Übersetzung: Felicitas Oehler

Vorwort
Liebe Leser,

Ich habe diesen Workshop erstellt, um die Frage "Wie mache ich einen Flugsimulator mit 3D GameStudio?" einer Antwort näher zu bringen. Gleichzeitig befasse ich mich einigen neuen Features, die mit der neuen Version 4.19 verfügbar geworden sind.

Dieser Workshop ist, wie schon der Venture–Workshop, hauptsächlich für Anwender gedacht, die bereits einige Vorerfahrung mit 3D GameStudio haben. Ich setze voraus, dass Sie die Tutorials durchgearbeitet haben und mit den Werkzeugen (WED,MED und WDL) umgehen können.

Dieser Text soll die Dokumentation, die mit 3D GameStudio mitgeliefert wird ergänzen, nicht ersetzen. Wenn Ihnen etwas unklar ist, lesen Sie bitte im Referenz-Handbuch nach. Für jedwede unklare Ausdrucksweise, fehlerhafte Codes, Irrtümer oder Versäumnisse entschuldige ich mich im Voraus.

Ich hoffe, sie empfinden diese Übungen als informativ und haben Spass dabei.

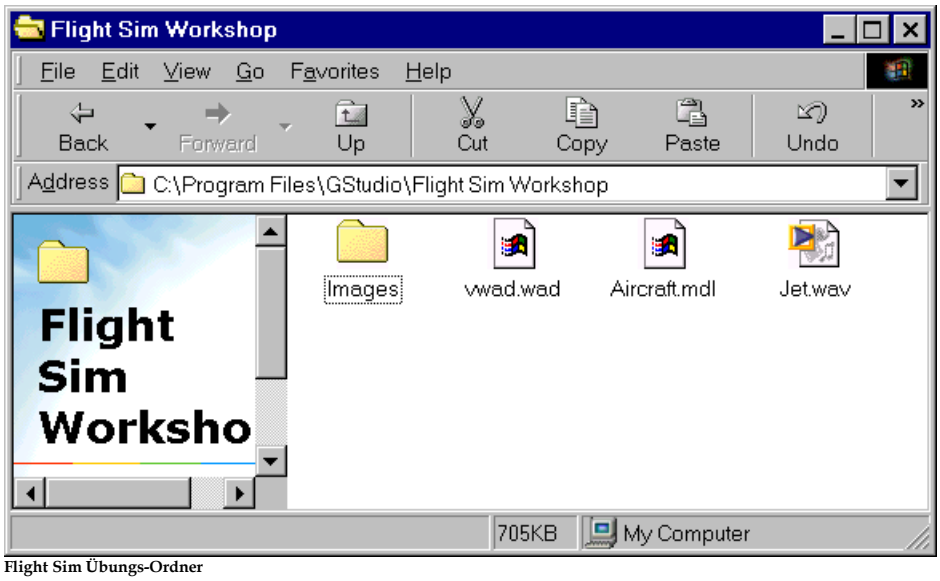
Doug Poston.

Besorgen Sie sich die neueste Version
Bevor Sie anfangen, vergewissern Sie sich, dass Sie die neueste Version von 3D GameStudio (4.19 oder neuer) haben. Wir werden nämlich einige der neu hinzugekommenen Features verwenden. Ausserdem werden Sie den neuesten Levelcompilierer (World Builder) brauchen, denn für einen Flugsimmulator benötigen wir eine extrem weiträumige Welt.

Bereiten Sie Ihr Arbeitsumfeld vor
Erstellen Sie einen Ordner mit dem Namen "Flight Sim Workshop" in Ihrem Gstudio – Verzeichnis. Das ist der Ordner, in dem alle Ihre Spieleelemente gespeichert werden.

Das erste, was wir in unserem Ordner unterbringen, sind die Models und Bilddateien. Sollten Sie diese noch nicht haben, gehen Sie auf die Conitec – Downloadseite (http://www.conitec.net/a4update.htm) und suchen nach dem FlightSim – Level. Entzippen Sie den Inhalt in Ihren Ordner. Mindestens die folgenden Dateien sollten sich nun darin befinden:

compass.pcx
mount16.pcx
sky1.pcx
nsky1.pcx
aircraft.mdl
jet.wav

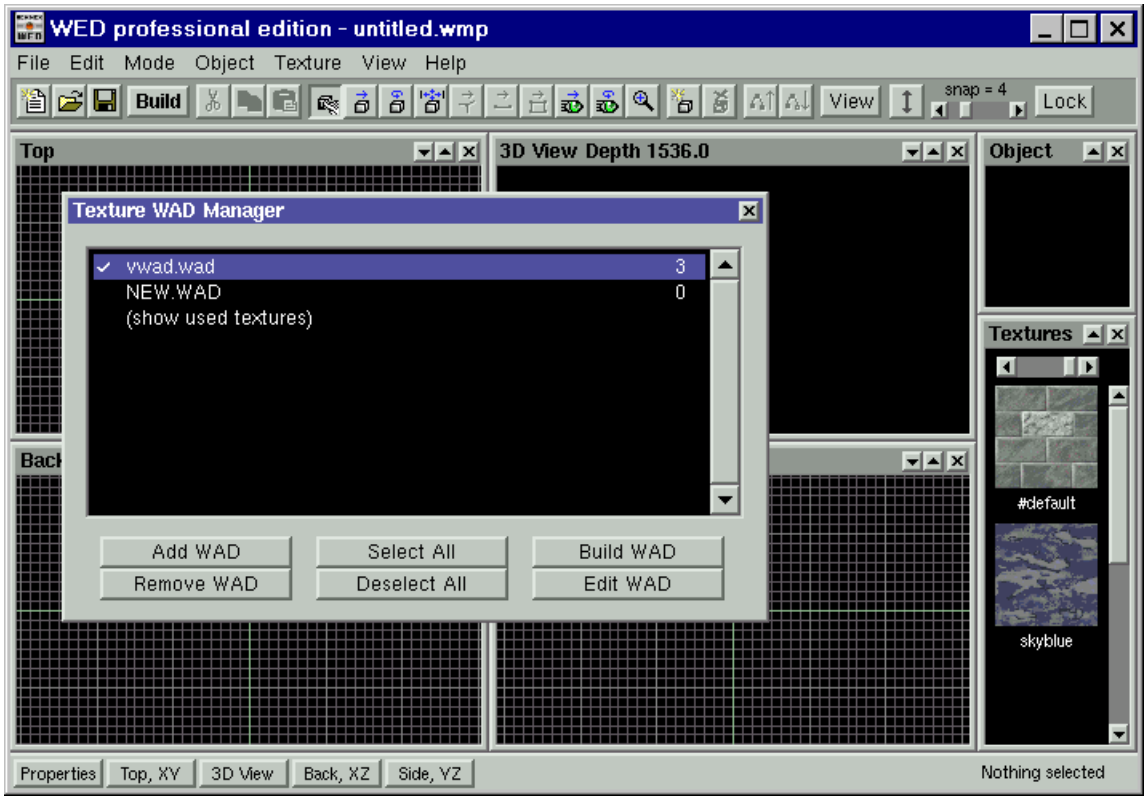


Flight Sim Übungs-Ordner

Erstellen des Levels
Unsere Levelumgebung (Map) wird wieder sehr einfach: eine grosse, flache Ebene mit einem Himmel.

Öffnen Sie WED und wählen Sie "File->New".

Öffnen Sie den Texturmanager ("Texture->Texture Manager"). Gehen Sie auf 'Add WAD', suchen Sie nach Ihrem "Flight Sim Workshop" – Ordner und fügen Sie die standard.wad hinzu. Verlassen Sie den Texturmanager.



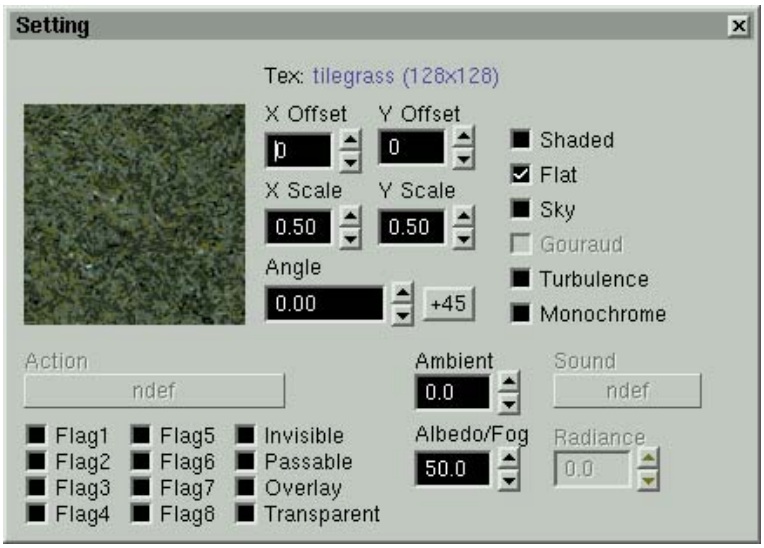
WED: Texturmanager

Wählen Sie “Add Primitive -> Cube(large)”.

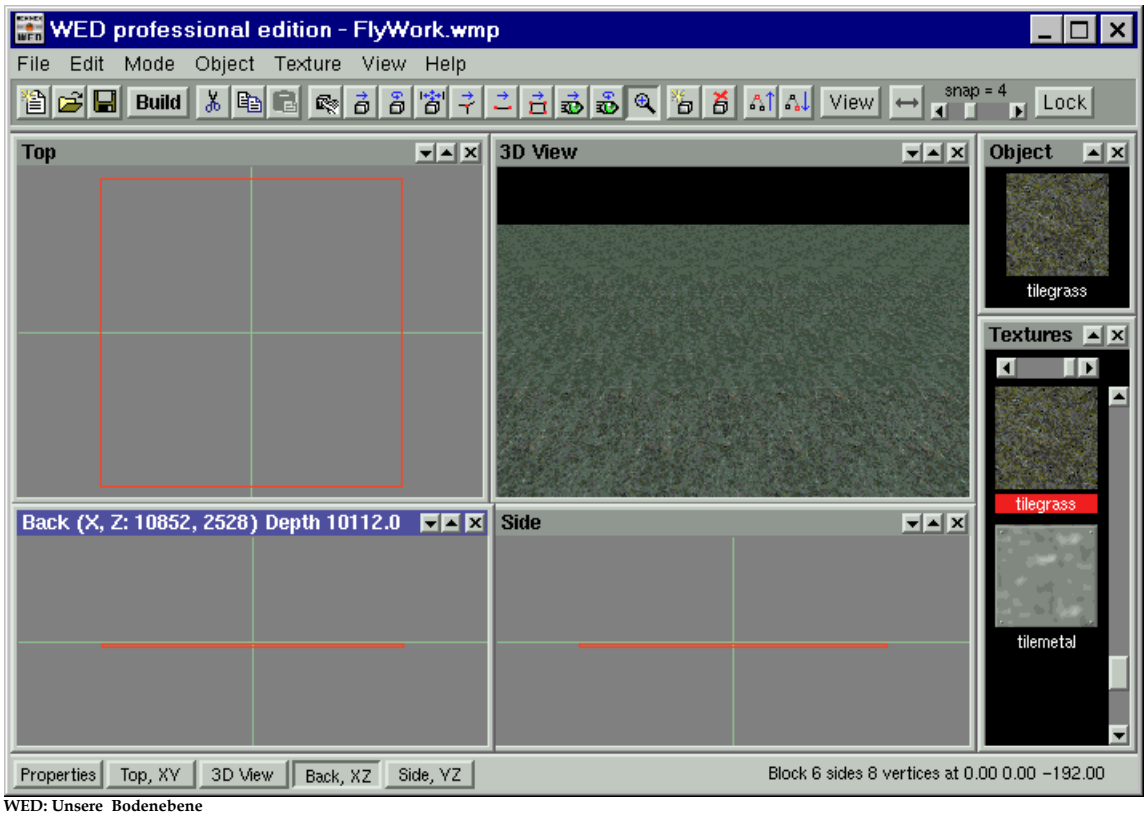
Skalieren Sie den Würfel und erstellen Sie so eine äusserst lange und breite aber dünne Oberfläche in der Grösse von etwa 40000x40000x50 Quants (hart an der Grenze der maximalen Levelgrösse von 50.000 Quants). Der einfachste Weg, dies zu erreichen ist es, die Sichttiefe (View Depth) je in der Rück- und Seitenansicht (mittels ‘+’ – Taste) bis zur gewünschten Ausdehnung zu erhöhen. Zentrieren Sie die Box und skalieren sie in der ‘Top’–Ansicht bis sie in den ‘Side’– und ‘Back’–Ansichtsfenstern gerade eben verschwindet. Skalieren Sie sie dann langsam zurück, bis sie wieder auftaucht.

Das wird Ihr Untergrund werden. Da dieser so weit ausgedehnt ist, wollen wir keine schattierte Textur dafür nehmen. Schattierte Texturen teilen die betreffende Oberfläche zur dynamischen Lichtgebung automatisch in kleine Quadrate auf und beanspruchen Video–Speicher für den Schattenwurf. Dadurch würden nur sowohl Compilierungszeit, Grösse der Leveldatei als auch die Anforderungen an den Videospeicher unnötig aufgebläht. Deshalb nehmen wir hier also eine flache Textur.

Klicken Sie rechts auf die 'tilegrass'–Textur im Textur–Balkenfenster, gehen Sie auf 'settings'. Dort machen Sie ein Häkchen bei 'flat' und setzen die Skalierung auf 0.5 herunter. Wir skalieren sie ein wenig kleiner, weil wir, um unsere Welt noch ein wenig grösser erscheinen zu lassen, auch herunterskalierte Models verwenden werden. Auf diese Weise erreichen wir, dass eine Welt mit ihren 40,000 x 40,000 Quants so wirkt, als wäre sie einige Quadratkilometer gross.

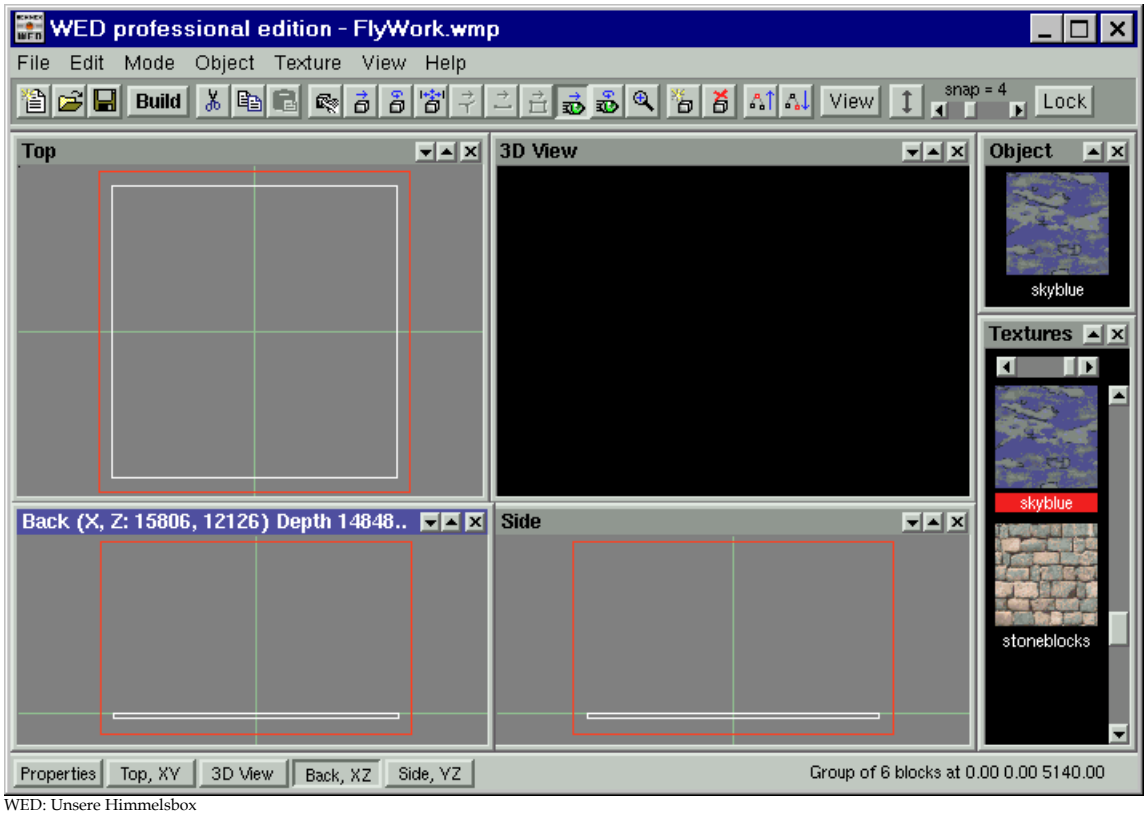


Applizieren Sie nun die herunterskalierte 'tilegrass' – Textur auf die Bodenebene und bewegen Sie diese dann so, dass sie knapp unterhalb des Nullpunkts liegt.



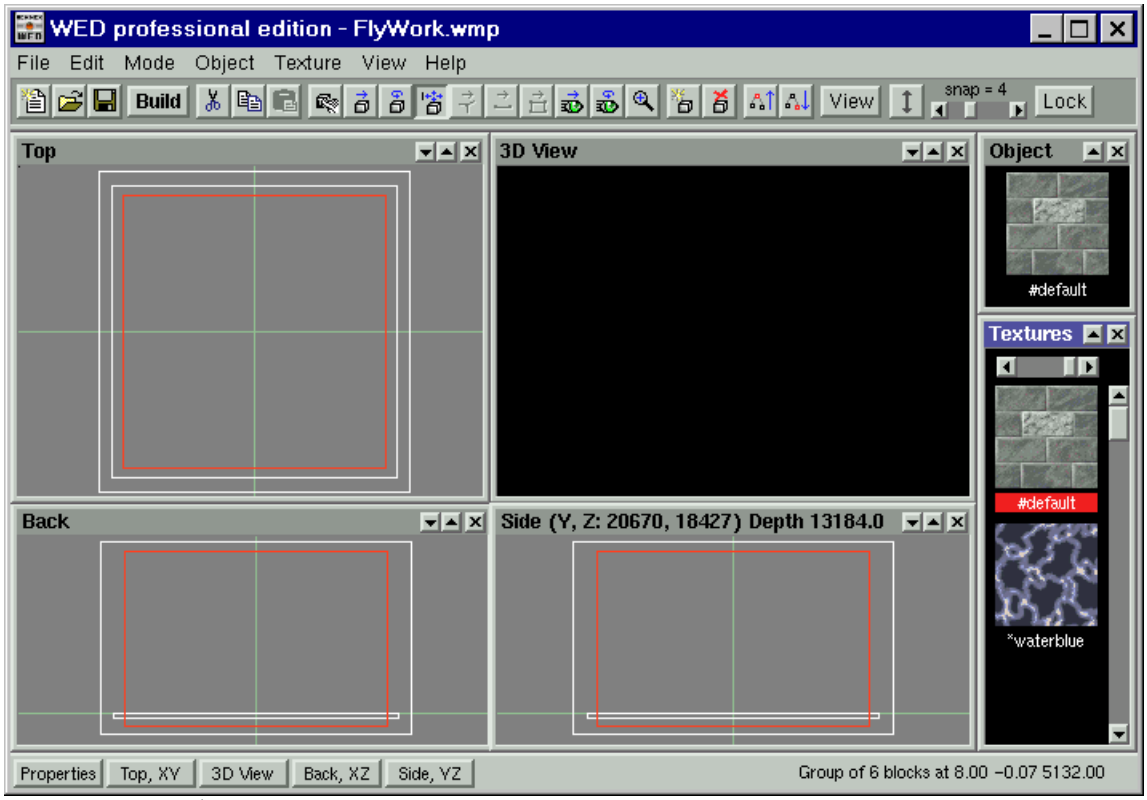
WED: Unsere Bodenebene

Fügen Sie einen weiteren Würfel hinzu und skalieren Sie ihn grösser, so dass er die Grundfläche gerade eben umgibt. Es macht nichts, wenn die Bodenebene ein wenig über die Box hinausragt, aber gehen Sie sicher, dass Sie viel Platz über dem Boden lassen. Höhlen Sie diesen Block aus (ALT + H) und geben Sie ihm die 'skyblue'–Textur. Das ist unsere Sky-Box.



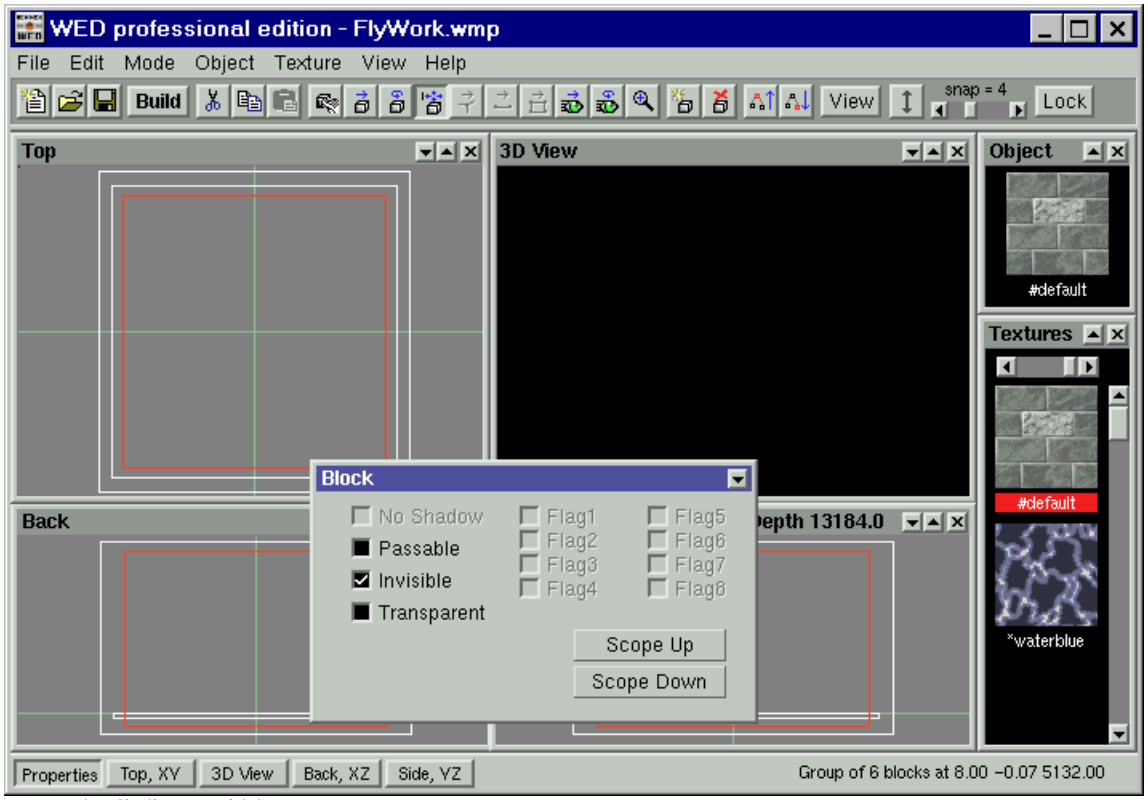
WED: Unsere Himmelsbox

Mit markierter Himmels- oder Skybox verwenden Sie den Duplizierungsbefehl (Strg + D) und erstellen einen Klon dieser Box. Diesen skalieren Sie kleiner, so dass er in die ursprüngliche Himmelsbox hineinpasst. Diese Box muss aber immer noch jede Menge Platz über der Grundebene lassen, sich unterhalb der Grundfläche ausdehnen und sie an ihren Kanten kreuzen.



WED: Innere Begrenzungsbox

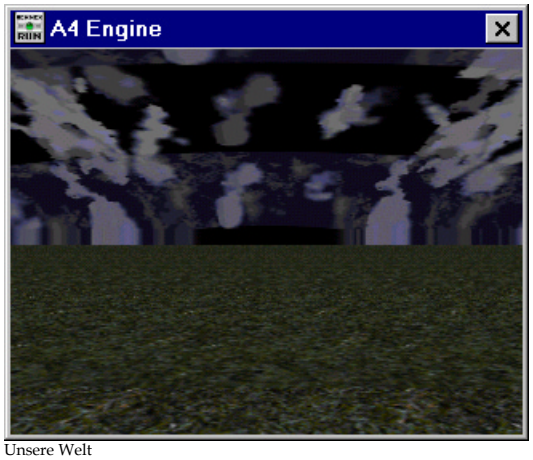
Markieren Sie bei der ‘default’ – Textur den Flag 'Flat' unter ‘Settings’ und weisen Sie diese dem Block zu. Öffnen Sie dann das Eigenschaftsfenster des Blocks. Markieren Sie den Flag ‘invisible’ (unsichtbar) und achten Sie darauf, dass kein Häkchen an einer der anderen Optionen ist. Diese Box wird unsere Aussengrenze darstellen und dafür sorgen, dass der Spieler nicht aus unserer Welt hinausfliegen kann. Später können wir Berge an die Grenzen unserer Welt setzen, im Moment tut es aber erstmal die Begrenzungsbox.



WED: Machen Sie die Box unsichtbar

Speichern Sie das Level in Ihren 'Flight Sim'-Übungsordner (nennen Sie es "FlyWork"), compilieren Sie das Level per 'BUILD' (mit Häkchen bei 'Level Map') und starten Sie es.

Sie sollten nun eine hübsche Grasfläche sehen, die sich meilenweit hinzieht.

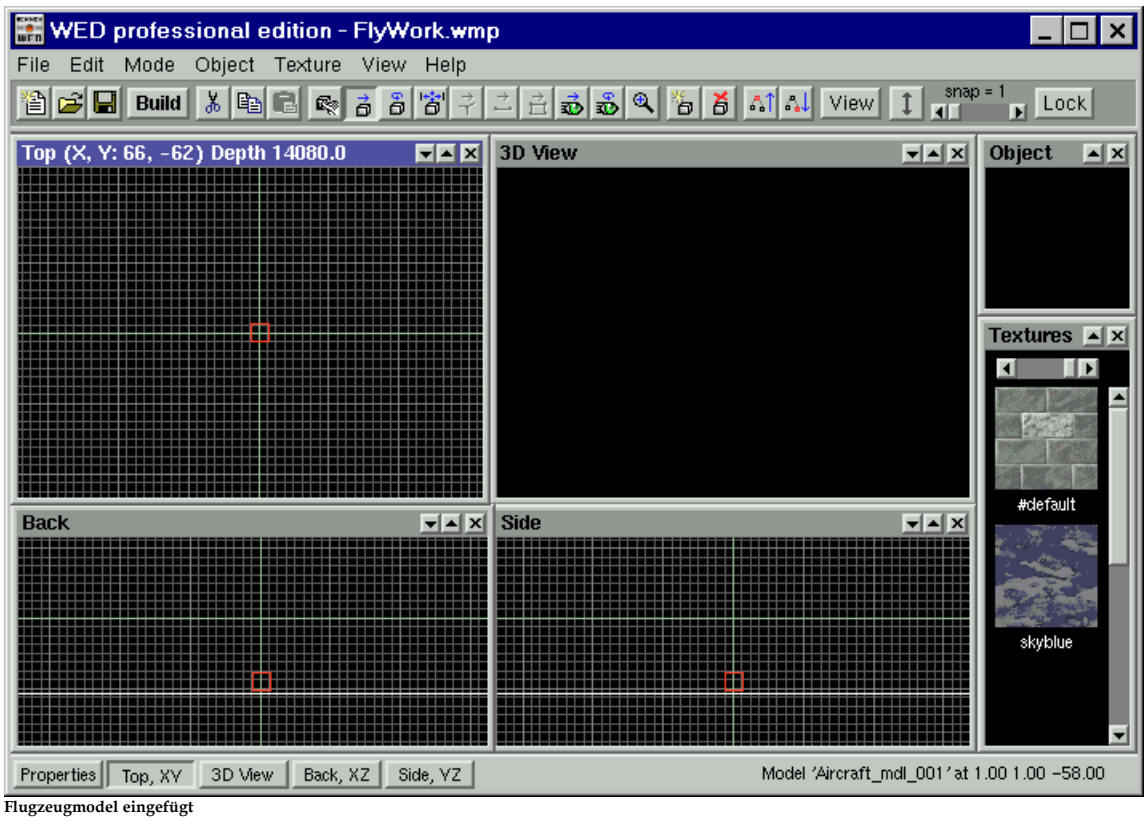


Unsere Welt

Dies ist die Minimalmap für einen Flugsimulator; einfach nur Platz, damit man loslegen kann. Um das Ganze etwas interessanter zu gestalten, bleibt es Ihnen überlassen, dem Level später andere Gebilde hinzuzufügen (Häuser, Berge etc.).

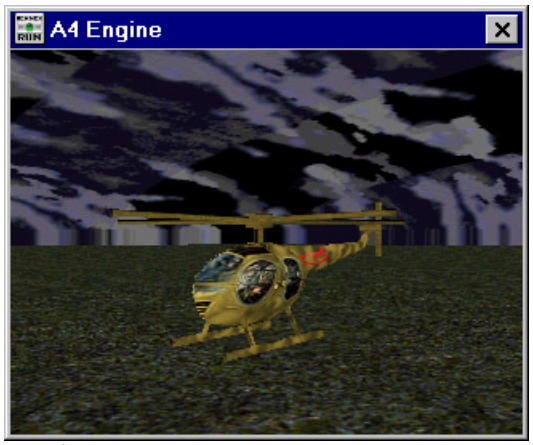
Das Flugzeug einfügen
Nun werden wir unser Flugzeugmodell einbauen. Für unseren Flugsimulator nehmen wir das Helikoptermodell (auch wenn wir zur Steuerung die Flugeigenschaften eines normalen Flugzeuges benutzen).

Wählen Sie "Object->Load Entity..." und suchen Sie mit Hilfe des Ordnersuchfesters das Model mit dem Namen "aircraft.mdl". Das Model sollte dann in Ihrer Welt auftauchen. Plazieren Sie es so in die Nähe des Zentrums Ihrer Ebene, dass es auf der Oberfläche steht.



Flugzeugmodel eingefügt

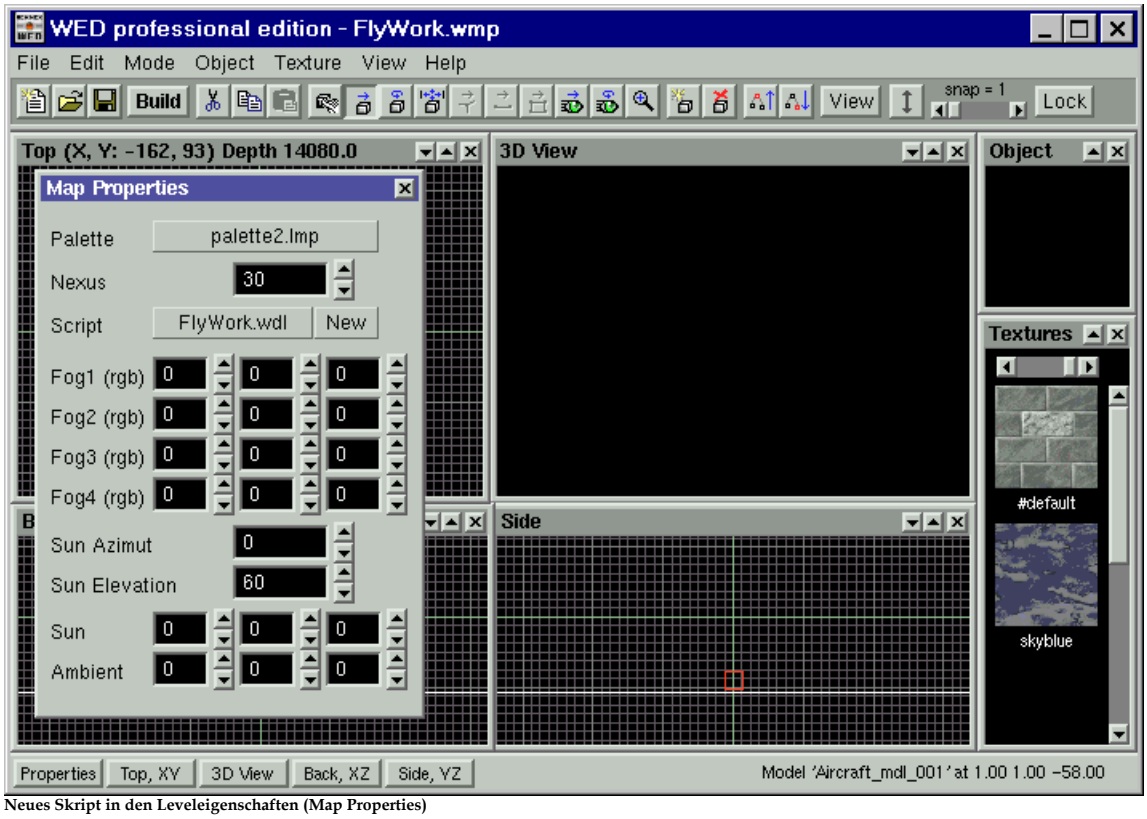
Speichern, compilieren und starten Sie das Level. Sehen Sie sich unser Flugzeug an. Sorgen Sie dafür, dass es auf oder nur leicht überhalb der Grundebene ist. Passen Sie die Plazierung so lange an, bis Sie mit dem Ergebnis zufrieden sind.



Unser Flugzeug

Erstellen Sie Ihr Skript

Nun brauchen Sie ein Skript für Ihr Level. Öffnen Sie Ihr Level–Eigenschafts–Fenster ("File->Map Properties..") und drücken Sie den 'New'–Knopf. Der Knopf neben "Script" sollte sich von "ndef" in "flywork.wdl" verändern.



Neues Skript in den Leveleigenschaften (Map Properties)

Öffnen Sie Ihren Levelordner, wählen und öffnen Sie (Doppelklick) die "flywork.wdl" – Datei. Falls Windows fragt, welche Applikation zum Öffnen der Datei verwendet werden soll, entscheiden Sie sich für "notepad" oder einen anderen reinen Texteditor.

Sie wissen ja, dass automatisch ein typisches Spiele–Skript für Sie erstellt wurde. Das ist gut und schön für die meisten Projekte, aber wir wollen hier ja nun etwas weiter gehen und etwas Komplexeres machen. Markieren Sie darum schlichtweg allen Code (in Microsoft notepad verwenden Sie dazu "Edit->Select All") und drücken Sie auf die Löschtaste. Jetzt fangen Sie bei Null an!

Pfade einfügen
Lassen Sie uns mit dem Definieren der Pfade beginnen, die unser Programm verwenden wird. Pfade werden gebraucht, um der Engine zu sagen wo sie die in unserem Projekt benuzten Dateien finden kann (Bilder, Sounds, andere Skripte, etc.). Der Ordner, in dem wir uns befinden ("Flight Sim Workshop") ist bereits (automatisch) eingefügt. In unserem Fall müssen wir also lediglich den "template"- und den "images"–Ordner einfügen.

Tippen Sie die folgende Zeile:

```
PATH    "..\\template"; // Pfad zum WDL-Template-Unterverzeichnis
```

Beachten Sie, dass alle Pfade in Relation zu unserem Levelordner stehen. Die obige Zeile sagt also das Folgende: "Gehe ins nächsthöhere (übergeordnete) Verzeichnis ("...") und gehe dort in den Ordner namens 'template' ("\\template")".

Nun wollen wir unseren "images"–Ordner auf dieselbe Weise einfügen. Schreiben Sie diese Zeile:

```
PATH    "images";       // Pfad zu unserem Bilder-Unterverzeichnis
```

Da sich der "images"–Ordner im selben Verzeichnis wie unser Level befindet, brauchen wir nur folgendes anzugeben: "schau in den Ordner mit dem Namen 'images' in diesem Ordner".

Achtung: Die Reihenfolge, in der diese beiden Zeilen erscheinen, legt fest, welcher Ordner zuerst aufgesucht wird. Falls Sie mehrere Dateien mit demselben Namen verwenden, wird in Ihrem Level nur die erste Datei, die die Engine findet, verwendet werden. Sie wollen zum Beispiel den Mauspfeil ("ARROW.PCX") dahingehend verändern, dass er weiss anstatt rot ist. Sie können hierzu Ihren neuen, weissen Pfeil kreieren und als "ARROW.PCX" in Ihren Levelordner abspeichern. Nun wird der neue Pfeil anstelle des Defaultpfeiles aus 'template' in Ihrem Level dargestellt. Wenn Sie diesen Pfeil aber im Ordner 'images' ablegen, werden Sie immer noch den alten 'template' - Pfeil im Level haben (Sie haben ja der Engine gesagt, dass sie dort zuerst nachschauen soll). Sie können die Reihenfolge der Abfrage verändern, indem Sie die Reihenfolge in der die PATH – Zeilen erscheinen verändern. Ändern Sie beispielsweise die beiden Zeilen, die sie eingegeben haben folgendermassen:

```
PATH    "IMAGES";               // Pfad zu unserem Bilder-Unterverzeichnis
PATH    "..\\template"; // Pfad zum WDL-Schablonen-Unterverzeichnis
```

Nun wird die Engine zuerst in Ihren Levelordner, dann in den "images" – Ordner und zuletzt in den 'template' – Ordner schauen.

Includes (Einfügungsanweisungen) hinzufügen
Nachdem wir unsere Pfade eingerichtet haben, sollten wir unsere Standard-Skriptdateien hinzunehmen. Schreiben Sie also das Folgende unter die Pfade:

```
// Einfügungsdateien
INCLUDE <movement.wdl>;
INCLUDE <messages.wdl>;
INCLUDE <particle.wdl>;
INCLUDE <doors.wdl>;
INCLUDE <actors.wdl>;
INCLUDE <weapons.wdl>;
INCLUDE <war.wdl>;
INCLUDE <menu.wdl>;
```

Der 'INCLUDE' – Befehl weist die Engine an, diese Zeile durch den Inhalt der Datei innerhalb der eckigen Klammern '<>' zu ersetzen. Es ist dasselbe, als würden Sie die betreffende Datei öffnen, sämtlichen Code darin kopieren und an dieser Stelle in Ihr Skript einfügen.

Das ist ein sehr wirkungsvolles Werkzeug, denn es erlaubt uns die Wiederverwendung von Code aus anderen Projekten. Auch können wir den Vorteil aus Updates innerhalb der eingefügten Dateien ziehen, ohne den Code umschreiben zu müssen. Die Update – Version 4.19 zum Beispiel beinhaltet Code, der dem Spieler das Schwimmen in Wasser ermöglicht. Also wird nun jedes Projekt, das "movement.wdl" eingefügt hat, diese neuen Schwimmcode automatisch verwenden.

Genau wie bei Pfaden, ist auch die Reihenfolge der INCLUDE – Zeilen wichtig, denn manche Skripte verwenden Werte, die in anderen Skripten festgesetzt werden. In der "actors.wdl" wird zum Beispiel die Variable 'force' verwendet, welche in der "movement.wdl" festgelegt wird. Wenn wir also "actors.wdl" vor "movement.wdl" setzen, werden wir Fehlermeldungen kriegen.

Es ist in Ordnung, Skripte auch dann per INCLUDE einzufügen, wenn man keine ihrer Features verwendet. Die meisten Dateien im 'template'–Ordner sind miteinander verflochten. Daher sollten Sie, auch wenn Sie eigentlich nur eine wirklich anwenden wollen, sicherheitshalber alle Dateien einfügen. Die Ausnahme zu dieser Regel ist die "Venture.wdl"–Skriptdatei. Diese wird von keiner anderen Skriptdatei benutzt, verwendet ihrerseits aber viel von den anderen.

Startwerte für die Engine
Nun werden wir einige wichtige Werte setzen, die eine Rolle dabei spielen, wie der Simulator auf dem Bildschirm dargestellt werden wird. Diese Werte betreffen Auflösung, Farbtiefe, Framerate (Geschwindigkeit des Bildaufbaus) und Beleuchtung. Fügen Sie die folgenden Zeilen nach den 'INCLUDE' – Zeilen ein:

```
// Startwerte für die Engine
```

```
IFDEF LORES;
var video_mode = 4;     // 320x240
IFELSE;
var video_mode = 6;     // 640x480
ENDIF;
var video_depth = 16;   // D3D, 16 bit Auflösung
var fps_max = 40;       // 40 fps max (frames/bilder per sekunde)
var floor_range = 200;  // verhindert plötzliche Helligkeitsschwankungen  im
                                // Flugzeug
```

Falls Sie mit früheren Versionen von 3D GameStudio bereits WDL – Skripte verfasst haben, wird Ihnen nun auffallen, dass wir 'var' anstelle von 'SKILL' verwenden. Das machen wir, weil 'var' ein wenig kürzer ist und ausserdem ist es kompatibel zu Javaskript. Das bedeutet nun, dass die alte Form "`SKILL VIDEO_MODE { VAL 6; }`" durch den eleganteren Java – Stil ersetzt wird: "`var video_mode = 6;`".

'fps_max' und 'floor_range' sind neu in 4.19. Sie können im WDL – Handbuch darüber nachlesen. Kurz gefasst limitiert 'fps_max' die Framerate, der definierte Wert kann also nicht überschritten werden. Indem 'fps_max' auf 40 gesetzt wird, begrenzen wir uns selbst auf 40 Bilder pro Sekunde, auch (oder gerade) dann, wenn das System, das wir verwenden, mehr anzeigen kann.

'floor_range' wurde mit dem Gedanken an Flugsimulatoren im Hinterkopf entwickelt. Dieser Wert begrenzt die Reichweite, innerhalb welcher das Licht einer Entity von Helligkeit und Schatten der Bodenfläche darunter beeinflusst wird. Dies bedeutet, dass sich die Beleuchtung unseres Flugzeugs, wenn es z.B. hoch über eine Stadt hinweg fliegt und einen Schatten überquert, nicht verändert.

Beachten Sie, dass es sich bei 'video_mode' und 'video_depth' um "read only", also "nur – lese" – Werte handelt. Das hat zur Folge, dass sie, wenn das Programm erst einmal läuft, nicht direkt gesetzt werden können. Sie können nur durch Aufrufen des "SWITCH_VIDEO' – Befehls verändert werden.

Die Main-Funktion

Bei jedem Projekt brauchen Sie eine 'Haupt' (main)–Funktion. Dies ist die erste Funktion, die bei Programmstart automatisch gestartet wird. In den meisten Fällen ist die Hauptfunktion denkbar einfach und unsere macht da keine Ausnahme. Schreiben Sie bitte das Folgende unter Ihre letzte Zeile:

```
// Desc: unsere HAUPT(main)-Function, Aufruf bei Spielstart
function  main()
{
IFNDEF NOTEX; //schalte bei schwachen Karten d3d_texreserved durch -d notex ab
        D3D_TEXRESERVED = min(12000,D3D_TEXMEMORY/2);
ENDIF;

// Priorität: den Himmel vor dem Laden des Levels setzen
        init_environment();
// 'Umgebung' (Himel, Wolken, etc.) einsetzen
        LOAD_LEVEL <flywork.WMB>;
        load_status();  // zurücksetzen der globalen Skills
```

```
}
```

Jede dieser Zeilen werden wir der Reihe nach besprechen.

Funktionen oder Aktionen?
Sie werden bemerkt habe, dass wir statt "`ACTION main {..}`" "`function main() {..}`" geschrieben haben. Funktionen sind etwas Neues in 4.19. Sie verhalten sich fast identisch zu Aktionen, allerdings erscheinen sie nicht als anhängbare Entity – Aktionen in der WED – Pop–Up – Liste. Wenn Sie also eine Funktion erstellen, die direkt an eine Entity angehängt werden soll (z.B. my_player), benutzen Sie das Schlüsselwort "action", falls nicht, nehmen Sie "function".

D3D_TEXRESERVED (und und die min(x,y) - Funktion)
Zuallererst müssen wir in der Hauptfunktion die Videokarte des Anwenders untersuchen und versuchen Speicherplatz zu reservieren, damit wir unsere Texturen vorab laden und bereitstellen können. Dadurch werden "Rucke", die beim Erstladen einer Textur während des laufenden Spieles auftreten können, vermieden.

Ich sagte 'versuchen', da es nicht alle Videokarten erlauben, Speicherplatz zu reservieren. Um mit diesen Karten klar zu kommen, haben wir den Code innerhalb eines "IFNDEF NOTEX; ... ENDIF;" – Absatzes geschrieben. Falls der Anwender D3D_TEXRESERVED nicht benutzen kann (also eine "schwache Karte" hat, wie z.B. ältere Voodoo-Versionen), muss als Befehlszeilenoption "-d notex" angegeben werden, ehe das Programm gestartet wird. So weiss die Engine, dass sie diesen Abschnitt im Skriptcode überspringen soll.

Wir wollen aber nicht unseren gesamten Video - Speicherplatz zum Vorabladen von Texturen verbrauchen. Wenn wir dies täten, hätte die Engine überhaupt keinen Platz mehr, um andere Texturen einzulesen, während das Level läuft (das würde zu eine Fehlermeldung führen). A4 reserviert automatisch 2 MB Speicher für diese Texturen, aber, um auf der sicheren Seite zu bleiben, sollten wir nicht mehr, als die Hälfte des verfügbaren Speicherplatzes belegen. Die Menge des zur Verfügung stehenden Speicherplatzes variiert von System zu System, kann jedoch jederzeit durch Überprüfung des Wertes in D3D_TEXMEMORY festgestellt werden.

Um also 12 MB auf Systemen mit 24 MB oder mehr Videospeicher zu reservieren oder die Hälfte des verfügbaren Speichers auf Karten mit weniger als 24 MB zu bekommen, sollten wir die Hälfte des Videospeichers nehmen und mit 12 MB vergleichen. Was weniger ist, nehmen wir dann.

Wir könnten es etwa so machen:

```
if((D3D_TEXMEMORY/2) < 12000)
{
        D3D_TEXRESERVED = D3D_TEXMEMORY/2;
}
ELSE
{
```

```
        D3D_TEXRESERVED = 12000;
}
```

Oder wir machen es eleganter mit einer einzigen Zeile, indem wir "min(x,y)" verwenden. Das nämlich nimmt zwei Werte und gibt den kleineren zurück:

```
D3D_TEXRESERVED = min(12000,D3D_TEXMEMORY/2);
```

init_environment() (die neue Art des Aufrufs von Funktionen)
Unser erster Aufruf einer Funktion zeigt die 'neue Art' eine Funtion zu initialisieren. Die alte Methode "CALL init_environment;" wurde durch die neue "init_environment();" ersetzt. Beide Methoden arbeiten auf dieselbe Weise, aber die neue Variante ermöglicht es uns, eigene Funktionen und Aktionen auf dieselbe Art zu verwenden, wie wir auch vordefinierte Funktionen (wie 'min' und 'max') benutzen.

Lassen Sie uns die 'init_environment'–Funktion (unterhalb der Hauptfunktion) eingeben:

```
// unsere Umgebungsbilder
BMAP my_sky = <sky1.pcx>,0,0,128,128;
BMAP my_clouds = <sky1.pcx>,128,0,128,128;
BMAP my_mountains = <mount16.pcx>;

// Desc: inutialisiere den Himmel, Wolken und Hintergrundbilder
function init_environment()
{
        sky_map = my_sky;
        cloud_map = my_clouds;
        scene_map = my_mountains;

        scene_field = 180;
        scene_angle.tilt = -10;

        SKY_SPEED.X = 0;
        SKY_SPEED.Y = .025;
        CLOUD_SPEED.X = 3;
        CLOUD_SPEED.Y = 4.5;
        SKY_SCALE = 1.0;
        SKY_CURVE = 1;
}
```

Die ersten drei Zeilen definieren unsere drei Himmels-Bitmaps. Die beiden ersten Grafiken ('my_sky' und 'my_clouds') werden aus der rechten und der linken Hälfte unserer Himmelstextur ("sky1.pcx")geladen. Dagegen wird 'my_mountains' aus der Bitmap mit dem Bergbild gelesen. Diese Bitmaps werden dann in die drei Bitmap-Synonyme geladen, die unseren Himmel darstellen ('sky_map', 'cloud_map', and 'scene_map').

Mehr Informationen über diese Bitmaps und weitere Werte, die hier gesetzt werden, können im WDL - Handbuch nachgelesen werden (Kapitel 7: Engine – Variable).

Speichern Sie Ihre Arbeit und starten Sie das Level. Sie sollten unseren gerenderten Himmel mit darüber hinwegziehenden Wolken sehen, und im Hintergrund erkennen Sie ein Gebirge. Sie werden ausserdem feststellen, dass Sie, wenn Sie bis zum Rand unserer Welt gehen, unter dem Fuss dieser Berge hindurchsehen können. Sie können dies um einige Grad beheben, indem Sie die 'scene_map' grösser machen und herunterziehen (durch Anpassen des 'scene_angle.tilt' – Wertes). Auch können Sie die Kanten des Levels verstecken, indem Sie Gebäude, Bäume oder Berge entlang seiner Ränder aufstellen.

Gestaltung unseres Flugzeugs

Was aus unserem Projekt einen Flugsimulator und eben kein Autorennspiel, eine Panzersimulation oder sonst eine Art von Spiel macht, ist die Eigenschaft unseres Flugzeuges, fliegen zu können. Darum untersuchen wir jetzt, wie man eine einfache Flugzeug–Action erstellt, die an unser Flugzeug–Modell angehängt werden kann, um ihm vom Boden in die Luft zu verhelfen.

Grundmodell der Flugfähigkeit

Was Flugsimulatoren (neben Grafiken und der Auswahl von Flugzeugen zur Fortbewegung) von allen anderen unterscheidet, ist die Funktionsweise des Flugverhaltens. Manche dieser Flugmodelle sind sehr komplex: Sie können reale Flugdaten verwenden und setzen komplizierte mathematische Berechnungen um. Damit berechnen sie aus Luftdruck, Strömungsgeschwindigkeit und Flügelprofil exakt, was geschehen würde, wenn Sie z.B. Ihren Klappenwinkel um 3 Grad erhöhten.

Andere Flugsimulatoren gehen das anders an: Sie setzen einige einfache Regeln und Werte, mit deren Hilfe sie Resultate erzielen, die dem physikalischen Verhalten eines Flugzeuges sehr nahe kommen. Neben der Tatsache, dass diese einfacher zu schreiben sind, können diese 'einfachen' Simulatoren auch eine Menge Spass machen (besonders, wenn Sie sich nicht erst hinsetzen und ein dickes Anleitungsbuch lesen wollen, ehe Sie mit einem fliegen).

In unserer Übungsreihe gehen wir den 'einfachen' Weg. Das Flugmodell, das hier vorgestellt wird, berücksichtigt die vier Grundkräfte, die auf das Flugzeug wirken (Auftrieb, Schub, Schwerkraft und Luftwiderstand), erstellt ein paar Regeln und Beschränkungen und berechnet, wie sich das Flugzeug verhalten wird.

Einrichten der Variablen

Lassen Sie uns mit dem Setzen der Werte beginnen, die von unserem Flugprogramm benutzt werden. Fügen Sie diese Zeilen ans Ende Ihrer Skriptdatei.

```
// Flugzeug DEFINEs und Werte
DEFINE _MODE_PLANE,16;
DEFINE _RPM,SKILL31;            // Motor Geschwindigkeit

// Flugzeugwerte
var stallspeed = 8;     // Flugzeug sackt unterhalb dieser Geschwindigkeit ab
var climbrate = 1.5;    // maximale Steigrate
var climbfactor = 0.1;  // Flügelprofil
```

```
var height_max = 750;   // maximale Höhe
var speed_max = 25;     // maximale Fluggeschwindigkeit
var max_rpm = 7;        // maximale Motorengeschwindigkeit

// Sound - Werte
var enghandle = 0;
SOUND engsound = <jet.wav>;     //  Motorengeräusch

// Animationa - String
string anim_fly_str = "fly";

define _FLYSCALE 0.3;   // herunterskalieren des Flugzeugs
```

Das erste DEFINE setzt einen der beiden Bewegungsmodi (MOVEMODE), die wir in unserem Flugprogramm kontrollieren werden (der zweite ist _MODE_DRIVING, der in der "movement.wdl" definiert ist). Das zweite DEFINE besagt, dass wir SKILL31 als RPM–Wert (Motordrehzahl) unseres Musters verwenden.

Die Flugzeugwerte sind konstante Werte, die zur Berechnung dienen, wie verschiedene Kräfte auf das Flugzeug einwirken werden (und die Grenzen zu denen diese Kräfte führen können). Wir werden uns mit diesen Werten so, wie sie im Code erscheinen, beschäftigen. Wenn Sie diesen Übungskurs absolviert haben, sollten Sie sich nicht scheuen, diese Werte neu abzustimmen, um zu sehen, wie sie das Flugverhalten verändern.

Die nächsten Werte regeln das Motorengeräusch ('enghandle' und 'engsound'), den Namen der Propelleranimation unseres Flugzeugmodells ('anim_fly_str') und die Skalierung, die unser Flugzeug mit dem Ziel, die Welt grösser erscheinen zu lassen, bekommen wird.

Schreiben der Aktion
Da diese Funktion innerhalb des Levels an das Flugzeug – Model angehängt werden wird, schreiben wir sie als Action. Es ist eine grosse Aktion, die in diverse Abschnitte unterteilt ist.

Initialisieren der Variablen
Das erste, was wir tun, ist unsere Entity zu initialisieren:

```
// Desc: unsere Flugaktion
ACTION player_aircraft
{
        // Init values
        if (MY.client == 0) { player = MY; } // auf dem Server generiert?
        MY._TYPE = _TYPE_PLAYER;
        MY.ENABLE_SCAN = on;    // damit Feinde mich aufspüren können

        IF(MY._FORCE == 0) {  MY._FORCE = 1.5; }
        IF(MY._MOVEMODE == 0) { MY._MOVEMODE = _MODE_PLANE; }
// herunterskalieren des Flugmodells, um die Welt zu vergrössern
        MY.scale_x *= _FLYSCALE;
        MY.scale_y *= _FLYSCALE;
        MY.scale_z *= _FLYSCALE;
```

```
drop_shadow(); // _movemode muss vorher gesetzt sein
```

Bis hierher ist die Aktion der 'player_move' – Aktion in der "movement.wdl" sehr ähnlich. Wir überprüfen ob wir auf einem Server sind, setzen unseren Spielertypus, machen die Entity empfänglich für das Geschehen ihrer Umwelt, setzen ihre Werte zur Einflussnahme (falls diese nicht bereits im Model selbst definiert sind) und setzen ihren Bewegungsmodus ('MOVEMODE') auf den des Flugzeuges.

Wenn nun all diese Werte gesetzt sind, rufen wir 'drop_shadow' auf damit unser Flugzeug auch einen Schatten auf den Boden wirft.

Die WHILE – Schleife des Flugprogramms
Nun geben wir den Kern unserer Aktion ein:

```
// solange wir im gültigen Bewegungsmodus (MOVEMODE) sind
WHILE(MY._MOVEMODE == _MODE_PLANE || MY._MOVEMODE == _MODE_DRIVING)
{
        _player_force();  // setze Kraft-und Gegenkraftwerte vom Spieler input
        scan_floor();     // setze floor_normal, my_height, und floor_speed
```

Wir bleiben in dieser Schleife, bis die Entity nicht länger fliegt oder fährt (in unserem Fall sollten wir immer in einer dieser beiden Zustände sein).

Als erstes lesen wir die Eingabewerte (Input) des Spieler und der Umgebung. Mit einem einfachen Aufruf von '_player_force' bekommen wir die gesamte Bewegungsinformation von Tastatur, Maus und Joystick des Spielers und legen diese Ergebnisse in den Variablen 'force' und 'aforce' ab. 'scan_floor' teilt uns mit, wie dicht wir uns über der nächsten Oberfläche befinden (sowie Neigung und eventuelle Strömungsgeschwindigkeit dieser Oberfläche).

Maschine (Schub)
Als nächstes berechnen wir die Antreibsgeschwindigkeit:

```
// Berechnung der Schubkraft des Motors (beschleunige die
// Maschine mittels [HOME], [END])
// MY._RPM = rpm / 1000
        MY._RPM += 0.0025*force.Z*TIME;
// begrenze _RPM zwischen 0 and 7000 rpm max
        MY._RPM = max(0,min(max_rpm,MY._RPM));
```

Indem er die [Pos1]– bzw. [Ende]–Taste drückt, setzt der Spieler die Antriebsgeschwindigkeit. So wird die Spielerkontrolle über den Gashebel des Flugzeugs simuliert. 'player_force' wandelt den Tastendruck auf [Pos1] bzw. [Ende] in einen positiven/negativen 'force.Z' – Wert um. Durch Multiplizieren des 'force.Z' – Wertes mit der Zeit des letzten Frames (TIME) und eines konstanten Wertes (der zur Abstimmung, wie rasch wir unseren Schub verändern, gebraucht wird) können wir die Änderung der Motordrehzahl berechnen.

Wir wollen die Geschwindigkeit unserer Maschine begrenzen, so dass sie einerseits 'max_rpm' nicht überschreitet und andererseits positiv (über Null) bleibt. Das können wir mit einigen 'IF' – Anweisungen erreichen, aber schneller geht es, wenn wir zusammengesetzte min/max–Funktionen verwenden. Die Anweisung "max(0, min(max_rpm, MY.RPM))" liesst sich folgendermassen: "nimm den Minimalwert von max_rpm und MY.RPM, nimm nun den Maximalwert dieses Wertes und Null".

Motorengeräusch
Nun wollen wir uns dem Motorengeräusch widmen. Wir wünschen einen gleichmässigen Hintergrundsound, solange der Motor läuft. Ausserdem wäre es schön, das Geräusch zu verändern, um damit dem Spieler auch akustisch einen Eindruck zu vermitteln, wie schnell seine Maschine fliegt.

```
// starte, stoppe oder stimme das Motorengeräusch ab
if(MY == player)
{
        // wenn der Motor läuft...
        if(MY._RPM > 0)
        {
                if(enghandle == 0)  // kein Geräusch
                {
                        // starte Motorengeräusch
                        PLAY_LOOP engsound,25;
                        enghandle = RESULT;
                }
                // stimme den Sound je nach speed_ahead ab
                temp = (MY._RPM + MY._SPEED_X * 0.2) * 60;
                TUNE_SOUND enghandle,25,temp;
        }
        else  // Motor ist abgestellt...
        {
                if(enghandle != 0)      // Sound wird abgespielt
                {
                        // stoppe Motorengeräusch
                        STOP_SOUND enghandle;
                        enghandle = 0;
                }
        }
} // END falls (MY == player)
```

Zunächst einmal müssen wir sichergehen, dass diese Aktion auch durch den Spieler aufgerufen wird (wenn wir die Aktion einer anderen Entity zugewiesen haben, wollen wir nicht, dass sie sich auf unseren Sound auswirkt). Stimmt die Zuweisung, überprüfen wir als nächstes, ob der Motor läuft (MY._RPM>0). Läuft der Motor, kontrollieren wir, ob das Motorengeräusch spielt, indem wir prüfen, ob ein Wert in 'enghandle' steht. Falls wir gegenwärtig kein Motorengeräusch abspielen, ist der Wert von 'enghandle' auf Null gesetzt.

Um das Motorengeräusch zu starten und gleichmässig im Hintergrund laufen zu lassen, können Sie die PLAY_LOOP–Anweisung verwenden. So wird der Sound im Hintergrund in einer Schleife ständig wiederholt, bis wir die STOP_SOUND – Anweisung aufrufen. PLAY_LOOP liefert einen Wert in der RESULT–Variablen zurück.

Dieser Wert ist die Kontrollnummer fürs Motorgeräusch. Es ist wichtig, dass wir dieses Nummer speichern, denn sie bietet uns die einzige Möglichkeit, diesen Sound, wenn er einmal läuft, zu verändern oder zu stoppen. Darum werden wir die Kontrollnummer in der Variablen 'enghandle' speichern.

Das nächste, was wir tun, solange die Motoren laufen, ist das Abstimmen der Tonhöhe des Motorengeräusches in Abhängingkeit von der Drehzahl und Geschwindigkeit. Das erreichen wir, indem wir die Formel (MY._RPM + MY._SPEED_X * 0.2) * 60 zur Berechnung des Prozentsatzes der Normalfrequenz verwenden, die wir dann wiedergeben. Eine höhere Frequenz wirkt, als liefe der Motor schneller, eine tiefere Frequenz hört sich langsamer an.

Berechnen des Auftriebs
Nun werden wir berechnen, wie die gegenwärtige Geschwindigkeit und der Winkel des Flugzeugs den Auftrieb unseres Fluggerätes beinflussen.

```
// Berechnung der Auftriebskraft, abhängig von Geschwindigkeit,
// Schräglage und seitlichem Rollwinkel
force.z = (climbfactor*MY._SPEED_X)
        * (0.05 * (90 + ang(MY.tilt)- (0.5*abs(ang(MY.roll))) ));
```

In dieser Berechnung ermitteln wir den Auftrieb (force.z), indem wir drei Faktoren kombinieren.

Der Auftrieb eines Flugzeuges entsteht durch Luft, die rasch über seine Flügel strömt. Je schneller sich ein Flugzeug durch die Luft bewegt, umso mehr Auftrieb entsteht. Unser "Grundauftrieb" wird so berechnet: ("climbfactor*MY._SPEED_X"). Dabei ist 'MY._SPEED_X' die Vorwärtsgeschwindigkeit des Flugzeugs und 'climbfactor' ist ein konstanter Wert, der die Form des Flügels und seine Fähigkeit Auftrieb zu erzeugen darstellt (der Effekt, der den Flügel zur Tragfläche macht).

Der Auftriebswert wird dann durch Schräglage und seitliche Rollbewegung des Flugzeuges modifiziert. Zeigt die Nase des Flugzeuges nach oben, wird es steigen. Wenn die Nase nach unten zeigt, wird das Flugzeug zur Erde hin absacken. Bekommt das Flugzeug Seitenlage, wird der Auftrieb, den die Tragflächen produzieren, reduziert. Die Kombination aus diesen beiden Winkeln wird berechnet, indem man 1/20 der Schräglage nimmt und die Hälfte des absoluten Wertes der Rollbewegung davon abzieht. Diese Werte sind weit von wirklichen physikalischen Daten entfernt, aber sie produzieren in den meisten Situationen vernünftige Werte.

Um in der Luft zu bleiben, müssen Flugzeuge eine bestimmte Mindestgeschwindigkeit beibehalten. Gerät ein Flugzeug unter diese Mindestgeschwindigkeit, geschieht das, was Piloten einen 'stall' (Sackflug) nennen. Wenn ein Flugzeug absackt, produzieren seine Tragflächen keinerlei Auftrieb mehr. Das können wir simulieren, indem wir diese einfache Regel einfügen:

```
// wenn unsere Geschwindigkeit unter der kritischen Absackgrenze liegt...
if (MY._SPEED_X < stallspeed)
{
```

```
        // stall (no lift)
        force.z = 0;
}
```

Die Kraft, die der Auftrieb zu überwinden sucht, ist die Schwerkraft. Als nächstes reduzieren wir den Auftriebseffekt durch die Gravitationskraft:

```
// Flugzeug wird durch Schwerkraft beeinflusst
force.z -= gravity;
```

Um sicher zu gehen, dass wir nicht plötzlich in die Luft hochschiessen, begrenzen wir den Wert, um den wir steigen können, auf unsere maximale Steigrate ('climbrate'):

```
// limitiere die Auftriebskraft
force.z = min(force.z,climbrate);
```

Bewegung auf dem Boden

Flugzeuge verbringen einen Grossteil ihrer Zeit auf dem Boden. Mit den folgenden Schritten entwickeln wir die Fortbewegung des Flugzeuges, solange wir nicht in der Luft sind.

Als erstes testen wir, ob wir auf dem Boden sind (my_height < 5). Wenn dem so ist, setzen wir unseren '_MOVEMODE' auf '_MODE_DRIVING':

```
// falls am Boden, fahre herum
if (my_height < 5)
{
        MY._MOVEMODE = _MODE_DRIVING;
```

Dann berechnen wir die Vorwärtskraft (force.x) indem wir die Hälfte der Maschinengeschwindigkeit minus 1 nehmen (wir benutzen den Maximalwert, um sicher zu gehen, dass wir niemals rückwärts fahren):

```
// die Vorwärtskraft hängt von der Motorgeschwindigkeit ab
force.X = max(0,0.5*(MY._RPM - 1));
```

Da das Lenken auf dem Boden von beidem, dem Stick (oder linker/rechter Pfeiltaste) und den Pedalen gesteuert wird, überprüfen wir, ob der Spieler den Joystick zum Steuern nimmt. Tut er das, wird der aforce.pan – Wert bereits gesetzt sein, falls nicht, verwenden wir irgendeine Kraft, die von den Pedalen kommt (die ist in force.Y gespeichert):

```
        // lenken mit Joystick und Pedalen [<] [>]
        if (aforce.pan == 0)
        {
                aforce.pan = force.Y;
        }
```

Um das Flugzeug auf dem Boden in der Waagerechten zu halten, wird der Rollwinkel durch eine entgegengesetzte der Roll–Kraft (aforce.roll) reduziert:

```
// wenn der Rollwinkel nicht Null war,
// füge eine Seitwärtsrollkraft an, um den Winkel zurückzusetzen
aforce.roll = -0.2*ang(MY.roll);
```

Nun benutzen wir die Winkel – Kraftwerte (aforce), um die die Winkel betreffende Geschwindigkeit des Flugzeuges zu setzen:

```
// erhöhe nun die Winkel-Geschwindigkeit und setze die Winkel
friction = min(1,TIME*ang_fric);
MY._ASPEED_PAN += (TIME * 0.3 * aforce.pan)
                        - (friction * MY._ASPEED_PAN);
MY._ASPEED_ROLL+= (TIME * aforce.roll)
                        - (friction * MY._ASPEED_ROLL);
```

Und dann verwenden wir die mit einem Bruchteil der Vorwärtsgeschwindigkeit (MY._SPEED_X) multiplizierte Winkelgeschwindigkeit des Flugzeuges, um den seitlichen Schwenkwinkel, und die Schräglage des Flugzeuges zu berechnen während es über den Boden rollt:

```
MY.pan += TIME * MY._ASPEED_PAN * MY._SPEED_X * 0.025;
MY.roll+= TIME * MY._ASPEED_ROLL;
MY.tilt = 0;
```

Wir verwenden dieselbe Art der Berechnung, um die Vorwärtsgeschwindigkeit des Flugzeuges zu berechnen:

```
friction = min(1,TIME*gnd_fric*0.3);
// erhöhe die relative Geschwindigkeit der Entity durch die Kraft
MY._SPEED_X += TIME*force.X - friction*MY._SPEED_X;
dist.X = TIME * MY._SPEED_X;
dist.Y = 0;
dist.Z = 0;
```

Nun benutzen wir den Auftriebswert (force.z), den wir aus den früheren Auftriebsberechnungen gewonnen haben, um das Flugzeug vom Boden hoch zu bekommen:

```
MY._SPEED_Z += TIME*force.z - friction*MY._SPEED_Z;
absdist.X = 0;
absdist.Y = 0;
absdist.Z = TIME*MY._SPEED_Z;
```

Die absolute senkrechte Bewegungsstrecke des Flugzeuges (absdist.Z) wird durch den Auftrieb (der am Boden niemals unter Null beträgt) und der Eindring-Tiefe in den Untergrund (was das Flugzeug auf der Bodenfläche hält) bestimmt:

```
// addiere die durch die Bodenelastizität entstehende Geschwindigkeit
absdist.Z = max(0,absdist.Z) - min(0,max(my_height,-10));
```

Zum Schluss addieren wir die horizontale Geschwindigkeit einer eventuellen sich bewegenden Plattform (das ist nötig, wenn Ihr Flugzeug auf einem Flugzeugträger geparkt war):

```
        // Falls das Flugzeug auf einer sich fortbewegenden
        // Plattform steht, addiere Bodengeschwindigkeit
        absdist.X += my_floorspeed.X;
        absdist.Y += my_floorspeed.Y;
}
```

Bewegung in der Luft
Das ist ja nun die spezifische Art der Fortbewegung an der wir bei einem Flugsimulator vor allem interessiert sind. Hier steuern wir das fliegende Flugzeug:

Als erstes setzen wir den '_MOVEMODE' auf '_MODE_PLANE':

```
ELSE // in der Luft
{
        MY._MOVEMODE = _MODE_PLANE;
```

Dann benutzen wir die 'force' - und 'aforce' – Werte, die in '_player_force' gesetzt sind und berechnen Auftrieb, seitliche Rollbewegung und Schwenkwerte des Flugzeuges. Um die natürliche Drehtendenz eines ungesteuerten Flugzeuges (der Joystick steht unberührt in der Mitte) zu simulieren, nehmen wir einen Bruchteil des gegenwärtigen Winkels (pan, tilt oder roll) und ziehen ihn von den Winkelkräften ab:

```
        // Schräglage wird durch auf/ab - Tasten bestimmt
        // und schwingt zurück nach loslassen
        aforce.tilt = (-0.1*force.X) - (0.01*ang(MY.tilt));
        // seitl. Rollen wird durch rechts/links - Tasten bestimmt
        // und schwingt zurück nach loslassen
        aforce.roll = (-0.1*aforce.pan) - (0.02*ang(MY.roll));
        // Schwenkkraft hängt von den Rudern und dem Rollwinkel ab
        aforce.pan = (0.01*force.Y) - (0.03*ang(MY.roll));
```

Die Vorwärtsgeschwindigkeit (Schub) hängt direkt mit der Motorengeschwindigkeit zusammen. Dieser Wert wird dann vom Kippwinkel modifiziert (verringern der Geschwindigkeit, wenn wir hochziehen, schneller werden, wenn wir die Nase nach unten drücken) und zu einem geringeren Grad vom absoluten Wert des seitlichen Rollens:

```
        // die Vorwärts-Schubkraft wird durch die
        //Motorengeschwindigkeit, den Kipp- und Rollwinkel bestimmt
        force.X = (0.5*MY._RPM)   // Motorengeschwindigkeit
                -(0.01*ang(MY.tilt))      // minus Kipp-Faktor
                -(0.005*abs(ang(MY.roll)));// minus Roll-Faktor
```

Nun verwenden wir die Winkelkräfte, die wir oben berechnet haben, um die winkelabhängige Geschwindigkeit des Flugzeuges zu ermitteln:

```
        // beschleunige die winkelabhängige Geschwindigkeit und
        // ändere die Winkel
        friction = min(1,TIME*ang_fric);
        MY._ASPEED_PAN  += (TIME*aforce.pan)
                              - (friction*MY._ASPEED_PAN);
        MY._ASPEED_TILT += (TIME*aforce.tilt)
                              - (friction*MY._ASPEED_TILT);
        MY._ASPEED_ROLL += (TIME*aforce.roll)
                              - (friction*MY._ASPEED_ROLL);
```

Diese "Winkelgeschwindigkeiten" werden nun zur Änderung der Winkel am Flugzeug benutzt:

```
        MY.pan  += TIME * MY._ASPEED_PAN;
        MY.roll += TIME * MY._ASPEED_ROLL;
        MY.tilt += TIME * MY._ASPEED_TILT;
```

Um zu verhindern, dass unser Flugzeug zu hoch oder zu schnell fliegt, reduzieren wir die Auftriebs- (lift.Z) und Schub- (force.X) –Werte um einen Bruchteil der gegenwärtigen Höhe und Geschwindigkeit abzüglich ihrer Maximalwerte:

```
        // kuenstliche Limits
        // verhindert über die Welt hinauszusteigen
        force.Z -= max(0, 0.01*(my_height - height_max));
        // verhindert zu hohe Geschwindigkeit
        force.X -= max(0, 0.1*(MY._SPEED_X - speed_max));
```

Nun brauchen wir den Schub (force.X), um die Vorwärtsgeschwindigkeit des Flugzeuges zu berechnen. Wir werden diesen Wert zur Berechnung der relativen Strecke verwenden, die das Flugzeug in diesem Frame zurücklegt:

```
        // erhöhe die relative Geschwindigkeit der Entity
        friction = min(1,TIME*gnd_fric*0.2);
        MY._SPEED_X += (TIME*force.X) - (friction*MY._SPEED_X);
        dist.X = TIME * MY._SPEED_X;
        dist.Y = 0;
        dist.Z = 0;     // Auftriebskraft kontrolliert nur die
                        // absolute Geschwindigkeit
```

Dasselbe machen wir mit der Auftriebskraft (force.Z). Der einzige Unterschied zur vorhergehenden Berechnung der Vorwärtsbewegung ist, dass Auftrieb immer gegen die Erdanziehungskraft arbeitet (absdist.Z):

```
        // addiere Auftriebs- und Anziehungskraefte
        MY._SPEED_Z += (TIME*force.z) - (friction*MY._SPEED_Z);
        absdist.X = 0;
        absdist.Y = 0;
        absdist.Z = TIME * MY._SPEED_Z;
}  // END 'airborne'
```

Abschliessen der 'player_aircraft' Aktion
Jetzt, nachdem wir beide, die relative und die absolute Distanz, die das Flugzeug in diesem Frame zurücklegen wird, berechnet haben (ob nun am Boden oder in der Luft), können wir das Flugzeug um diese Strecke fortbewegen, seine Animation ('aircraft_anim') "updaten", den Kamerablick ('move_view') bewegen und für einen Frame die Kontrolle an andere Funktionen abgeben ('wait(1)'):

```
                // Bewege mich (ME) jetzt um die relative und absolute Distanz
                YOU = NULL;     // YOU - Entity ist durch MOVE passable
                MOVE ME,dist,absdist;

                aircraft_anim(); // animiere das Flugzeug

                // Falls ich der einzige Spieler bin, führe Kamera mit
                if (client_moving == 0) { move_view(); }

                // warte einen Tick und wiederhole dann
                wait(1);
        }
}
```

Die Flugzeuganimations – Funktion (aircraft_anim)
Jetzt müsssen wir nur noch die 'aircraft_anim' – Funktion definieren. 'aircraft anim' lässt anhand der gegenwärtigen RPM der Maschine die Propeller routieren. Das machen wir, indem wir den '_ANIMDIST' – Wert des Flugzeugs um seinen '_RPM' multipliziert mit der Zeit, die zur Animation dieses Frames benötigt wird erhöhen. Dieser Wert wird dann in einen Wert zwischen 0 und 100 (unter Verwendung einer WHILE – Schleife) "eingepackt". Anhand dieses Wertes wird dann mit 'SET_CYCLE' das geeignete Animationsbild ausgewählt. Die Funktion sieht folgendermassen aus:

```
function aircraft_anim()
{
        MY._ANIMDIST += MY._RPM * TIME;
        // verpacke die Animation MALGENOMMEN mit einem Wert zwischen 0 und 100
        while (MY._ANIMDIST > 100) { MY._ANIMDIST -= 100; }
        // setze das Bild aus dem Prozentsatz
        SET_CYCLE MY,anim_fly_str,MY._ANIMDIST;
}
```

Einbinden der Aktion
Speichern Sie Ihre Skriptdatei und gehen Sie zurück in den WED. Suchen Sie Ihr Flugzeug – Model und markieren Sie es. Über sein Eigenschaftsfenster ändern Sie seine Aktion in 'player_aircraft'. Speichern Sie das Level und kompilieren sie es (da wir ausschliesslich eine Map – Entity verändert haben, können Sie Zeit sparen wenn Sie dabei "Update Entities" auswählen).
Gehen Sie im "File" – Menue auf "Run Level..." und geniessen Sie das Fliegen in Ihrem ersten Flugsimulator.

das fertige Level laeuft

Abschluss

Mit diesem Kurs haben wir einen umfassenden Bereich abgedeckt. Wir haben ein Skript von Grund auf neugeschrieben, etwas über die neuen 4.19 Befehle und ihre Syntax gelernt und ein einfaches Flugmodell, das zu fliegen Spass macht, generiert.

Dieser Übungskurs deckt nur die absoluten Grundzüge von Flugsimulatoren ab. Einige Dinge können Sie durch schlichtes Hinzufügen verbessern: Verbessern des Flugmodells (realistischer machen), Modifikation der Landschaft in WED (Gegenstände wie Bäume, Berge etc. einbinden), Verwenden von Panels, um damit ein Cockpit mit Messgeräten, Kompass usw. zu kreieren. Der zweite Teil dieser Übungsreihe wird sich mit Luftkampf befassen.