Kamera Tutorial Version 2

Oder: Wie kriege ich es hin, daß diese Views funktionieren?

Von Gnometech

Vorwort zu Version 2

Ich möchte allen danken, die mir so positive Rückmeldungen über dieses Tutorial gaben. Allerdings gibt es immer noch Dinge, die bislang nicht geklärt wurden und die Kameras und Views betreffen. Eine der am häufigsten gestellten Fragen im 3DGS Forum in den letzten Wochen war: "Wie mache ich eine Resident Evil Kamera, also eine, die den Spieler die ganze Zeit ansieht, aber immer an derselben Stelle verbleibt?"
Dies ist nicht wirklich schwer zu programmieren, aber eine zweite Frage folgte meist auf dem Fuß: "Wie kann ich zwischen mehrerer dieser Kameras umschalten?" Dies ist etwas kniffliger und ich werde versuchen, diese Fragestellungen in den neuen Kapiteln (ab Kapitel 5) zu beantworten. Ich werde außerdem die Chance nutzen, etwas über Pointer (Zeiger) und Handles zu erklären.
Wie immer, viel Spaß hierbei. März 2002

Kapitel 1: Erstellen der Map und Vorbereiten der Sitzung

Gruß!

Mein Name ist Gnometech und dies ist mein Kamera Tutorial. Ich werde keine Versprechungen machen, wie "Du wirst mit Sicherheit in der Lage sein, eigene Kameraansichten zu erstellen, wenn Du diesen Text durchgearbeitet hast", aber ich hoffe doch, daß die Dinge hinterher etwas klarer sind. Auch erlaube ich mir, beim unformalen "Du" zu bleiben.

Als erstes erstelle eine kleine Map oder nehme eine von Deinen Maps, die schon erstellt wurden. Ich gehe davon aus, daß Du weißt, wie das funktioniert, denn ich werde hier nicht weiter darauf eingehen. Plaziere den Spieler irgendwo in der Map und weise ihm Deine persönliche Lieblings-Movement Action zu, z.B. die vordefinierte player_move aus den Templates oder Deine eigene.

Damit die Kameras funktionieren, muß die Spielerentity durch ein Synonym global bekannt sein. In der movement.wdl wird dieses Synonym "player" genannt und definiert, ich gehe einfach davon aus, daß es bei Dir genauso heißt, falls Du Deine eigenen Bewegungsskripte verwendest.

Da Du nun Deinen eigene Code für die Kameras schreiben möchtest, ändere die Zeile, wo die vordefinierte "move_view" aufgerufen wird. Unsere Funktion wird "update_views" heißen. Für Templatenutzer: einfach move_view suchen und ersetzen (ggf. mehrfach). Falls Du Dein eigenes Skript verwendest, stelle nur sicher, daß "update_views" einmal pro Frame Zyklus aufgerufen wird (und sei es in einer eigenen While Schleife)

Erstelle nun eine neue WDL Datei namens "cam.wdl" und füge sie mittels "include" in Dein Skript ein. Los geht's!

Kapitel 2: Die Ego Ansicht

Diese Sicht ist recht einfach. Folgende Zeilen müssen in die neue cam.wdl:

```
view 1st_person
{
	layer = 1;
	pos_x = 0;
	pos_y = 0;
}

function init_cameras()
{
	camera.visible = off;
	1st_person.size_x = screen_size.x;
	1st_person.size_y = screen_size.y;
```

```
	1st_person.genius = player;
	1st_person.visible = on;
}

function update_views()
{
	1st_person.x = player.x;
	1st_person.y = player.y;
	1st_person.z = player.z;
	1st_person.pan = player.pan;
	1st_person.roll = player.roll;
	1st_person.tilt = player.tilt;
}
```

Irgendwo in der Main Funktion, nach dem Laden des Levels, muß der Aufruf "init_cameras();" erfolgen.
Speicher das Skript und starte das Level. Wenn alles korrekt ist, sollte die Kamera den Bewegungen des Spielers folgen.
Was haben wir getan? Wir haben eine neue "View" definiert. Man kann sich eine View vorstellen, als die Verbindung in die Spielwelt. Es ist ein "Fenster", welches einen Ausschnitt dieser Welt zeigt. Die Position und Größe dieses Fensters auf dem 2D Bildschirm wird durch pos_x, pos_y, size_x und size_y definiert. In unserem Beispiel ist die obere linke Ecke der View identisch mit der oberen linken Ecke des Bildschirms (0,0) und die Größe ist exakt die Bildschirmgröße. Wenn man die View verkleinert und eine zweite hinzufügt, kann man auf diese Weise sehr einfach Splitscreeneffekte erzielen.
Die Funktion update_views wird einmal pro Frame Zyklus aufgerufen und setzt neue Werte für x, y und z. Dies sind die Koordinaten der View in der Spielwelt, bestimmen also, welcher Ausschnitt gezeigt wird. In diesem Fall, werden diese Werte jeweils denen des Spielers angepaßt, somit erreichen wir, daß die Kamera dem Spieler folgt, bzw. immer "in seinem Innern" bleibt.
Das Skript ist noch nicht perfekt. Zum Beispiel ist die Kamera im Zentrum des Spieler Modells positioniert und nicht an seinem Kopf, die View ist also etwas zu tief. Außerdem können wir noch nicht hoch oder runter schauen.

Das erste ist schnell behoben. Wir müssen nur eine Variable am Anfang des Skriptes neu definieren:

```
var eye_height = 20;
```

Und dann, anstelle von:

```
1st_person.z = player.z;
```

schreiben wir:

```
1st_person.z = player.z + eye_height;
```

Damit wird die Kamera um 20 Quants höher angesetzt. Falls dieser Wert nicht den Anforderungen entsprechen sollte, kann man ihn ändern, sogar während das Spiel läuft.

Als nächstes wollen wir, daß der Spieler hoch oder runter sehen kann. Dafür benötigen wir weitere Variablen:

```
var tilt_1st = 0;
var cam_turnspeed = 2;
var max_tilt_1st = 40;
```

Nun ändere folgende Zeile:

```
1st_person.tilt = player.tilt;
```

in:

```
1st_person.tilt = player.tilt + tilt_1st;
```

Füge außerdem zwei Funktionen hinzu:

```
function look_up()
{
	if (tilt_1st < max_tilt_1st) { tilt_1st += cam_turnspeed; }
}

function look_down()
{
	if (tilt_1st > -max_tilt_1st) { tilt_1st -= cam_turnspeed; }
}
```

Diese Funktionen müssen während des Spiels aufgerufen werden. Man könnte z.B. schreiben:

```
on_pgup = look_up;
on_pgdn = look_down;
```

Speicher das Skript und starte die Map. Hmmm... nicht schlecht, aber nicht das Gewünschte. Wenn eine der Tasten gedrückt und gehalten wird, ändert sich der Winkel etwas und nicht weiter. Man muß die Taste mehrmals drücken, um richtig nach oben oder unten schauen zu können und das ist immer noch sehr langsam.

Wie ändern wir das? Als erstes ändern wir die on_pgup und on_pgdn Definitionen:

```
on_pgup = handle_pageup;
on_pgdn = handle_pagedown;
```

Und dann noch zwei weiter Funktionen:

```
function handle_pageup()
{
	while (key_pgup)
	{
		look_up();
		wait(1);
	}
}

function handle_pagedown()
{
	while (key_pgdn)
	{
		look_down();
		wait(1);
	}
}
```

Auf diese Weise werden unsere "look_up" und "look_down" einmal pro Framezyklus aufgerufen, solange die entsprechenden Tasten gedrückt werden. Falls die Kamera sich zu schnell bewegen sollte, ändere einfach den Wert von cam_turnspeed. Ebenso kann man max_tilt_1st ändern, welche den Winkel angibt, bis zu dem sich die View ändern kann. Ein Wert von 90 bedeutet hier, daß der Spieler direkt nach oben bzw. unten sehen kann. Werte darüber lassen ihn einen Salto machen... ;)

Genug für dieses Kapitel. Die Ego Perspektive funktioniert nun, auf geht es zur Außenkamera.

Kapitel 3: Frei schwenkbare Außenkamera

Nun wollen wir eine schwenkbare Außenkamera erstellen. Es gibt dafür zwei Möglichkeiten: entweder wir definieren eine Kamera, die außerhalb des Spielers ist und sich nicht bewegt, auch wenn er sich dreht, die ihn also immer aus einer bestimmten Richtung ansieht. Die andere Möglichkeit wäre eine Kamera, die sich dreht, wenn der Spieler es tut, also immer hinter ihm bleibt (oder vor ihm, oder seitlich oder...)

Ich werde die Kamera so konstruieren, daß sie frei dreh- und zoombar ist. Falls Du für Dein Spiel eine Sicht von oben oder von der Seite brauchst, kein Problem, wähle einfach die nötigen Werte und füge keine Funktionen ein, um sie während des Spiels zu ändern, dann bleibt die Kamera immer oben, an der Seite oder wo auch immer.

Auf geht es. Ich wähle die erste Möglichkeit, also eine Kamera, die den Spieler immer von derselben Richtung ansieht. Falls sie sich außerdem mit dem Spieler drehen soll, ist nur eine geringe Änderung nötig, wie wir sehen werden.

Definiere also eine zweite View:

```
view 3rd_person
{
	layer = 1;
	pos_x = 0;
	pos_y = 0;
}
```

Füge folgende Zeilen in die init_cameras:

```
...
	3rd_person.size_x = screen_size.x;
	3rd_person.size_y = screen_size.y;
...
```

Wir werden diese Kamera noch nicht auf "sichtbar" schalten. Statt dessen möchten wir zwischen der Außenkamera und der Egosicht umschalten wollen:

```
var cam_mode = 0; // 0 meaning 1st Person, 1 meaning 3rd Person

function toggle_cams()
{
	if (cam_mode == 0)
	{  // Change to 3rd Person
		1st_person.visible = off;
		3rd_person.visible = on;
		cam_mode = 1;
	}
	else
	{  // Change to 1st Person
		3rd_person.visible = off;
		1st_person.visible = on;
		cam_mode = 0;
	}
}

on_f8 = toggle_cams;
```

Wenn wir nun F8 drücken, können wir damit die eine View ein- und die andere ausblenden. Allerdings bringt das noch nicht viel, da die x, y und z Koordinaten der Außenkamera noch nicht definiert wurden. Wir werden nun also die "update_views" Funktion ändern, das war diejeinige, die einmal pro Framezyklus aufgerufen wird.

Doch vorher sollten wir überlegen... wir können wir eine Außemkamera definieren, die sich beliebig um den Spieler drehen läßt? Sie sollte (zunächst) in derselben Ebene sein, wie der Spieler, also ist schonmal ihre Z Koordinate dieselbe wie die vom player. Aber wie berechnen wir die x und y Koordinaten? Etwas Mathematik hilft hier.

Stell Dir den Spieler vor, wie er von oben aussieht. Mache am besten einen Punkt M auf ein Blatt Papier, um den Spieler zu repräsentieren. Male einen Kreis um diesen Punkt. Dies soll der Kreis sein, auf dem die Kamera sich um den Spieler dreht. Dieser Kreis hat einen bestimmten Radius. Wenn man einen bestimmten Punkt O auf dem Kreis (z.B. der "oberste", also der am weitesten "nördliche" Punkt) fixiert, so kann man jeden anderen Punkt P auf dem Kreis eindeutig durch den Winkel beschreiben, der von den Verbindungslinien OM und PM eingeschlossen wird.

Wir fassen zusammen: dadurch, daß wir eine Distanz vom Spieler (Kreisradius) und einen Winkel in der Ebene zwischen 0° und 360° angeben, ist die Kameraposition eindeutig bestimmt. Ich gebe die Formel an, man kann sie mit etwas Trigonometrie leicht selbst herleiten.

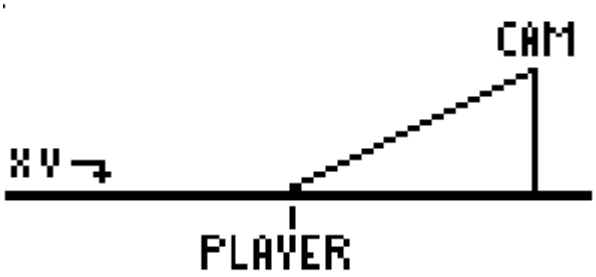
```
var dist_planar =  300; // distance from the player
var cam_angle = 0;

function update_views()
{
	if (cam_mode == 0)
	{
		1st_person.x = player.x;
		1st_person.y = player.y;
		1st_person.z = player.z + eye_height;
		1st_person.pan = player.pan;
		1st_person.roll = player.roll;
		1st_person.tilt = player.tilt + tilt_1st;
	}
	else
	{
		3rd_person.x = player.x - cos (cam_angle) * dist_planar;
		3rd_person.y = player.y - sin (cam_angle) * dist_planar;
		3rd_person.z = player.z;
		3rd_person.pan = cam_angle;
		3rd_person.roll = 0;
		3rd_person.tilt = 0;
	}
}
```

Speicher das Skript und starte die Map. Drücke F8, um die Außenkamera zu aktivieren. Du solltest nun den Spieler von außerhalb sehen. Wenn Du ihn durch die Map lenkst, sollte die Kamera stets denselben Abstand vom Spieler einhalten und sich nicht mitdrehen, auch wenn er es tut.
Falls Du den Wert "dist_planar" während des Spiels änderst (z.B. durch Druck auf TAB und Eingabe von dist_planar = 150; oder indem Du eine Funktion definierst, die den Wert ändert... das solltest Du nun allein können!), zoomt die Kamera näher, bzw. weiter weg. Eine Änderung von cam_angle läßt die Kamera sich um den Spieler drehen, wobei sie ihn immer ansieht. (3rd_person.pan = cam_angle sorgt dafür)

Es gibt aber noch ein Problem mit Wänden. Die Kamera fährt einfach hindurch und oft wird es Hindernisse geben, die die direkte Sicht zum Spieler blockieren. Wir werden dieses Problem im nächsten Kapitel lösen, zunächst wollen wir den Code noch etwas abändern. Wir wollen nun die Kamera auch nach oben bewegen, aber so, daß sie nach wie vor auf den Spieler gerichtet ist. Auch die Distanz zum Spieler soll gleichbleiben. Die Kamera soll sich also nicht mehr auf einem Kreis um den Spieler, sondern auf einer Kugel um ihn bewegen können.

Wie können wir das erreichen? Stell Dir den Spieler von der Seite vor. Wenn die Kamera pberhalb der XY Ebene des Spielers ist und Du eine Linie von der Kamera zum Spieler ziehst, erhältst Du einen Winkel zwischen dieser Linie und der XY Ebene. Dieser Winkel definiert die Position der Kamera.

Natürlich ändert sich die Distanz bezüglich der Ebene, wenn die Grunddistanz die gleiche sein soll! Ein kleines Bild:



Hier also die Änderungen am Code:

```
var dist_total = 300; // Change THIS value to zoom in or out
var tilt_3rd = 0;

function update_views()
{
...
	else
	{
		dist_planar = cos (tilt_3rd) * dist_total;
		3rd_person.x = player.x - cos (cam_angle) * dist_planar;
		3rd_person.y = player.y - sin (cam_angle) * dist_planar;
		3rd_person.z = player.z + sin (tilt_3rd) * dist_total;
		3rd_person.pan = cam_angle;
		3rd_person.roll = 0;
		3rd_person.tilt = - tilt_3rd;
	}
```

Indem wir nun die Werte tilt_3rd, cam_angle und dist:_total ändern, können wir die Kamera frei um den Spieler bewegen. Sie bleibt immer auf einer Kugel, deren Radius durch Ändern von dist_total geändert werden kann.
Es ist nun Dir überlassen, die Funktionen zu schreiben, welche diese Werte ändern können und ihnen Tasten zuzuweisen. Sie sollten auch begrenzt werden (ähnlich wie bei tilt_1st), damit man nicht unbegrenzt hinein oder hinauszoomen kann.
Im nächsten Kapitel werden wir das Problem mit den Wänden lösen.

Kapitel 4: Wie man die Kamera von Wänden fernhält

Eine Sache vorweg, es gibt mehrer Lösungen hierfür und ich beanspruche nicht, daß meine die beste ist. Es hat für mich soweit ganz gut funktioniert, seht es euch einfach an. Es ist nicht weiter schwer zu verstehen.

Die Idee ist es, vom Spieler aus zu der berechneten Kameraposition einen "trace" auszuführen und zu prüfen, ob ein Hindernis im Weg ist. Falls ja, wird einfach die Distanz zwischen dem Spieler und der Kamera geändert und zwar um genau die Distanz, die zwischen Spieler und Hindernis liegt, so daß sich die Kamera flüssig "an der Wand entlang schmiegt".

Zur technischen Seite. Nach der Berechnung der 3rd_person Werte, füge die folgende Zeile ein:

```
function update_views()
{
...
```

```
			3rd_person.roll = 0;
			3rd_person.tilt = - tilt_3rd;
			validate_view();
		}
...
}
```

Diese Funktion definieren wir hier:

```
var dist_traced;

function validate_view()
{
		my = player;
		trace_mode = ignore_me + ignore_passable;
		dist_traced = trace (player.x, 3rd_person.x);
		if (dist_traced == 0) { return; } // No obstacles hit... fine
		if (dist_traced < dist_total)
		{
			dist_traced -= 5; // Move it out of the wall
			dist_planar = cos (tilt_3rd) * dist_traced;
			3rd_person.x = player.x - cos (cam_angle) * dist_planar;
			3rd_person.y = player.y - sin (cam_angle) * dist_planar;
			3rd_person.z = player.z + sin (tilt_3rd) * dist_traced;
		}
}
```

Die Formeln zur Bestimmung der Kameraposition sind exakt dieselben, man verwendet nur eine neue Distanz zum Spieler, nämlich dist_traced, die vom trace geliefert wird. Die Distanz wird noch etwas modifiziert, um die Kamera ein Stück aus der Wand rauszubewegen.

Nun sollte die Kamera Wände und Hindernisse vermeiden.

Eine Sache noch, bevor dieses Tutorial endet... zu erreichen, daß die Kamera immer hinter dem Spieler bleibt. Das ist sehr einfach, ändere folgende Zeilen:

```
			3rd_person.x = player.x - cos (cam_angle) * dist_planar;
			3rd_person.y = player.y - sin (cam_angle) * dist_planar;
```

zu

```
			3rd_person.x = player.x - cos (cam_angle + player.pan) * dist_planar;
			3rd_person.y = player.y - sin (cam_angle + player.pan) * dist_planar;
```

Alles, was Du also tun mußt, ist den player.pan zu dem cam_angle zu addieren. Mach dasselbe in validate_view und auch bei der Bestimmung von 3rd_person.pan.

Falls die Kamera beim Start des Spieles nun nicht hinter dem Spieler ist, sondern z.B. vor ihm, ändere einfach cam_angle, bis es Deinen Wünschen entspricht und lasse den Spieler diese Variable nicht mehr ändern. Damit bleibt die Kamera ständig hinter dem Spieler.

Kapitel 5: Zeiger und Handles

Nun gut, wie versprochen ist nun unser nächstes Ziel eine "Resident Evil" Kamera. Für alle, die das Spiel nicht kennen, das sind Kameras, die an einer Position bleiben (z.B. in der Ecke des Raumes), aber den Spieler immer im Blick behalten. Wenn der Spieler sich zu weit fort bewegt, schaltet das Spiel automatisch zu einer weiteren solchen Kamera um usw.
Damit wir mehr als eine Kameraposition vernünftig verwalten können, wird es sehr nützlich sein, mit Zeigern und Handles zu arbeiten. (Anmerkung: Zeiger heißt im englischen auch "Pointer", ich gebrauche beide Begriffe... also nicht verwirrt sein, es bedeutet dasselbe) Aber was genau bedeutet es? Ich werde versuchen, im

Folgenden einen kurzen Überblick über diese Strukturen zu geben. Natürlich finden Zeiger auch Verwendung, wenn es nicht um Kameras und Views geht, man begegnet ihnen immer wieder, wenn man mit WDL arbeitet.

Fangen wir mit einem Zeiger an. Im Prinzip ist ein Zeiger eine Variable. Aber dort werden keine Zahlen gespeichert (nunja, eigentlich schon, da der Computer intern alles irgendwie numerisch darstellt, aber diese Daten werden nicht als Zahlen interpretiert). Ein Zeiger enthält Informationen, die es erlauben, auf eine bestimmte Entity zuzugriefen. Ein bekanntes Beispiel wird in den Templates verwendet und es wurde weiter oben auch schon benutzt: player.
Der Zeiger "player" ist ein Zeiger auf die Spieler Entity. Mit seiner Hilfe haben wir Zugriff auf alle Daten, die zu dieser Entity gehören. Jede Entity hat ihren eigenen "Datenraum", wo z.B. die Position der Entity (x, y, z), ihre Orientierung im Raum (pan, tilt, roll) und andere Dinge gespeichert sind, z.B. auch die 48 skills und 8 flags. Weil es normalerweise sehr viele Entities in einem Spiel gibt, können wir nicht einfach "x" in den Code schreiben und erwarten, daß der Compiler direkt weiß, welche Entity wir meinen. Statt dessen sagen wir es ihm mit einem Zeiger:

```
player.x += 50;
```

Dieses Kommando erhöht den Wert in der Variablen x von der Entity, auf die der Zeiger "player" zeigt um 50. Zwei besondere Zeiger sind my / me und you / your. (Die Namen sind äquivalent, Du kannst beide verwenden, um auf den jeweiligen Pointer zuzugreifen)

Der my Zeiger wird automatisch richtig gesetzt, wenn wir uns in der Action einer Entity befinden (entweder über WED zugewiesen, oder mittels ent_create) und er zeigt immer auf die Entity, zu der diese Action gehört. Es ist ganz einfach, eine Action wie:

```
action turn
{
	while(1)
	{
		my.pan += 3;
		wait(1);
	}
}
```

wird den pan derjenigen Entity verändern, der sie zugewiesen ist. Bei mehr als einer Entity mit dieser Action in der Map gibt es keine Probleme, sie drehen sich alle, da "my" immer die korrekte Entity meint.
"You" funktioniert etwas anders. Dieser Pointer wird von einigen Events und Funktionen direkt von der Engine gesetzt. Im Handbuch steht das ganze mit mehr Details, hier nur einige Beispiele: bei jedem trace, das eine Entity trifft, bei jedem Scan, das etwas findet oder wenn eine Entity mit einer anderen kollidiert, wird jeweils "you" auf die andere Entity gesetzt.
Wie wird ein Zeiger nun deklariert? Ähnlich wie eine Variable, schreibe einfach:

```
entity* this_is_a_pointer;
```

Das Sternchen hinter dem Typnamen bedeutet einfach, daß dies einen Zeiger auf ein Objekt "Entity" deklariert (C Syntax)
Es ist wichtig zu beachten, daß ein solcherart deklarierter Pointer immer global ist, genau wie eine Variable, die so definiert wird. Das kann mitunter zu Problemen führen. Es ist natürlich völlig in Ordnung, einen Zeiger namens "player" zu haben, in den meisten Fällen gibt es ja nur eine Spieler Entity (Multiplayer Spieler müssen mit dem Problem auch anders umgehen) Ein Zeiger namens "enemy" hingegen wird in den meisten Fällen nutzlos sein, da dieser sich immer nur auf einen Gegner beziehen kann und nicht auf alle zugleich. Da wäre es doch schön, wenn man ein Array von Zeigern anlegen könnte. Dies geht nicht direkt, aber es geht trotzdem und dafür brauchen wir die Funktionen handle und ptr_for_handle.

Die erste Funktion erhält einen Zeiger und konvertiert ihn in eine Zahl. Die zweite Funktion erhält eine so generierte Zahl und macht wieder den ursprüunglichen Zeiger draus. Es ist eine Art Übersetzungsmechanismus und damit können wir Zeiger überall dort speichern, wo wir auch Zahlen speichern können: in Arrays, in skills, ...

Eine Sache noch: wenn ein Zeiger deklariert wird, zeigt er auf nichts. Das heißt, sein vordefinierter Inhalt ist "null". Falls Du nun versuchst, einen skill oder eine andere Variable der Entity über einen Null-Zeiger zu bekommen, dann gibt es eine Fehlermeldung (empty pointer). Da ist ein wenig Vorsicht geboten.

Da wir dies nun alles wissen, sind wir gewappnet für die versprochenen Resident Evil Kameras.

Kapitel 6: Feste Kameras

Ich gehe davon aus, daß die anderen Kameratypen von oben nicht benötigt werden, wenn man feste Kameras hat, ich beginne also von vorn. Die verschiedenen Typen zu kombinieren ist auch nicht schwer, oben steht ja schon ein Beispiel, wie ich zwischen Egoperspektive und Außenkamera umschalten kann, das geht dann prinzipiell genauso.

Also, flugs ein neues Skript aufgemacht und folgendes hineingetippt:

```
entity* fixed_cam;

view fixed
{
	layer = 1;
	pos_x = 0;
	pos_y = 0;
}

function init_cameras()
{
	camera.visible = off;
	fixed.size_x = screen_size.x;
	fixed.size_y = screen_size.y;
	fixed.visible = on;
}

function update_views()
{
	if (fixed_cam == null) { return; }
	vec_set (fixed.x, fixed_cam.x);
	vec_set (temp, player.x);
	vec_sub(temp, fixed.x);
	vec_to_angle(fixed.pan, temp);
}
```

Binde das Skript mit "include" in Deine Haupt-WDL Datei ein und rufe init_cameras von der main auf und update_views aus dem inneren der Schleife der Spieleraction. Was geschieht hier nun? In update_views wird die Position der View auf die Position der Entity fixed_cam gesetzt. Dann wird die View gedreht, so daß die Kamera den Spieler im Blick hat.
So kann es aber noch nicht funktionieren, da fixed_cam nirgends gesetzt wird (null-pointer!). Um dies zu ändern, muß die folgende Action her:

```
action set_fixed_pos
{
	fixed_cam = me;
	my.invisible = on;
	my.passable = on;
}
```

Plaziere nun eine Entity (völlig egal welche) in die Map und zwar genau an die Stelle, wo die Kamera sein soll und weise diese Action zu. Starte das Level dann.

Die Kamera sollte an der Position sein, wo die Entity plaziert wurde (die selbst nicht zu sehen sein sollte) und sie sollte so schwenken, daß der Spieler immer in der Mitte bleibt. Hey, das war ja einfach.
Allerdings besteht ein Level in den meisten Fällen aus mehr als einem Raum und daher möchte man eventuell die Kamera an eine neue Position bringen. Das wird die nächste Sache sein, die wir in Angriff nehmen.

Füge nun mehr solcher "dummy" Entities in Dein Level ein und gib allen die Action set_fixed_pos. Um sie voneinander zu unterscheiden, gib jeder Position einen anderen Wert für "skill1", beginnend mit 0 für die erste. Dies werden die Kameras sein.

Modifiziere dann die Action:

```
var cam_array[200]; // a maximum of 200 cameras

action set_fixed_pos
{
	if (my.skill1 >= 200) { return; }
	cam_array[my.skill1] = handle(me);
	my.invisible = on;
	my.passable = on;
}
```

Dieser Ansatz ist etwas anders. Als wir noch eine Kamera hatten, konnten wir einen globalen Zeiger nehmen und die View auf diese Position setzen. Nun haben wir aber mehr als eine solche Entity. Und so benutzen wir Handles, um die Zeigerwerte in einem Array zu speichern.

Was nun noch fehlt ist eine Funktion, die auf eine bestimmte Kamera umschaltet:

```
function set_view_to_cam(cam_number)
{
	if (cam_array[number] == 0) { return; }
	fixed_cam = ptr_for_handle (cam_array[number]);
}
```

Der fixed_cam Zeiger ist derjenige, der für update_views zählt. Und dieser wird von der Funktion geändert. Beachte, daß die Funktion einen Parameter erhält. Der Zweck ist klar: wenn die View auf eine bestimmte Kamera (z.B. Kamera Nr. 4) umschalten soll, muß einfach:

```
	set_view_to_cam (4);
```

aufgerufen werden.

Man könnte z.B. kleine Entities in der Map verteilen mit Trigger Events, die die Kamera umschalten, wenn der Spieler in die Nähe kommt... man könnte den Spieler selbst nach der nächsten Kamera scannen lassen und die View dorthin schalten... man könnte die nächstgelegene Kamera mit Hilfe von Vektorrechnung bestimmen und mit Hilfe eines trace feststellen, ob der Spieler auch gesehen wird... es gibt eine Menge Möglichkeiten, wie das Umschalten vonstatten gehen soll, und diese hängen von Deiner Phantasie, aber auch von der Art des Spiels und dem Level Design ab.

Kapitel 7: Die Kamera folgt einem Pfad

Für alle, die es interessiert, ich habe kürzlich eine kleine "Kamera folgt Pfad" Funktion geschrieben. Diese hat eigentlich nichts mit dem Rest des Tutorials zu tun und ich werde auch nicht im Detail darauf eingehen, ich werde einfach den Code hier hinzufügen, für diejenigen, die es interessiert. Hier ist er:

```
DEFINE _speed, skill1;
DEFINE _turnspeed, skill2;
DEFINE _crit_dist, skill3;

DEFINE _turnto1, flag1;

entity* cam_ent;
var to_angle[3];
var pan_diff;
var tilt_diff;

function turn_towards_point()
```

```
{
	vec_set(temp, my.skill21);
	vec_sub (temp, my.x);
	vec_to_angle (to_angle, temp);
	my.pan = ang(my.pan);
	my.tilt = ang(my.tilt);
	to_angle.pan = ang(to_angle.pan);
	to_angle.tilt = ang(to_angle.tilt);
	pan_diff = abs(my.pan - to_angle.pan);
	if (pan_diff > 180) { pan_diff = 360 - pan_diff; }
	tilt_diff = abs(my.tilt - to_angle.tilt);
	if (tilt_diff > 180) { tilt_diff = 360 - tilt_diff; }
	if (pan_diff > 6 * time * my._turnspeed) { my.skill48 = time * my._turnspeed; }
	else {my.skill48 = pan_diff / 6; }
	if (((my.pan > to_angle.pan) && (abs(my.pan - to_angle.pan) < 180)) || ((my.pan < to_angle.pan) &&
(abs(my.pan - to_angle.pan) >= 180)))
	{ my.pan -= my.skill48; } else { my.pan += my.skill48; }
	if (tilt_diff > 6 * time * my._turnspeed) { my.skill48 = time * my._turnspeed; }
	else {my.skill48 = tilt_diff / 6; }
	if (((my.tilt > to_angle.tilt) && (abs(my.tilt - to_angle.tilt) < 180)) || ((my.tilt < to_angle.tilt) &&
(abs(my.tilt - to_angle.tilt) >= 180)))
	{ my.tilt -= my.skill48; } else { my.tilt += my.skill48; }
}

// uses _speed, _turnspeed, _crit_dist, _turnto1
action mobile_cam
{
	my.invisible = on;
	my.passable = on;
	cam_ent = me;
	if (my._speed == 0) { my._speed = 5; }
	if (my._turnspeed == 0) { my._turnspeed = 6; }
	if (ent_path("cam_path") == 0) { return; }
	my.skill10 = ent_waypoint (my.skill21, 0);
	if (my._turnto1 == on)
	{
		vec_set (target, my.skill21);
		vec_sub (target, my.x);
		vec_to_angle (my.pan, target);
	}
	while(1)
	{
		turn_towards_point();
		my.skill24 = my._speed * time;
		ent_move (my.skill24, nullvector);
		if (vec_dist(my.x, my.skill21) <= max(time * my._speed, my._crit_dist))
		{
			ent_waypoint(my.skill21, my.skill9);
			my.skill9 += 1;
			if (my.skill9 >= my.skill10) { my.skill9 = 0; }
		}
		vec_set (camera.x, my.x);
		vec_set (camera.pan, my.pan);
		wait(1);
	}
}
```

Benutzung: den Code mit Hilfe von include einfügen, einen Pfad in die Map legen, der den Namen "cam_path" erhält, eine Entity mit der Action "mobile_cam", die die folgenden Skills hat:

skill1 = Kamerageschwindigkeit
skill2 = Schwenkgeschwindigkeit

skill3 = Distanz zum nächsten Wegpunkt, ab der die Kamera zum übernächsten abdrehen soll
flag1 = Falls es gesetzt ist, wird die Kamera automatisch auf den ersten Punkt des Pfades ausgerichtet, wenn das Spiel startet, ansonsten dreht es sich in der Bewegung dorthin

Bitte beachte, daß die vorgefertigte View "camera" aktiv sein muß, damit das funktioniert (oder ändere einfach "camera" in "1st_person" oder wie auch immer die View heißt, die Du verwendest...)

Das wär's fürs Erste. Bei Fragen, die das Tutorial betreffen, einfach im 3DGS Forum fragen oder mir eine eMail schicken:

gnometech@gmx.de

Viel Spaß.

Gnometech