = mtl_wavingGrass;
}
```

Wir können für Gräser nun mit folgender Funktion unseren schönen Wedel-Effekt realisieren:

```
void env_weed ()
{
   fx_wavingGrass();
   set(my, PASSABLE);
   reset(my, DYNAMIC);
}
```

Für Büsche stehen die Modelle „bushA“ bis „bushC.mdl“ und für Gräser die Datei „weedA.mdl“ zur Verfügung. In den waving-grass Shader ist bereits ein Alphatest eingebaut, sodass die engine den Alphakanal der Modelle korrekt wiedergibt. In der Shaderdatei selbst gibt es am Anfang zwei Variablen, mit denen man die Geschwindigkeit und Ausschwenkung der Gräser definieren kann – falls Sie die Windeinstellungen ändern wollen.

Die Gräser sehen im Spiel hinterher so aus:

Nun, zum Nordpol gehören auch Eisschollen und hin und wieder herausguckende (wenn auch kleine) Eisberge dazu. Dafür stehen die Modelle „iceFloeA“ bis „iceFloeC.mdl“ und „iceBerg.mdl“ bereit. Man kann die beiden Sachen ganz schnöde ins Level packen, was natürlich gehen würde. Meine Idee war, dass die Modelle so ähnlich wie das Wasser eine Skybox erhalten, die als Reflexion auf das Model gemapped wird. Dieses Verfahren nennt man Environmental-Mapping und wird in Spielen sehr gerne genutzt um Reflexionen zu realisieren. Dabei soll jedoch die Skybox transparent auf das Model gelegt werden, sodass man die Textur noch sehen kann.

Der Shader ist in der Datei „shiny.fx“ realisiert und wird mit dem folgenden Material eingebunden:

```
MATERIAL* mtl_shiny = {
   effect = "shiny.fx";
}
```

Der Shader erwartet im ersten Material Skin die cubemap. Außerdem benötigt das Material eine sogenannte Material-Event Funktion. So eine Funktion wird in der Regel benutzt, um laufend Daten zu beziehen, die der Shader ohne Weiteres nicht berechnen kann. Darunter fällt in diesem Fall die Informationen über die Kameraposition, die wir brauchen, um das korrekte Environment-Mapping zu berechnen.

Im Folgenden nun die Funktion, die den Shader implementiert. Die Funktion kann einen BMAP* Zeiger erhalten, in dem die cubemap steckt. Eine Initialisierungsfunktion weist die cubemap zu und sorgt auch dafür dass der Material-Event des Shaders gesetzt wird:

```
void fx_shiny (BMAP* cube)
{
   my.material = mtl_shiny;
   mtl_shiny_init(cube);
}

   void mtl_shiny_init (BMAP* cube)
   {
      mat.skin1 = cube;
      mat.event = mtl_shiny_func;
      set(mat, ENABLE_VIEW);
   }

      void mtl_shiny_func ()
      {
         mat_set(mtl.matrix,matViewInv);
         mtl.matrix41=0;
         mtl.matrix42=0;
         mtl.matrix43=0;
      }
```

Theoretisch können wir den Nordpol-Skycube für die Reflexionen nehmen, allerdings hat dieser Skycube den Nachteil, dass die untere Hemisphäre (also alles was unter Z = 0 liegt), dunkel gefärbt ist, um einen echten Horizont zu zeichnen. Wenn wir jetzt ein Model mit dieser cubemap ausstatten, dann würden wir schwarze Stellen auf dem Model sehen. Deshalb habe ich einen zweiten Skycube eingefügt, den wir nur für solche Reflexionsgeschichten verwenden wollen. Die BMAP* Referenz und der String für die Datei sehen so aus:

```
BMAP* fx_cube_shiny;
char* FX_CUBE_SHINY = "cubeShiny+6.tga";
```

Die Cubemap müssen wir natürlich genauso wie den Nordpoly-Skycube laden:

```
void shader_startup ()
{
   //(...)

   fx_cube_shiny = bmap_create(FX_CUBE_SHINY);
   bmap_to_cubemap(fx_cube_shiny);
}
```

Jetzt können wir z.B. für die Eisscholle folgende Funktion schreiben, die die Scholle passable, nicht dynamisch und reflektierend schaltet:

```
void env_iceFloe ()
{
   fx_shiny(fx_cube_shiny);

   set(my, PASSABLE);
   reset(my, DYNAMIC);
}
```

Das selbe Prinzip wenden wir für den Eisberg an:

```
void env_iceberg ()
{
   fx_shiny(fx_cube_shiny);

   set(my, PASSABLE);
   reset(my, DYNAMIC);
}
```

Theoretisch könnten wir beide Objekte unter einer Funktion zusammenfassen (da beide Funktionen dasselbe tun), aber man weiß ja nie, wie man ein Objekt später noch erweitert.

Als ich den Eisschollen und dem Eisberg diese reflektierenden Shader zugewiesen hatte, dachte ich, wieso das Environment nicht auch wenig glänzt. Kurzum habe ich den Detailmapping-Shader für das Environment so umgeschrieben, dass zusätzlich eine cubemap auf das detail-gemappedte Model abgebildet wird. Der Shader steht in der seperaten Datei „detailShiny.fx“. Der Grund, warum wir den originalen Shader nicht anfassen, ist simpel: wenn der User weniger Leistung hat und die Shader zurückschrauben will, würden wir für das Environment dann den „normalen“ Detail-Shader laden anstatt den kombinierten Shader mit der cubemap. Diese Optimierungen werden wir jedoch erst später durchführen.

Der Shader wird durch folgendes Material eingebunden:

```
MATERIAL* mtl_detailShiny = {
   event = mtl_detailShiny_init;
   flags = ENABLE_VIEW;
   effect = "detailShiny.fx";
}
```

und die dazugehörigen Funktionen, die den Shader implementieren, lauten:

```
void fx_detailShiny ()
{
   my.material = mtl_detailShiny;
}

   void mtl_detailShiny_init ()
   {
      mtl.skin1 = fx_cube_shiny;

      mtl.event = mtl_detailShiny_func;
   }

      void mtl_detailShiny_func ()
      {
         mat_set(mtl.matrix, matViewInv);

         mtl.matrix41 = 0;
         mtl.matrix42 = 0;
         mtl.matrix43 = 0;
      }
```

Hier wird die cubemap in den Material-Skin 1 geladen. Im Entity Skin 1 wird die ganz normale Textur erwartet und in der Entity Skin 2 wird die Detail-Textur erwartet. Der Shader basiert auf dem Detail- und dem Shiny-Shader, deshalb kann man den detailShiny-Shader auf dieselbe Art editieren wie die beiden anderen. Wir haben in dieser Implementierung die fx_cube_shiny – cubemap statisch zugewiesen. Dem Leser sei es überlassen, die Funktionen so anzupassen, dass man beliebige cubemaps zuweisen kann.

Die neue Funktion für das Environment sieht dann so aus:

```
void env_ground ()
{
   fx_detailShiny();
   set(my, POLYGON);
   reset(my, DYNAMIC);
}
```

Die ganzen Funktionen werden wir später so umbauen, dass wir auf verschiedene Detailstufen für Shader reagieren können. Wir implementieren die Shader erstmal alle so, wie es in der höchstmöglichen Detailstufe beabsichtigt ist.

Generelle Objekte im Level

Es gibt eine ganze Palette von Objekten, mit dem wir unsere Umgebung ausstaffieren wollen. Dabei trennen wir sauber die Elemente, die direkt mit der Levelumgebung zusammenhängen (siehe oben) und den Objekten, die wir dort unabhängig platzieren. Schilder, Iglus, Feuerstellen, Mauern usw. sind eigenständige Objekte, wie z.B. das Level-Tor, welches wir schon im Objekt-Modul (Datei „objects.c/.h“) erfasst haben. Als erstes wollen wir das Tor mit einem der neuen Shadereffekte ausstatten: wir lassen es ein wenig glänzen. Dazu implementieren wir den shiny-Shader:

```
void obj_gate ()
{
   // Initialization
      fx_shiny(fx_cube_shiny);

   //(...)
```

Das Gleiche können wir auch für das Tor-Environment machen:

```
void obj_gateEnvr ()
{
   fx_shiny(fx_cube_shiny);
   //(...)
}
```

Für den Nordpol-Level werden wir in erster Linie Objekte erfassen, die genau dort auch vorkommen. Wenn wir in späteren Levels Objekte erfassen, dann können wir die dort neu eingeführten Objekte im Umkehrschluss natürlich auch im Nordpol-Level einbauen. Für den Nordpol haben wir u.a. ein kleines Eskimodorf geplant, indem Iglus, Eskimos und typische Accessoires platziert sind. Eskimos wollen wir noch nicht behandeln, da sie Akteure und keine
Objekte sind – dies tun wir in einem späteren Kapitel.

Das Iglumodel („iglu.mdl“) ist – wie es für ein Iglu typisch ist – rund. Deshalb schalten wir eine polygonale Kollisionserkennung an.

```
void obj_iglu ()
{
   set(my, POLYGON);
   reset(my, DYNAMIC);
}
```

Eskimos neigen desweiteren dazu, Fische zu verspeisen und jagen diese. Fische werden in dem Level als Akteure vorkommen, dennoch auch in einer „toten“ Variante. Für den Fischfang habe ich eine Art Gestänge mit Fischernetzen gebaut („leverage.mdl“), auf dass wir als Stafette tote Fische legen können (der Fisch liegt als Datei „fish.mdl“ bereit).

```
void obj_leverage ()
{
   set(my, POLYGON);
   reset(my, DYNAMIC);
}

void obj_fishDead ()
{
   set(my, PASSABLE);
   reset(my, DYNAMIC);
}
```

Wir müssen dem Fisch eine explizite Action geben, weil die engine bei Modellen ohne Action die Animationsframes automatisch durchläuft. Dies wollen wir nicht, weil der Fisch dann ja noch zappeln würde – aber er soll ja tot sein!

Zum braten der Fische – oder einfach nur weil es hübsch aussieht – wollen wir noch ein Lagerfeuer zu den Iglus dazustellen. Das Lagerfeuer liegt als Datei „campfire.mdl“ vor. Jetzt wäre es hübsch, wenn wir auch noch einen schönen Feuereffekt für die Lagerfeuer hätten – aber da wir nicht unbedingt jede Feuerstelle angezündet haben wollen, müssen wir unterscheiden, ob es mit Feuer und ohne Feuer gestartet werden soll. Das kann man auf mehrere Weisen lösen, ich habe mich dafür entschieden, zwei unterschiedliche Funktionen anzubieten, die dann als Vorlagen einfach zugewiesen werden können:

```
void obj_fireOff ()
{
   set(my, PASSABLE);
   reset(my, DYNAMIC);
}

void obj_fireOn ()
{
   set(my, PASSABLE);
   reset(my, DYNAMIC);
   //todo: fire effect
}
```

Den Feuereffekt werden wir später implementieren. Es fällt auf, dass wir die Feuerstelle als „passable“ kennzeichnen – das ist auch gut so, weil es unnatürlich und „unfair“ wirken würde wenn Rudi gegen so eine Feuerstelle kollidieren würde.

Für die Klippe, auf der die Pinguine nisten, habe ich ein Modell eines Pinguinnestes gebaut. Es liegt in zwei Varianten vor: einmal mit und einmal ohne Eier („nestEmpty.mdl“ und „nestEggs.mdl“). Das leere Nest ist so gebaut, das eine Pinguinmutter dort perfekt reinpasst. Für das Nest brauchen wir aber nur eine Funktion, da wir an das Nest keine Effekte koppeln. Genau wie die Feuerstelle soll das Nest auch passabel sein:

```
void obj_nest ()
{
   set(my, PASSABLE);
   reset(my, DYNAMIC);
}
```

Für die Eishotelszenerie habe ich ein kleines „sub-environment“ gebaut.. also eine zusammenhängende Umgebung, die in einer anderen Umgebung „gekapselt“ ist. Lange Rede, kurzer Sinn: das Hotel besteht aus einer Hotelfront („hotelFront.mdl“) und einer Baustelle von Zimmern. Das Zimmermodell („hotelRooms.mdl“) kann mehrfach platziert werden, um eine Art „Irrgarten“ zu bauen, der den Geschicklichkeitsgrad an der Stelle im Level hochschraubt. Zu der Hotelszene gehört auch noch ein großes Baustellenschild („hotelSign.mdl“), auf dem „Santa's Wellness Icehotel“ steht, um dem Spieler zu sagen, was das überhaupt ist (ansonsten wäre die Szene sehr sinnlos! - frei nach dem Motto „eine Baustelle am Nordpol,.. was soll das denn?“). Alle drei Modelle sollen eine Kollision erlauben und damit es so aussieht, dass das Hotel auch aus Eis gebaut wird, wollen wir das Räume-Modell mit dem shiny-Shader ausstatten:

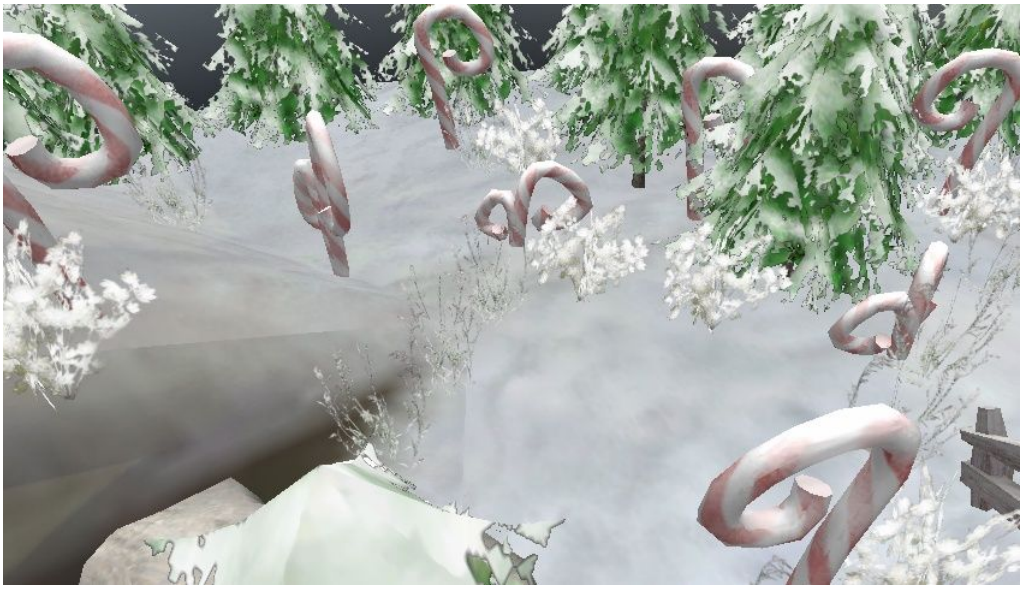
```
void obj_hotelRoom ()
{
   fx_shiny(fx_cube_shiny);
   set(my, POLYGON);
   reset(my, DYNAMIC);
}

void obj_hotelFront ()
{
   set(my, POLYGON);
   reset(my, DYNAMIC);
}

void obj_hotelSign ()
{
   set(my, POLYGON);
   reset(my, DYNAMIC);
}
```

Im östlichen Teil der Map soll es einige Dinge geben, die andeuten, dass der Weihnachtsmann in der Nähe wohnt. Außerdem wollen wir im Stil des Tores durchaus „Kitsch“ im Level platzieren, wie zum Beispiel „wild wachsende Zuckerstangen“ - völlig sinnlos, aber kreativ und irgendwie im wahrsten Sinne des Wortes „süß“. Die Kinder wird es freuen!

Für die frei platzierbaren „wilden“ Zuckerstangen habe ich drei Modelle angefertigt („caneA.mdl“ bis „caneC.mdl“). Eigentlich als Accessoire für das Environment gedacht, sollten wir trotzdem zwei Funktionen erstellen, die die Stangen auf passabel stellen oder eben die Kollisionerkennung aktivieren. Da die meisten Stangen wohl außerhalb von Rudis Reichweite stehen werden, ist die passable Variante Standard und die

kollisionsaktivierte Fassung die spezielle Fassung. Beide Stangentypen sollen glänzend sein:

```
void obj_candyCane ()
{
   set(my, PASSABLE);
   reset(my, DYNAMIC);
}

   void obj_candyCaneColl ()
   {
      set(my, POLYGON);
      reset(my, DYNAMIC);
   }
```



Als zusätzlichen Gag wollen wir ein Schild in Form einer Zuckerstange mit einer Hängelampe ins Level stellen, das als Wegweiser den Weg zum Nordpol und den Weg zu Santa weist („santaSign.mdl“). Da das Schild wohl für Rudi erreichbar sein wird (damit der Spieler es auch sieht!), wird es zwangsläufig auch Kollision unterstützen:

```
void obj_santaSign ()
{
   set(my, POLYGON);
   reset(my, DYNAMIC);
}
```

Um einige Grenzen besser abzustecken, sind Zäune eine gute Wahl - ein paar rustikale Zäune lockern die Umgebung etwas auf! Ein Zaunelement besteht normalerweise aus zwei Pfosten und Querstreben, wenn sie allerdings verbunden sind, dient ein Pfosten in der Mitte als Verbindung zweier Elemente. Deshalb haben wir zunächst zwei Modelle: einmal ein Element, das vollständig ist und eines, dessen rechter Pfosten fehlt, damit er rechts an ein nächstes Element anschließen kann („fenceA.mdl“ und „fenceA2.mdl“). Die Funktionen sind trivial und lauten:

```
void obj_fence ()
{
   set(my, POLYGON);
   reset(my, DYNAMIC);
}

   void obj_fencePass ()
   {
      set(my, PASSABLE);
      reset(my, DYNAMIC);
   }
```

Im Anschluss an dieses Kapitel finden Sie einen Exkurs, indem Sie lernen, einen Schneemann zu modellieren. Dieser Schneemann liegt auch schon als Datei vor: „snowman.mdl“. Wenn Sie den Schneemann im Level als Dekoration platzieren, können Sie die folgenden Funktionen zuweisen:

```
void obj_snowman ()
{
   set(my, PASSABLE);
   reset(my, DYNAMIC);
}

   void obj_snowmanColl ()
   {
      set(my, POLYGON);
      reset(my, DYNAMIC);
   }
```

Abschluss und Ausblick

Wir haben nun alle wesentlichen Leveldaten erfasst und für alle Dinge und Objekte entsprechende Funktionen gebaut, womit wir sofort das Level ausstatten können. Das Level des Spiels ist bereits fertig gebaut und auch schon mit Dingen und Funktionen ausgestattet, die wir in späteren Kapiteln betrachten.. nichtsdestotrotz ist es eine gute Übung mit den vorhandenen Dingen seine eigene Levelversion zu bauen. Sie können auch schon ein oder zwei Pakete in das Level setzen, um eine direkt voll spielbare Version des Levels zu haben.

Wir werden uns in den nächsten Kapiteln noch mehr auf das Level konzentrieren, indem wir Akteure für das Level schreiben, Effekte aufsetzen und noch andere tolle Sachen machen. Bis dahin viel Spaß bei der Gestaltung des Levels und dem „tweaking“ (dem Verändern) der shader!

Exkurs: Einen Schneemann modellieren

In diesem Exkurs, das freundlicher Weise von Felix Caffier zur Verfügung gestellt wurde, wird Ihnen gezeigt, wie Sie mit der kostenlosen Freeware Wings3D einen Schneemann modelliert und texturiert. Viel Spaß dabei!

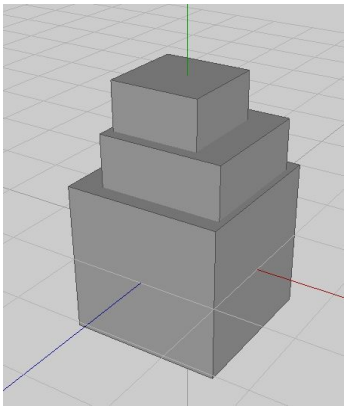
Es entstehe ein Schneemann

Wir beschäftigen uns in diesem Exkurs damit, ein Umgebungsobjekt zu erstellen, zu unwrappen und zu skinnen. Es handelt sich um den Schneemann, den wir mithilfe des OpenSource-Programms „Wings3d“, mein bevorzugtes Programm für diese Arbeiten, erstellen. Das Programm finden Sie unter http://www.wings3d.com für die Plattformen Windows, Linux, Mac.

Wings3D ist ein Boxmodelling-Programm, das es mit wenigen Werkzeugen und dem Befehl „smooth“ erlaubt, in kürzester Zeit komplexe Modelle zu erstellen. Gesteuert wird Wings zumeist über die Maus, auf elementare Dinge wie die Wings-Oberfläche und Bedienung will ich hier nicht eingehen, da es viele sehr gute Tutorials für diese Basics gibt. In diesem Rahmen werde ich Ihnen ein paar Kniffe zeigen, die in den üblichen Wings-Manuals nicht vorkommen oder nur am Rande behandelt werden.

Zunächst stellen wir das Kontextmenü auf „advanced“ um, um einige Befehle als Auswahloption zu erhalten, die in den Standardmenüs nicht vorkommen oder nur über Umwege anwählbar sind (Edit / Preferences / Advanced / Häkchen bei „Advanced Menus“).

Für den Schneemann beginnen mit einem Grundmodell: 3 ineinander geschachtelten Blöcken für die 3 Kugeln, aus dem er besteht.

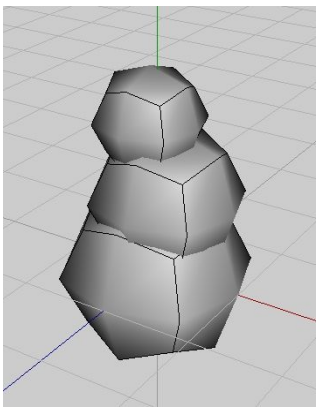


Wir arbeiten mit Blöcken als Grundmodellen, denn gegenüber Kugeln haben Blöcke weniger Polygone und lassen sich leichter in ihrer Form manipulieren.

Wings ist ein Subdivision/Boxmodellingprogramm, bei dem Sie durch das Smoothen von simplen Objekten organische Formen erhalten können. Um ein Objekt zu smoothen, wählen Sie es aus (face- oder object-Mode) und drücken dabei die Taste „S“.

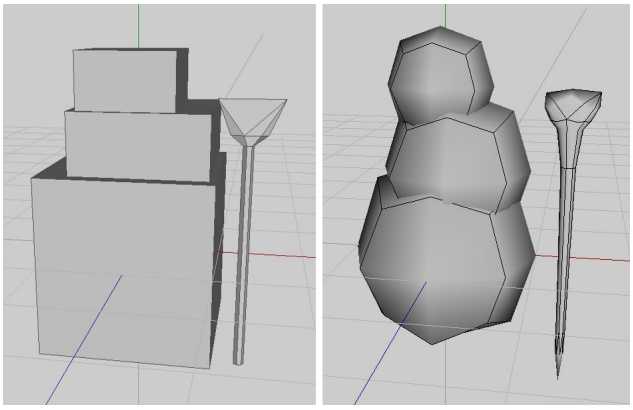
Smoothing sollten Sie immer als letzten Schritt applizieren, wenn das gesamte Modell fertig designedt ist, weil Sie dann die besten Ergebnisse erhalten. Um während des Designens eine Vorstellung von dem zu erhalten, was Sie als gesmoothtes Ergebnis erhalten werden, drücken Sie - ohne etwas ausgewählt zu haben (Leertaste) - die Tastenkombination „Shift+Tab“, um zwischen normalem und gesmoothtem Preview umzuschalten. Die Linien schalten Sie mit „W“ wie „wireframe“ aus. Damit das Model schattiert wird, drücken Sie die Taste „TAB“.

So sieht also unser Modell in der Smooth Preview aus:



Als nächstes nehmen wir uns den Besenstiel vor. Wir erstellen also einen Zylinder...halt! Der Wings-Standardzylinder hat 16 Kanten, das sind weit mehr, als wir brauchen. Falls Sie sich schon einmal gefragt haben, wozu der „Kasten“ hinter dem „cylinder“-, „sphere“- und „torus“- Eintrag ist: Es handelt sich um einen Knopf! Hier können wir die Parameter zur Erstellung unseres Zylinders einstellen. Ändern Sie den Eintrag von „16“ auf höchstens „7“.

Im nächsten Bild habe ich den Zylinder bereits skaliert und mit einem Kopf versehen, hier werden später die Reisigruten draufgemapped, da ein Ausmodellieren der Stöckchen zuviele Polygone erzeugen würde. Das würde die Engine nicht überfordern, aber es wäre eine maßlose Verschwendung von Leistung.

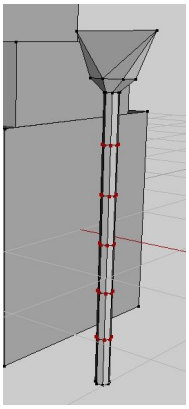


Wie Sie sehen können, unterscheidet die Smooth Preview nicht zwischen den Objekten - den Besen sollten Sie besser nicht smoothen. Es ist also immer wichtig zu wissen, wie man den Preview interpretiert, nicht jedes Modell sieht mit eingeschaltetem smooth besser aus.

Wir stellen fest, dass der Besen noch viel zu gerade ist. Wir werden nun erstmals eine Funktion nutzen, die nur in den „advanced menus“ zu finden ist: „bend“.

Bend ist ein Befehl, der Ihre Modelle anhand von zwei Eckpunkten und einer Normalen verbiegt. Er funktioniert nur mit Vertices und erfordert Einiges an Planung, da gebogene Modelle sich wesentlich schwerer texturieren lassen als gerade. Texturieren Sie daher vorzugsweise ihre Modelle, bevor Sie sie verbiegen. Doch das ist ein Vorgriff aufs nächste Kapitel, kümmern wir uns zunächst darum, den „Bend“-Befehl zu erlernen.

Dazu müssen wir zunächst den Besenstiel feiner unterteilen. Dazu gehen wir in den Edge Mode, Selektieren alle sieben vertikalen Kanten und drücken „6“. Die Ansicht springt um und Sie haben vier neue Vertices pro Seitenkante, d.h. die Kante wurde in sechs Abschnitte unterteilt (vier in der Mitte und die zwei Vertices an den Außenkanten). Dieser Trick funktioniert mit jeder Zahl auf dem Keyboard, alternativ können Sie auch über das Kontextmenü (rechte Maustaste) den Eintrag „cut“ und dann einen Wert für die Unterteilung wählen. Mit den neu erzeugten und noch selektierten Knoten drücken Sie nun die Taste „C“ für „connect“ und Sie haben den ehemals eingliedrigen Zylinder in sechs gleich lange Teile unterteilt:

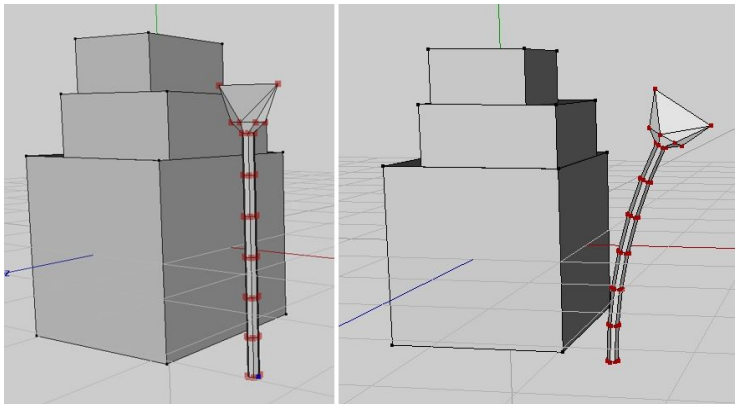


Nun wollen wir den Besen biegen. Selektieren Sie dafür den gesamten Besen im Vertices-Mode und wählen Sie über das Kontextmenü den Eintrag „bend“. Das Modul Bend ist eigentlich recht einfach zu verstehn, wenn man weiß, wo man hinschauen muss: in die linke untere Ecke der Statusleiste. Wings wird Ihnen dort mitteilen, was es als nächsten Schritt von Ihnen erwartet. So erscheint dort die Nachricht „L: Pick Rod Center“, wobei mit „L“ die linke Maustaste gemeint ist.

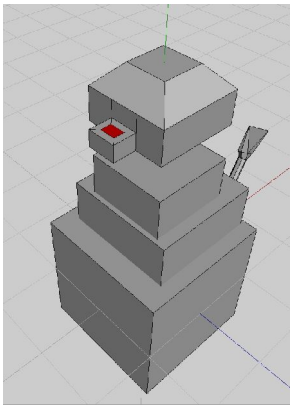
Zuerst sollen wir also den center Vertex markieren, denn Wings schlägt es uns vor. Erfahrungswert: Tun Sie das nicht! In der Regel wollen Sie nämlich nicht etwas um einen Mittelpunkt biegen, sondern von etwas von einem Punkt zum anderen, zum Beispiel ein Kabel oder wie hier den Besen. Markieren Sie also den Knoten, der nicht gebogen werden soll - bei uns ist es der Boden (im Bild blau). Schließen Sie die Markierung mit der rechten Maustaste ab.

Danach folgt der „Rod Top (Vertex)“, den wir mit der linken Maustaste markieren sollen, wählen Sie hier einen Punkt unterhalb des Reisigkopfes. Bis zu diesem „Top Vertex“ wird Ihr Modell später verbogen, alle Vertices oberhalb dieses Vertex werden starr mitgebogen, ohne eine Krümmung zu erhalten. Die rechte Maustaste beendet diese Abfrage.

Danach ist die „bend normal“ an der Reihe. Dieser Vektor gibt die Richtung vor, in die Sie Ihren Besen verbiegen. Selektieren Sie hier einen oder mehrere Vertices, bis der blaue Pfeil in die gewünschte Richtung zeigt. Hat alles geklappt, sollte ihr Besen in etwa so aussehen:



Wenden wir uns den Henkeln des Topfes zu. Wie bekommt man ein Loch in diese Henkel?

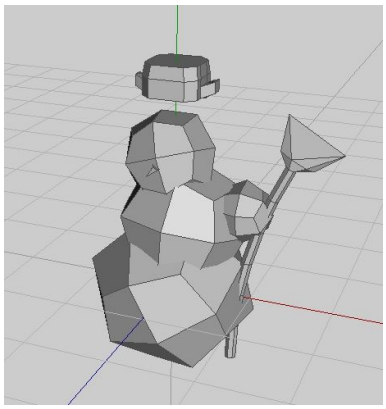


Erstellen Sie eine Form wie auf dem Bild. Wie sie sehen, ist auch der Topf aus einem Quader hervorgegangen. Ich habe Ihn einmal längs unterteilt, um ihn nach oben hin zu verjüngen. Dann habe ich zwei kleine Flächen extrudiert und diesen mit dem Befehl „inset“ eine weitere Innenfläche zugefügt. Diese werden uns helfen, Löcher in die Henkel zu stanzen. Wir werden dafür den Befel „Bridge“ einmal anders einsetzen, als Sie es wahrscheinlich gewohnt sind.

Dafür benötigen Sie zwei übereinander liegende Faces mit gleicher Anzahl an Vertices, da Bridge genau dieses von Ihnen verlangt. Wählen Sie über das Kontext-Menü den Befehl „Bridge“ aus - Sie haben soeben ein Loch gestanzt! Sie sehen also, „Bridge“ verbindet nicht nur zwei Flächen oder Objekte, sondern bietet auch bei geschicktem Einsatz die Möglichkeit, Löcher zu stanzen.

Fügen wir nun noch die Mohrrübe hinzu und ordnen die Objekte etwas nett an.
Jetzt ist der Zeitpunkt gekommen, „smooth“ final einzusetzen: die 4 Kugeln werden gesmoothed, nicht aber der Besen oder Topf – Sie erinnern sich doch noch an unsere Smooth Preview, dann ist auch klar, warum Sie den Besen besser nicht smoothen sollten.
Rotieren Sie die Kugel ein wenig, um ein natürliches, „schiefes“ Ergebnis zu bekommen, nichts ist schlimmer als drei vollkommen gleiche Objekte mit gleichen Kanten und gleicher Vertexverteilung in einem Modell.

Wir erhalten wir folgendes Resultat:



Den Topf habe ich absichtlich abseits gelassen, im nächsten Tutorial werden Sie sehen, warum. Im nächsten Schritt wollen wir unseren Schneemann texturieren.

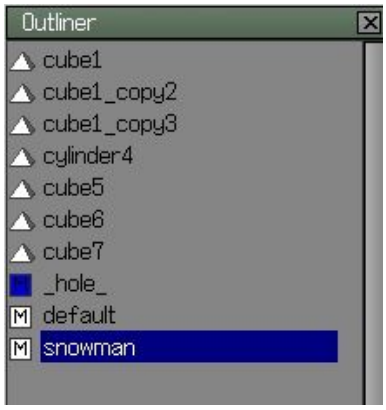
Den Schneemann anziehen: UV skinning und die Textur

Nun wollen wir unseren Schneemann „anziehen“. Seit Wings3D 0.98.31 kann man dies ebenfalls in diesem Programm sehr komfortabel erledigen. Es bringt das Paket „AutoUV“ mit, das das Auffalten von 3D-Objekten zum Kinderspiel werden lässt. Es fügt sich dabei vollkommen in die bestehenden Wings3D-Konzepte ein und erfordert

nur geringe Einarbeitungszeit.

Die Wings3D-Standardansicht ist jedoch für diesen Vorgang ungeeignet, wir benötigen ein weiteres Fenster, den „Outliner“, sozusagen ein Strukturbaum der gesamten Szene, der Geometrie, Lichter, Materialien, Bilder etc. auflistet. Selektieren Sie zunächst nichts (Leertaste). Klicken Sie in der Menüleiste oben auf „Windows“, „Outliner“. Damit schalten Sie ein neues Fenster aktiv, das die Struktur Ihrer Szene widerspiegelt.

Texturen werden in Wings3D über Materialien auf das Modell transferiert, ein Material bestimmt (wie in MED) das Aussehen eines Modells über Texturen, Farben für Spekularität, Grundierung, Transparenzwerte sowie weitere Texturen für Normal- und Gloss Map, die aber in MED nicht eingelesen werden. Wählen Sie aus dem Kontextmenü der rechten Maustaste nun „Material...“. Erstellen Sie so ein neues Material und nennen Sie es „snowman“. Es sollte in Ihrem Outliner zu sehen sein:



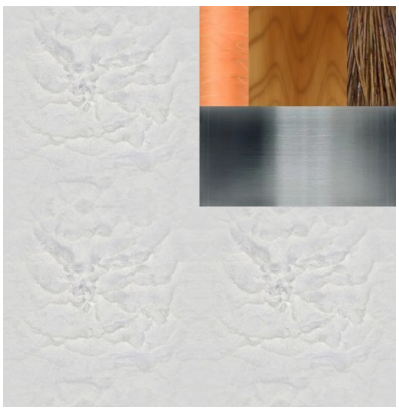
Generell empfiehlt es sich, den Outliner aktiv zu lassen, er bietet guten Überblick und ist für das Arbeiten mit Texturen und Materialien unabdingbar. Das vorangegangene Bild stellt somit auch das Standardlayout meines Wings3D-Workspaces dar.

Als nächstes benötigen Sie eine Textur. Wir werden uns aus bereits bestehenden Bildern oder Fotografien eine Textur zusammenkleben. Dies ist nicht die eleganteste Art, aber Sie geht schnell von der Hand und kann in jedem beliebigen Grafikprogramm nachvollzogen werden. Stellen wir zunächst zusammen, was wir an Bildmaterial brauchen: etwas Schnee, eine Mohrrübe, einen Topf, einen Besenstiel/Holz, Reisig für den Besen und Knöpfe für die Augen.

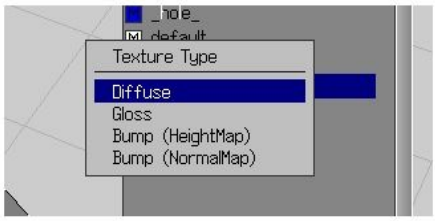
Es ist gibt keinen Standard für Texturgrößen, allerdings sollten Sie für Modelle immer abschätzen können, was das Modell in etwa für eine Textur braucht. Nehmen Sie dann mindestens die nächstgrößere Texturgröße (die Seiten immer in Potenzen von 2 halten!), damit Sie zur Not später die Textur auch kleiner machen können. Wir wollen nun mit einer 512x512-Textur arbeiten.

Jetzt müssen Sie eine Entscheidung treffen: Soll das Modell nur eine Textur beinhalten, oder erstellen Sie das Modell aus vielen kleinen Texturen und separaten Materialien? Ersteres hat den Vorteil, dass sie das Modell später problemlos in andere Modeller einladen und es animieren können. Zweiteres ist einfacher zu erstellen und bietet die Möglichkeit, kachelbare Texturen zu verwenden. Wir werden uns nur mit der ersten Version beschäftigen, da sie die gebräuchlichere im Bereich Gamedesign ist und für weniger Probleme beim Programmieren von Shadern und Portieren in animationsfähige 3D-Software sorgt.

Wir haben unsere Ressourcen aus dem Internet, Texturendatenbanken und aus diversen „Contributions“ in Foren zusammengesucht, die man frei verwenden darf. Nun beginnt die Planung: Wir haben drei Schneekugeln, eine nur aus Schnee, eine mit Knöpfen für den Bauch und eine mit Knöpfen für die Augen. Das macht dreimal einen Platz von 256x256 Pixeln, damit bleiben einmal eine Fläche von 256x256 für die übrigen Texture frei. Unsere restlichen Texturen passen wir so an, dass sie sich diesen Raum ohne Überschneidungen teilen können: Holz (128x128), Topf (128x64), Karotte (64x128) und Reisig (64x128). Angeordnet sieht das so aus:



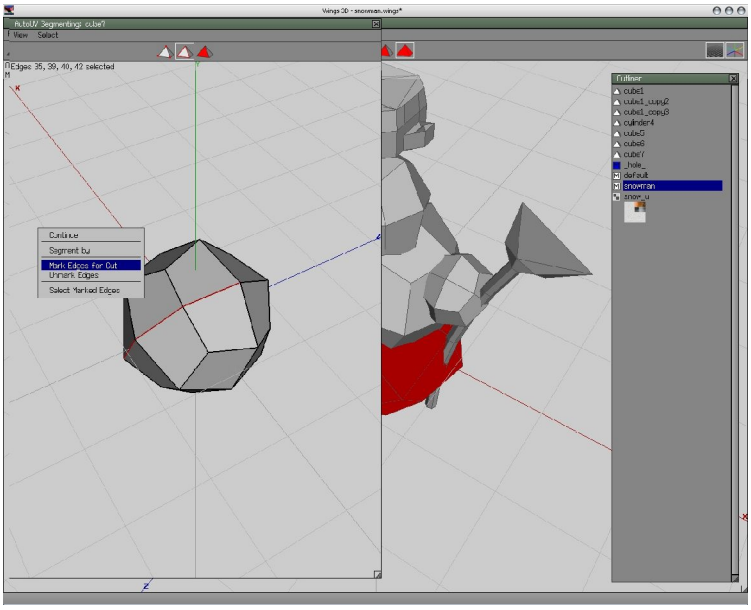
Laden Sie Ihre Textur über File>Import Image in den Outliner. Selektieren Sie die Textur im Outliner und ziehen Sie diese mit gedrückter linker Maustaste auf das im Outliner gelistete Material „snowman“. Im aufpoppenden Dialog wählen Sie die Option „diffuse“, um das Bild als Color Map (die Textur an sich) einzufügen.



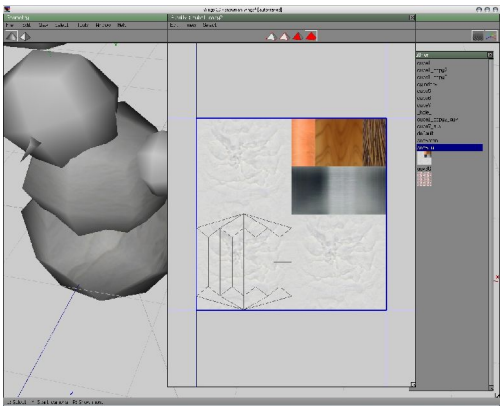
Wechseln Sie in den Object Mode und selektieren Sie die unterste Kugel. Wählen Sie aus dem Kontextmenü die Option „UV Mapping“, um mit dem Texturierungsprozess zu beginnen. Dabei legen Sie fest, auf welche Art und Weise ihr dreidimensionales Objekt in den zweidimensionalen Texturraum projiziert wird. Diesen Vorgang nennt man auch „UV Mapping“. Ein neues Fenster öffnet sich („Auto UV“), in dem Sie nur Ihre Kugel sehen. Ziel ist es, Kanten zu selektieren, an denen entlang das Modell aufgeschnitten wird. Dies ist notwendig, da Sie eine geschlossene Kugel schwerlich auf eine Ebene aufklappen können. Stellen Sie sich eine Orange vor, die Sie schälen möchten, mindestens ein Schnitt ist notwendig, um Sie aus ihrer Schale zu befreien – genau das gleiche Problem entsteht beim entrollen der UV-Map für kugelhafte Körper.

Wir selektieren also unseren halben Kugelumfang mit dem Kantentool und wählen aus dem Kontextmenü „mark edges for cut“. Nun färben sich die Kanten grün, sobald Sie die Selektierung aufheben. So können Sie noch später sehen, wo Sie das mesh zum Schneiden markiert haben. Heben Sie alle Selektierungen auf und wählen Sie aus dem Kontextmenü „Continue“ und dann „Spherical Mapping“, um das Modell auf eine Ebene aufzuklappen. „Spherical Mapping“ ist ein Aufklappalgorithmus, der für Kugeln optimiert wurde. Denken Sie an die Orange zurück: Es sollte Ihnen recht schwer fallen, mit nur einem Schnitt die Orange zu schälen und die Schale stauchungsfrei vor sich auf die Küchenplatte zu legen. Spherical Mapping kann dies weitgehend stauchungsfrei und automatisch für Sie erledigen, so dass die Textur so wenig wie möglich verzerrt wird.

Das AutoUV-Fenster verschwindet und ein Neues öffnet sich. Sie erblicken die Wings-Standardtextur und das Gitternetz der bearbeiteten Kugel. Wings hat die Eigenart, für jedes bearbeitete Objekt ein neues Material anzulegen, vergessen Sie daher bitte nicht, dass Sie nun nicht am Material „snowman“ arbeiten, sondern, wie in meinem Falle, an dem Material „cube7_auv“. Wir werden das Problem am Ende elegant lösen.

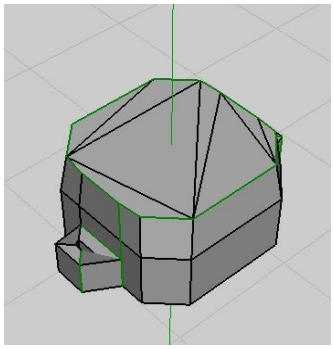


Ziehen Sie nun Ihr Texturbild in das Texturfenster. Daraufhin wird die Standardtextur durch Ihre Textur ersetzt. Nun justieren Sie das Gitternetz passend zur Textur, das heißt über die volle Fläche einer der 256x256-großen Schneequadrate. Verfahren Sie so mit den verbleibenden vier Kugeln.

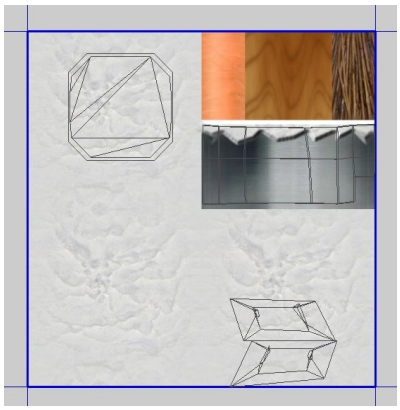


Wählen Sie die Schnittkanten geschickt, am besten so, dass sie von anderen Objekten überdeckt werden. Die Texturnähte sind unschön, können aber nicht immer vermieden werden. Setzen sie daher bevorzugt kachelbare Texturen ein, bei denen automatisch ein nahtloser Übergang entsteht, sobald die Außenkanten aufeinander treffen.

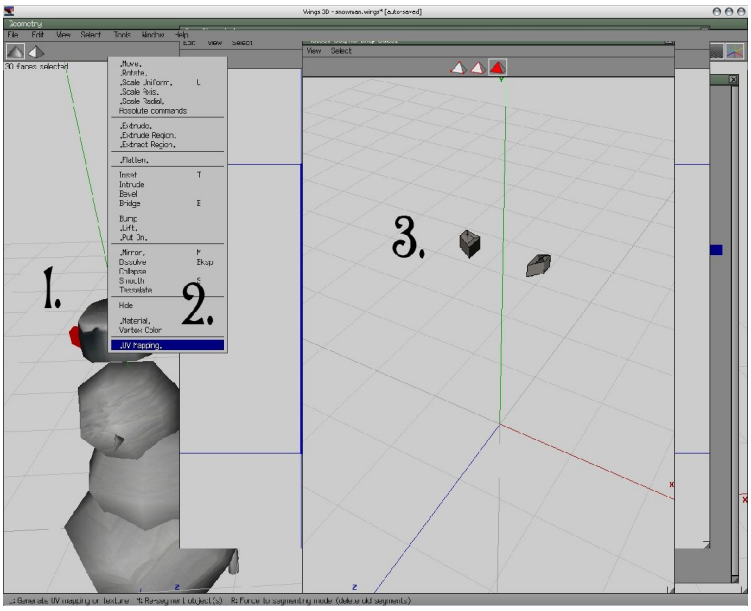
Als nächstes wenden wir uns dem Topf zu. Da es sich hierbei um keine Kugel handelt, müssen wir ihn auch anders zerschneiden. Ich habe Henkel, Boden und Deckel abgetrennt und in die Wand einen vertikalen Schnitt gesetzt. Wählen wir nun „Continue“ und dann „Unfolding“ aus, so wird das Objekt entfaltet. „Unfolding“ ist wohl der Begriff, den Sie beim Texturieren in Wings3D am häufigsten brauchen werden, da sich nicht jeder Körper über Kugeln und Spherical Mapping abstrahieren lässt. Mit „Unfold“ haben Sie eine Allzweckwaffe selbst für die wildesten Oberflächenformen.



Wie Sie sehen können, habe ich ein paar Schneeüberhänge in die Topfwand eingemalt. Die Henkel sind uns nicht gelungen. Sie werden des öfteren feststellen, dass ihre ursprünglichen Schnitte nicht zu einem schönen uv-Ergebnis führen.



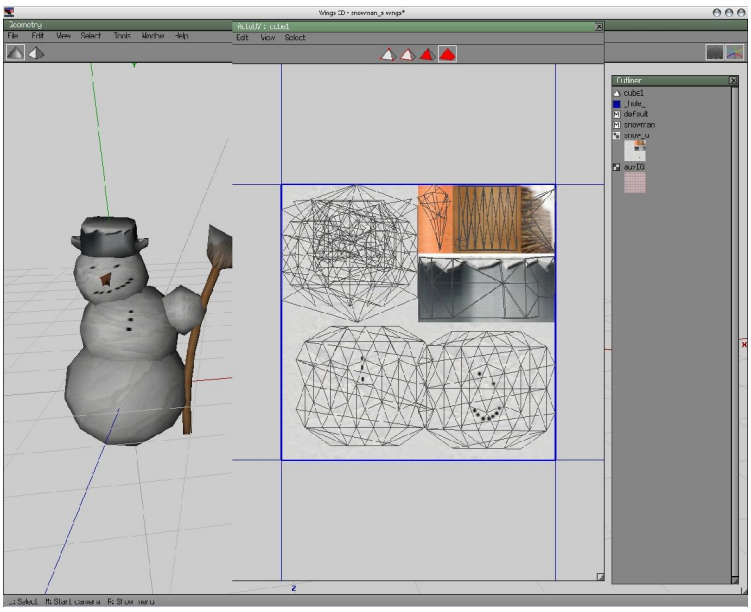
Selektieren Sie die missratenen Polygone und wählen Sie aus dem Kontextmenü „delete“. Fahren Sie nun mit Ihrer Maus über das 3D-Fenster (die gelöschten Polygone sind immer noch selektiert) und wählen Sie aus dem Kontextmenü erneut „UV Mapping“ aus. Das bekannte AutoUV-Fenster öffnet sich und Sie können die misslungenen Bereiche neu zerteilen. Die bereits fertigen werden von dieser Änderung nicht berührt!



Den fertig texturierten Topf können Sie nun verschieben ohne dass sich die Textur verschiebt. Bringen Sie ihn in die Position, die Ihnen am meisten zusagt.

Nun haben Sie die Gelegenheit, einmal zu zeigen, was Sie gelernt haben! Ich überlasse Ihnen das UV-Mappen das Besens. Einen Tipp gebe ich Ihnen jedoch: zerschneiden Sie das Objekt in sinnvolle Abschnitte, wie zum Beispiel Stiel und Kopf. Der Kopf hat zwei Seiten, die sich gegenseitig nicht unterscheiden. Der Stiel besteht aus sieben identischen Zylindern. Ich denke, sie wissen nun, wo Sie Ihre Schnitte zu setzen haben.

Sind Sie mit dieser Arbeit fertig, ist es an der Zeit, das eingangs besprochene Auto-Materialproblem zu lösen. Selektieren Sie hierzu alle Teile des Objekts im Face-Mode und wählen Sie über das Kontextmenü „Material“ das Material „snowman“ aus. Nun teilen sich alle Objekte das Material „Snowman“ und sie können die übrigen, ungenutzten Materialien getrost löschen (Rechtsklick und dann „delete“). Unser UV- Muster sieht nun so aus:



Ich habe dem Schneemann noch Augen und Knöpfe gegeben, desweiteren musste ich die Anordnung des Reisigs verändern. Ansonsten hat unsere Planung vom Anfang Stand gehalten.

Wings3D ermöglicht Ihnen, auch nach dem Texturieren das Modell weiter zu bearbeiten. Unser Schneemann hat extrem wenig Polygone erhalten und eignet sich schwerlich für eine Nahaufnahme. Jedoch besitzt er nur 400 Polygone und eignet sich für eine entfernte Darstellung.

Sie können das Modell weiter smoothen, doch schießt dabei die Polygonzahl in die Höhe. Hier ein Shot mit rund 1200 Polygonen:

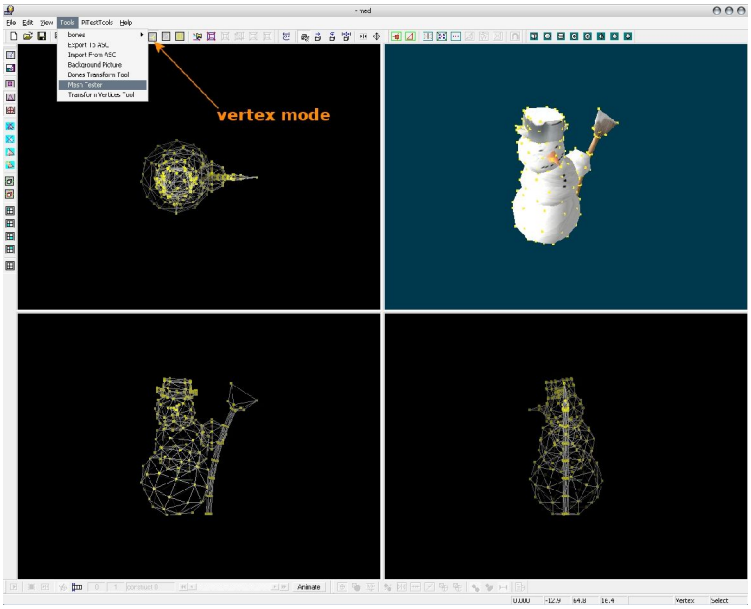
Finden Sie einen gesunden Mittelweg! Dass es möglich ist, beweist der folgende Shot, lediglich 572 Tris sind hier verbaut:

Er sieht nur minimal schlechter aus als der Schneemann mit 1200 Polygonen, es können aber auch doppelt so viele in der Szene dargestellt werden, Sie sehen also, es lohnt sich! Und Ihr Programmierer wird sich freuen!

Der letzte Schritt ist die Portierung nach MED. Da es sich um ein unanimiertes Modell handelt, nutzen wir die *. 3ds-Exportfunktion in Wings3D. Drücken Sie File>Export as>3ds.

Laden Sie Ihr Modell in MED über File>Import>From 3ds ein. Med wird Sie höchstwahrscheinlich nach der Texturdatei fragen, es sei denn, sie haben sie im selben Ordner wie Ihr 3ds-File gespeichert. Geben Sie den Pfad an und drücken Sie auf den „OK“ Knopf. Das Modell ist duch das Schneiden in Wings3D nicht gemerged, das heißt, es handelt sich um ein offenes Modell. So können Sie z.B. keine Stencil Shadows auf das Model anwenden. Wir wollen es nun schließen.

Wechseln Sie dazu in den Vertex Mode und nutzen Sie den Mesh-Tester im Menü „Tools“.



Nun werden alle übereinanderliegenden Vertices selektiert. Drücken Sie einfach auf Merge und Sie sind fertig.

Abschluss und Ausblick

Wir sind am Ende und haben einen netten Schneemann gebaut. Ich hoffe, Sie haben auf diesem kleinen Exkurs mit Wings3D etwas Neues hinzugelernt und hatten ein wenig Spaß beim Lesen und Ausprobieren. Ich wünsche Ihnen viel Erfolg beim Erstellen und Texturieren Ihrer Modelle!

Kapitel 12: Akteure

Akteure und NPCs

In unserer Spielewelt bewegt sich im Moment nur Rudi durch das Level – der Rest ist statisch und wie angewurzelt. Wir wollen nun unseren Nordpol beleben, indem wir ein paar Pinguine, Eskimos und Fische in das Level stellen, die dort rumgeistern und ihren Spaß haben. Alle diese Wesen nennt man auch NPCs („non playable characters“, auf deutsch: „nicht spielbare Charaktere“). Allgemein bezeichnet man in der Softwaretechnik diese als Akteure: ein Akteur abstrahiert dabei von realen Nutzern, indem er für eine Rolle steht, die von verschiedenen Nutzern in gewissen Prozessen eingenommen werden können. Um dies an einem Beispiel zu verdeutlichen: wenn wir wollen, dass einige Pinguine und auch Eskimos an einer Stelle stehen und dem Spieler hinterherschauen, wenn er nah genug ist, dann hätten in diesem Fall beide NPCs ein und dieselbe Rolle: sie gucken Rudi hinterher. Sie unterscheiden sich nur durch ihren Charakter: während der Eskimos sich einfach umdreht und ihm hinterherschaut, quiekt der Pinguin, wenn Rudi ihm zu nahe kommt.

Auf diese Weise wollen wir uns eine kleine Sammlung von sehr allgemeinen Akteuren, bzw. akteurbezogenen Funktionen schreiben, die ein Verhalten, bzw. eine Rolle verallgemeinern. Die spezialisierenden Implementierungen für die NPCs beschreiben wir dann seperat. Für dieses System legen wir ein Akteurmodul und ein NPC-Modul an, indem wir im game Ordner die Dateien „actors.c/.h“ und „npc.c/.h“ erzeugen und in der „rudi.c“ inkludieren.

Ein Akteur mit einer sich wiederholenden Animation

Als erstes wollen wir einen ziemlich simplen Akteur schreiben: er soll die ganze Zeit nur eine einzige Animation abspielen. Dafür werden wir exemplarisch den Eskimo („eskimo.mdl“) nehmen, denn er hat eine Sitz-Animation, die er in Schleife abspielen soll. Da die NPCs den Akteur implementieren, können die NPCs auch unterschiedliche Merkmale haben, wie zum Beispiel unterschiedliche Animationsgeschwindigkeiten oder eben den Namen der abzuspielenden Animation.

Die Akteurfunktion empfängt diese Merkmale über einen Parameter (der Animationsname) und einen Skill (die Animationsgeschwindigkeit im Skill1). Die implementierende Akteurfunktion sieht so aus:

```
void act_loop_anim (STRING* animName)
{
   // Use my real size
      c_setminmax(my);
      vec_scale(my.min_x, 0.6);  // Shrink
      vec_scale(my.max_x, 0.6);

   // Animation offset
      var seed = random(360);

   while (1) {
      ent_animate(my, animName, ((total_ticks + seed) * my.skill1)%100, ANM_CYCLE);
      wait(1);
   }
}
```

Damit nicht viele nebeneinandersitzende Eskimos im gleichen Takt im Sitzen wippen, benutzen wir ein Offset über den Wert in seed, um die Animation zu shiften. Außerdem benutzen wir eine geshrinkedte Boundingbox, die sich an der realen Größe der Entity ausrichtet, um eine leichter spielbare Kollision zu haben. Die NPC-Funktion des Eskimos sieht so aus:

```
void npc_eskSit ()
{
   act_loop_anim("sit");
   my.skill1 = 10;
}
```

Die Akteurfunktion wird aufgerufen und mit „sit“ wird die Animation angegeben. In my.skill1 wird die Animationsgeschwindigkeit angegeben. Wenn wir einen Eskimo ins Levelstellen und die Action npc_eskSit zuweisen, dann sitzt er fröhlich rum.

Die Elfen, also die fröhlichen Helferlein des Weihnachtsmannes, wollen Rudi natürlich verabschieden, wenn er sich auf seine Reise macht. Wir können den gerade geschriebenen Akteur auch für die Elfen benutzen, denn das Elfen-Modell („elf.mdl“) besitzt eine Wink-Animation. Die NPC Funktion des Elfen sieht daher so aus:

```
void npc_elfWave ()
{
   act_loop_anim("wave");
   my.skill1 = 10;
}
```

Der Rudi hinterherschauende Akteur

Nun wollen wir einen etwas anspruchsvolleren Akteur schreiben: er soll einfach nur rumstehen und wenn Rudi in der Nähe ist, soll er sich zu ihm drehen – das war es! Wenn wir diesen Akteur geschrieben haben, erzeugen wir zwei NPCs, die dieses Verhalten implementieren: einmal für den Pinguin („penguin.mdl“) und für den Eskimo.

Die NPCs haben nun wieder unterschiedliche Merkmale, wie zum Beispiel unterschiedliche Animationsgeschwindigkeiten. Der Akteur soll Rudi hinterherschauen, wenn er in der Nähe ist – die Nähe wäre eines dieser Merkmale. Wenn der Akteur rumsteht, dann soll er seine Stand- bzw. Idle-Animation abspielen, währenddessen er seine Lauf-Animation abspielen soll, wenn er sich drehen muss (ansonsten würde es so wirken, als ob der Akteur „schwebend“ rotieren würde). Da die NPCs unterschiedlich sind, sollen sie auch unterschiedliche Animationsgeschwindigkeiten angeben dürfen.

In der ersten Fassung des Akteurs wollen wir ihn erstmal initialisieren und eine Hauptschleife schreiben, die die Distanz checkt. Ist er zuweit entfernt, soll er seine Idle-Animation abspielen. Die Parameter sollen von den NPCs an die Akteurfunktion über Skills angegeben werden. Skill1 soll die Geschwindigkeit für die Idle- und Skill2 für die Walk-Animation speichern, währenddessen Skill3 die Distanz angibt, auf die der Akteur reagiert, wenn Rudi zu nahe ist. Die Akteurfunktion nennen wir act_watchRudi und schreiben sie in die „actors.c“:

```
void act_watchRudi ()
{
   // Use my real size
      c_setminmax(my);
      vec_scale(my.min_x, 0.6);  // Shrink
      vec_scale(my.max_x, 0.6);

   // Animation offset
      var seed = random(360);

   while (1) {

      if (Rudi) {

         // Player approaches the actor
         if (vec_dist(my.x, Rudi.x) < my.skill3) {

            //todo: turn

         } else {
            // Player is far away, so just play the idle animation
            ent_animate(my, "idle", ((total_ticks + seed) * my.skill1)%100, ANM_CYCLE);
         }
      }

      wait(1);
   }
}
```

Wir nehmen die Boundingbox als Kollisionshülle und schrumpfen sie ein wenig, damit Rudi auch eng dran vorbeigehen kann. Der „seed“ Wert wird generiert, um die Animation zu shiften. Das ist ganz nützlich, weil dann alle Akteure unterschiedlich animiert sind und nicht „im selben Takt“ animiert sind. Als Laufvariable für die Animation tricksen wir und „missbrauchen“ die total_ticks Variable.

Kommt Rudi zu nahe, wird die primäre Verzweigung ausgeführt, in die wir bisher noch nichts geschrieben haben. Dort müssen wir nun Folgendes tun: als erstes berechnen wir den Winkel zu Rudi und interpolieren den Winkel dann, damit er sich nicht plötzlich, sondern „gemächlich“ dreht. Wir vergleichen den Winkel des Akteurs dann vor und nach der Interpolation, um zu schauen, um wieviel Grad wir uns eigentlich gedreht haben. Übertrifft dieser Wert einen gewissen Schwellenwert, wissen wir, dass wir uns so „schnell“ drehen, dass wir die Walk-Animation abspielen müssen, ansonsten die Idle-Animation. Das sieht dann so aus:

```
// Player approaches the actor
if (vec_dist(my.x, Rudi.x) < my.skill3) {

   // Calculate direct rotation
      vec_diff(vecTemp.x, Rudi->x, my.x);
      vec_to_angle(vecTemp.x, vecTemp.x);

   // Interpolate rotation and save in skill10 the
   // amount of the rotation (angle difference)

      my.skill10 = ang(my.pan);
      ang_lerp(my.pan, my.pan, vecTemp.x, 0.3 * time_step);
      my.skill10 -= ang(my.pan);

      my.roll = 0;
      my.tilt = 0;

   // Depending on the amount of the rotated degrees
   // the actor stands still (idle) or plays the walk
   // animation to prevent "hovered" rotation

      // Above a certain rotation we play the walk animation
      if (abs(ang(my.skill10)) > 1) {
         ent_animate(my, "walk", ((total_ticks + seed) * my.skill2)%100, ANM_CYCLE);
      } else { // Stand still
         ent_animate(my, "idle", ((total_ticks + seed) * my.skill1)%100, ANM_CYCLE);
      }

} else {
   // Player is far away, so just play the idle animation
```

```
   ent_animate(my, "idle", ((total_ticks + seed) * my.skill1)%100, ANM_CYCLE);
}
```

Damit ist der Akteur eigentlich vollständig beschrieben. Um den Akteur jetzt zu benutzen, muss die Funktion einmal von dem NPC aufgerufen werden. Außerdem müssen die Skillwerte gesetzt werden, sonst macht der Akteur gar nichts. Deshalb kann die Akteurfunktion so nicht den Entities direkt zugewiesen werden im WED!

Für den Pinguin und den Eskimo können wir nun in der Datei „npc.c“ ihre Funktionen schreiben, die den Akteur implementieren:

```
void npc_penWatch ()
{
   act_watchRudi();

   my.skill1 = 10;   //idle speed
   my.skill2 = 10;   //walk speed
   my.skill3 = 600;  //viewing range
}

void npc_eskWatch ()
{
   act_watchRudi();

   my.skill1 = 5; //idle speed
   my.skill2 = 7; //walk speed
   my.skill3 = 700;  //viewing range
}
```

Wenn wir nun ein Pinguin- oder Eskimo-Modell ins Level stellen und die jeweilige Funktion zuweisen, dann können wir beobachten, wie er Rudi nachschaut, wenn er dem NPC zunahe kommt.

Let's go fishin'! - Die Fischmutter und ihre Fischkinder

Vor allem im Teich und dann noch zusätzlich in der Nähe der Eiskante im Meer wollen wir ein paar Fische schwimmen lassen. Damit wir festlegen, wo sie schwimmen, wollen wir einfache, ringförmige Pfade benutzen. Diese kann man im WED setzen. Diese Fische sollen von kleineren Fischkindern gefolgt werden, sodass wir kleine Fischfamilien erzeugen können. Deshalb sind die „größeren“ Fische, die an den Pfaden entlangschwimmen die Fisch-Mütter und die kleinen die Fisch-Kinder.

Bevor wir die NPCs implementieren können, müssen wir Akteur-Funktionen schreiben, die uns dabei unterstützen, an den Pfaden entlangzulaufen. Wir wollen zunächst erstmal Entities mit Pfaden verbinden, indem wir den Pfadnamen, den wir im WED setzen können, als Pfadreferenz angeben. Diese Funktion ist sehr einfach und schreiben wir in die „actors.c“:

```
void act_path_assign (ENTITY* ent, char* pathName)
{
   path_set(ent, pathName);
}
```

Als zweites benötigen wir eine Funktion, die eine Entity auf dem Pfad bewegt. Wir wollen für die Pfadbewegung der Fische die Spline-Interpolation benutzt, die uns die Engine bietet. Die Funktion sieht so aus:

```
void act_path_run (ENTITY* ent, var speed, var offset)
{
   //skill90 stores the progress on the path
   //skill91-93 the last position (xyz)

   // Move on the path
      path_spline(ent, ent->x, offset + ent->skill90);
      ent->skill90 += speed * time_step;

   // Rotate so that the entity looks along the path
      vec_diff(vecTemp, ent->x, my.skill91);
      vec_to_angle(vecTemp.x, vecTemp);
      ang_lerp(my.pan, my.pan, vecTemp.x, 0.2 * time_step);
      vec_set(my.skill91, ent->x);
}
```

Die Akteur-Entity wird mit ent übergeben und in speed steht die Geschwindigkeit drin, die der Akteur in diesem Frame auf dem Pfad zurücklegen soll. Offset ist eine Art Zufallswert, der die Startposition auf dem Pfad verschiebt, sodass mehrere Akteure auf einem Pfad an unterschiedlichen und nicht an derselben Position sind. Die Funktion bewegt die Entity und rotiert sie so, dass sie immer entlang des Pfades schaut. Intern werden die Skills 90 bis 93 benutzt, um von frame zu frame Daten festzuhalten und sollten daher nicht von anderen NPC Funktionen oder dergleichen verwendet werden.

Das reicht auch schon aus, um unsere Fischmutter zu schreiben. Der Fisch soll sich initialisieren und einen Pfad sich selbst zuweisen. Damit wir im WED bestimmen können, welchen Pfad der Fisch folgt, schreiben wir in den String2 der Entity im WED den Namen des Pfades. Die Bewegung auf dem Pfad soll aber keiner konstanten Bewegung entsprechen. Ich habe mir eine Kombination aus einer Grundgeschwindigkeit und zwei sich immer verändernden Sinus-Werten benutzt, um den Fisch periodisch langsamer und schneller schwimmen zu lassen. Während der Fisch dann die ganze Zeit dem Pfad entlangschwimmt, soll er auch etwas nach oben und unten gleiten.. immer hin und her. Die Funktion schreiben wir in die „npc.c“:

```
void npc_fishMom ()
{
   // Initialization

      npc_fish_init();                 // Make me a fish!
      act_path_assign(my, my.string2); // Assign path

      var seed = random(10000);
      var speedSeed = random(5);
      var originalZ = my.z;

   // Swim along the path
   while (1) {

      varTemp[0] = 12 + speedSeed;                        // Base speed
      varTemp[0] += sin(total_ticks + seed) * 3;          // Variation 1
      varTemp[0] += sin((total_ticks + seed) * 2) * 3;    // Variation 2

      // Motion
         act_path_run(my, varTemp[0], seed);              // Run path
         my.z = originalZ + sin(total_ticks + seed) * 20; // Float up & down

      // Animation
         ent_animate(my, "swim", (var)(my.skill90 * 0.25) % 100, ANM_CYCLE);

      wait(1);
   }
}
```

Wobei die allgemeine Fisch-Initialisierung trivial ist:

```
void npc_fish_init ()
{
   set(my, PASSABLE);
}
```

Das Fischkind ist etwas anders gestrickt. Zunächst einmal muss das Fischkind wissen, welche Entity seine Mutter ist, damit es sich danach ausrichten kann. Um die Fischmutter als solche zu kennzeichnen, geben wir der Fischmutter eine Markierung in Form einer Entity-ID. Dazu definieren wir in der „game.h“ einen Skill, der diese ID speichern soll:

```
#define entID skill80
```

Standardmäßig ist dieser Skill gleich 0. Wenn also dort was drin steht, dann ist die Entity explizit markiert. In der „npc.h“ können wir nun eine ID für die Mutter vergeben:

```
#define ID_NPC_FISH_MOM 50001
```

wobei das Tag nach folgender Regel gestaltet ist: erst das ID Tag, dann das Tag für NPC's, dann der NPC Typ (FISH) und dann die Kennzeichnung dass es die Mutter (MOM) ist. Die Zahl die dahinter steht, sollte eindeutig sein. Ich habe jetzt z.B. eine funfstellige genommen, das sollte für genügend andere ausreichen. Die ID müssen wir der Mutter geben:

```
void npc_fishMom ()
{
   // Initialization

      npc_fish_init();                // Make me a fish!
      my.entID = ID_NPC_FISH_MOM;     // I am a mother
      act_path_assign(my, my.string2); // Assign path

      //(...)
```

Damit können nun Fischmütter identifiziert werden. Damit die Kinder auch „ihre“ Mutter finden, wollen wir in jede Mutter im Skill1 eine Zahl schreiben, die die Mutter eindeutig kennzeichnet. In das Fischkind wollen wir dann auch im Skill1 im WED die Zahl eintragen. Findet das Fischkind also eine Mutter, die denselben Skill1-Wert hat, gehören beide zusammen. Um nun eine Fischmutter mit diesem Skill zu finden, können wir schonmal folgende Suchfunktion festhalten:

```
void act_fish_searchMom (ENTITY* kid, var search)
{
   // Goes through the entity list and searches for a fish
   // mother (indicated by the ID_ACT_FISH_MOM entID). SEARCH
   // determines the connected mother. The mother's entity
   // pointer is saved in skill2 of the fish kid.

   you = ent_next(NULL);
   while (you != NULL) {

      if ((you.entID == ID_NPC_FISH_MOM) && (you.skill1 == search)) {
         kid->skill2 = you;
         break;
      }

      you = ent_next(you);
   }
}
```

In diesem Fall wird im Skill2 des Fisch-Kindes der Entity-Zeiger auf die Mutter gespeichert. Die Funktion des Fischkindes lautet dann:

```
void npc_fishKid ()
{
   // Initialization

      npc_fish_init();                    // Make me a fish!
      wait(3);                            // Wait for mom-ents
      act_fish_searchMom(my, my.skill1);  // Search my mom
      vec_scale(my.scale_x, 0.5);         // Make me smaller

      var seed = random(10000);
```

```
      var speedSeed = random(5);
      var originalZ = my.z;

   // Follow my mom
   while (1) {

      // Get mom
      you = (ENTITY*)(my.skill2);
      if (you) {

         // Rotate target point around my mother
            vec_set(vecTemp.x, vector(150, 0, 0));
            vec_rotate(vecTemp.x, vector(total_ticks + seed, 0, 0));
            vec_add(vecTemp.x, you.x);
            vec_diff(vecTemp.x, vecTemp.x, my.x);

         // Calculate speed

            varTemp[0] = 7 + speedSeed;                        // Base speed
            varTemp[0] += sin(total_ticks + seed) * 3;         // Variation 1
            varTemp[0] += sin((total_ticks + seed) * 2) * 3;   // Variation 2

            vec_normalize(vecTemp, varTemp[0]);
            vec_scale(vecTemp, time_step);
            my.skill90 += vec_length(vecTemp);

         // Move
            vec_add(my.x, vecTemp);

         // Rotation
            vec_to_angle(vecTemp, vecTemp);
            ang_lerp(my.pan, my.pan, vecTemp.x, 0.2 * time_step);

         // Float p & down
            my.z = originalZ + sin(total_ticks + seed) * 20;

         // Animation
            ent_animate(my, "swim", (var)(my.skill90 * 0.25) % 100, ANM_CYCLE);
      }

      wait(1);
   }
}
```

Der Code ist eigentlich eher trivial und durch die Kommentare gut lesbar. Interessant ist, warum ich nicht den Targetpoint der Bewegung direkt auf die Position der Mutter gesetzt habe, sondern den Punkt um die Mutter herum zirkulieren lasse. Der Grund ist einfach: wenn ich das tun würde, dann würden die Fische, wenn sie nahe genug sind, in den Bauch der Mutter schwimmen und jetzt schwimmen Sie immer um die Mutter herum.

Eskimos auf Patrouille und eine Pinguin-Karavane

Bisher stehen, bzw. sitzen unsere Pinguine und Eskimos nur rum. Sehr statisch das ganze und auch irgendwie langweilig. Lassen wir doch einfach ein paar Eskimos und Pinguine rumlaufen! Eine Idee, die ich hatte, war auch eine Pinguinkaravane ins Level zu stellen – einfach, damit es knuffiger wird. Was müssen wir also nun dafür tun? Zunächst einmal können wir in etwa den gleichen Weg gehen wie mit den Fischen: wir definieren einen Pfad im WED, an dem unsere NPCs entlanggehen sollen. Wir müssen nur dafür sorgen, dass die Akteure auch auf ihren Füßen stehen, wenn sie so durchs Level gehen.

Die Akteure sollen außerdem eine Kollsion erhalten, damit sie quasi „passive“ Gegner sind. Weil es komisch aussieht wenn eine Figur die ganze Zeit durch die Gegend geht, bauen wir eine Art „Stop and Go“ Mechanismus ein, der die Figur erst etwas stehen lässt, dann läuft sie ein wenig, dann steht sie wieder und so weiter. Damit Gegner nicht mit voller Absicht in Rudi reinlaufen, bauen wir außerdem noch einen Check ein, der guckt, ob Rudi in der Nähe ist. Wenn dem so ist, bleibt die Figur einfach stehen. Bei der Karavane sieht das ein wenig anders aus: wir müssen dafür sorgen, dass sobald nur eine Figur Rudi sieht, alle stehen bleiben. Wir verwenden für all diese vielen Sachen Skill-Parameter, die von den NPC Funktionen gesetzt werden sollen. Darunter fallen dann Animationsgeschwindigkeiten für die stand- (Skill1) und walk-Animation (Skill2), der Entfernung, bei der der

Akteur in der Nähe von Rudi stehen bleibt (Skill3), einer Zeitangabe, wie lange der Akteur mindestens laufen soll (Skill4), einer Geschwindigkeit, mit der sich der Akteur bewegen soll (Skill5). Optional kann die NPC die „seed“ Variable voreinstellen (Skill6) indem der Skill mit einem Wert ungleich 0 initialisiert wird. Dieser seed-Wert gibt darüber Auskunft wo der Akteur seinen Weg auf dem Pfad startet. Die optionale Angabe dieses Wertes brauchen wir, wenn wir die Karavane einfügen, da wir sichergehen wollen, dass alle beteiligten Akteure hintereinander und nicht irgendwie auf dem Pfad starten. Der Pfad selber wird über den my.string2 im WED angegeben.

Weil es wahrscheinlich komplizierter wäre jetzt den Code einzeln herzuleiten, stelle ich hier die Funktion an sich vor und erkläre danach.

```
void act_patrolPath ()
{
   // Initialization

      // Wait to enable skills
         wait(3);

      // Assign path
         act_path_assign(my, my.string2);

      // Collisionhull
         c_setminmax(my);
         vec_scale(my.min_x, 0.75);
         vec_scale(my.max_x, 0.75);


      // Internals

         var seed;
         var stopAndGo = 1;

            if (my.skill6 == 0) {
               seed = random(30000);
            } else {
               seed = my.skill6;
               stopAndGo = 0;
            }

         var speedSeed = my.skill5;

         var flag = 0;
         var timeout = 0;
         int state = 1;

   // Main loop

   while (1) {

      flag = 0; // Reset stop condition

      // Player approaches actor
      if (Rudi) {
         if (vec_dist(my.x, Rudi->x) < my.skill3) {
            flag += 1;

            // If we are part of a caravane, register frame
            if (!stopAndGo) {act_patrolPathCaravaneStop = total_frames;}
         }

      }

      // Default patrol state check
      if (stopAndGo) {

         // State time is over
         if (timeout < 0) {
            state = (++state % 2); // 1->0 or 0->1

            // Randomize state length
            timeout = 16 * ((random(my.skill4) + 3) +
                     ((1-state) * random(my.skill4)));
```

```
            } else { // Decrease timer
               timeout -= time_step;
            }

            // If we are in waiting state, enable flag
            flag += state;

         } else {

            //We are part of a caravane and only stop if all stop
            flag += (abs(act_patrolPathCaravaneStop - total_frames) < 3);

         }

         // Walking
         if (flag == 0) {

            // Run on path
               act_path_run(my, speedSeed, seed);

            // Trace down to stand on my feet
               c_trace(my.x, vector(my.x, my.y, my.z - 1000), IGNORE_ME | IGNORE_PASSABLE |
IGNORE_SPRITES | IGNORE_PUSH);
               my.z = target.z - my.min_z;

            // Animation
               ent_animate(my, "walk", my.skill2 * (total_ticks + seed), ANM_CYCLE);

         } else { // Standing

            // Animation
               ent_animate(my, "idle", my.skill1 * (total_ticks + seed), ANM_CYCLE);
         }

         wait(1);
      }
   }
```

Die meisten Initialisierungsdinge sollten relativ klar sein. Wirklich interessant ist jedoch die Hauptschleife. Wir verfolgen hierbei das Konzept eines additiven Flags, das genau dann NICHT geschaltet ist, wenn es einen Wert von > 0 besitzt. Dem liegt folgende Logik zugrunde: wir nehmen an, dass das Flag geschaltet ist. Geschaltet bedeutet in diesem Zusammenhang, dass der Akteur läuft. Also setzen wir in jedem frame das flag auf 0. Jetzt klopfen wir alle Möglichkeiten ab, warum das flag vielleicht aus sein sollte und addieren in jedem dieser Fälle eine 1 auf das Flag auf. Wenn dann am Ende keine Möglichkeit aktiv ist, ist das das Flag = 0, ansonsten ist es aus.

Zunächst schauen wir ob Rudi in der Nähe ist. Ist die lokale Variable stopAndGo eingeschaltet, dann ist der Akteur eine normale Patrouille, ansonsten ein Teil einer Karawane. Die Variable act_patrolPathCaravaneStop ist eine globale Variable und ist als var in der header-Datei definiert. Wenn ein Teil einer Karawane Rudi sieht, dann wird dieses globale Flag gesetzt, damit alle anderen Karawanen-Mitgleider – die Rudi vielleicht noch nicht sehen – auch stehen bleiben.

Dann schauen wir uns auch eben noch die Sache mit der Variablen state an. Wenn wir laufen, steht state auf 0 und wenn wir stehen auf 1. Wenn wir den state wechseln, weil der Timer abgelaufen ist, dann berechnen wir:

```
state = (++state % 2);
```

Dies ist einfache Moduloarithmetik. Zunächst erhöhen wir die Variable state um 1, indem wir davor den ++ Operator setzen. Der Wert wird dann mit Modulo 2 gerechnet. Das bedeutet: wenn der Wert von state 1 war und dann auf 2 steht, dann modulo 2 gerechnet wird, kommt 0 dabei heraus. Wenn state auf 0 stand, auf 1 wechselt und dann modulo 2 gerechnet wird, kommt dabei 1 heraus. Somit wird immer von 0 auf 1 oder von 1 auf 0 geschaltet.

Wenn wir also mit dem Akteur laufen, dann bewegen wir uns entlang des Pfades. In jedem Frame machen wir dann einen Trace nach unten und setzen ihn auf die Position der Füße, damit die Entity auch richtig steht.

Die Funktion für unseren Eskimo in der „npc.c“ sieht dann in etwa so aus:

```
void npc_eskPatrol ()
{
   my.skill1 = 5;    //Idle-Animation
   my.skill2 = 9;    //Walk-Animation
   my.skill3 = 400;  //Sight distance
   my.skill4 = 5;    //Run time min
   my.skill5 = 10;   //Walk speed

   act_patrolPath();
}
```

und für den Pinguin:

```
void npc_penPatrol ()
{
   my.skill1 = 5;
   my.skill2 = 10;
   my.skill3 = 400;
   my.skill4 = 6;
   my.skill5 = 12;

   act_patrolPath();
}
```

Wir können nun Pfade und darauf patrouillierende NPCs ins Level stellen, indem wir die Funktionen zuweisen und – ganz wichtig! - den Namen des zugewiesenen Pfades in den my.string2 schreiben.

Die Pinguinkaravane erzeugen wir, indem wir in den Skill6 Werte eintragen. Für den ersten Pinguin tragen wir z.B. 100 ein und für den nächsten 150, dann 300 usw., damit die Pinguine voneinander entfernt sind und nicht alle auf einem Platz stehen. Den Rest erledigt die Akteur-Funktion automatisch!

Möwen

Möwen am Nordpol? Ja genau! :) Wir wollen ein paar Möwen über dem Pinguinplateu und dem kleinen Teich kreisen lassen. Die dazu passende Implementation ist allerdings recht einfach und ähnlich im Vergleich mit dem Code für die Fische. Die Möwen „laufen“ einfach nur einem Pfad entlang, spielen dabei ihre Animation ab und das war es auch schon.

Der Code sieht so aus:

```
void npc_seagull ()
{
   // Initialization

      npc_seagull_init();
      act_path_assign(my, my.string2); // Assign path

      var seed = random(30000);
      var speedSeed = 13 + random(10);

   // Main loop
   while (1) {

      // Motion
         act_path_run(my, speedSeed, seed);               // Run path

      // Animation
         ent_animate(my, "fly", 10 * (total_ticks + seed), ANM_CYCLE);

      wait(1);
   }
}

   void npc_seagull_init ()
   {
      set(my, PASSABLE);
   }
```

Kapitel 13: Effekte, Effekte, Effekte

Wo Schnee liegt, da schneit es auch! - der Schneefall

Irgendwie ist es klar, dass es am Nordpol schneit.. sonst würde da ja auch nicht soviel Schnee rumliegen. Auch wir wollen es in unserem Level schneien lassen, damit das Level gleich viel besser aussieht. Außerdem wirkt es dann auch viel „voller“ und „bewegt“, wenn wir die ganze Zeit Schnee um Rudi herum herunterrieseln sehen.

Für den Schneefall sind Partikel prädestiniert: wir können viele davon gleichzeitig bei gleichzeitig schneller Ausführung darstellen. Dies ist vor allem sehr wichtig für Wettereffekte, da man dort meist besonders viele Partikel benötigt, um einen anspruchsvollen und dichten Wettereffekt zu modellieren.

Die Frage ist nur, wie wir das technisch lösen. Betrachten wir das ganz objektiv: über uns befinden sich Wolken und dort entstehen die Schneeflocken, fallen dann herunter und das wars dann auch schon – und das geht dann immer weiter so. Das würde bedeuten, dass wir ständig Schneeflocken neu erzeugen und dann löschen müssen. Das ist logisch, wird aber auch etwas ausbremsend wirken, weil das Erzeugen immer ein kostspieliger Vorgang ist, vor allem dann, wenn man viele Partikel ständig neu erzeugt.

Eine Alternative ist die Folgende: anstatt ständig Partikel zu löschen und neu zu erzeugen, erzeugen wir einmal beim Levelstart eine ganze Reihe von Partikeln einmal, die unendlich lange existent sind (allerdings werden sie entfernt, sobald das Level gewechselt wird). Es gibt dann eine gedachte Box, in der sich die Kamera befindet. Wenn die Kamera woanders hinbewegt wird, wird die Box immer mitbewegt. Die Partikel sollen sich nun innerhalb dieser Box befinden. Wenn sie sich außerhalb der Box befinden, werden sie so verschoben, dass sie sich wieder dort an geeigneter Stelle befinden. Während ihrer Lebenszeit führen die Partikel eine einfach gestrickte Partikelfunktion aus, die die Bewegung der Partikel beschreibt. Somit kann man einen relativ dichten Partikeleffekt schreiben, der nur einmal initialisiert wird.

In dem Bild sehen wir bereits die fertige Fassung, aber wir sehen auch eine „kastenförmige“ Anordnung der Sprites. Dabei habe ich das Spiel angehalten und bin mit der Kamera weggefahren, wobei die Sprites an ihrem Ort bleiben – die Kamera hat sich vorher in diesem Cluster von Sprites befunden!

Der Schneeeffekt soll konfigurierbar sein: einmal sollen die Ausmaße der Box und die Dichte (die Anzahl der Partikel) einstellbar sein. Die Bewegung der Schneeflocken soll durch den Wind bestimmt sein. Durch diese wenigen Parameter können wir dann in verschiedenen Levels unterschiedliche Schnee-Settings haben (z.B. ganz leichtes Schneetreiben, einen Schneesturm, normaler Schneefall usw. - angepasst an das jeweilige Level). Aus performance-technischen Gründen werden wir darauf verzichten, für jede einzelne Schneeflocke einen einzige Partikel zu nehmen. Alternativ werden wir einen etwas größeren Sprite benutzen, auf dem mehrere Schneeflocken abgebildet sind („effSnowFlakes.tga“).

Den Schneeeffekt wollen wir im Effekte-Modul („effects./.h“) erfassen. Wir definieren erstmal die globalen Variablen, die wir dafür benötigen. Wir brauchen einen Vektor, der die Wetter-Box erfasst, einen Vektor für die Windrichtung und -stärke und einen Bitmap-Zeiger, der uns den Schneesprite bereithalten wird:

```
VECTOR eff_weatherbox;
VECTOR eff_wind_dir;

BMAP* eff_snowFlakes_bmap;
```

Die Effekt-Funktion, die wir für die einzelnen Partikel aufrufen ist relativ simpel: der Partikel erzeugt sich irgendwo in der Wetterbox und initialisiert sich dann mit seiner Bitmap, einer Zufallsgröße und startet dann seine Funktion:

```
void eff_snow (PARTICLE *p)
{
   // Create a random position in the weatherbox
      vec_set(p.x, vector( camera.x+random(eff_weatherbox.x*2) - eff_weatherbox.x,
                           camera.y+random(eff_weatherbox.y*2) - eff_weatherbox.y,
                           camera.z+random(eff_weatherbox.z*2) - eff_weatherbox.z));

   // Initialisation
      p.bmap = bmapCheck(eff_snowFlakes_bmap, "effSnowFlakes.tga");
      p.size = 700 + random(700);
      set(p, MOVE);

   // Snowflake function
      p.event = eff_snow_func;
}
```

Die Funktion eff_snow_func ist dann die eigentliche Partikelfunktion. Sie implementiert die Beziehung zur Wetterbox, die vom Wind beeinflusste Bewegungsrichtung und das konstante Weiterleben des Partikels:

```
void eff_snow_func (PARTICLE* p)
{
   // Keep the particle within the weather-box
      vec_set(p->x, vector(cycle(p->x, camera.x - eff_weatherbox.x, camera.x +
                           eff_weatherbox.x), cycle(p->y, camera.y – eff_weatherbox.y,
                           camera.y + eff_weatherbox.y), cycle(p->z, camera.z -
                           eff_weatherbox.z, camera.z + eff_weatherbox.z)));

   // Update the movement speed and direction
      vec_set(p->vel_x, vector(eff_wind_dir.x * time_step,
                               eff_wind_dir.y * time_step,
                               eff_wind_dir.z * time_step));

   p.lifespan = 100; // Live forever
}
```

Damit wäre der Schneeeffekt an sich zwar programmiert – allerdings fehlt noch die Erzeugung des kompletten Wettereffektes inklusive der Einstellung aller Parameter. Dazu benutzen wir eine Funktion, die genau dies erfasst:

```
void eff_snow_create(VECTOR* size, var density)
{
   // Set weatherbox
      vec_set(eff_weatherbox, size);
      vec_scale(eff_weatherbox, 0.5);

   // Create particle cluster
      effect(eff_snow, density, nullvector, nullvector);
}
```

Die Wetterbox wird halbiert, da dann die Berechnungen in der Partikelfunktion einfacher sind. Die Größe und die Dichte werden als Parameter entgegengenommen und zugewiesen. Damit wir in unseren Levels aus einfachen Vorlagen wählen können, definieren wir uns eine Funktion, die so eine Vorlage realisiert und über eine Dummy-Entity implementiert wird:

```
void eff_snow_medium ()
{
   vec_set(eff_wind_dir, vector(10,10,-25));
   eff_snow_create(vector(3000,3000,2500), 100);
   ent_remove(my);
}
```

Sie platzieren dann ähnlich wie beim skycube eine Dummy-Entity im Level, weisen ihr die Funktion zu und schon rieselt Schnee in ihrem Level!

Eis- und Wasserdampf

Wenn Sonne auf benässte Flächen oder Eis scheint, dann entsteht Dampf, weil das Wasser gelöst wird und verdunstet. Nun, ich will nicht andeuten, dass wir auf dem gesamten Level nun Dampf haben wollen, nur fände ich es „cool“, wenn an einigen Stellen, wo wir (Eis-)Wasser im Level haben, dort Dampf aufsteigen zu lassen. Für diesen Effekt benutze ich Sprites, die einen Alphakanal haben. Ich benutze deshalb einen Sprite, damit ich ihn horizontal ins Level legen kann und damit flächenbasiert diesen Dampf- oder auch „Nebel“-Effekt realisieren kann. Um den Effekt in einem Level einzubauen, will ich es so halten: ich nehme den Sprite, der für den Effekt an der Stelle verwendet werden soll und lege ihn dort hin, wo der Effekt beginnen soll. Außerdem skaliere ich den Sprite bereits so, wie die erzeugten „Layer“-Sprites dann auch skaliert sein sollen. Dem Sprite weist man dann folgende Funktion zu:

```
void eff_horFog ()
{
   // Generator will be not seen ingame
      set(my, PASSABLE | INVISIBLE);
      reset(my, DYNAMIC);
```

```
   // Create in large intervals a new fog layer
      while (1) {
         you = ent_create(my.type, my.x, eff_horFog_cloud);
         vec_set(you.scale_x, my.scale_x);
         vec_set(you.pan, my.pan);
         wait(-(5 + random(8)));
      }
}
```

Die Funktion des eigentlichen Nebel-Sprites lautet eff_horFog_cloud. Der Sprite fährt dabei nach oben und fadet ein und dann wieder aus:

```
void eff_horFog_cloud ()
{
   set(my, TRANSLUCENT | PASSABLE | BRIGHT);
   my.alpha = 0;

   vec_set(vecTemp, vector(random(30), 0, random(40)));
   vec_rotate(vecTemp, vector(random(360), 0, 0));
   vec_add(my.x, vecTemp);

   my.pan = random(360);

   wait(1);

   vec_scale(my.scale_x, 1.1-random(0.2));

   while (my.alpha < 100)
   {
      my.alpha += 2 * time_step;
      my.z += 0.5 * time_step;
      my.pan += 0.25 * time_step;
      wait(1);
   }

   while (my.alpha > 0)
   {
      my.alpha -= 0.15 * time_step;
      my.z += 0.5 * time_step;
      my.pan -= 0.5 * time_step;
      wait(1);
   }

   ent_remove(my);
}
```

Für den Nebel benutze ich den Sprite „horSteamIce.tga“ und platziere ihn im Teich und an einigen Stellen an der Eiskante um dort den Effekt zu haben.

Lichteffekte

Im Bereich des Eishotels hängen ein paar Lampen an Seilen herum. Um den Lichteffekt dort hervorzuheben und plastischer zu machen, wollen wir den Lichtschein simulieren, indem wir einen Lichtsprite dort hinstellen. Sprites sind dazu prädestiniert: wir können den Schein nach außen hin wegfaden lassen und das ganze auch nuch unscharf zeichnen, damit es erst recht „realistisch“ aussieht. Die Hängelampen schauen nach unten, sodass der Lichtschein nach unten einem Zylinder entspricht. Der Sprite wird dementsprechend gestaltet und dort platziert. Wenn der Sprite sich mit der Kamera mitdreht, sieht es wirklich so aus, als ob dort ein „volumetrischer“ Lichtschein ist. Die Funktion dafür ist sehr simpel:

```
void eff_flare ()
{
   set(my, PASSABLE | TRANSLUCENT | BRIGHT);
   vec_set(my.pan, nullvector);
   my.alpha = 36;
   reset(my, DYNAMIC);
}
```

Die Funktion heißt eff_flare, weil wir die gleiche Funktion allen anderen flare-Effekten („Blendeffekte“) zuweisen könnten.

Die Seile zwischen den Schlitten

Bisher hatten wir zwischen den Schlitten als Verbindung – rein gar nichts! Die Schlitten haben sich sprichwörtlich auf magische Weise bewegt ohne jedwede Verbindung zum Vordermann. Wir wollen das jetzt ändern und ein Seil zwischen zwei Schlitten spannen. Das gilt auch für Rudi, denn der erste Schlitten wird ja von ihm gezogen.

Als Verbindung wollen wir ein Model nehmen. Es wird eine leichte U Form haben, damit, wenn die Distanz zwischen zwei Schlitten nicht so groß ist, dass Seil dann etwas durchhängt. Das Model heißt „ropeToSledge.mdl“ und ist besonders aufgebaut: der Origin des Models liegt nämlich an genau einem Ende des Seils. Wenn wir das Seil genau auf die Kupplung des Hintermann-Schlittens setzen, ist dass eine Ende des Seils dann nämlich direkt richtig platziert. Wir müssen dann nur noch das andere Ende richtig platzieren.

Wenn ein Schlitten erzeugt wird, wollen wir dann auch gleich das Seilmodell erzeugen – jeder Schlitten wird so eine Seilentity besitzen. Damit diese Assozation auch gespeichert und aufrufbar wird, werden wir einen Skill einrichten, indem wir den Zeiger der Seilentity speichern:

```
#define sledgeRopeEnt skill23

Wir können dann während der Schlitten-Initialisation die Seilentity erzeugen und speichern:

void sl_init ()
{
   //(...)

   // Rope
      my.sledgeRopeEnt = ent_create("ropeToSledge.mdl", vector(0,0,-1000), sl_rope_ent);
}
```

Wir erstellen das Modell deshalb an der Position (0, 0, -1000), damit das Modell im ersten frame nicht sichtbar ist. Das hat einen Grund: wenn das Model erzeugt wird, ist es noch nicht richtig ausgerichtet. Dies wollen wir in einer eigenen Funktion schreiben und weil diese erst ein Frame nach der Initialsierung aufgerufen wird, wird das Seil im ersten Frame halt nicht richtig „aussehen“. Deshalb stellen wir es „ganz woanders“ hin, sodass es nicht gesehen wird. Die Kontrolle über das Seil wird der Schlitten selbt übernehmen, da nur er die Informationen über den Vordermann haben soll. Deshalb sieht die Entity-Funktion sl_rope_ent dementsprechend simpel aus:

```
void sl_rope_ent ()
{
   set(my, PASSABLE);
}
```

Wie bereits gesagt, soll der Schlitten das Seil kontrollieren. Das tut er in einer eigenen Funktion namens sl_rope, die er in jedem Frame aufruft:

```
void sl_main ()
{
   //(...)

   while (my.sledgeDeath == 0) {

      // (...)

      sl_rope();

      wait(1);
   }

   //(...)
}
```

Die Funktion sl_rope sieht wie folgt aus:

```
void sl_rope ()
{
   // Get predecessor
   you = (ENTITY*)(my.sledgeFront);
   if (you) {

      // Calculate my coupling point
         vec_set(vecTemp, vector(91.5, 0, 43));
         vec_rotate(vecTemp, my.pan);
         vec_add(vecTemp, my.x);

      // Set coupling point data for our predecessor
      if (you == Rudi) {   // Rudi
         //todo
      } else {             // Sledge
         vec_set(vecTemp2, vector(-50, 0, 30.3));
         varTemp[0] = 248;
      }

      // Calculate precedessor's coupling point
         vec_set(vecTemp2, vecTemp2);
         vec_rotate(vecTemp2, you.pan);
         vec_add(vecTemp2, you.x);

      // Get rope
      you = (ENTITY*)(my.sledgeRopeEnt);
      if (you) {

         // Sets the rope to my coupling point. Then we scale it, so
         // that the rope is long enough to couple my to my pred.
         // Then we rot. the rope to the pred. so that we're coupled

            vec_set(you.x, vecTemp);
         vec_diff(vecTemp3, vecTemp2, vecTemp);
         you.scale_x = vec_length(vecTemp3) / varTemp[0];
         vec_to_angle(you.pan, vecTemp3);

      }
   }
}
```

Das sieht alles sehr wild aus, aber das Prinzip ist einfach: zunächst holt sich die Funktion den Vordermann. Wenn dieser existiert, wird die Position der eigenen Kupplung berechnet. Die Zahlenwerte habe ich mir aus dem Model herausgelesen. Alternativ könnte man auch einen Vertex nehmen und dessen Position abfragen, aber da wir unterschiedliche Schlittenmodelle mit unterschiedlich vielen Vertexen benutzen, ist diese Variante besser. Wenn der Vordermann ein Schlitten ist, dann berechnen wir dessen Kupplungsposition. Wenn die Seilentity existiert, dann machen wir Folgendes: wir bewegen das Seil an unsere Kupplung. Dann skalieren wir das Seil so, dass es exakt so lang ist, wie die Distanz zum Kupplungspunkt. Dazu teilen wir die Länge zur Kupplung durch die Länge des Seils und erhalten den richtigen Skalierungsfaktor. Damit das Seil nur länger und nicht dicker wird, wenden wir die

Skalierung nur auf den scale_x Faktor an. Danach drehen wir nur noch das Seil in Richtung des Vordermanns und schon sind wir fertig!

Nun fehlt noch das Seil vom ersten Schlitten zu Rudi. Allerdings hat Rudi noch keinen Zaum (engl.: „bridle“), an den das Seil angeknüpft ist. Der Zaum liegt als seperate Modeldatei vor („rudiBridle.mdl“), sodass wir beliebig das Model an Rudi anhängen können, wenn wir wollen. Außerdem besitzt der Zaum die gleichen Animationsframes wie Rudi und ist daher synchron mit Rudi bei jeder Bewegung. Wenn Rudi noch keinen Schlitten an sich gekettet hat, sieht das recht merkwürdig aus, wenn er einen Zaum trägt, also zeigen wir ihn nur an, wenn er mindestens einen Schlitten besitzt. Ähnlich wie bei den Schlitten wollen wir den Zaum auch als Entity vorher laden und dann von Rudi aus kontrollieren und in jedem Frame justieren.

Die Entityreferenz speichern wir genauso wie bei den Schlitten in einem Skill namens

```
#define playerBridle skill12
```

und erzeugen die Entity in Rudis Init-Funktion:

```
void pl_rudi_init ()
{
   //(...)

   // Create bridle
      my.playerBridle = ent_create("rudiBridle.mdl", vector(0,0,-1000), pl_rudi_bridle);

   //(...)
}
```

Die Funktion des Zaums ist identisch mit der der Seile:

```
void pl_rudi_bridle ()
{
   set(my, PASSABLE);
}
```

Nun ist es wichtig, an der richtigen Stelle im Code von Rudi den Zaum zu justieren: denn erst wenn wir die finale Position und Rotation erreicht haben, können wir mit Gewissheit den Zaum neu einstellen. Dies tun wir nach dem Aufruf der Interpolationsfunktions kurz vor dem wait(1) in der Hauptschleife von Rudi:

```
void pl_rudi ()
{
   //(...)

   while (my.playerDeath == 0)
   {
      //(...)

      pl_interpolate();

      pl_bridle();

      pl_log();

      wait(1);
   }

   //(...)

}
```

Die Funktion pl_bridle kopiert dann quasi nur Rudis Parameter in das Zaummodel, sofern Rudi einen Schlitten hinter sich herzieht. Ansonsten wird der Zaum unsichtbar geschaltet.

```
void pl_bridle ()
{
   you = (ENTITY*)(my.playerBridle);

   if (you) {

      if (my.sledgeFirst != NULL) {

         vec_set(you.x, my.x);
         vec_set(you.pan, my.pan);
         you.frame = my.frame;
         you.nextframe = my.nextframe;

         reset(you, INVISIBLE);

      } else {
         set(you, INVISIBLE);
      }
   }
}
```

Interessant sind die Zeilen, indem der frame und nextframe Wert kopiert wird: diese Werte geben an, welcher Frame bei Rudi gerade angezeigt wird und welcher Frame als nächstes angezeigt wird. Auf diese Weise synchronisieren wir den Zaum mit Rudi und es sieht so aus, als ob Rudi den Zaum auch „in echt“ tragen würde, da der Zaum sich Rudi anpasst (und zwar nur deshalb, weil beide Modelle ursprünglich miteinander animiert worden sind).

Damit bei einem Crash, bzw. dem Levelende der Zaum auch richtig weiteranimiert wird, müssen wir den aufruf von pl_bridle(); auch noch in die animierenden Schleifen in die Funktionen pl_death_crash und pl_death_levelend einfügen.

Jetzt können wir auch das Seil vom ersten Schlitten zur Rudis Zaum spannen. An dem Code von sl_rope ändern sich nur Kleinigkeiten. Zunächst einmal setzen wir andere Daten, um den Ort der Kupplung festzustellen:

```
// Set coupling point data for our predecessor
if (you == Rudi) {   // Rudi
   vec_set(vecTemp2, vector(-46, 0, 70)); //offset
   varTemp[0] = 78;                       //rope length
} else {
   //(...)
}
```

Nachdem wir dann den Kupplungsort berechnet haben, müssen wir nun folgendes feststellen: da der Zaum mit Rudi mitanimiert wird, „wippt“ der Zaum die ganze Zeit, während Rudi läuft. Das bedeutet auch, dass der Kupplungspunkt am Zaum hoch und runter wippen müsste. Das tut er allerdings nicht: das Seil würde immer an der gleichen Stelle „hängen“, und wenn der Zaum hinten etwas höher liegt, dann hängt das Seil in der Luft. Deshalb nehmen wir einen Vertex, berechnen dessen Position und modifizieren dann die aktuelle Kupplungsposition. Dies muss unmittelbar vor der Justierung des Seils geschehen:

```
// Calculate precedessor's coupling point
   //(...)

// If predecessor is Rudi, adapt coupling to bridle
   if (you == Rudi) {
      vec_for_vertex(vecTemp3, (ENTITY*)(Rudi->playerBridle), 84);
      vecTemp2.z = vecTemp3.z;
   }

//(...)
```

Jetzt haben wir die gesamte Schlittenkette mit schönen Seilen miteinander verknüpft was das Ganze nun realistischer – und schöner! - macht – toll!

Rudis rote Nase

Was wäre Rudi ohne seine rot leuchtende Nase? Gar nichts, nämlich nur ein stinknormales Rentier! Zwar haben wir schon ein knalliges Rot an seiner Nase, aber um das ganze etwas effektvoller wirken zu lassen, wollen wir ihm eine rot leuchtende Nase geben!

Wir wollen dies erreichen, indem wir für den Schein (engl.: „glow“) der Nase einen roten Sprite benutzen („effRudiNoseGlow.tga“, im effects-Ordner). Während sich Rudi dann durch das Level bewegt, soll sich der Sprite immer an seiner Nase ausrichten. Dazu benutzen wir das Mesh von Rudi: wir nehmen uns einen Vertex an der Nase heraus und lesen dann in jedem Frame die Position des Vertexes aus. An dieser Stelle stellen wir dann den Sprite hin, sodass er immer an seiner Nase „klebt“.

Die Funktion des Sprites ist relativ simpel: er stellt sich auf passable und orientiert sich, falls Rudi existiert, an seiner Nase – der Vertex wird als Nummer identifiziert:

```
void eff_rudiNoseGlow_func ()
{
   set(my, PASSABLE | TRANSLUCENT | BRIGHT);

   while (1) {

      if (Rudi) {
         reset(my, INVISIBLE);
         vec_for_vertex(my.x, Rudi, 223);
      } else {
         set(my, INVISIBLE);
      }

      my.scale_x = my.scale_y = 1.5 + sin(total_ticks) * 0.5;
      my.alpha = 40 + sin(total_ticks * 4) * 5;

      wait(1);
   }
}
```

Damit der Sprite erzeugt wird, erstellen wir folgende initialisierende Funktion:

```
void eff_rudiNoseGlow ()
{
   if (!eff_rudiNoseGlow_ent) {
      eff_rudiNoseGlow_ent = ent_create("effRudiNoseGlow.tga", vector(0,0,-1000),
                                        eff_rudiNoseGlow_func);
   }
}
```

Damit der Sprite dann erzeugt wird, muss Rudi nur noch in seiner Initialisationsfunktion den Aufruf von eff_rudiNoseGlow(); tätigen!

Lagerfeuereffekt

Wir haben bereits einige Lagerfeuermodelle ins Level gestellt und Funktionen (obj_fireOn und obj_fireOff) zugewiesen, die angeben sollen, ob an das Lagerfeuer an dem Modell auch brennen soll. Nun wollen wir diesen Effekt auch programmieren.

So ein Feuer besteht intuitiv gesehen nur aus Feuer und Rauch. Dies wollen wir mit Hilfe zweier Partikeleffekte ausdrücken. Dafür liegen zwei Grafiken bereit: „eff_light_fire.tga“ für das Feuer und „eff_light_smoke.tga“ für den Rauch. Das Verhalten der Partikel ist recht primitiv – im Prinzip sollen die Feuerpartikel nur ausfaden und irgendwohin schwirren. Die Funktion sieht so:

```
void eff_bigFire_fire (PARTICLE *p)
{
   vec_set(p.vel_x, vector(random(2) - 1, random(2) - 1, random(2) + 1));
   p.alpha = 40 + random(20);
```

```
   p.bmap = bmapCheck(eff_bigFire_fireBmap, "eff_light_fire.tga");
   p.size = 60 + random(50);
   set(p, MOVE | BRIGHT);
   p.lifespan = 100;
   p.event = eff_bigFire_fire_func;
}

   void eff_bigFire_fire_func (PARTICLE *p)
   {
      p.size -= 0.5 * time_step * (p.size > 1);
      p.alpha -= 1 * time_step;
      p.lifespan = (p.alpha > 0) * 10;
   }
```

Der Raucheffekt sieht in etwa genauso aus, nur wollen wir dort eine kleine Besonderheit einbauen: ein Partikel kann auch eingefärbt werden, was wir dafür nutzen können, dass der Rauch, je weiter er „weggezogen“ ist, dunkler wird. Wir wollen dies in einer Abhängigkeit zur Alpha-Transparenz ausdrücken, da wir den Rauch ausfaden lassen wollen. Außerdem soll der Rauch auch etwas weiter oben starten, weil der Rauch in der Regel am oberen Ende des Feuers entsteht. Der Code sieht daher so aus:

```
void eff_bigFire_smoke (PARTICLE *p)
{
   p.z += 50 + random(50);
   vec_set(p.vel_x, vector(random(1) - 0.5, random(1) - 0.5, random(4) + 2));
   p.alpha = 50 + random(10);
   p.bmap = bmapCheck(eff_bigFire_smokeBmap, "eff_light_smoke.tga");
   p.size = random(80)+20;
   set(p, MOVE);
   p.lifespan = 100;
   p.event = eff_bigFire_smoke_func;
}

   void eff_bigFire_smoke_func (PARTICLE *p)
   {
      p.size      -= time_step * (p.size > 1);
      p.alpha     -= 1 * time_step;
      p.lifespan  = (p.alpha > 0) * 10;

      p.red    = p.alpha * 2.5;
      p.green  = p.red;
      p.blue   = p.red;
   }
```

Wir können nun die Funktion obj_fireOn mit dem Effekt ausstatten, indem wir in bestimmten Zeitabständen den Effekt am Ort des Lagerfeuers starten:

```
void obj_fireOn ()
{
   set(my, PASSABLE);

   while (1) {

      if (my.skill1 > 3) {

         effect(eff_bigFire_fire, 1, my.x, nullvector);
         effect(eff_bigFire_smoke, 1, my.x, nullvector);

         my.skill1 -= 3;

      }

      my.skill1 += time_step;

      wait(1);
   }
}
```

Wenn wir allerdings mehrere Feuerstellen im Level haben, würde an jeder Entity zu jeder Zeit der Effekt gestartet werden. Um dies zu vermeiden, könnten wir ja immer genau dann den Effekt starten, wenn die Lagerfeuer erst im Bildschirm sind. Dass können wir so abfragen: jede Entity hat ein Flag namens CLIPPED, dass aussagt, ob eine Entity im letzten Frame angezeigt worden ist oder nicht. Wir können dies nun abfragen und dann im diesem Fall den Effekt starten. Die Entityflags stehen in der Eigenschaft my.eflags, die Abfrage sieht dann so aus:

```
while (1) {

   if (!(my.eflags & CLIPPED)) {

      if (my.skill1 > 3) {

         effect(eff_bigFire_fire, 1, my.x, nullvector);
         effect(eff_bigFire_smoke, 1, my.x, nullvector);

         my.skill1 -= 3;

      }

      my.skill1 += time_step;

   }

   wait(1);
}
```

Ein Schatten für Rudi - gefakedte Softshadows

Bisher läuft Rudi schattenlos durch das Level, was nicht unbedingt ideal ist. Erst durch einen Schatten können wir dem Spieler ein echtes Gefühl von Tiefenwahrnehmung vermitteln. Wir könnten für Rudi einen sogenannten Stencil-Shadow geben, der die echte Form von Rudi auf den Grund wirft. Das ist zwar sehr dynamisch und auch irgendwie realistisch - aber dann auch wieder nicht, da ein Stencil-Schatten scharfkantig ist. Eine Alternative ist ein sogenannter Soft-Shadow Shader, der einen scharfen Schatten berechnet und dann mit einem Unschärfefilter "weich", also realistisch, macht. Soetwas finden Sie in aktuellen Spielen und ist mittlerweile zur Praxis geworden. Wir wollen für Rudi nicht darauf verzichten, nur wollen wir auch keine große Leistung darauf verschwenden. Ich will Ihnen in diesem Abschnit erklären, wie Sie in gewissem Maße Soft-Shadows nachbilden können, ohne dass Sie einen Soft-Shadow Shader benutzen müssen.

Es gibt einige Gründe, warum man in der Lage sein kann, Softshadows zu faken. Zunächst einmal ist das Level komplett von oben beleuchtet - wir müssen also keine speziellen Sonnenwinkel in Betracht ziehen. Desweiteren haben wir über die Vertexe des Modells Informationen über den aktuellen Zustand des Meshes. Kennen Sie Blop-Shadows? Das sind Schattensprites, die rund sind und nach außen wegfaden. Man legte sie früher unter Modelle, um deren Schatten zu simulieren. Im Prinzip wird der Blop-Shadow immer an der Mitte des Modells ausgerichtet. Rudi hat aber z.B. Hufe, die sich die ganze Zeit bewegen und zwar außerhalb der Reichweite eines Blop-Shadows,

wenn er unter Rudi liegen würde. Der Trick ist, dass wir nun ganz viele Blop-Shadows benutzen wollen und die Position des Sprites über die Vertexe des Models bestimmt wird.

Wir hängen also an alle 4 Hufe, dem Kopf und an einigen zentralen, wichtigen Körperstellen von Rudi mehrere Blop-Shadows dran, indem wir sie an Vertexe "anheften". Für die Höhe könnten wir jedes Mal einen Trace nach unten schicken, um den genauen Auftreffpunkt zu bestimmen, aber Rudi steht zumeist komplett auf dem Boden, sodass wir den Grund der Boundinbox als Basis für die Höhe des Sprites nehmen. Wir wollen das so programmieren: wir rufen eine Funktion auf, die die Ziel-Entity entgegennimmt, den Vertex und einen Wert für die Größe, bzw. die Skalierung des Sprites. Der Sprite für den Schatten liegt als "eff_shadowSpot.dds" im effects-Ordner vor. Die Funktion nennen wir effShadSoft und lautet so:

```
void effShadSoft(ENTITY* ent, int vertex, var scale)
{
   // Create spot
      ENTITY* spot = ent_create("effShadowspot.dds", vector(0,0,-1000), effShadSoft_ent);

   // Scale it
      spot->scale_x = scale;
      spot->scale_y = scale;

   // Pass parameters
      spot->skill1 = ent;     // Attached entity
      spot->skill2 = vertex;  // Attached vertex
}
```

Wir erledigen alles hier bis auf die Ausrichtung - das soll die Entity selber machen. Als Parameter werden in den Skill1 und 2 die Entity, die mit dem Schatten ausgestattet wird, übergeben und den Vertex, an den sich der Schatten-Blop dranheftet.

Bevor wir uns nun an die Entityfunktion effShadSoft_ent machen, müssen wir einige Vorüberlegung anstellen. Zunächst einmal weiß eine Entity nichts davon, dass sie einen Schatten hat. Das macht das ganze mit der Detailstufe auch einfacher, weil entweder wir erzeugen einen Schatten - oder nicht. Die betroffene Entity sollte diese nicht auch noch managen. Normalerweise lebt ja auch eine Entity nur in einem Level, bis sie entfernt oder das Level gewechselt wird. Bei einem Wechsel wäre das ja auch nicht so schlimm, da dann die betroffene Entity und die Schatten-Sprites gemeinsam gelöscht werden. Bei ersterem ist es aber ein Problem. Bei ersterem müssen wir dafür sorgen, dass eine Entity ihre Schatten - falls vorhanden - beliebig löschen kann. Dies wollen wir über ein Skilldefine lösen, dass wir dann ganz einfach auf 1 setzen, damit die Schatten gelöscht werden. Das define definieren wir in effects.h:

```
#define deleteShadow skill13
```

Ansonsten sieht der Code so aus:

```
void effShadSoft_ent ()
{
   wait(3); // Wait for valid parameter values

   // Initialization
      set(my, PASSABLE | TRANSLUCENT); // Passable and not solid
      my.tilt = 90;                    // Flat on the floor

   while (1) {

      you = (ENTITY*)(my.skill1);   // Get entity
      if (you) {                    // If it exists...

         // If the engine explicitely wants the shadow to
         // be destroyed, do it

            if (you.deleteShadow) {
               break;
            }

         // Get vertex and pose me there
            vec_for_vertex(my.x, you, my.skill2);
```

```
            // Move to the feet
               my.z = you.z + you.min_z + 10;
               //We add some quants, because otherwise the sprite would flicker

         } else { // Entity doesnt exists
            break;
         }

         wait(1);
      }

      // Remove shadow
         ent_remove(my);
   }
```

Erst warten wir kurz, damit die Skills auch richtig gefüllt sind (die Parameter in Skill1 und 2!), dann schalten wir den Sprite auf passabel und transparent. In der Schleife lesen wir dann in jedem Frame die globale Position des Vertex aus und setzen den Sprite auf diese Position. Daraufhin berechnen wir die Höhe des Grundes der Boundingbox und setzen uns auf diese Höhe. Wir rechnen vorher nochmal ein paar Quants drauf, weil sonst der Sprite in einer Ebene mit dem Grund wäre, was zu einem Flackern führen würde. Wird während der Laufzeit der deleteShadow Skill der Zielentity aktiviert, dann löschen wir uns.

Wir können nun in Rudis Code in der Init-Funktion eine Funktion aufrufen, die den gefakedte Softshadow aktiviert:

```
void pl_rudi_init ()
{
   //...

   pl_rudi_init_shad();
}
```

Die Funktion sieht so aus:

```
void pl_rudi_init_shad ()
{
   effShadSoft(Rudi, 313, 4);
   effShadSoft(Rudi, 169, 4);
   effShadSoft(Rudi, 323, 4);
   effShadSoft(Rudi, 179, 4);
   effShadSoft(Rudi, 27, 4);
   effShadSoft(Rudi, 22, 4);
   effShadSoft(Rudi, 18, 4);
}
```

Natürlich sind die Vertexzahlen abhängig von dem Modell - die Werte sind jetzt perfekt auf das "Rudi.mdl" Modell zugeschnitten - bei einem anderen Modell müssen Sie natürlich andere Werte eingeben! Wenn Sie jetzt das Level starten, hat Rudi einen schönen, dynamischen und Softshadow-ähnlichen Schatten unter sich - toll! Aber Achtung: Sie sollten, bevor Sie in pl_death_crash ein Level neustarten,

```
my.deleteShadow = 1;
```

setzen, damit die alten Schatten gelöscht werden, bevor die neuen erstellt werden!

Simple Schatten-Sprites (mit und ohne trace)

Da wir auch unsere herumlaufenen Pinguine / Eskimos und auch die herumfliegenden Möwen mit Schatten versehen wollen (oder besser sollten! - damit wird nämlich eine bessere Tiefenwahrnehmung erzeugt), aber nicht dutzende Sprites ausgeben wollen - wie bei Rudi - , nehmen wir einen einfachen Schattensprite und legen ihn an die Füße der Entity. Manche Entities bewegen sich immer auf dem Boden (die Eskimos, bzw. die Pinguine), sodass wir einfach den Sprite ausrichten müssen. Wenn wie im Fall der Möwen die Entity „irgendo“ ist, müssen wir einen trace-Strahl nach unten schicken, um zu wissen, wo der Boden ist. Außerdem sollte jede Entity bestimmen dürfen, was für einen Schatten-Sprite sie benutzt, damit der Sprite eventuell an die Gestalt der Entity angepasst ist.

Zunächst schauen wir uns die Lösung ohne Trace an. Dazu rufen wir eine Funktion auf, die einen Schattensprite erzeugen soll:

```
void effShadBlob (ENTITY* ent, var scale, char* file)
{
   ENTITY* spot = ent_create(file, vector(0,0,-1000), effShadBlob_ent);
   spot->scale_x = scale;
   spot->scale_y = scale;
   spot->skill1 = ent;
}
```

Der Parameter ent gibt die Entity an, zu der der Schatten zugehörig ist, eine Skalierung und den Dateinamen des Sprites. Der Sprite wird dann erstellt und die Parameter in die Skills geschrieben. Der Code ist dann eigentlich sehr einfach:

```
void effShadBlob_ent ()
{
   wait(3);

   proc_mode = PROC_LATE;

   set(my, PASSABLE | TRANSLUCENT);
   my.tilt = 90;

   while (1) {

      you = (ENTITY*)(my.skill1);
      if (you) {

         if (you.deleteShadow) {
            break;
         }

         vec_set(my.x, you.x);
         my.z += you.min_z + 5;
         my.pan = you.pan;

      } else {
         break;
      }

      wait(1);
   }

   ent_remove(my);
}
```

Die Entity schaut, ob ihre zugehröige Entity da ist und ob das deleteShadow Flag nicht gesetzt ist. Andernfalls löscht sie sich. Ist sie weiterhin existent, orientiert sie sich am unteren Boden der Kollisionshülle und bewegt sich nochmal ein paar Quants höhe, um keine Überlappungseffekte zu erzeugen.

Wir können den Schatten nun ganz einfach zuweisen. Für die patrouillierenden Eskimos/Pinguine machen wir dann einfach Folgendes:

```
void npc_penPatrol ()
{
   //...
   effShadBlob(my, 4, "effShadPen.tga");
}

void npc_eskPatrol ()
{
   //...
   effShadBlob(my, 5, "effShadEsk.tga");
}
```

Für die Möwen brauchen wir wie gesagt eine Schattenfunktion, die mit einem Trace-Strahl arbeitet. Die Erzeugerfunktion sieht dann genauso aus:

```
void effShadBlobTrace (ENTITY* ent, var scale, char* file)
{
   ENTITY* spot = ent_create(file, vector(0,0,-1000), effShadBlobTrace_ent);
   spot->scale_x = scale;
   spot->scale_y = scale;
   spot->skill1 = ent;
}
```

aber die Hauptfunktion ist anders:

```
void effShadBlobTrace_ent ()
{
   wait(3);

   set(my, PASSABLE | TRANSLUCENT);
   my.tilt = 90;

   while (1) {

      you = (ENTITY*)(my.skill1);
      if (you) {

         if (you.deleteShadow) {
            break;
         }

         vec_set(vecTemp.x, you.x);
         vecTemp.z -= 0.5 * (you.max_z - you.min_z);
         vec_set(my.x, vecTemp.x);
         vecTemp.z -= 10000;

         c_trace(my.x, vecTemp.x, IGNORE_ME | IGNORE_YOU | IGNORE_PASSABLE | IGNORE_SPRITES);

         vec_set(my.x, target.x);
         my.pan = you.pan;
         my.z += 5;

      } else {
         break;
      }

      wait(1);
   }

   ent_remove(my);
}
```

Anstatt den Sprite nun an der unteren Fläche der Boundingbox auszurichten, tracen wir nach unten und stellen dann die richtige Höhe ein. Die Möwen erhalten demnach mit

```
void npc_seagull ()
{
   if (detail == DETAIL_LOW) {
      ent_remove(my);
      return;
   }

   effShadBlobTrace(my, 2, "effShadSeaGull.tga");

   //...
```

einen getracedten Schattensprite.

Kapitel 14: Musik

Der Spielsoundtrack

Wir wollen für unser Spiel schöne Weihnachtslieder benutzen. Allerdings muss man solche Musik erstmal besitzen, um sie einsetzen zu können. Wir können – auch wenn wir keine Gewinnabsicht oder soetwas haben – keine Musik von Künstlern benutzen, die kein Einverständniss dafür gegeben haben, dass wir ihre Musik in unserem Spiel benutzen. Glücklicherweise gibt es mittlerweile den Trend, dass viele Künstler (das reicht von Amateuren bis hin zu waschechten Profis) gegen die Kommerzialisierung von Musik vorgehen und ihre Musik frei zugänglich für den privaten Gebrauch machen. Ganz besonders im Rampenlicht stehen die sogenannten Creative Commons.

Die Creative Commons ist eine gemeinnützige Gesellschaft, die im Internet verschiedene Standard-Lizenzverträge veröffentlicht, mittels derer Künstler an ihren Werken – wie eben zum Beispiel Musik – anderen Menschen in der Öffentlichkeit gewisse Nutzungsrechte einräumen können. Man kann dabei die Lizenz nach bestimmten Kriterien zusammenmischen, um die Nutzung stark oder fast gar nicht einzuschränken. Es gibt unter anderen Musikplattformen, wie z.B. Jamendo.com, auf der haufenweise Künstler ihre Alben und Musik für kostenlos mit der Angabe der jeweilige CC Lizenz hochladen.

Die Musik, die ich für Rudi ausgesucht habe, stammt von einem amerikanischen Künstler namens Kevin MacLeod, der seine Musik auf seiner Seite Incompetech.com kostenlos auf Basis dieser Creative Commons Lizenzen anbietet. Er hat – zufällig – als ich nach Musik für Rudi gesucht hatte, selber ein Set von verschiedenen „lustigen“ Weihnachtsliedern, die zum Schunkeln anregen, gearbeitet. Deshalb hat das Spiel auch so eine breite Palette an Musiken in Petto, das Spiel hat quasi einen eigenen Soundtrack.

Natürlich kann man fragen, wofür man so viele Lieder braucht.. nunja. Wenn man es ganz nüchtern betrachtet, kann man sich die Gründe, bzw. die Situationen, in denen man Musik im Spiel hört, an einer Hand abzählen:

- in der Splashscreensequenz
- im Menü
- im Abspann
- im Game-Over – Bildschirm
- und natürlich im Spiel, also im Level, selbst

Da der Spieler so ein Level wahrscheinlich länger als ein oder zwei Minuten spielt, bietet es sich an, ein paar Lieder hintereinander abzuspielen, damit etwas Abwechslung in die ganze Sache reinkommt – aus diesem Grunde sieht man schnell, dass man eine etwas breitere Palette an Musikstücken benötigt, um das Spiel nicht musikalisch eintönig werden zu lassen.

Das Musikformat

Das Format der Musikstücke ist auch ein Thema für sich. Jeder kennt einige geläufige Musikformate wie Wave, MP3 oder WMA.. In Spielen darf man aber nicht jedes Format benutzen, was man will, bzw. man sollte auch einige Format nicht benutzen. Das Wave-Format ist für Musik total ungeeignet, da die Dateien riesengroß sind für ein dreiminütiges Lied. Bei MP3 Dateien muss man Lizenzgebühren bezahlen. Bei WMA weiß ich über die Rechtslage nicht genau Bescheid, aber es könnte auch sein, dass Microsoft dafür Lizenzgebühren verlangt. Allerdings gibt es einen Ausweg aus dieser Misere: das OGG (-Vorbis) Format. OGG ist ein open-source Audioformat. Was bedeutet, dass man es kostenlos benutzen darf, ohne Lizenzgebühren zu bezahlen. Der Pferdefuß ist aber, dass man eventuell dafür sorgen muss, dass das OGG Format beim Endbenutzer installiert ist, sonst kann der Spieler die Musik nicht hören. Dies kann man am geschicktesten über eine Integrierte Installation der Treiber bei der Spiel-Installation lösen.

Wie auch immer: das OGG-Format speichert Musikdateien ähnlich wie das MP3-Format sehr stark komprimierend ab – Musikdateien sind in der Regel nicht sehr groß. Mit dem kostenlos nutzbaren Tool „Audacity“ kann man

übrigens Musik als OGG Dateien exportieren und dabei auch die Kompressionseigenschaften kontrollieren, sodass man wirklich sehr kleine Musikdateien erzeugen kann. Unser Musik werden wir im OGG-Format abspeichern, damit auch die Größe des Spiels an sich nicht sehr groß ist. Die Dateien habe ich im „media“ Ordner gespeichert, da die Dateien gestreamedt werden müssen.

Die Musikimplementierung

Für den Einbau der Musik wollen wir ein eigenes Modul eröffnen, dass wir in Form der Dateien „music.c“ und „music.h“ neu erstellen und in der „rudi.c“ inkludieren. Die Idee ist, dass wir für jede Situation, in der wir Musik abspielen, ein eigenes Handle zur Verfügung stellen. Werden Streams über die Engine geöffnet, erhalten wir eine Kennnummer des Streams, die man „Handle“ nennt. Mit diesem Handle haben wir dann Zugriff auf den Stream, um ihn z.B. zu stoppen oder die Lauststärke zu ändern. Wir oben beschrieben, bereiten wir in der „music.h“ folgende Handles vor:

```
var music_splash;
var music_menu;
var music_level;
var music_gameOver;
var music_credits;
```

Zusätzlich zu diesen Handles müssen wir für die Musikimplementierung auch wissen, welche Dateien wir abspielen wollen. Wie bereits oben gesagt, werden alle Situationen bis auf das Level mit einer Musik bedacht. Wir definieren die abzuspielende Musik über Zeichenketten:

```
char* music_splashFile     = "media\\jinglebells.ogg";
char* music_menuFile       = "media\\jinglebells2.ogg";
char* music_gameOverFile   = "media\\deckthehalls.ogg";
char* music_creditsFile    = "media\\sugarplum.ogg";
```

Für die Musiken der Leveldatei wollen wir eine Liste von Zeichenketten bereithalten. Außerdem müssen wir wissen, welchen Eintrag der Liste als nächstes kommt. Für die Liste nehme ich ein TEXT-Objekt und für das Festhalten der aktuell abgespielten Musik eine Variable:

```
int music_levelFiles_current = 0;
TEXT* music_levelFiles = {
   strings = 4;
   string = "media\\jinglebells.ogg",
            "media\\housetop.ogg",
            "media\\jinglebells2.ogg",
            "media\\deckthehalls2.ogg";
}
```

In der Datei „music.c“ wollen wir nun so vorgehen: für jede Situation definieren wir zwei Funktionen, die jeweils den Abspielvorgang starten und stoppen. Das Beispiel für die Splashscreen-Sequenz ist ganz einfach gehalten:

```
void music_splash_start ()
{
   music_splash = media_play(music_splashFile, NULL, volMusic);
}

void music_splash_stop ()
{
   media_stop(music_splash);
}
```

Es wird einfach die Datei gestartet und das Handle in die Variable geschrieben. Für die Lautstärke benutzen wir volMusic, dessen Wert im Konfigurationsmenü einstellbar ist. Wir werden im Folgenden für jede Situation so eine Start/Stop Funktion bauen, sodass die Prototypenliste in der Datei „music.h“ so aussieht:

```
void music_splash_start();
void music_splash_stop();
void music_menu_start();
```

```
void music_menu_stop();
void music_level_start();
void music_level_stop();
void music_gameOver_start();
void music_gameOver_stop();
void music_credits_start();
void music_credits_stop();
```

Damit wir nicht jedesmal bestimmte Streams explizit ausschalten müssen, wollen wir in den Startfunktionen alle Stop-Funktionen aller möglichen Vorgänger-Situationen aufrufen, um das automatisch zu erledigen. Das Menü kann z.B. aus allen Situationen aus aufgerufen werden, sodass die Implementierung so aussieht:

```
void music_menu_start ()
{
   music_splash_stop();
   music_level_stop();
   music_gameOver_stop();
   music_credits_stop();

   music_menu = media_play(music_menuFile, NULL, volMusic);
}

void music_menu_stop ()
{
   media_stop(music_menu);
}
```

Die Implementierungen der Funktionen für den Game Over – Bildschirm und den Abspann sehen ähnlich aus:

```
void music_gameOver_start ()
{
   music_level_stop();

   music_gameOver = media_play(music_gameOverFile, NULL, volMusic);
}

void music_gameOver_stop ()
{
   media_stop(music_gameOver);
}

void music_credits_start ()
{
   music_menu_stop();
   music_level_stop();

   music_credits = media_play(music_creditsFile, NULL, volMusic);
}

void music_credits_stop ()
{
   media_stop(music_credits);
}
```

Die Implementierung für das Level ist nun ein wenig trickreicher. Wir müssen nämlich ziemlich viel tun. Zunächst müssen wir dafür sorgen, dass die Variable music_levelFiles_current immer zwischen 0 und der Länge der Musik-Liste - 1 ist. Dann müssen wir diese Variable nutzen, um den entsprechenden Eintrag abzuspielen – wir müssen als auf die Strings in dem TEXT Objekt zugreifen. Dann lassen wir die Funktion solange warten, bis der Stream zuende ist oder abgebrochen worden ist. Wurde er nicht abgebrochen, erhöhen wir music_levelFiles_current um 1 und starten dann das nächste Lied usw. Dies erledigt die folgende Funktion:

```
void music_level_start ()
{
   music_menu_stop();

   music_levelFiles_current %= (int)(music_levelFiles->strings);
   music_level = media_play((music_levelFiles.pstring)[music_levelFiles_current], NULL, volMusic);

   while ((music_level != 0) && (media_playing(music_level))) {wait(1);}
```

```
   wait(1);

   if (media_playing(music_menu))      {return;}
   if (media_playing(music_gameOver))  {return;}
   if (media_playing(music_credits))   {return;}

   music_levelFiles_current++;
   music_level_start();
}
```

Am Ende ruft sie sich rekursiv auf, aber auch nur dann, wenn das Lied zuende war. Dann wird durch den rekursiven Aufruf das nächste Lied abgespielt. Durch die Modulorechnung am Anfang vermeiden wir einen out-of-bounds Fehler beim Zugriff auf die Stringliste und erreichen damit außerdem noch, dass nach dem Durchspielen aller Lieder die Liste von neuem startet. Die Stop-Funktion ist wieder recht trivial:

```
void music_level_stop ()
{
   media_stop(music_level);
   music_menu = 0;
}
```

Die Musiksteuerung

Wir haben zwar die Musik an sich implementiert, aber steuern die Ausgabe noch nicht. Ist ja auch klar, die Startfunktionen werden auch noch gar nirgends aufgerufen!

Wir wollen einigermaßen nach Reihenfolge vorgehen. Als erstes starten wir die Musik für die Splashscreens. Dazu fügen wir einfach in der Funktion splash_show den entsprechenden Aufruf hinzu:

```
void splash_show ()
{
   splash_init ();   //load all ressources
   wait(3);          //tripple buffering

   splash_ready = 0; //reset global flag

   music_splash_start();

   //(...)
```

Danach ist das Menü dran. Wie bereits kurz erwähnt, wird das Menü aus allen möglichen Situationen heraus aufgerufen. Zunächst wird nach den Splashscreens einmalig die Funktion menu_main zur Initialsierung des Hauptmenüs aufgerufen. Dort fangen wir an:

```
void menu_main ()
{
   menu_init();

   // First time: play music
      music_menu_start();

   while (1) {

      //(...)
```

Wenn der Spieler das Spiel durch das Ingame-Menü zurück ins Hauptmenü verlässt:

```
void menu_game_mainmenu ()
{
   menu_mode = MENU_MODE_MAIN;

   music_menu_start();

   wait(1);
   level_load("");
```

```
   freeze_mode = 0;
}
```

Wenn der Spieler tot ist und zurück ins Hauptmenü geht:

```
void menu_gameover_main ()
{
   music_menu_start();
   menu_mode = MENU_MODE_MAIN;
}
```

Schlussendlich, wenn der Spieler den Abspann sieht und dann ins Hauptmenü zurückgeleitet wird:

```
void cr_tfp ()
{
   //(...)

   // Go back to main menu
      music_menu_start();
      menu_main();
}
```

Als nächstes schauen wir uns die Musik für das Level an. Wichtig ist, dass wir die Musik solange kontinuierlich spielen, wie wir noch leben. Wenn Rudi also gegen eine Wand läuft und sofort wieder wiederhergestellt wird, soll die Musik nicht „neu“ starten, sondern einfach weiterlaufen. Folglich starten wird die Musik erstmal, wenn das erste Level geladen wird:

```
void game_start ()
{
   //(...)

   level_load("northpole.wmb");
   music_level_start();

   //(...)
}
```

Da wir noch keinen Level-Lader für weitere Levels gebaut haben, sind wir für die Musik des Levels auch schon fertig. Das nächste ist der Game Over – Bildschirm. Die Musik rufen wir auch dort auf, wo die Leveldatei geladen wird:

```
void pl_death_crash ()
{
   //(...)

   if (lifes == 0) {

      music_gameOver_start();

      //(...)
}
```

Die Creditsmusik wird genau dann abgespielt, wenn die Credits in der Kamera-Entity der Credits gestartet werden:

```
void cr_camStart ()
{
   //(...)

   // Play credits music
      music_credits_start();
```

Das wars auch schon: die Musik ist erfolgreich eingebunden!

Kapitel 15: Detailstufen und Optimierungen

Der Grasseffekt

Wir haben zwar bereits den Grass-Shader eingebaut, aber Sie haben vielleicht bemerkt, dass wenn Sie ganz viele Büschel ins Level stellen, dass die Leistung der Engine (je nach Stärke ihres Rechners), in die Knie geht. Wir haben bereits die Variable detail eingeführt, die die Werte DETAIL_LOW, DETAIL_MEDIUM und DETAIL_HIGH annehmen kann. Wir können z.B. nur dann den Grasswaving Shader starten, wenn der Spieler die höchste Detailstufe angewählt hat:

```
void fx_wavingGrass ()
{
   if (detail == DETAIL_HIGH) {
      my.material = mtl_wavingGrass;
   }
}
```

Shiny shader

Der Shiny-Shader wird von vielen Objekten benutzt. Allerdings, wenn wir den Shader in Verbindung mit sehr vielen Objekten einsetzen, dann geht die Performance doch beträchtlich in die Kniee. Deshalb sollte man bei Objekten, die jetzt nicht standardmäßig den Shader besitzen sollen, eine Abfrage der Art

```
if (detail == DETAIL_HIGH) {
   fx_shiny(fx_cube_shiny);
}
```

hinzufügen. Dies betrifft insbesondere die Objekt-Funktionen env_iceberg, obj_gate und obj_gateEnvr. Die Eisschollen sind etwas öfter im Bild, dort habe ich dann den Shader ab dem mittleren Detailgrad eingeschaltet:

```
if (detail >= DETAIL_MEDIUM) {
   fx_shiny(fx_cube_shiny);
}
```

Schnee

Den Schneeeffekt haben wir bereits eingebaut und auch schon eine Methode gefunden, wie wir eine sehr große Menge an Schneeflocken als Partikel umgehen: indem wir eine größere Bitmap mit mehreren Schneeflocken benutzen. Allerdings wirkt sich der Schneefall genau wie alle anderen Effekte auf die Leistung aus. Die Funktion env_snowMedium erstellt standardmäßig einen mittleren Schneefall. Was ist aber, wenn wir einen niedrigeren Detailgrad eingestellt haben? In diesem Fall wäre es gut, wenn wir den Schneefall reduzieren würden. Wir könnten z.B. über einen switch-case-tree verschiedene Generatorfunktionen aufrufen, wobei der eigentliche Schneeeffekt erst im hohen Detailgrad voll zur Geltung kommt. Darunter werden wir eine Vorlage für weniger Schnee benutzen. Das sieht dann so aus:

```
void env_snowMedium ()
{
   switch (detail) {
      case DETAIL_LOW:     eff_snow_verylight(); break;
      case DETAIL_MEDIUM:  eff_snow_light(); break;
      case DETAIL_HIGH:    eff_snow_medium(); break;
   }

   ent_remove(my);
}
```

Die Generatorfunktionen in der Datei effects.c sehen dann so aus:

```
void eff_snow_medium ()
{
   vec_set(eff_wind_dir, vector(10,10,-25));
   eff_snow_create(vector(2500,2500,2250), 50);
}

void eff_snow_light ()
{
   vec_set(eff_wind_dir, vector(10,10,-25));
   eff_snow_create(vector(2000,2000,2000), 25);
}

void eff_snow_verylight ()
{
   vec_set(eff_wind_dir, vector(10,10,-25));
   eff_snow_create(vector(1800,1800,1800), 12);
}
```

Kapitel 16: Das Spiel fertig verpacken

Das folgende Kapitel wurde freundlicherweise von Ulf Ackermann zur Verfügung gestellt.

Nach vielen schweisstreibenden Stunden ist das Spiel nun fertig und wir wollen es in kompilierter Form unter das Volk bringen. Dafür brauchen wir ein Programm, welches aus den Spieldateien eine fertige Installation erzeugt, mit der die Spieler in der Lage sind, das Spiel problemlos zu installieren bzw. zu deinstallieren.

Zum Kompilieren des Spiels benutzen Sie einfach die makeEXE.bat, die Sie am Anfang des Workshops kennengelernt haben. Angenommen, Sie möchten nun doch ein anderes Icon als Jenes, mit dem A7 Logo für Ihre ausführbare Spieldatei RUDI.exe benutzen. Dazu legen sie vor dem Kompilieren eine Sicherheitskopie der acknex.exe im Gamestudio Verzeichnis an. Jetzt können Sie mit einem Ressourceneditor (z.B. ResHacker) das Icon der original acknex.exe nach Ihren Wünschen abändern und speichern. Kompiliert man nun das Spiel erneut, so wird eben dieses abgeänderte Icon der acknex.exe für die RUDI.exe benutzt.

Bevor wir uns nun ein Installationsprogramm erzeugen, räumen wir erst einmal den Spieleordner auf. Dazu können Sie alle Dateien im Spieleordner löschen, die temporär sind und die zum Beispiel der WED benötigt um Arbeitseinstellungen zu speichern. Dazu gehören Daten der Endung .$$*, .bak, .wed und die unkompilierten Level im .wmp Format.

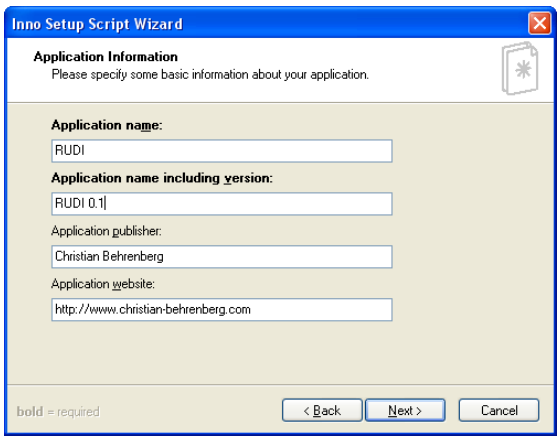
Beachten Sie bitte unbedingt, dass bei RUDI und allen Spielen die mit der GPL Lizenz versehen sind unbedingt der Sourcecode beiliegen muss. Im Falle von A7 als Engine sind das die Dateien mit der Endung .c bzw. .wdl in denen Ihre neu erworbenen Programmierkenntnisse für die Ewigkeit konserviert sind. Die zugehörigen Headerfiles mit der Endung .h und Shadercode mit der Endung .fx., genauso wie die Content-Daten wie z.B. wmp, mdl, bmp, dds, tga, pcx usw.

Angenommen Sie haben also etwas am Code von RUDI geändert und möchten gern Ihre eigene Version mit verbesserten Shadern in Umlauf bringen, so müssen Sie unbedingt die Lizenztexte die bei RUDI in Form eines Dokumentes beiliegen (z.B. license.pdf) mit ausliefern – sonst verletzen Sie die GPL Lizenz des Spiels und das wollen Sie ja nicht.

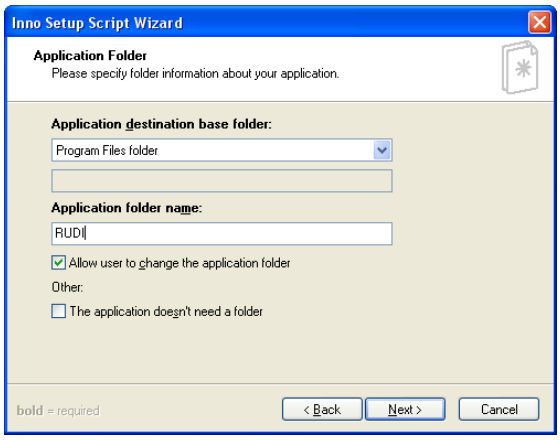
Im Endeffekt muss bei einem Spiel ohne GPL Lizenz jedoch nur die .exe Datei erhalten bleiben, die fertig kompilierten Level im .wmb Format, sowie alle anderen Dateien die nicht unmittelbar zum Spiel gehören. Denken Sie dabei zum Beispiel an ein Handbuch im .pdf Format, welches sich logischerweise auch im Spieleordner befindet.

Um nun aus dem Spiel eine fertige Installation zu erzeugen, besorgen wir uns von der Inno Setup Webseite das gleichnamige, kostenlose Tool unter: **http://www.jrsoftware.org/isdl.php** Dazu das 'stable release' in der 'self-installing-package' herunterladen, installieren und den Inno Setup Compiler aus der entsprechenden Gruppe im Startmenü starten.

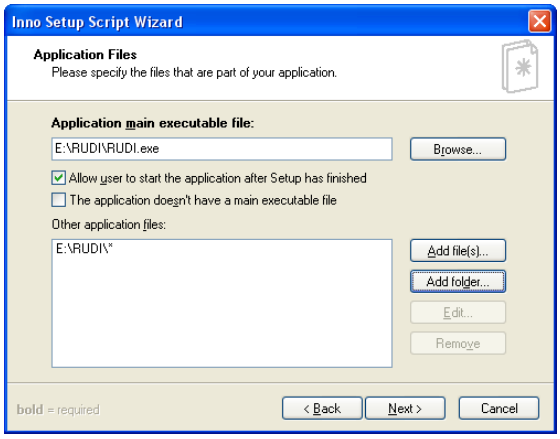
Jetzt erzeugen wir ein neues Skript indem wir auf 'File'->'New' oder das weisse Blatt in der Leiste klicken. Dann erscheint der Wizard, dort darf kein Haken bei 'Create a new empty script file' gesetzt sein. Nun können wir auf 'Next' klicken und schon einige Daten über das Spiel eintragen. Das ganze sollte dann so hier aussehen.



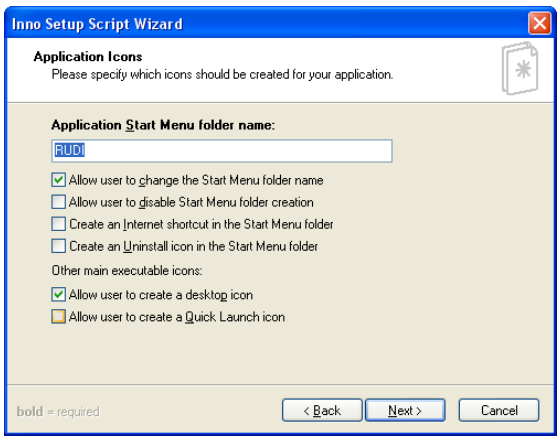
Mit einem erneuten Klick auf 'Next' fahren wir fort um einzustellen, wohin das Spiel installiert wird. In unserem Fall in das legendäre Programmverzeichnis 'Program Files' oder Deutsch 'Programme' in einen Ordner Namens RUDI. Der Haken bei 'Allow User to change the application folder' erlaubt es dem Nutzer den Ordner während der Installation selbst zu wählen.



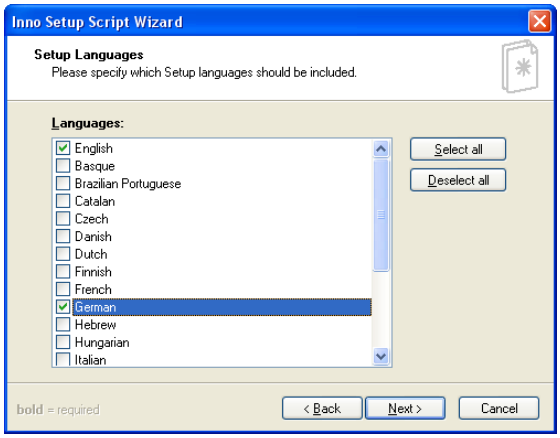
Es kann weitergehen, Sie ahnen es bereits, dazu müssen wir auf 'Next' klicken. Im folgenden Fenster stellt man die ausführbare Datei des Spiels ein, hier RUDI.exe. Natürlich müssen wir auch alle Dateien die sich im Spielordner und den Unterordnern befinden mit auf dem Zielsystem installieren und nicht nur die RUDI.exe. Dazu genügt ein klick auf 'Add folder...' und wir browsen zum Verzeichnis in dem sich das Spiel befindet. In meinem Fall ist das E:\RUDI. Die Frage 'Should files in subfolders of ... also be included' beantworten wir mit JA.
Das Resultat sollte in etwa so aussehen, bevor sie den sagenumwobenen 'Next' Button drücken dürfen.



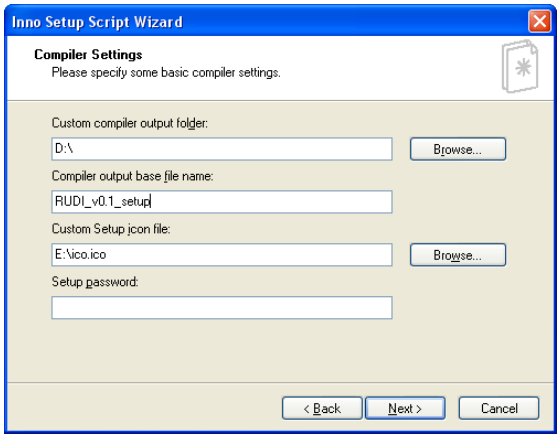
Im nächsten Fenster haben wir die Möglichkeit dem Benutzer zu erlauben den Namen der Gruppe im Startmenü zu ändern, sowie ein Icon auf dem Desktop automatisch mit zu erstellen. Da wir beides wollen, brauche wir nichts gross zu ändern. Überprüfen Sie kurz ob es bei Ihnen auch so aussieht und zappen Sie per 'Next' zum nächsten Fenster.



Dort können wir Lizenzinformationen einstellen, die der Nutzer vor der Installation aktzeptieren muss, um überhaupt forzufahren. In unserem Fall wäre das die GPL Lizenz, die als Text im Spielverzeichnis vorliegen sollte. Ich verzichte an dieser Stelle auf einen Screenshot und nehme an Sie haben bereits auf 'Next' geklickt.
Nun stellen wir die Sprachen ein, in der die Installation ausgeliefert werden soll. Also Englisch und Deutsch. Siehe folgendes Bild.



Weiter geht’s wie immer mit einem beherzten Klick auf 'Next'. Dort haben wir jetzt 4 freie Felder die nur darauf warten, gefüllt zu werden. Also los gehts! Ins Erste kommt das Verzeichnis in welches Inno Setup später nach dem kompilieren unsere Installationsdatei kopiert. Ins Zweite der Name eben dieser Installationsdatei, hier RUDI_v0.1_setup. Im Dritten können wir ein extra Icon für die RUDI Installation bestimmen, was wir auch getan haben. Das Icon mit der Endung .ico wurde vorher mit einem Programm wie z.B. (Inkscape) erstellt und in einen Ordner ausserhalb des RUDI Ordners kopiert. Wie immer ein kleines Bildchen zum eben eingetragenen.



'Next' bringt uns zum Ende des Wizards und wir dürfen endlich auf 'Finish' klicken um das Skript automatisch erzeugen zu lassen.

Jetzt ist es ganz wichtig das Skript zu speichern. Dazu klicken wir auf 'File' -> 'Save as...' und wählen einen guten Ort aus an dem wir das Skript später wiederfinden und der nicht das RUDI Verzeichnis ist.

Nun müssen wir jedoch noch einige Änderungen am Skript durchführen um dem Installationsprogramm für unser Spiel den letzten Schliff zu verpassen. Keine Angst, ich werde alles ausführlich behandeln und es ist wirklich kinderleicht. Der Grossteil ist schon geschafft und Inno Setup hat ein hervorragendes Skript für uns erstellt.

Fangen wir also an. RUDI nutzt .ogg Musikdateien, da der Nutzer der sich später das Spiel installiert, aber noch keinen directshow Filter hat, müssen wir ihm diesen in der Installation mit anbieten, damit er auch in den Genuss der Musik kommt. Dafür gibt es die Datei OggDS0995.exe welche kostenlos und im \Gstudio\media Ordner von A7 zu finden ist. Wir kopieren sie zu diesem Zweck einfach in den Spieleordner RUDI und fügen folgende Zeile hinter den Eintrag [Run] im Skript.

```
Filename: "{app}\OggDS0995.exe"; Description: "{cm:LaunchProgram,ogg filter installation";
                                 Flags: postinstall skipifsilent unchecked
```

Das hat den Effekt, dass der Spieler nach erfolgreicher Installation von RUDI noch auswählen kann, ob er gern den Filter installieren möchte, oder lieber nicht.
Auch das Handbuch möchten wir dem Benutzer nach der Installation anbieten. Wie oben schon beschrieben, erreichen wir dies so.

```
Filename: "{app}\manual.pdf"; Description: "{cm:LaunchProgram,read manual}";
                              Flags: postinstall skipifsilent unchecked
```

Wie Sie sicher bemerkt haben, steht beim Eintrag für RUDI.exe im [Run] Abschnitt zusätzlich noch das Flag nowait. Welches bewirkt, dass nicht gewartet wird bis das Spiel beendet wird um das Setup zu beenden. In unserem speziellen Fall würde das Spiel nun also erst starten, wenn die OggFilter Installation abgeschlossen ist, und der Nutzer das Handbuch gelesen und wieder geschlossen hat. Vorausgesetzt natürlich er hatte vorher alles per Haken ausgewählt, praktisch oder?

Jetzt wollen wir ein Icon auf dem Desktop erzeugen, dazu setzen wir einfach einen voreingestellten Haken, indem wir unter [Tasks] das Flag 'unchecked' entfernen. Die einzigste Zeile die dort noch stehen darf sieht nun so aus.

```
Name: "desktopicon"; Description: "{cm:CreateDesktopIcon}";
                     GroupDescription: "{cm:AdditionalIcons}";
```

Wir möchten ausserdem, dass der Haken bei Spiel starten, den das Skript automatisch für nach der Installation eingestellt hat verschwindet. Dazu reicht es das Attribut 'unchecked' hinten an den 'Flags:' Teil für den RUDI.exe Eintrag anzufügen. Fertig sieht das so aus.

```
Filename: "{app}\RUDI.exe"; Description: "{cm:LaunchProgram,RUDI}";
                            Flags: nowait postinstall skipifsilent unchecked
```

Das hätten wir, weiter im Programm. Jetzt fällt Ihnen plötzlich ein, sie wollten gern, dass man das Spiel doch lieber mit 'start RUDI v0.1' im Startmenü und auf dem Desktop starten sollte.
Dazu ändern Sie einfach beide Einträge unter [Icons] in folgenden Text.

```
Name: "{group}\start RUDI v0.1"; Filename: "{app}\RUDI.exe"
Name: "{commondesktop}\start RUDI v0.1"; Filename: "{app}\RUDI.exe"; Tasks: desktopicon
```

Sie ergänzen also einfach den Text um RUDI, so dass es so ausschaut wie bei mir. Jetzt wollen Sie der Einfachheit halber gern noch das Handbuch mit in das Startmenü packen. Nichts einfacher als das! Fügen wir einfach folgende Zeile zwischen die beiden, die wir gerade geändert haben. Beachten Sie das Flag shellexec, was unbedingt gesetzt sein muss, da das Handbuch ein Dokument und kein ausführbares Programm ist. Ausserdem braucht Setup das Flag waituntilterminated, damit wirklich mit dem Spielstart gewartet wird, bis der Nutzer seinen Pdf Reader schliesst.

```
Filename: "{app}\manual.pdf"; Description: "read manual";
                              Flags: postinstall skipifsilent unchecked shellexec waituntilterminated
```

Fast hätten wir vergessen das Deinstallationsprogramm auch im Startmenü anzubieten. Das geht mit folgendem Eintrag auch in der [Icons] Sektion.

```
Name: "{group}\{cm:UninstallProgram,RUDI}"; Filename: "{uninstallexe}" ; WorkingDir: {app}
```

Abrunden werden wir das Ganze, indem wir noch eine Internetadresse einrichten, so dass der Nutzer bequem nach neuen Versionen suchen kann.

```
Name: "{group}\{cm:ProgramOnTheWeb,christian behrenberg}";
      Filename: "http://www.christian-behrenberg.com"
```

Damit ist der komplizierte Teil auch schon geschafft! Jetzt verschönern wir das Setup noch ein wenig. Dazu haben Sie sicher schon 2 .bmp Bitmaps gemalt. Ein Grosses der Grösse 164x314 für den linken Rand und ein Kleines für Oben Rechts in einer 55x58 Auflösung.

Prima sieht das aus! Diese zwei bauen wir nun ein, indem wir einfach an den [Setup] Abschnitt diese zwei Zeilen anfügen.

```
WizardImageFile=E:\side.bmp
WizardSmallImageFile=E:\picto.bmp
```

Wie Sie sehen, sollten auch diese Dateien in einem Ordner liegen, der nicht der RUDI Spieleordner ist.

Jetzt heisst es noch einmal 'File' ->'Save' im Menü oder das blaue Diskettenicon in der Leiste und wir können kompilieren. Die Kompilierung wird mit Klick auf den grünen Play Button gestartet, oder per Menüeintrag unter

'Build'->'Compile'

Gratulation zu Ihrem ersten selbstgemachten Installationsprogramm. Die erzeugte .exe Datei finden Sie fertig zum Verteilen in dem Ordner, den Sie dafür im Wizard angegeben hatten. Na Erinnern Sie sich noch? Richtig! In unserem Falle war das direkt das Laufwerk D:\.

Es war mir eine Freude, Sie durch das Tutorial zu führen. -Ulf Ackermann

Hier noch einmal als Referenz das fertige Skript, wie wir es gerade gemeinsam erarbeitet haben:

```
; Script generated by the Inno Setup Script Wizard.
; SEE THE DOCUMENTATION FOR DETAILS ON CREATING INNO SETUP SCRIPT FILES!

[Setup]
AppName=RUDI
AppVerName=RUDI 0.1
AppPublisher=Christian Behrenberg
AppPublisherURL=http://www.christian-behrenberg.com
AppSupportURL=http://www.christian-behrenberg.com
AppUpdatesURL=http://www.christian-behrenberg.com
DefaultDirName={pf}\RUDI
DefaultGroupName=RUDI
OutputDir=D:\
OutputBaseFilename=RUDI_v0.1_setup
SetupIconFile=E:\RUDI\ico.ico
Compression=lzma
SolidCompression=yes
WizardImageFile=E:\side.bmp
WizardSmallImageFile=E:\picto.bmp

[Languages]
Name: "english"; MessagesFile: "compiler:Default.isl"
Name: "german"; MessagesFile: "compiler:Languages\German.isl"

[Tasks]
Name: "desktopicon"; Description: "{cm:CreateDesktopIcon}";
                     GroupDescription: "{cm:AdditionalIcons}";

[Files]
Source: "E:\RUDI\RUDI.exe"; DestDir: "{app}"; Flags: ignoreversion
Source: "E:\RUDI\*"; DestDir: "{app}"; Flags: ignoreversion recursesubdirs createallsubdirs;

[Icons]
Name: "{group}\start RUDI v0.1"; Filename: "{app}\RUDI.exe"
Name: "{group}\read manual"; Filename: "{app}\manual.pdf"; WorkingDir: {app}
Name: "{group}\{cm:UninstallProgram,RUDI}"; Filename: "{uninstallexe}"; WorkingDir: {app}

Name: "{group}\{cm:ProgramOnTheWeb,christian behrenberg}";
       Filename: "http://www.christian-behrenberg.com"

Name: "{commondesktop}\start RUDI v0.1"; Filename: "{app}\RUDI.exe"; Tasks: desktopicon


[Run]
Filename: "{app}\OggDS0995.exe"; Description: "{cm:LaunchProgram,ogg filter installation}";
Flags: postinstall skipifsilent unchecked

Filename: "{app}\manual.pdf"; Description: "read manual";
          Flags: postinstall skipifsilent unchecked shellexec waituntilterminated

Filename: "{app}\RUDI.exe"; Description: "{cm:LaunchProgram,RUDI}";
          Flags: postinstall skipifsilent unchecked nowait
```

Abschluss

Sie haben Sich nun mehr oder weniger durch diesen Workshop gearbeitet, gelesen oder vielleicht auch gequält – letztenendes haben Sie nun aber ein Spiel in der Hand, dass Sie (vielleicht) selbst programmiert haben – und darauf können Sie stolz sein! Wenn Sie das nicht getan haben, haben Sie aber zumindestens ansatzweise eine Vorstellung davon, wie anstrengend und vielleicht auch aufreibend die Umsetzung auch eines so kleinen Spieles sein kann – und die content creation haben wir noch nicht einmal richtig behandelt!

Mit dem 3D Gamestudio haben Sie eine exzellente Möglichkeit, ihre „einfachen“ oder auch „komplexen“ Spielideen relativ zügig und „relativ problemlos“ umzusetzen, da Ihnen vieles abgenommen wird. Auch wenn einige (vielleicht auch Sie?) meinen, RUDI wäre ein triviales Spiel – so irren sie sich. Sie haben fundamentale Grundsätze von vielen kleinen und großen Dingen behandelt, auf die sie immer mal wieder in ihrem Leben als Spieleentwickler treffen werden und dann mit einem gewissen Grad an Know-How dann wieder entgegentreten können.

Sie haben gelernt, wie Sie herausfinden, was Sie eigentlich für ein Spiel „wollen“, was Sie für Kernmechanismen benötigen, was für technische Vorraussetzungen und Limits es gibt, wie sie performanten C Code schreiben, welche Schritte nötig sind, um andere Dinge abzuarbeiten, wie Sie eine schöne Grafik mit Hilfe von Effekten, Shadern und Tricks auf den Bildschirm zaubern, wie Sie Musik und Soundeffekte einbinden, wie Menüdialoge gestaltet werden können, was wichtig ist, um ein Spiel „rund“ zu machen, wie sie eine Spielewelt durch Ereignisse und NPCs beleben und so weiter und so fort. Diese Liste kann beliebig verlängert werden und detaillierter dargestellt werden – ein Spiel umfasst ein riesengroßes Spektrum an Dingen, die zu tun sind, damit alles stimmig ist und die Idee des Spiels sich vollständig entfalten kann. Wenn das Spiel dann auch noch Spaß macht, haben Sie und ihre Spieler gewonnen – denn nichts ist schlimmer als ein Spiel, was keinen Spaß macht.

Aber die Wahrheit ist: auch wenn RUDI ihren Spielern Spaß machen sollte, kann die Entwicklung weniger spaßig sein. Dieser Workshop blendet furchtbar einfach all die Situationen und Momente weg, in denen stundenlang rumprobiert worden ist, bis etwas gut aussah (das Wasser, Licht & Schatten, die Charaktere usw.), etwas funktionierte (das Track-System, das Menü, das resetten des Spielers) oder etwas einfach nicht lief und man keine Ahnung hatte, wieso. Für viele Leute kann das frustrierend sein, aber – wie für mich – auch ein Antrieb.

Ich und einige Freunde vertreten daher die Meinung „wir hassen Spaß“ wenn es um Spieleentwicklung geht, da man nur mit Ernsthaftigkeit und Disziplin ein Spiel durchziehen kann. Der Spaß kommt dann von ganz alleine wenn man irgendwann die Früchte seiner Arbeit sieht, aber wer nur dann Spiele (weiter-) entwickelt, wenn er/sie „mal eben Spaß“ daran hat.. der kommt nicht weit.

Der Sinn, warum RUDI als open source Software released worden ist, hat nicht nur historische Gründe, sondern auch ein „höheres Ziel“ (auch wenn sich das jetzt für einige wie Schwachsinn anhört): ich glaube, dass ein kleines, nicht zu triviales, abgeschlossenes und homogenes Spiel (in Bezug auf die Grafik und das Gameplay) und vor allem als sauber programmiertes Spiel vielen Leuten einen „AHA!“ Effekt geben kann, den Leser lernen lässt oder einfach nur Interessierte den Quell-Code lesen und dann sogar Verbesserungen schreiben oder gar „Mods“ des Spiels (also Modifikationen, .. andere Varianten des Spiels) unter derselben GPL Lizenz herausbringen.

Ich freue mich daher, dass ich in Zusammenarbeit mit einigen Freunden aus der Szene dieses Projekt vorerst fertigstellen konnte. Wir haben uns zwar viel mehr vorgenommen (u.a. auch diesen Workshop zeitgleich auf Englisch herauszubringen und drei statt einem Level anzubieten), aber ich glaube wir haben die richtigen Entscheidungen getroffen. Das Spiel ist ansehnlich geworden und ich bin zufrieden, dass der Code schön und strukturiert ist. Ich hoffe, Sie können soviel Nutzen für ihr Know-How aus diesem Workshop ziehen wie möglich, damit Sie später auch ein tolles, eigenes Spiel auf die Beine stellen können.

Ausblick

Zu jedem Abschluss gehört auch ein Ausblick, wie ich finde. Es ist wichtig, über den Tellerrand zu schauen und in diesem Fall meine ich nicht Ihren Tellerrand, sondern den dieses Projektes. Das Spiel wurde in der Version 0.1 als Open-Source Software veröffentlicht unter der GPL veröffentlicht.

Trotz all dieser zukunftsweisenden Lizenzmodelle (ich erhoffe mir ja dass einige Leser verbesserten oder neuen Code einreichen!), steht die Zukunft des Projektes selber allerdings noch nicht fest. Aus der Konzeptphase dieses Projektes ergaben sich diverse Ideen, wie man die nachfolgenden Levels gestalten kann und wie man das Gameplay sinnvoll erweitern, bzw. ergänzen kann. So gesehen, wurde auch noch gar nicht das komplette Gameplay implementiert, wie es ursprünglich geplant war – das Boni/Mali System fehlt vollständig. Im Zuge von weiteren Levels ergeben sich auch neue Fragestellungen, die beantwortet werden müssen, wie z.B. ein Levelloader, ein Levelmanagement (zur Verwaltung von freigespielten und nicht freigespielten Levels), eventuellen Zwischensequenzen, usw. Auch fehlen noch einige aus meiner Sicht wichtigen Dinge, wie das dynamische Zuweisen von Tastenbelegungen, ein kleines Tutorial und solche Dinge.

Die content-Creation würde sehr wahrscheinlich je nach „Neuheitsgrad“ des contents pro Level immer schneller vorran gehen als in der Designphase der Version 0.1, da wir nicht nur Modelle, Artworks, Animationen, Shader usw. anfertigen mussten, wir mussten auch diese Dinge alle designen. Dies müsste man auch bei neuen Sachen machen, aber die Levels werden immer einen gewissen Prozenzsatz von bereits vorhandenen Sachen weiterverwerten. Von daher wäre es nicht die Frage, ob wir ein weiteres Level anfertigen und das Spiel ausbauen, sondern inwiefern der Workshop weitergeführt wird.

Wie bereits erwähnt, ist es schon sehr schwierig die Kommentare im Quellcode und eine Dokumentation aufrecht und aktuell zu halten. In diesem Zusammenhang war die Erstellung und der ständige Abgleich des Workshops mit dem Spiel echt zeit- und nervenaufreibend. Daher weiß ich nicht genau, ob ich für weitere Versionen des Spiels zusätzliche Kapitel schreiben will oder einfach nur das Spiel (weiterhin unter der GPL Lizenz) weiterentwickeln will. Dies hängt auch so ein bißchen von der Resonanz ab, die ich auf diesen Workshop erhalte.

Sie können Sich in das Projekt jederzeit miteinbringen: als Shader-Programmierer, als Lieferant für Texturen, Effekte, Modelle und Animationen, oder in Form von Skizzen, Designs usw. oder auch Geld in Form von Paypal Donations (ID: christian@behrenberg.de).

Ich hoffe, Ihnen hat das Lesen dieses Workshop und das Spielen des finalen Spiels ein wenig Freude gemacht und dass Sie nun bereit für die Umsetzung Ihrer eigenen Spielideen sind.

Mit spielerischem Gruß,
Christian Behrenberg

ENDE