3D GameStudio

Workshop
2D Animation

für A5 Engine 5.10
von Alain Brégeon September 2001

Aktuelle Neuigkeiten, Demos, Updates, diverses Werkzeug sowie das Anwender-Forum und alles zum jährlichen Wettbewerb findet sich auf den GameStudio Internetseiten unter **http://www.3dgamestudio.com**.

Inhalt

Vorwort

Lieber Leser,

Ich habe diesen Workshop erstellt, um Ihnen bei der Beantwortung der Frage: "Wie mache ich eine **2D-Animation** mit 3D GameStudio?" created this workshop in order to help you to find an answer to the question "How to make a **2D Animation** with 3D GameStudio?" Die in diesem Workshop verwendeten Features sind ab der Version **4.25** oder neuer verfügbar.

Wie andere Kurse zuvor zielt auch dieser Workshop auf Anwender ab, die über eine gewisse Vorerfahrung mit 3D GameStudio verfügen. Ich setze voraus, dass Sie die Tutorials durchgearbeitet haben und mit den diversen Werkzeugen (WED,MED and WDL) umzugehen wissen.

Dieser Text ergänzt die zu 3D GameStudio gehörige Dokumentation und kann diese nicht ersetzen. Sollte Ihnen irgendetwas in diesem Workshop unklar sein, lesen Sie daher bitte im mitgelieferten Manual nach. Für eventuelle Unklarheiten im Ausdruck, fehlerhaften Code, Irrtümer oder Versäumnisse entschuldige ich mich hiermit bereits im Voraus.

Ich hoffe, Sie empfinden diesen Workshop als Informativ und dass er Ihnen Spass macht.

Alain Brégeon
mailto:alainbregeon@hotmail.com

Besorgen Sie sich die neueste Version

Bevor Sie loslegen vergewissern Sie sich bitte, dass Sie auch die neueste Version von 3D GameStudio (**4.25** oder neuer) haben.

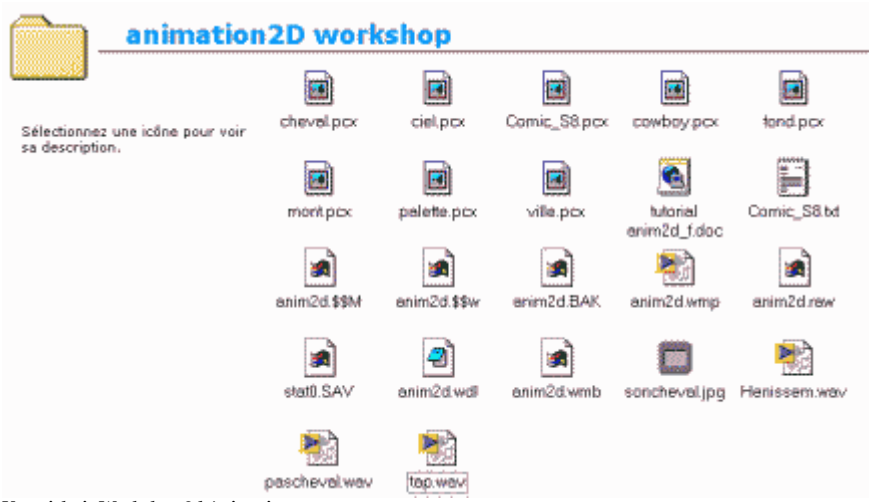
Bereiten Sie Ihre Arbeitsumgebung vor

Erstellen Sie einen Ordner namens "animation2d Workshop" in Ihrem GStudio-Verzeichnis. Das ist nun der Ordner in dem Sie sämtliche Spielelemente verwalten werden.

Als erstes werden wir in diesen Ordner die Entities, die wir für unser Spiel brauchen hineinkopieren. Sollten Sie diese noch nicht haben, finden Sie sie auf der Download-Seite von Conitec unter http://www.conitec.net / a4update.htm.

Entzippen Sie den Inhalt in Ihren Ordner, er sollte dann die folgenden Dateien enthalten:

```
cheval.pcx
ciel.pcx
comic_s8.pcx
fond.pcx
mont.pcx
palette.pcx
ville.pcx
tutorial.doc
comic_s8.txt
henissem.wav
pascheval.wav
tap.wav
```



Verzeichnis Workshop 2d Animation

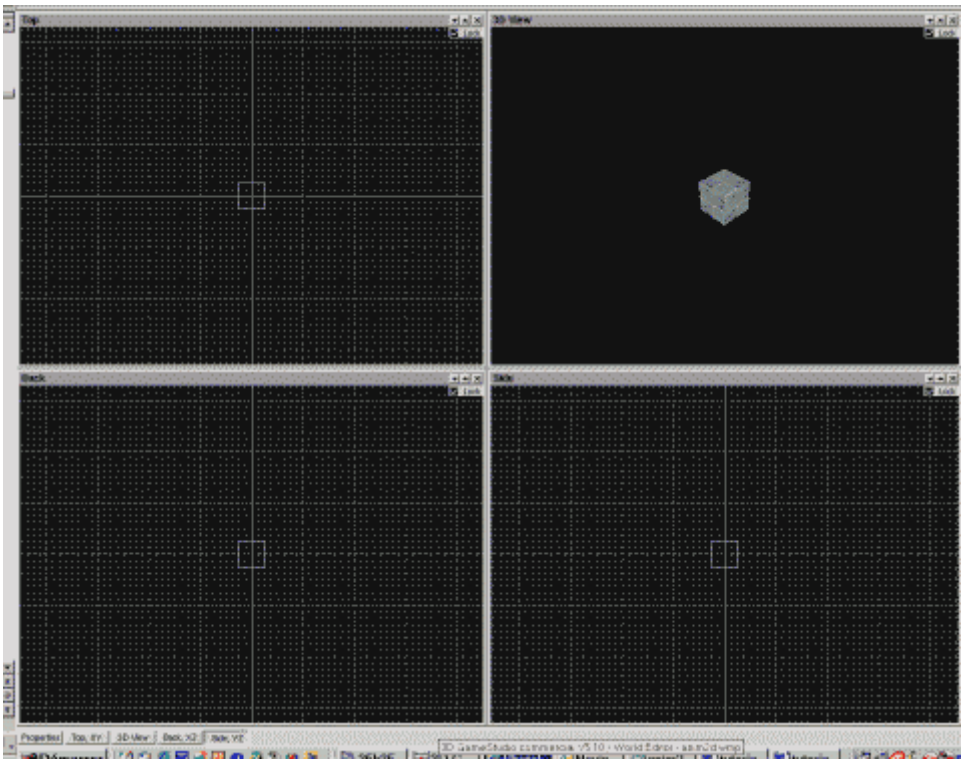
Erstellen Sie Ihr Level

Unser Level wird sehr einfach: schlicht leerer Raum. Damit wir aber die Engine überhaupt starten können, brauchen wir wenigstens ein Dummy-Objekt. Lassen Sie uns also folgende Map erstellen:

Öffnen Sie **WED** und wählen **File-> New**

Dann gehen Sie auf **Add Primitive - > Cube(small)**.

Wir bringen die Defaulttextur an, speichern das Level und nennen es **`anim2d`**. Dann kompilieren (**build**) und starten wir es: Gar nichts passiert, das ist normal.



Das Level

Erstellen Sie Ihr Skript

Erstellen Sie nun ein Skript für Ihr Level. Dazu öffnen Sie einfach Ihr Eigenschaftsfenster (**File > MapProperties**) und klicken auf den **new-**Knopf. Der Knopf daneben müsste nun von **`ndef`** auf **`anim2d.wdl`** umspringen.

Öffnen Sie Ihren Levelordner, wählen und öffnen (Doppelklick) Sie die Datei **`anim2d.wld`**. Falls Windows fragt mit welcher Applikationssoftware es die Datei öffnen soll, suchen Sie sich am besten "Notepad" (oder eben einen anderen reinen Texteditoren) aus.

Sie werden feststellen, dass die Standard Spiele-"Templates" bereits für sie erstellt wurden. Das ist für die meisten Projekt sehr schön aber wir wollen ja etwas für Fortgeschrittene machen. Darum nur zu, markieren Sie alles (in Microsoft Notepad use **Edit->Select All**) und drücken Sie die Löschtaste. So, jetzt fangen Sie bei Null an!

Pfade Einfügen

Lassen Sie uns mit dem Definieren der Pfade, die unser Programm benutzen wird, beginnen. Diese Pfade werden dazu gebraucht, der Engine zu sagen, wo sie die in unserem Projekt verwendeten Dateien finden kann (Bilder, Sounds, andere Skripte usw.). Der Ordner in dem wir uns befinden("Anim2d Workshop") ist bereits eingefügt, wir müssen in unserem Fall also nur noch die **`template1`**-Ordner hinzufügen.

Schreiben Sie die folgende Zeile:

```
path  "..\\template";    // Pfad zum WDL Templates-Unterverzeichnis
```

Beachten Sie, dass sämtliche Pfade in Relation zu unserem Levelordner stehen. Die obige Zeile besagt folgendes: "Gehe ein Verzeichnis höher ("...") und dort in den Template-Ordner

(**\\template**)".

'Includes' einfügen

Nachdem wir unsere Pfade eingerichtet haben, sollten wir unsere sogenannten Include-Dateien hinzunehmen. Schreiben Sie also unter **path**:

```
include <movement.wdl>;  // Bibliothek von WDL-Funktionen
include <messages.wdl>;
include <menu.wdl>;      // menu muss VOR doors and weaponsincludet werden
include <particle.wdl>; // wegnehmen, wenn keine Partikel gebraucht werden
```

Der **include**-Befehl sagt der Engine, dass sie diese Zeile durch den Inhalt zwischen den beiden spitzen Klammern (<...>) ersetzen soll. Es ist dasselbe, als wenn Sie in die betreffende Datei gehen, den gesamten Code kopieren und an der Stelle in Ihrem Skript einfügen würden.

Dies ist ein sehr wirksames Werkzeug, denn es erlaubt uns Code aus anderen Projekten wieder zu verwenden und die Vorteile von Überarbeitungen innerhalb der **include**-Dateien zu nutzen ohne den eigenen Code umschreiben zu müssen. Die Version 4.19 z.B. beinhaltete Code, der das Schwimmen in Wasser ermöglichte. Seither kann in jedem Projekt, das die **movement.wdl** 'included" hat dieser neue Schwimmcode verwendet werden.

Da einige Skripte Werte verwenden, die in anderen Skripten definiert sind, ist die Reihenfolge der **include**-Zeilen genauso wichtig wie bei den Pfaden. Die **actors.wdl** z.B. verwendet die Variable **force**, die aber in der **movement.wdl** definiert ist. Setzen wir also die **actors.wdl** vor die **movement.wdl**, kriegen wir hässliche Fehler.

Es ist vollkommen in Ordnung Skripte auch dann zu '**include**en', wenn Sie daraus gar keine Features in Ihrem Code verwenden. Die meisten vorgefertigten Dateien (Templates) sind voneinander abhängig, so dass Sie, sollten Sie vorhaben eine zu verwenden, zur Sicherheit am besten gleich alle per **include** in Ihren Code aufnehmen sollten. Die Ausnahme zu dieser Regel heisst **venture.wdl**. Auf dieses Skript wird von keinem anderen verwiesen, es selbst allerdings benutzt einige der anderen.

Startwerte für die Engine

Nun werden wir einige zur Bestimmung der Simulationsdarstellung wichtige Werte festlegen. Diese Werte betreffen die Auflösung, Farbtiefe, Framerate und Helligkeit. Fügen Sie also die folgenden Zeilen unterhalb der **include**-Zeilen ein:

```
// Startwerte der Engine
ifdef lores;
var video_mode = 4; // 320x240
ifelse;
var video_mode = 6; // 640x480
endif;
var video_depth = 16; // D3D, 16 bit Aufloesung
var fps_max = 50; // 50 fps max
```

Unsere Spielevarianten

Hier ist der Platz an dem wir unsere Spielevarianten je nach Notwendigkeit eingeben:

```
//our skills ************************************************
```

Anzeigen des the A4/A5-Logos

We apply the display of the logo, which is located in the templates directory:

```
////////////////////////////////////////////////////////////
// definiere einen Splash-Screen mit dem gewünschten A4/A5-Logo
bmap splashmap = <logodark.bmp>; // das default A5 logo in templates
panel splashscreen { bmap = splashmap; flags = refresh,d3d; }
```

Die Haupt ('Main')-Function

Für jedes Projekt brauchen Sie eine Haupt- oder **main**-Funktion. Das ist die allererste Funktion, die beim Programmstart aufgerufen wird. In den meisten Fällen ist die **main**-Funktion ziemlich einfach und unsere **main** ist da keine Ausnahme. Schreiben Sie also bitte die folgenden Zeilen hinter Ihre bisher letzten:

```
function main()
{
   fps_max = 50;
   warn_level = 2;    // melde schlechte Texturgroessen und fehlerhaften wdl code
   tex_share = on;    // map entities teilen ihre Texturen

// nentriere den Splash-screen für nicht-640x480 Auflösungen
   splashscreen.pos_x = (screen_size.x - bmap_width(splashmap))/2;
   splashscreen.pos_y = (screen_size.y - bmap_height(splashmap))/2;
   splashscreen.visible = on; // mache ihn sichtbar
   wait(3); // warte 3 Frames (für dreifaches Buffering) bis gerendert und zum Vordergrund geflippt ist.
// jetzt lade den Level
   load_level (<fighting.wmb>);
// warte die benötigte Sekunde und schalte dann den Splash-Screen aus.
   waitt(16);
   splashscreen.visible = off;
   bmap_purge(splashmap);   // Nimm die Logo-Bitmap aus dem Videospeicher
   load_status(); // lade einige globale Variablen, wie Soundvolumen
```

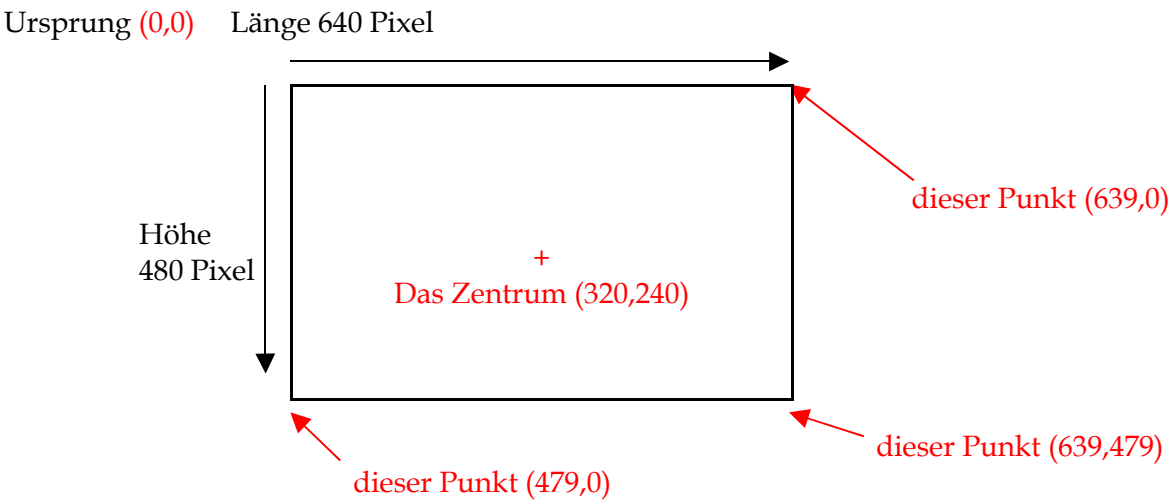
Jede dieser Zeilen ist mit einem Kommentar versehen, so dass sie keiner weiteren Erklärungen bedürfen.

Die Grundlagen von 2D

Um die Vorteile von 2D nutzen zu können, ist es wichtig, einige gewisse Dinge, die wir nun beschreiben werden, zu wissen.

Koordinaten

Die Bildschirmgrösse wird in Pixeln angegeben, Länge und Höhe z.B. 640 x 480. Die erste Eigenheit ist die, dass der Ursprung nicht, wie man vermuten würde, untel links, sondern in der oberen rechten Ecke liegt. Jeder Punkt kann mit (x,y) beschrieben werden:



Die Palettenanpassung:

Die Erläuterung dieses Konzepts kommt Ihnen vielleicht etwas barbarisch vor. Zuerst öffnen wir ein wohlbekanntes Bild, das A5-Logo:

Das A5 Logo

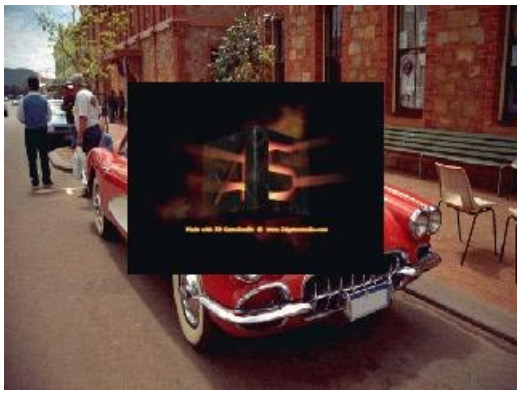
Dann nehmen wir zwei wunderschöne Bilder:

Landschaftsidyll

Nette Kiste

Und wir kopieren (copy/paste) unser Logo in beide. Das kommt dabei heraus:

????????



...Und Was Zum Teufel Soll Das?

Das Logo sieht nicht mehr so prächtig aus wie vorher. Nun, falls Sie keinen Unterschied erkennen können, dann kopieren Sie eben das Auto in das Bild mit dem idyllischen See und Sie werden es bestimmt besser verstehen:

Sieht Nach Illegaler Müllentsorgung Aus

Erläuterungen:

Zunächst müssen Sie wissen, dass dieser Effekt nur im 8-Bit-Modus auftaucht. Aber was um Himmels willen ist bloss ein 8-Bit-Modus? Wenn Sie das begriffen haben, wird alles andere klarer.

Im 8-Bit-Modus wird jeder Bildpunkt (Pixel) durch eine Farbnummer einer Werteskala zwischen 0 und 255 (in Binärcode 00000000 in 11111111, also 8 Bit) dargestellt. Jeder dieser Werte ist das Äquivalent eines Index, der aus 3 Farbwerten (Rot, Grnün und Blau) besteht und Teil einer Farbtabelle ist. Diese Tabelle ist die sogenannte **Palette**.

Nehmen wir ein kleines Beispiel: Den allerersten Pixel unseres Bildes (zum leichteren Verständnis hier die ersten7 Werte unserer Palette):

Position	0	1	2	3	4	5	6	…	255
Farbe									

Position	0	1	2	3	4	5	6	…	255
Farbe									

Mein erster Bildpunkt, der an der oberen linken Ecke meines Bildes hat den Wert 4, also hat er die Farbe, die an vierter Stelle in meiner Palette steht.

First palette :

Unser erster Pixel hat die Farbe der Nummer 4 **dieser Palette** und sieht demgemäs so aus:

Second palette :

Unser erster Pixel hat die Farbe der Nummer 4 **jener Palette** und sieht demnach so aus:

Mein Bild hat sich nicht verändert, der Wert seines ersten Bildbunktes ist immer noch 4. Was sich aber tatsächlich verändert hat, ist die grafische Darstellung der Werte: Im ersten Fall sagt die Palette gemäss der Position 4, ist die Farbe , während die zweite Palette eine andere Farbe an der 4. Stelle findet und diese logischerweise darstellt: .

Zur Beachtung:
Wenn man mit dieser Engine arbeitet, sollte die erste Farbe jeder Palette (**0**) immer schwarz (WerteR = 0, G = 0, B = 0) sein und die letzte (**255**) weiss (Werte R =255, G = 255, B = 255).

Natürlich arbeiten wir so oft es geht mit Bildern von 16 oder 32 Bit. Allerdings wollen wir ja auch, dass unser Spiel zu allen anderen Umgebungen kompatibel ist und wir wissen, dass für bestimmte ältere Maschinen 8 Bit empfohlen sind.

Man begreift sofort, dass das Auto seine Farbe verliert sobald man es auf das Bild mit dem See setzt. Es gibt nämlich überhaupt kein Rot innerhalb der See-Palette. Hier sind die beiden Beispielpaletten für See und Auto:

Palette Vom See

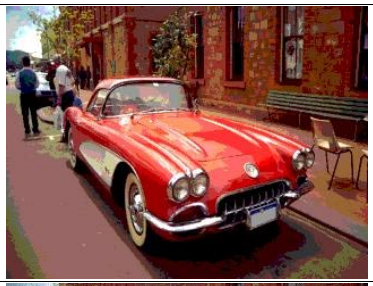
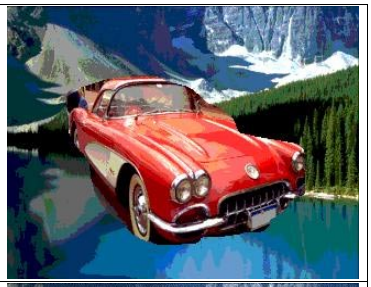
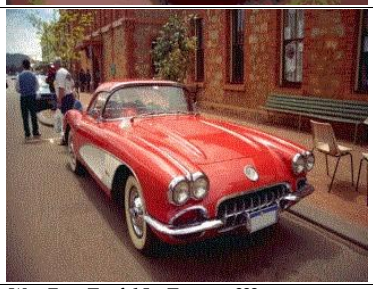
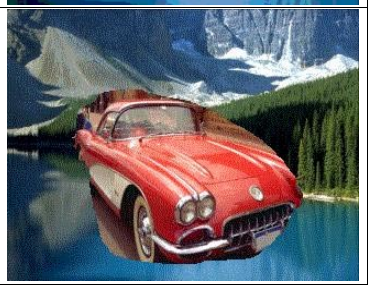
Palette Vom Auto

Wie kriegt man das nur unter einen Hut?

Die nächstliegende Lösung ist es mit den Standardpaletten von Windows zu arbeiten, allerdings ist das, was dabei herauskommt nicht gerade brilliant. Hier das Resultat, wenn Sie die sogenante Palette 666 verwenden:

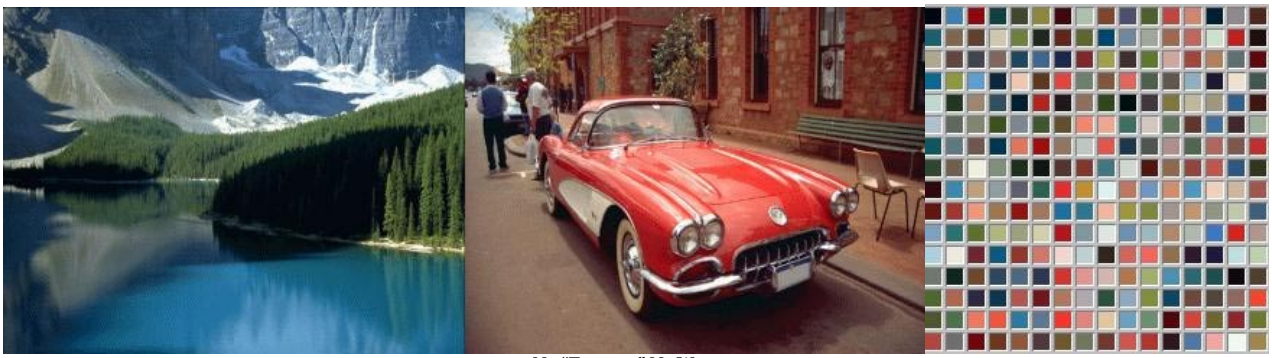
Nix Halbes / Nix Ganzes Und Von Jedem Was

Mein Malprogramm erlaubt mir die Wahl ob ich den "Tramage"-Modus will oder nicht. Hier die beiden Resultate:

Kein tramage			
Mit tramage			

Was Zum Teufel Ist Tramage???

Eine andere Lösung ist es, die beiden Bilder im 16-Bit-Modus zu einem zu vereinigen und das daraus erhaltene Bild in den optimierten 8-Bit-Modus zu konvertieren. Dieses Verfahren prüft sämtliche verwendeten Farben und generiert dann die optimale Kompromisspalette. Hier das Ergebnis:



No "Tramage" No Worry...

Das Overlay:

Overlay ist eine weitere ziemlich wichtige Bezeichnung über die man beim Erstellen eines 2D-Spieles Bescheid wissen sollte. Das WDL-Handbuch erklärt wie folgt:"Wenn dieser Falg gesetzt ist, wird die Farbe 0 (nicht unbedingt schwarz) eines Bildes nicht dargestellt..."

Ich habe auch wirklich ein Bild das ich auf meinem Grundbild auftauchen lassen will. Wie wäre es mit einem Heissluftballon, der über meinen See dahinschwebt?

Füge ich diesen Heisluftballon ein, kriege ich das hier:

Ein Komisches, Störendes Rechteck

Das ist ja nun nicht das, was wir wollten, also nehmen wir die Overlay-Funtion. Dazu müssen wir die Oberflächen, von denen wir wollen, dass sie unsichtbar sind (hier der weisse Hintergrund) durch die Farbe mit der Nummer 0 (meistens schwarz) ersetzen, damit sie nicht gezeichnet wird. Nun, hier ist das Ergebnis:

Warum Der Wohl Leer Ist?

Das ist schon viel besser.
Leider ist es damit aber vorbei, sobald etwas Schwarzes auf dem Ballon selbst drauf ist das wir aber als sichtbar behalten wollen. Nehmen wir an, da ist Werbung drauf, man kann sich leicht

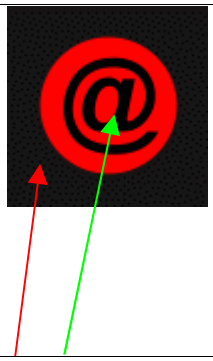
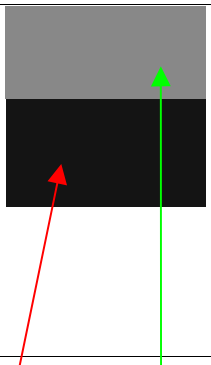
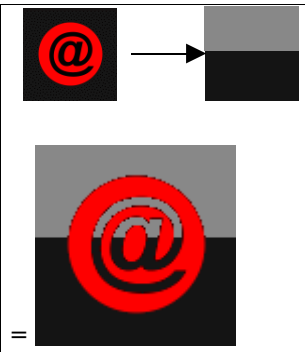
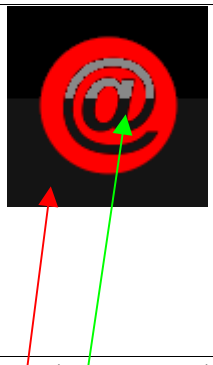
vorstellen, was dabei herauskommt:

Where Have All The @'s Gone?

Nun, das @, was ursprünglich in schwarz war, wurde durchsichtig und ist verschwunden. Ist das nicht furchtbar?

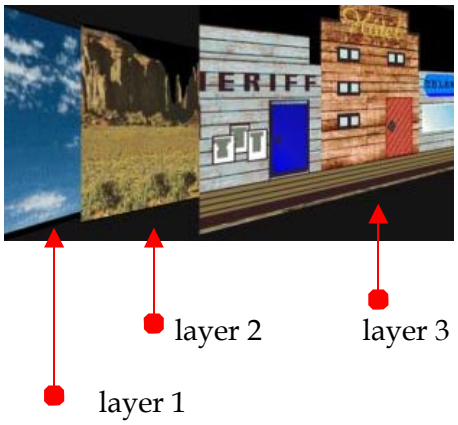
Ein kleiner Tipp von einem alten Programmierhasen:

In meinem Malprogramm setze ich das Bild des Heissluftballons auf einen dunkelgrauen Hintergrund. Die schwarze Farbe habe ich auf transparent gestellt, also werden alle schwarzen Pixel grau. Dann fülle ich das Äussere mit Schwarz und mache es, wie Sie anhand der Graphiken sehen können, auf diese Weise wiederum transparent. Damit man es besser sieht, habe ich hier das Dunkelgrau ein wenig aufgehellt.

			
Das Schwarz ist tief, dunktel, total schwarz (r = 0, g = 0 et b = 0)	Der ganze Untergrund ist normalerweise Dunkelgrau. Das Schwarz hier ist nicht tiefschwarz, sondern dunkelgrau (r = 20, g = 20 et b = 20)	Ich mache das Schwarz transparent und sehe dadurch meinen grauen Untergrund, dann verschmelze ich beides	aussenherum machen wir wieder alles schwarz (0,0,0) Outside, tdas innere schwarz ist bloss dunkelgrau (20,20,20) aber wer sieht das schon?

Layer

Layers legen die Reihenfolge von Panels fest. Layers determine the order of panels. Im Rahmen dieses Tutorials werden wir ein Dorf mit einem Berg dahinter und einem Himmel als Hintergrund aufbauen. Es gibt also 3 Bilder, die hintereinandergestapelt werden müssen und jedes bekommt eine Layer-Nummer.



Nun, genug der Theorie, schreiten wir zur Praxis.

Unser Ziel ist es, eine alte Westernstadt zu erbauen und wir wollen, dass ein Cowboy darin herumspaziert. Obwohl wir es hier mit 2D zu tun haben, wollen wir doch versuchen etwas 3D-mässige Effekte in das Ganze hineinzubringen.

Dazu werden wir uns eines Tricks bedienen, der normalerweise in Filmen verwendet wird. Wir bewegen unsere drei Layers mit verschiedenen Geschwindigkeiten und erhalten so den Eindruck von Tiefe.

Fangen wir mit der Positionierung unseres Sky an:

Zuerst öffnen wir die Dateie **anim2d.wdl** und definieren das Grundpanel. Dieses Panel ist eine Bitmap aus purem Schwarz und 640 x 480 Pixel gross. Wir nennen sie **fond.pcx.**

Wir definieren wie folgt (soll in die Variablen eingefügt werden):

```
bmap fond_map,<fond.pcx>;

panel fond_pan
{
   pos_x = 0; pos_y = 0;
   layer = 0;
   bmap = fond_map;
}
```

Und in unsere **game**-Funktion schreiben wir:

```
fond_pan.visible = on;
```

Wenn wir das Level jetzt starten, sollten wir nach dem Anzeigen des Logos einen schwarzen Bildschirm sehen.

Nehmen Sie also ein bischen Farbe und kreieren Sie einen schönen Himmel. Es ist eine Bitmap von 640 x 120 Pixeln und sie heist **ciel.pcx**. Defieniert wird sie wie folgt (wird in die Variablen eingefügt):

```
bmap ciel_map,<ciel.pcx>;

panel ciel_pan
{
```

```
   pos_x = 0; pos_y = 0;
   layer = 1;
   bmap = ciel_map;
   flags = refresh;
}
```

Und ans Ende unserer **game**-Funktion schreiben wir:

```
ciel_pan.visible = on;
```

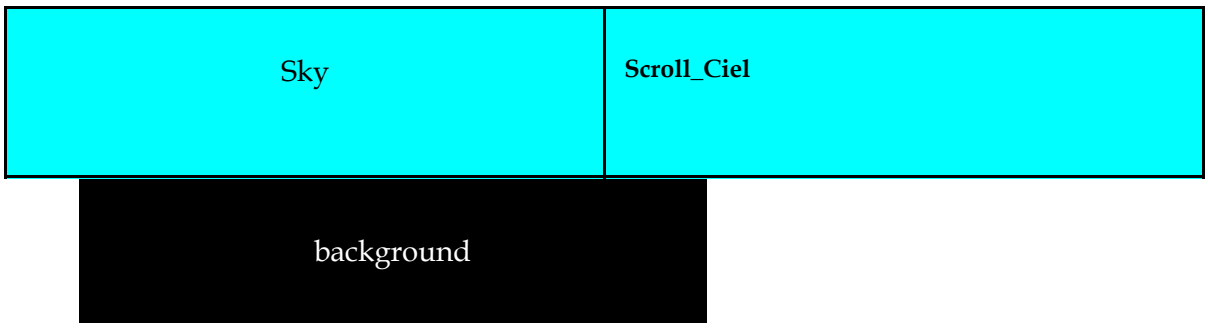
Wenn wir das Level jetzt starten, müssten wir einen wundervollen blauen Himmel haben. Es wäre perfekt, den nun noch zu animieren, meinen Sie nicht? Um das zu erreichen, gehen wir nach der folgenden Methode vor:

1 – wir definieren eine **scroll_ciel_pan** die mit **ciel_pan** identisch ist:

```
panel scroll_ciel_pan
{
   pos_x = 0; pos_y = 0;
   layer = 1;
   bmap = ciel_map;
   flags = refresh;
}
```

2 - Wir zeigen unseren Sky an der Position die dem Wert der gewünschten Verschiebung entspricht.

3 – Wir zeigen unsere **scroll_ciel** an (640 - der Position) die dem gewünschten Verschiebungswert entspricht.



Dabei kommt dann das:

```
var ciel_pos =0;//
```

... am Anfang unserer Variablen heraus.

Und am Ende unserer **game**-Funktion schreiben wir:

```
ciel_pan.visible = on;
scroll_ciel_pan.visible = on;
while(1)
{
     ciel_pan.pos_x = - ciel_pos;
     scroll_ciel_pan.pos_x = 640 - ciel_pos;
     avance_ciel();
     waitt (1);
}
```

Nun tippen wir unsere Funktion **avance_ciel**:

```
function avance_ciel()
{
      ciel_pos +=1;
      if (ciel_pos >= 640){ciel_pos = 0;}
}
```

Wenden wir uns nun dem Berg zu. Er besteht aus einer 640 x 200 Pixel grossen Bitmap und heist **mont.pcx**.

Definieren wir ihn also wie folgt (wird in die Variablen eingefügt)

```
bmap mont_map,<mont.pcx>;

panel mont_pan
{
   pos_x = 0;pos_y = 35; // y = 35 c'est à dire que la montagne est plus basse que le ciel
      layer = 2;
   bmap mont_map;
   flags = overlay,refresh; //overlay pour rendre le noir transparent
}
```

Und in die **game**-Funktion tippen wir vor **while(1)**:

```
mont_pan.visible = on;
```

Starten wir nun das Level, sollten wir eine Berglandschaft unter blauem Himmel haben. Nicht schlecht, wie gefällt es Ihnen?

Das Scrolling des Berges erledigen wir auf dieselbe Weise wie wir es beim Himmel gemacht haben, indem wir nun die folgenden Anweisungen hinzufügen:

Schreiben Sie am Anfang Ihrer Variablen:

```
var mont_pos = 0;
```

Dann unterhalb der Definition des Panels **mont_pan:**

```
panel scroll_mont_pan
{
   pos_x = 0;pos_y = 35; // y = 35 c'est à dire que la montagne est plus basse que le ciel
      layer = 2;
   bmap mont_map;
   flags = overlay,refresh; //overlay pour rendre le noir transparent
```

```
}
```

und in **game():**

```
scroll_ mont_pan.visible = on;
```

Dann fuegen wir noch in unsere **while(1)**-Schleife (rot geschrieben) ein:

```
   while(1)
   {

      ciel_pan.pos_x = - ciel_pos;
      scroll_ciel_pan.pos_x = 640 - ciel_pos;
      mont_pan.pos_x = - mont_pos;
      scroll_mont_pan.pos_x = 640 - mont_pos;
      avance_ciel();
      deplace();
      wait (1);
   }
```

Als nächstes schreiben wir unsere neue Funktion **deplace**:

```
function deplace()
{
   if (key_cur == 1) //droite
   {
      ciel_pos += 1;
      if (ciel_pos >= 640){ciel_pos = 0;}
      mont_pos  += 3;
      if (mont_pos >= 640){mont_pos = 0;}

   }
   if (KEY_CUL == 1) //gauche
   {
      ciel_pos -= 1;
      if (ciel_pos <= 0){ciel_pos = 640;}
      mont_pos  -= 3;
      if (mont_pos <= 0){mont_pos = 640;}

   }
}
```

Speichern und starten Sie das Level jetzt. Bewegen Sie sich mit den Pfeiltasten ein bischen hin und her. Schon hübscher, oder nicht?

Kümmern wir uns nun um den letzten Teil, die Stadt selbst. Dazu nehmen wir die Bitmap mit dem Namen **ville.pcx**. Der Rest müsste jetzt schon Routine für Sie sein:

```
bmap ville_map = <ville.pcx>;
var ville_pos = 0;

panel ville_pan
{
   pos_x = 0;pos_y = 85; // y = 85 c'est à dire que la ville est encore plus basse
   layer = 3;
   bmap ville_map;
   flags = overlay,refresh; //overlay pour rendre le noir transparent
}
```

Und in die **game**-Funktion schreiben wir vor **while(1)** :

```
ville_pan.visible = on;
```

Dann fügen wir in die **while(1)**-Schleife ein:

```
      ville_pan.pos_x = - ville_pos;
```

Für das Verschieben unserer Stadt müssen wir kein kreisförmiges Scrolling machen, denn wir sollten an jeder Ecke stoppen können. Ein einziges Panel ist daher ausreichend. Allerdings müssen wir das Scrolling von Himmel und Berg immer dann, wenn wir an einer der Kanten sind, stoppen.

Wir fügen die folgenden Anweisungen in unsere **deplace**-Funktion (rote Zeilen) ein:

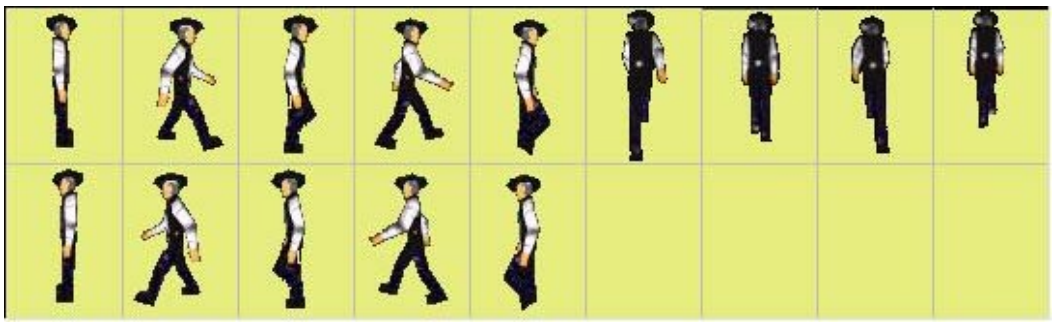
```
   if ((key_cur == 1) && (ville_pos < 1600)) //droite
   {
      ville_pos += 6;
- -  --  - - -  -

   if ((key_cul == 1) && (ville_pos >134)) //gauche
   {
      ville_pos -= 6;
```

Wir können unser Level starten und einen Spaziergang in sämtliche Richtungen durch die Stadt machen .

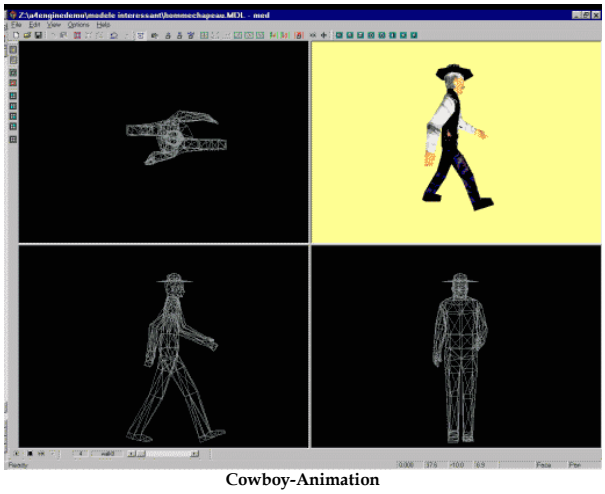
Darstellung des Cowboys

Erinnern wir uns unserer Theorie. Das Anzeigen des Cowboys verlangt einige Erklärung. In Wahrheit besteht der Cowboy aus einer Animations-Bitmap. Hier haben Sie das Bild, das wir verwenden werden:



Damit es besser sichtbar ist, habe ich hier einen gelben Hintergrund genommen. Vergessen Sie aber nicht, dass er die Farbe "0" (nicht zwingend, aber doch meistens schwarz) Ihrer Palette haben muss, damit der Cowboy auch als Overlay funktioniert. Falls Sie kein Palettenbild, sondern ein Echtfarbenbild verwenden, muss der Hintergrund schwarz (0, 0, 0) sein.

Die erste Frage, die Sie nun vermutlich stellen werden, ist die, wie man eine Figur in 3D animiert? Die Antwort ist sehr einfach: indem man MED benutzt und Screenshots macht. Hier ist z.B. die erste Gehphase:



Cowboy-Animation

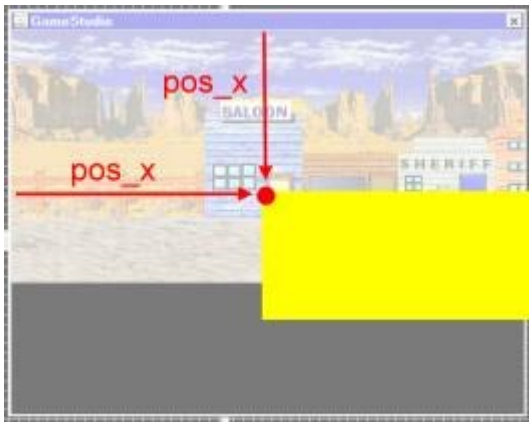
Nach einem gekonnten Schnitt und erfolgter Scalierung, erhalten wir das fertige Bild.

Wir verwenden das Fensterelement des Panels.

Die Anweisung schreibt sich so:

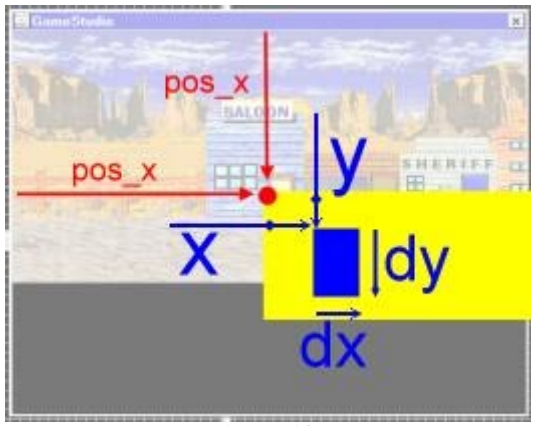
```
window = x,y,dx,dy,bmap,varx,vary;
```

Zuerst müssen wir unser Panel einbinden (hier in gelb):



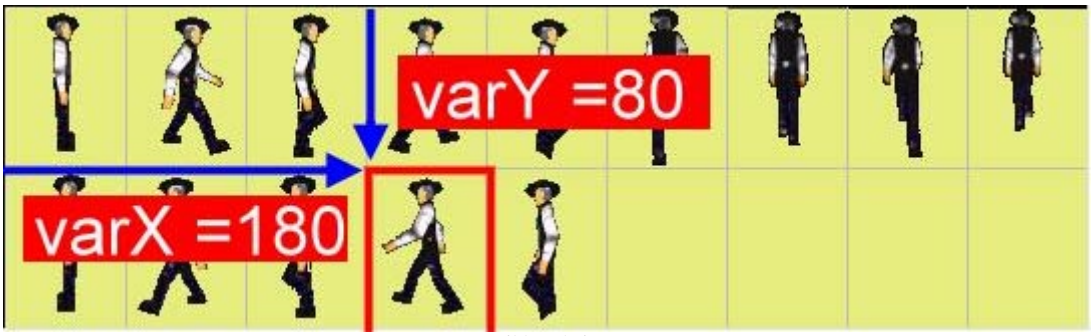
Setting The Cowboy Panel

Dann setzen wir ein Fenster (hier in blau) and die am Ursprung des gelben Panels ausgerichtete x/y-Position (**`pos_x / pos_y`**) an der Breite dx und der Höhe dy, wie hier:



Das Blaue Guckfenster

Dann definieren wir unser Bitmap-Bild und die Koordinaten eines Punktes, der es uns ermöglicht, einen Bildausschnitt von der Grösse unseres blauen Fensters herauszuschneiden (das Beispiel unten zeigt unseren 4. Cowboy):



Bildauschnitt

Und so schreiben wir das:

Innerhalb der Variablen...

```
var cowboy_frame_pos[2]=0,0;
```

Der erste wert ist unser **`varX`** und hat die Werte 0, 60, 120, 180 und 240 für das Gehen von der Seite.

Der zweite Wert ist unser **`varY`** und hat den Wert 0, wenn man nach rechts und 80 wenn man

nach links marschiert.

Wir definieren also unsere Panels wie folgt:

```
bmap cowboy _map,<cowboy.pcx>;

panel cowboy _pan
{
   pos_x = 310;pos_y = 200;
   layer = 7;
   window = 0,0,60,80,cowboy_map,cowboy_frame_pos.x,cowboy_frame_pos.y;
   flags = d3d,overlay,refresh;
}
```

Natürlich finden wir unsere **pos_x** und **pos_y**-Werte wieder. Diese Werte entsprechen der Startposition des Cowboys vor der Saloontür.

Warum Layer 7 obwohl wir doch nur 4 (0 -3) definiert haben? Einfach weil wir, wie jeder gute Programmierer, vorausschauend sind. Es könnte sein, dass später ja noch ein paar Dinge mehr zwischen den Häusern und dem Cowboy sind.

Als nächstes definieren wir unser Fenster, von dem wir wollen, dass es in der oberen linken Ecke des betreffenden Panels ist und es ist 60 Pixel breit und 80 Pixel hoch (0, 0, 60, 80). **Cowboy_map** heisst das Bild übrigends von dem wir uns unsere Frames ausschneiden werden.

Puh! Hier tat Aufklärung not.

Nun müssen wir das Panel nur noch anzeigen. Das geschieht indem die folgende Anweisung am Anfang von **game** und zwar vor **while(1)** geschrieben wird:

```
   cowboy_pan.visible = on;
```

Und die Cowboybewegungen werden von den folgenden Zeilen (in rot) verwaltet:

```
   if ((key_cur == 1) && (ville_pos < 1570)) //droite
   {
      cowboy_frame_pos.x +=60;cowboy_frame_pos.y = 0;
      if (cowboy_frame_pos.x >= 300){cowboy_frame_pos.x = 60;}
   - - - -  --  - - - -  -

   if ((key_cul == 1) && (ville_pos >128)) //gauche
   {
      cowboy_frame_pos.x +=60;cowboy_frame_pos.y = 80;
      if (cowboy_frame_pos.x >= 300){cowboy_frame_pos.x = 60;}
```

Jetzt wollen wir das Level mal starten und einen vergnüglichen Spaziergang durch unsere Westernstadt machen.

Unglücklicherweise wird uns der Spass schnell von all den kleinen ins Auge springenden Fehlerchen verdorben.

Fehlersuche

Erster Fehler: wenn Sie den Player während des Gehens anhalten, behält er sein Bein in der Luft und das sieht nicht so perfekt aus.

Das zweite was wir verbessern müssen: es wäre wirklich um einiges besser, wenn man nur einmal auf die Pfeiltaste drücken müsste, damit der Cowboy zur entsprechenden Tür geht. (Sie werden bereits gemerkt haben, dass es das Ziel ist, Gebäude zu betreten).

Und das brauchen wir dazu, diese Probleme zu beheben: eine Variable für die Richtung, ein Schritte-Zähler und eine Variable für die Bewegungen. Also, machen Sie sich an die Arbeit und schreiben Sie in die Variablen:

```
var waiting = 0;
var go_right = 1;
var go_left = 2;
var state = 0;

var look_right = 1;
var look_left = 2;
var look_at = 1;

var compteur_pas = 0;
```

Und wir modifizieren unsere **deplace**-Routine ziemlich merklich.

Hier ist sie in Gänze:

```
function deplace()
{
   if ((key_cul == 1) && (state == waiting))
   {
      state = go_left;
      compteur_pas = 20;
      if (key_ctrl == 1){ compteur_pas = 40;}
   }

   if ((key_cur == 1) && (state == waiting))
   {
      state = go_right;
      compteur_pas = 20;
      if (key_ctrl == 1){ compteur_pas = 40;}
}


   if ((state == go_right) && (look_at == look_left))
   {
      state = waiting;
      look_at = look_right;
      cowboy_frame_pos.x = 0;cowboy_frame_pos.y = 0;
   }

   if ((state == go_right) && (compteur_pas > 0))
   {
      look_at = look_right;
      if (ville_pos < 1570) //droite
      {
         cowboy_frame_pos.x +=60;cowboy_frame_pos.y = 0;
         if (cowboy_frame_pos.x >= 300){cowboy_frame_pos.x = 60;}
         ville_pos += 6;
         ciel_pos += 1;
         if (ciel_pos >= 640){ciel_pos = 0;}
         mont_pos  += 3;
         if (mont_pos >= 640){mont_pos = 0;}
```

```
         compteur_pas -=1;
         if (compteur_pas ==0)
         {
            state = waiting;
            cowboy_frame_pos.x = 0;cowboy_frame_pos.y = 0;
         }
      }
      else
      {
         compteur_pas = 0;
         state = waiting;
         cowboy_frame_pos.x = 0;cowboy_frame_pos.y = 0;
      }

   }

   if ((state == go_left) && (look_at == look_right))
   {
      state = waiting;
      look_at = look_left;
      cowboy_frame_pos.x = 0;cowboy_frame_pos.y = 80;
   }

   if ((state == go_left) && (compteur_pas > 0))
   {
      look_at = look_left;
      if (ville_pos >128) //gauche
      {
         cowboy_frame_pos.x +=60;cowboy_frame_pos.y = 80;
         if (cowboy_frame_pos.x >= 300){cowboy_frame_pos.x = 60;}
         ville_pos -= 6;
         ciel_pos -= 1;
         if (ciel_pos <= 0){ciel_pos = 640;}
         mont_pos  -= 3;
         if (mont_pos <= 0){mont_pos = 640;}

         compteur_pas -=1;
         if (compteur_pas ==0)
         {
            state = waiting;
            cowboy_frame_pos.x = 0;cowboy_frame_pos.y = 80;
         }
      }
      else
      {
         compteur_pas = 0;
         state = waiting;
         cowboy_frame_pos.x = 0;cowboy_frame_pos.y = 80;
      }
   }
}
```

Wenn Sie das Level jetzt starten, werden Sie feststellen, dass:

- Eine Richtungsänderung im Stehen nun schlicht und ohne Vorwärtsbewegung die Richtung ändert.

- Unser Cowboy der Tastenanweisung prompt folgt und er sich in Richtung der nächten Tür in angegebener Richtung bewegt.

Nicht schlecht, aber es wäre gut, dem Ganzen noch etwas mehr Leben einzuhauchen, meinen Sie

nicht auch?

Der Reiter

Wir werden einen Reiter erscheinen lassen, der unsere Stadt zufallsabhängig durchquert. Hier ist das Bitmapbild (der Hintergrund ist normalerweise schwarz und es gibt keine Trennungslinien):

Auf die übliche Frage:"wie belebt man ein Pferd?" ist die Antwort wieder diesselbe:"Man verwendet MED". Der Unterschied allerdings ist, dass ich kein vollanimiertes Pferd auftreiben konnte und daher selbst eines in all seinen Animationsdetails erstellen musste. (Da es ja nicht Gegenstand dieses Workshops ist, enthalte ich mich hier einer detaillierten Ausführung. Vielleicht finden Sie die Erklärungen hierzu in einem weiteren Mini-Tutorial oder, was wahrscheinlicher ist, in den AUM Anwendermagazin).

Für die Animation des Pferdes benutzen wir das gleiche Fensterelement wie beim Cowboy. Der Hauptunterschied zum Cowboy ist der, dass sich das Pferd viel weiter über den Bildschirm bewegt. Daher müssen wir auch das Panel bewegen. Wie geht das?

Wir machen das in zwei Schritten. Zuerst animieren wir das Pferd in der Bildschirmmitte:

Wir erstellen eine Variable:

```
var cheval_frame_pos = 0;
```

Dann definieren wir unser Panel:

```
bmap cheval_map,<cheval.pcx>;
panel cheval_pan
{
   pos_x = 400;pos_y = 200;
      layer = 8;
   window = 0,0,140,102,cheval_map,cheval_frame_pos.x,0;
         flags = d3d,overlay,refresh;
}
```

`Cheval_frame_pos.x` wird die Werte 0, 140, 280, 420 und 560 annehmen. Beim Wert 700 wird er auf 0 zurückgesetzt.

Bleibt uns noch, das Panel anzuzeigen. Das machen wir, indem wir diese Anweisung am Anfang unserer **`game`**-Funktion, vor der **`while(1)`**-Schleife einfügen.

```
   cheval_pan.visible = on;
```

Die Animation des Pferdes wird von den folgenden (roten) Zeilen verwaltet:

```
function deplace()
{
        cheval_frame_pos.x +=140;
        if (cheval_frame_pos.x >= 700){cheval_frame_pos.x = 0;}

   if ((key_cul == 1) && (state == waiting))
```

Starten Sie das Level und betrachten Sie es. Bitte vergessen Sie nicht zu applaudieren.

Jetzt müssen wir unser Pferd nur noch verschieben und dafür sorgen, dass es zufallsabhängig auftaucht.

Zu Testzwecken weisen wir die Zufalls-Funktion einer Taste zu (z.B. Leertaste):

```
on_space anim_cheval;

function anim_cheval
{
   cheval = 1;
}
```

und an den Anfang unserer Variablen tippen wir:

```
var cheval = 0;
```

Die Schwierigkeit (nur eine klein, versprochen) ist, dass wir das Pferd nicht hinsichtlich des Bildschirms verschieben, sondern im Bezug auf unsere Stadt, die wegen des Scrolling wiederum selbst verschoben wird.

In unseren Variablen fügen wir ein:

```
var cheval_pos = 0;
```

Und in der **game**-Funktion, gleich vor dem Aufrufen (**call**) der Funktion **deplace** schreiben wir:

```
cheval_pan.pos_x = cheval_pos - ville_pos;
deplace();
```

Und weil wir ungeduldigt sind, sehen wir gleich nach was geschieht. (Solange Sie die Leertaste nicht drücken, passiert gar nichts). Oh ist das nicht wundervoll...

Aber welch ein Frust: Was geschieht eigentlich wirklich vor und nach dem Anzeigen? Bewegt sich das Perd weiter? Sind wir sicher, dass alles so läuft, wie wir ers wollen? Und was, wenn Sie die Zufallsfunktion (per Leertaste)auslösen wollen und es taucht gar kein Pferd auf? Liegt das an einem Fehler in der Zufallsfunktion - liefert sie nicht den erwarteten Wert zurück? Dann ist das nicht länger nur Frust, es ist sowas wie die metaphysische Angst: “Bin ich gut oder bin ich es nicht?”

Selbstverständlich sind wir mit Debugern ausgerüstet aber das ist nicht das Richtige, um unsere Ängste zu zerstreuen. Wäre es nicht einfach perfekt, wenn wir die Variablen, über die wir Bescheid wissen wollen, während des ganzen Prozesses auf dem Bildschirm anzeigen könnten? Sicher, wir könnten die Funktion **D** dahingehend überarbeiten, dass sie unsere Variablen anzeigt, aber es ist besser, es auf die andere Art zu machen (wir sind doch hier um zu lernen, oder nicht?). Also bereiten wir jetzt das weitere Fortschreiten des Spieles vor, es gibt noch einiges zu schreiben:

Um einen Text darzustellen bedarf es eines Fonts. Was ein Font ist? Es ist die Datei, die uns das Bild des Buchstaben, den wir drucken wollen, liefert. Wir wollen zum Beispiel ein 'A' anzeigen und es erscheint die graphische Darstellung davon auf dem Bildschirm:

A or *A* or E or A or *A* or ***A*** or A or **A** or **A** or A or **A** or *A* or ✌ or ✌ oder B (warum nicht)…

Da der Buchstabencode genormt ist (zum Glück, sonst hätten wir Anarchie), hat 'A' im ASCII-

Standard (das ist der, den wir verwenden) z.B. immer den Wert 64. Der Platz, der den ersten druckbaren Buchstaben repräsentiert hat den Wert 32. Die graphische Darstellung der Werte 0, 48 etc. ist Ihnen überlassen. Wenn Sie wollen, können Sie auch beschliessen: 'wenn ich Buchstabe 64 angezeigt haben will, dann zeichne ein kleines Auto'.

Aber bevor wir anfangen, nehmen wir uns ein bischen Zeit zum Nachdenken und dann stellen wir die guten Fragen:

- Was brauche ich zur Darstellung? Handelt es sich ausschliesslich um Zahlen, reicht es, einen Font von nur 11 Zeichen zu machen (Leerfeld + 10 Zahlen). Verwende ich die 7-Bit-ASCII-Tabelle, wird mein Font 256 Zeichen haben.

- Wie gross sollen die Buchstaben sein?

- Werde ich einen bereits existierenden Font nehmen oder zeichne ich mir meine eigenen Buchstaben?

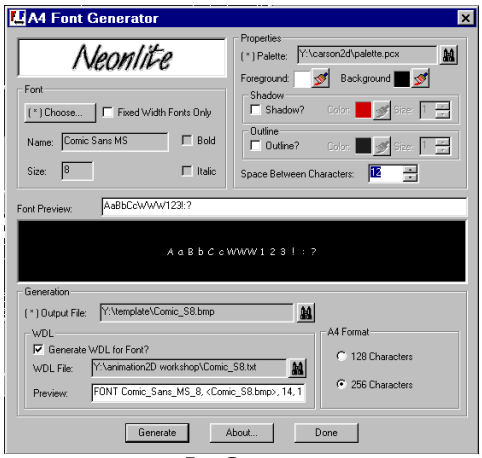
In unserem Fall, wo wir ja international sein wollen und daher auch die Eigenheiten verschiedener Sprache berücksichtigen müssen, sollten wir den vollen ASCII verwenden. Das ist ein Font mit 256 Zeichen.

Bitte beachten Sie, dass auch bei identischen Fonts ein Unterschied in der Darstellung unter Windows und 3D GameStudio besteht. Windows benutzt das proportionale Fonts (will heissen, dass der aktuell gebrauchte Platz je nach Buchstabenweite, Stil usw. variiert. Ein 'i' braucht daher weniger Platz, als z.B. ein 'W'.) A4 / A5 hingegen verwaltet Buchstaben als Komponenten einer festgelegten Grösse und beansprucht immer gleichviel Platz (starrer Font).

Zur Verdeutlichung haben Sie hier dieselbe Zeile in verschiedenen Fonts:

Font: arial corps 11	Abcdefghijkl WAIW	Proportionaler Font
Font: agency corps 11	Abcdefghijkl WAIW	Proportionaler Font
Font courrier corps 11	Abcdefghijkl WAIW	Starrer Font

Man hört, die 3D GameStudio-Gemeinde sei eine grossartige Gemeinschaft. Auf der Link-Seite finden Sie eine Anwendung, die es Ihnen ermöglicht, Ihren Font aus einem Windows-Font zu generieren. Und so sieht sie aus:



Font Generator

Und hier das Ergebnis:

Fertiger Font (bloss Wie Kriegt Man Das Jetzt In Eine Sinnvolle Reihenfolge?)

Zum Schluss müssen wir den Font noch in unserem Skript definieren (Sie können die TXT-Datei, die mit dem A4-Font-Generator erstellt wurde kopieren). Schreiben Sie bitte am Anfang Ihrer Variablen:

```
font comic, <comic_s8.bmp>,10,15;
```

Hier das erwünschte Ergebnis:

Die Textkoordinaten beziehen sich auf den Ursprung des Fensters (obere linke Ecke).	
	Die Koordinaten der Variablen beziehen sich auf den Ursprung des Panels in dem sie angezeigt werden.

Um unseren Text zu definieren, schreiben wir die folgenden Zeilen:

```
string mouchard = "cheval     cheval_pos     ville_pos";
text mouchard_txt
{
   pos_x = 10;pos_y = 350; // distance par rapport au 0,0 de l'écran
   layer = 10;
   font = comic;
   string = mouchard;
}
```

Dann bereiten wir die Anzeige unserer Variablen vor:

```
panel affiche_var_pan
{
   pos_x = 0;pos_y = 380; // position de notre panneau par rapport au 0,0 de l'écran
      layer = 11;
   bmap fond_map; //on reprend notre fond noir d'origine
   digits = 35,0,1,comic,1,cheval;
   digits = 140,0,4,comic,1,cheval_pos;
```

```
   digits = 270,0,4,comic,1,ville_pan.pos_x;
         flags = overlay,refresh;
}
```

Wie muss man das lesen? Nehmen wir die blaue Zeile als Beispiel:

Die Anweisung lautet: **`digits = x, y, len, font, factor, var;`**

Digits = 35, // platziere im Abstand von 35 Pixel von der Kante des Panels **`affiche_var`** ➔ **X** (horizontale Richtung)
0, // platziere im Abstand von 0 Pixel von der oberen Kante des Panels**`affiche_var`** ➔ **Y** (vertikale Richtung)
1, // Anzahl der anzuzeigenden Zeichen (unsere Variable kann die Werte 0 oder 1 annehmen, daher genügt ein Zeichen) ➔ **lengh**
comic, //verwende den Font namens **`comic`** ➔ **font**
1, // multipliziere das Ergebnis mit 1 ➔ **factor**
cheval ;// der Inhalt der Variable mit dem Namen **`cheval`** ➔ **var**

Der Faktor ist seht hilfreich, denn **`digits`** zeigt die gesamte Variable an. Ist Ihre Variable zur Zeit 0,123, zeigt **`digits`** 0 an. Nun multiplizieren Sie den Faktor = 1000 und **`digits`** liefert 123.

Ein Tipp: wenn Sie z.B. einen kleinen roten Kreis anstelle der 0 und einen kleinen grünen Kreis anstatt der 1 in Ihre Bitmap malen, wird der Wert der Variablen **`cheval`** nicht mehr als 0 oder 1 angezeigt, sondern als roter bzw. grüner Kreis.

Bleibt weiter nichts zu tun, als die Anzeige durch drücken beispielsweise der Taste **[F12]** auszulösen (fügen Sie also am Ende des Skriptes ein):

```
on_f12 espion;

function espion()
{
   if (mouchard_txt.visible == off)
   {
         affiche_var_pan.visible = on;
         mouchard_txt.visible = on;
        }
        else
   {
         affiche_var_pan.visible = off;
         mouchard_txt.visible = off;
        }
}
```

Wir starten das Level und durch Drücken von **[F12]** erscheint die Anzeige. Drücken Sie nun die **[Leertaste]**, stellen Sie fest, dass die Variable **`horse`** von 0 auf 1 springt und die Position des Pferdes zunimmt (bewegt sich über den Bildschirm). Wenn wir unseren Cowboy in Bewegung setzen, wird auch die Position der Stadt modifiziert. Während wir dies betrachten, dürfen wir feststellen dass das alles gut gemacht ist.

Prima, und jetzt wollen wir das Pferd nicht durch die **[Leertaste]** startem, sondern zufallsabhängig.

Wir modifizieren den Anfang unserer **`deplace`**-Function wie folgt (rote Zeile):

```
   if (cheval == 1)
   {
```

```
      cheval_frame_pos.x +=140;
      if (cheval_frame_pos.x >= 700){cheval_frame_pos.x = 0;}
      cheval_pos += 12;
      if (cheval_pos > 2185)
      {
         cheval_pos = 0;
         cheval = 0;
      }
   }
   else {cheval = int(random(100));}
```

Wir löschen die folgenden Zeilen (Sie können sie erstmal noch behalten aber vergessen Sie nicht, sie vor Fertigstellung des Spieles zu entfernen!):

```
on_space anim_cheval;

function anim_cheval
{
   cheval = 1;
}
```

Und verändern die folgende Zeile (Blau wird durch Rot ersetzt):

Vorher:

```
   digits = 35,0,1,comic,1,cheval;
```

Nachher:

```
   digits = 35,0,2,comic,1,cheval;
```

Actually we want 2 characters to be displayed to see the result of **int** (random (100)).

Sound

Aber was wäre unser Spiel so ganz ohne jegliche Soundeffekte, die Welt des Schweigens?

Beginnen wir mit dem einfachsten und augenscheinlichsten: den Schritten des gehenden Cowboys.

Wir erstellen 2 Variablen für den Sound und fügen sie in unsere Variablen ein:

```
sound tap, <tap.wav>;
var taphandle = 0;
```

Dann spielen wir den Sound ab solange sich der Cowboy bewegt und wir stoppen ihn in dem Moment, in dem der Cowboy stehen bleibt.

Kopieren Sie (nur die roten Zeilen) in unsere **movement**-Routine:

```
   if ((state == go_right) && (compteur_pas > 0))
   {
      if (taphandle ==0) //le son n'est pas joué
      {
         play_loop (tap,20);
         taphandle = result;
      }
```

```
      -   - - - - - - - - - - - - - -

    else
    {
       compteur_pas = 0;
       state = waiting;
       cowboy_frame_pos.x = 0;cowboy_frame_pos.y = 0;
    }
    if (state == waiting)
    {
       stop_sound (taphandle);
       taphandle = 0;
    }

      -   - - - - - - - - - - - - - -

if ((state == go_left) && (compteur_pas > 0))
{
    if (taphandle ==0) //le son n'est pas joué
    {
       play_loop (tap,20);
       taphandle = result;
    }
      -   - - - - - - - - - - - - - -

    else
    {
       compteur_pas = 0;
       state = waiting;
       cowboy_frame_pos.x = 0;cowboy_frame_pos.y = 80;
    }
    if (state == waiting)
    {
       stop_sound (taphandle);
       taphandle = 0;
    }
      -   - - - - - - - - - - - - - -
```

Das war nun nicht so schwierig, auch wenn es nicht gerade genial ist. Ich überlasse es Ihnen nach einem passenden Schritt-Sound, der Ihnen besser gefällt, zu suchen.

Interessant wird es an dem Punkt, wo wir zu den Pferdeschritten kommen. Es liegt auf der Hand, dass die Lautstärke des Sounds, je nach Abstand zwischen Pferd und Cowboy variieren muss.

Wenn Sie sich die Anweisungen, die Sounds betreffen gut angeschaut haben, so ist dies die einzige, die eine Soundabstimmung vorsieht:

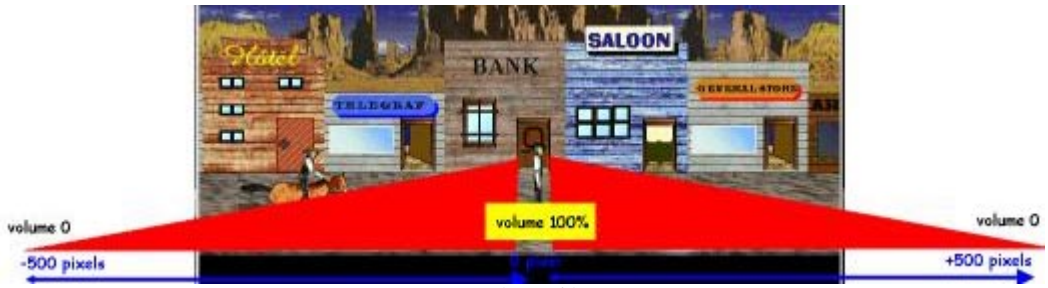
```
tune_sound(handle,varvol,varfreq);
```

Natürlich spielen nehmen wir die Variable **`varvol`**.

Wir wollen: je näher das Pferd herankommt, je lauter soll der Sound werden, je weiter es sich entfernt, desto leiser soll er werden.

Ich habe eine komplett willkürliche Regel aufgestellt die festlegt, dass bei einer Distanz von mehr als 500 Pixeln zwischen Pferd und Cowboy kein Sound zu hören sein soll. Innerhalv von 500 und 0 Pixeln nimmt die Lautstärke des Sounds zu.

Zur Veranschaulichung:



Sound Pyramide

Es versteht sich von selbst, dass es nicht die Position des Pferdes im Bezug auf uns ist, die uns interessiert, sondern der Abstand des Pferdes im Hinblick auf die Stadt und in Relation zu uns. Auch wenn wir uns immer in der Mitte des Bildschirms befinden, ist unsere Position im im Bezug zur Stadt nicht gleichbleibend. Diese Position ist bereits berechnet, da sie und dazu dient das Panel mit dem Pferd anzuzeigen und ihr Name ist: **cheval_pan.pos_x**. Wir ziehen sie relativ zum Bildschirm von unserer Position (die Mitte = 320). Als Ergebnis erhalten wir einen Wert der (mehr oder weniger) zwischemn 500 und 0 liegt. Nun dividieren wir dieses Resultat durch 5 um einen Wert zwischen 0 und 100 zu erzielen, denn dieses sind exakt die Grenzwerte unseres Lautstärkereglers. Selbstverständlich nehmen wir die Absolutwerte, damit wir das Vorzeichen loswerden.

Das kleine Problem, das dabei noch bleibt, ist die Tatsache, dass das gelieferte Resultat 0 ist, falls das Pferd nah bei uns ist und einen Wert von 100 hat sobald es am weitesten weg ist. Das kriegen wir in den Griff indem wir das erhaltene Ergebnis von 100 abziehen und das ganze dann an unsere Anweisung zurückliefern.

In Computeralgebra schreibt sich das so:

```
volume = 100-(abs(320-cheval_pan.pos_x)/5);
```

Trauen Sie mir, ich verspreche, dass es funktioniert.

Jetzt müsen wir das Ganze noch in unserem Skript unterbringen. Wir erstellen folgende Variablen:

```
sound pascheval, <pascheval.wav>;
var paschevalhandle = 0;
var volume = 0;
```

Dann fügen wir in unserer **movement**-Routine noch (rote Zeilen)hinzu :

```
function deplace()
{
   if (cheval == 1)
   {
      if (paschevalhandle ==0) //le son n'est pas joué
      {
         play_loop (pascheval,5);
         paschevalhandle = result;
      }
      else
      {
         volume = 100-(abs(320-cheval_pan.pos_x)/5);
         tune_sound (paschevalhandle,volume,0);
      }

      cheval_frame_pos.x +=140;
```

```
      if (cheval_frame_pos.x >= 700)
      {
         cheval_frame_pos.x = 0;
      }
      cheval_pos += 12;
      if (cheval_pos > 2185)
      {
         cheval_pos = 0;
         cheval = 0;
      }
   }
   else
   {
      cheval = int(random(100));
      stop_sound(paschevalhandle);
      paschevalhandle = 0;
   }
```

Starten Sie das Level und Sie haben ein nettes, kleines Vergnügen.

Nicht schlecht, trotzdem bin ich ein wenig enttäuscht. Ich habe den Helm auf dem Kopf, meine Soundkarte unterstützt Stereoeffekte. Es wäre viel besser, wäre der Sound links wenn das Pferd auf der linken Seite ist und rechts, wenn es dort ist. Na, wie wäre das?

Ich habe die Anweisungsparameter nochmal durchgelesen, aber vergeblich. Nichts erlaubt uns eine solche Differenzierung. Vielleicht habe ich ja nicht die richtige Anweisung genommen? Lassen Sie uns weiter im WDL-Handbuch lesen.

Würde die Anweisung **play_sound** (my,sound,Var), die einen 3D-Sound abspielt nicht besser passen? Ja, aber die Erläuterungen reden von Dingen wie **entity** und **my.** Solange wir nicht unter Drogen stehen, haben wir davon aber nichts in einem 2D-Spiel.

Nun ich werde Ihnen die Augen öffnen: Die A4 / A5 -Engine ist so phantastisch, dass wir auch in 2D eine Kamera haben und wir können auch Entities erstellen. Ist das nicht erstaunlich?

Also erstellen wir eine Entity als Aufhänger, den wir dann an unserem Pferd festmachen und die Kamera wird an die Player-Position gesetzt.

Nur zu, schreiben Sie diese Zeilen in die Variablen:

```
synonym bidon {type entity;}
var bidon_pos[3] =0,200,0;
```

Und in die **main**-Funktion schreiben Sie bitte die roten Zeilen:

```
   load_status();
   create <bidon.pcx>,bidon_pos,f_bidon;
   game();
}

function f_bidon
{
   bidon = me;
   while (bidon == null){wait 1;}
}
```

In der **game**-Funktion schreiben Sie dann die roten Linien in die Schleife (loop):I

```
   cheval_pan.visible = on;
   camera.x = 0;
   camera.y = 0;
   camera.z = 0;

   while(1)
   {

      ciel_pan.pos_x = - ciel_pos;
      scroll_ciel_pan.pos_x = 640 - ciel_pos;
      mont_pan.pos_x = - mont_pos;
      scroll_mont_pan.pos_x = 640 - mont_pos;
      ville_pan.pos_x = - ville_pos;
      //avance_ciel();
      bidon.x = cheval_pan.pos_x+60;
```

Und in unserer **deplace**-Funktion ersetzen Sie die blauen Zeilen durch die roten:

```
function deplace()
{
   if (cheval == 1)
   {
      if (paschevalhandle ==0) //le son n'est pas joué
      {
         play_loop (pascheval,5);
         paschevalhandle = result;
      }
      else
      {
         volume = 100-(abs(320-cheval_pan.pos_x)/5);
         tune_sound (paschevalhandle,volume,0);
      }


   if (cheval == 1)
   {
      if (paschevalhandle ==0) //le son n'est pas joué
      {
         play_entsound (bidon,pascheval,200);
         paschevalhandle = result;
      }
      else {if (snd_playing(paschevalhandle)==0){paschevalhandle =0;}}
```

Starten Sie das Level und machen Sie einen Spaziergang. Es ist mit einem Mal realistischer, meinen Sie nicht?

Nun, da Sie alles gut verstanden haben, geht es rasch, das Pferd in der Nähe des Players zufallsabhängig wiehern zu lassen.

Wir tippen die roten Zeilen:

```
sound pascheval, <pascheval.wav>;
var paschevalhandle = 0;
sound henni, <henissem.wav>;
var hennihandle = 0;
- - - - - - - - - - -

   if (cheval == 1)
   {
```

```
//henni si moins de 100 pixels du joeur et 1 fois sur 3
if ((abs(320-cheval_pan.pos_x) < 100) && (random(3) <1))
{
   if (hennihandle ==0) //le son n'est pas joué
   {
      play_entsound (bidon,henni,100);
      hennihandle = result;
   }
   else {if (snd_playing(hennihandle)==0){hennihandle =0;}}

}

if (paschevalhandle ==0) //le son n'est pas joué
```

Zum Schluss

Und ja, nun sind wir schon fertig. Diese Übungssoftware hat ihren Zweck erfüllt, wenn sie Ihnen erste Grundlagen zum Erstellen Ihrer eigenen 2D-Animation vermittelt hat.

Fussnote in der Historie: Carson City ist ein Spiel, das zuerst auf Spectrum sinclair und dann auf Amstrad assembler entwickelt wurde. Dann habe ich angefangen mit C++ und Directdraw an dem Projekt zu arbeiten, aber dann erschien 3D GameStudio underneath hat meine Pläne durcheinandergeworfen.

Diese Tatsache hat mich dazu gebracht, über eine 3D-Version nachzudenken. Aber dann, vielleicht weil 3D GameStudio auf jeden Fall einfacher als C++ ist, wurde die Idee geboren, das 2D-Spiel mit 3D GameStudio zu machen.

Vorerst hoffe ich, Sie hatten Spass mit mir ein Stück Wegs auf einer kleinen Strasse innerhalb des faszinierenden Universums des Spieleprogrammierens zu gehen und dass es mir gelungen ist, etwas von meinem Wissen und viel von meiner Passion zu vermitteln.

Besuchen Sie meine Seite: http://alainbrgeon.free.fr

PS: Sie finden sämtliche Dateien dieses Szenarios unter dem Namen **`anim2d.all`** .