Forward
Dear Reader,

I have produced this workshop to help answer the question "How do I make a flight simulator with 3DGameStudio?". Along the way I will bring up some of the new features that became available with the recent (4.19) update.

This workshop, like the Venture Workshop before it, is mostly aimed towards users with some previous 3DGameStudio experience. I assume that you have worked through the tutorials and understand how to use the tools (WED, MED, and WDL).

This text is meant to complement the rest of the documentation that comes with 3DGameStudio, not replace it. If something in this workshop is unclear to you please read through the manuals that came with 3DGameStudio. I apologize in advance you any unclear wording, faulty code, errors, or omissions.

I hope you find the workshops informative and enjoyable.

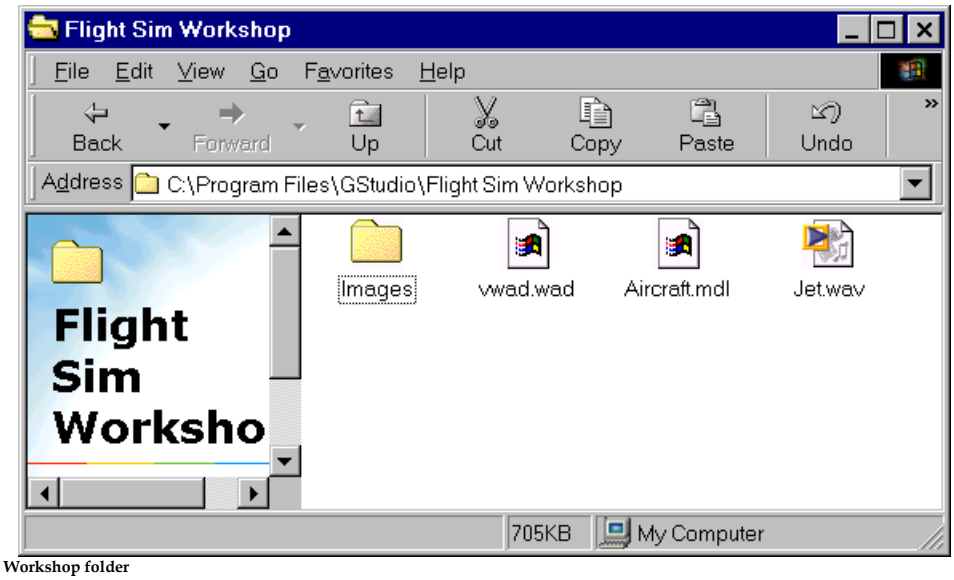
-doug.

Get the latest version
Before you begin, make sure you have the latest version of 3DGameStudio (4.19 or greater) since we are going to make use of several of the new features that have been added. Also you'll need the latest world builder, because for a flight simulator we'll need an extremely huge world.

Prepare your workspace
Create a folder called "Flight Sim Workshop" in your GStudio folder. This is the folder where all you game elements will be stored.

The first thing we are going to add to our folder are the game model and image folder. If you don't have them already, go to the Conitec download page (http://www.conitec.net/a4update.htm) and retrieve the FlightSim level. Unzip the contents into your folder. You folder should now contain at least the files:

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



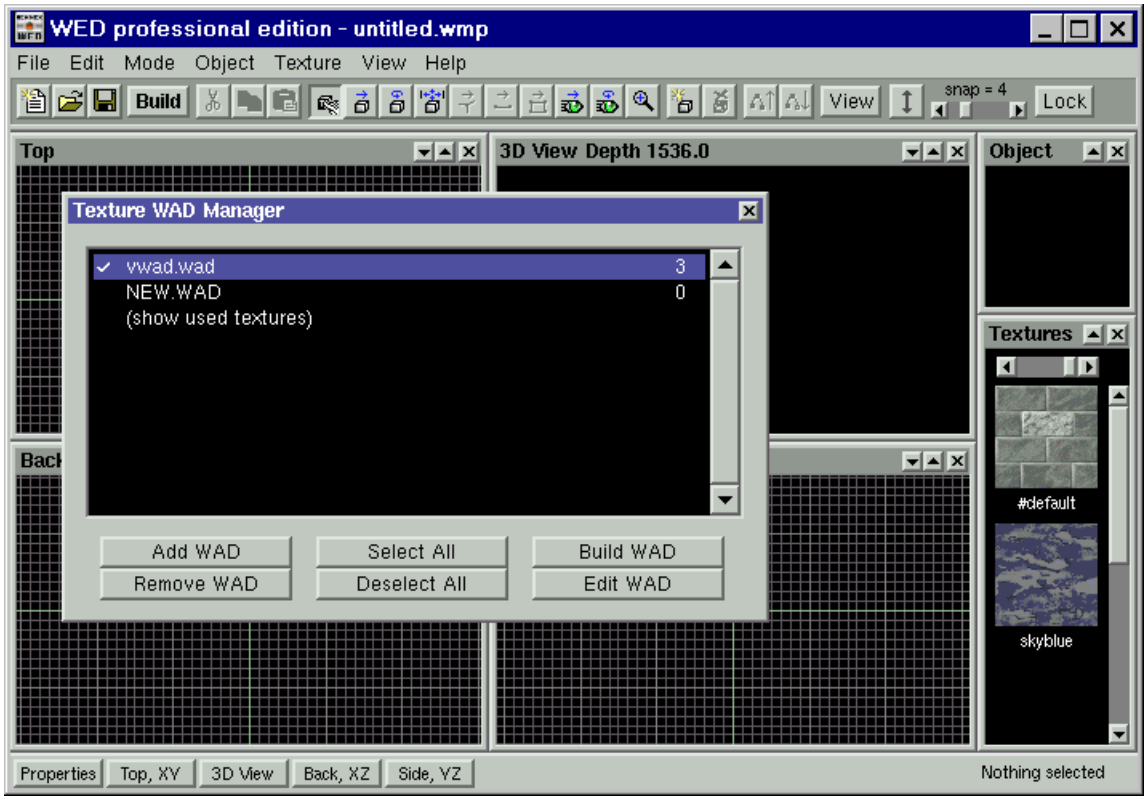
Flight Sim Workshop folder

Creating the level

Our level map is going to be simple. A large, flat ground surface with a sky.

Open WED and select “File->New”.

Open up the texture manager (“Texture->Texture Manager”). Select 'Add WAD', navigate into your "Flight Sim Workshop" folder, and add the standard.wad. Close the texture manager.



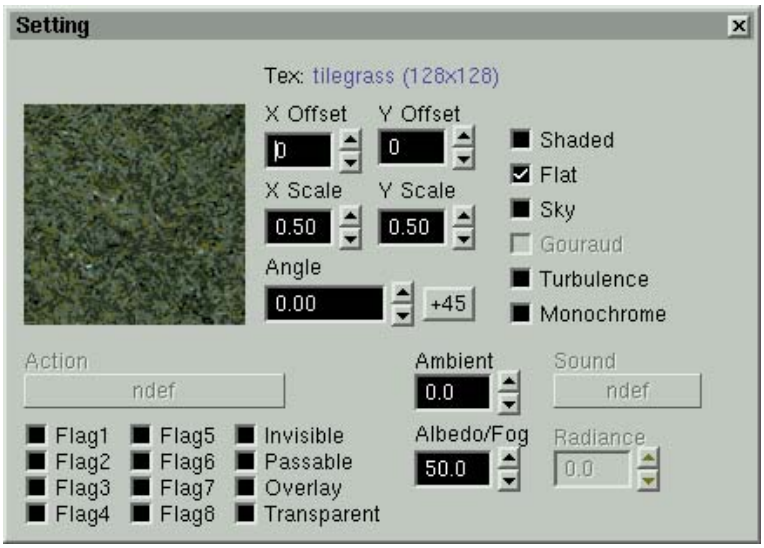
WED: Texture Manager

Select "Add Primitive -> Cube(large)".

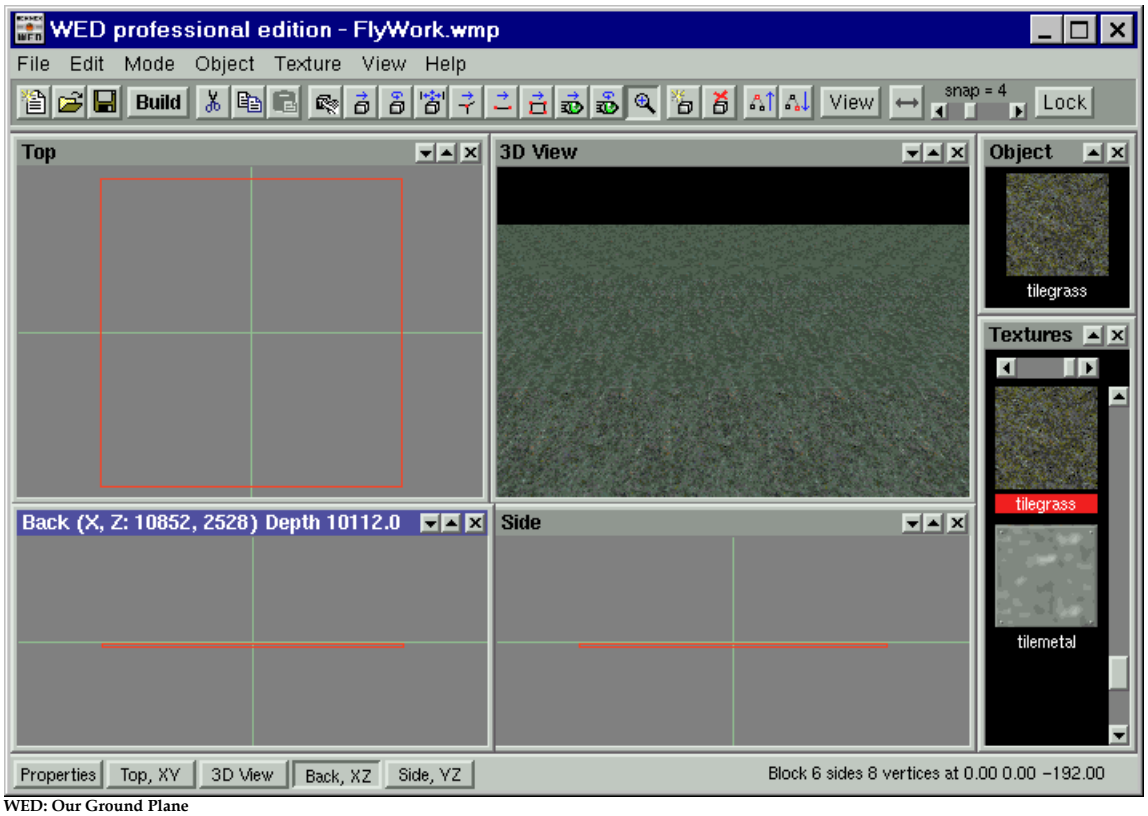
Scale the cube, creating an extremley long, wide, thin surface of around 40000x40000x50 quants in size (close to the max level size of 50,000 quants). The simplest way to do this is to increase the 'View Depth' of your 'Back' and 'Side' views (by using the '+' key) to the size you want the plane to be, center the box, and scale it in the 'Top' view until it just disappears in the 'Side' and 'Back' view windows. Then scale it back slowly until it reappears.

This is going to be your ground. Because it's so huge, we don't want to apply a shaded texture for it. Shaded textures automatically split the surface into small squares for dynamic lighting, and allocate video memory for the shadow map. Both will increase building time, level file size and video memory requirements. So we'll use a flat texture here.

Click right onto the 'tilegrass' texture in the texture bar, select 'settings', then check 'flat' and set the scales down to 0.5. We are scaling it down a little because for increasing our world further, we are also going to use scaled down models for it. This way we'll achieve a world whose 40,000 x 40,000 quants will appear as several square miles.

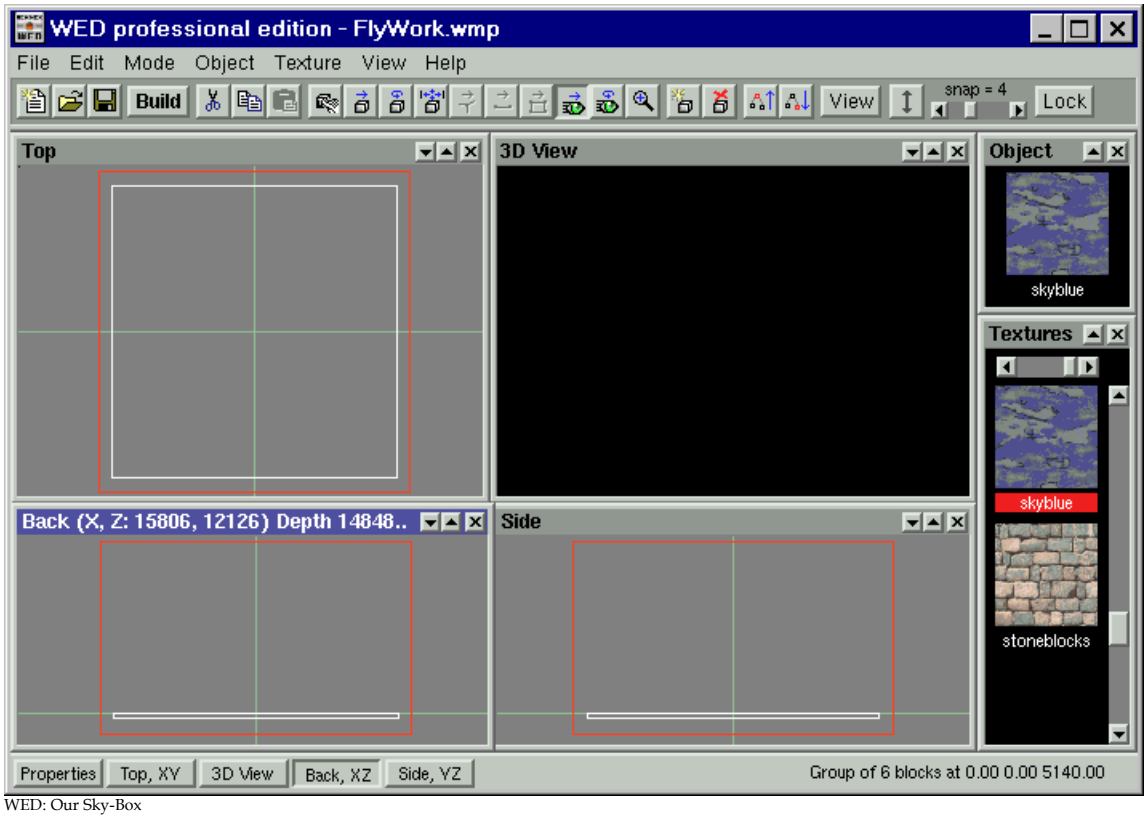


Now apply the flat, scaled-down 'tilegrass' texture to the ground plane and then move it so it is just below the center.



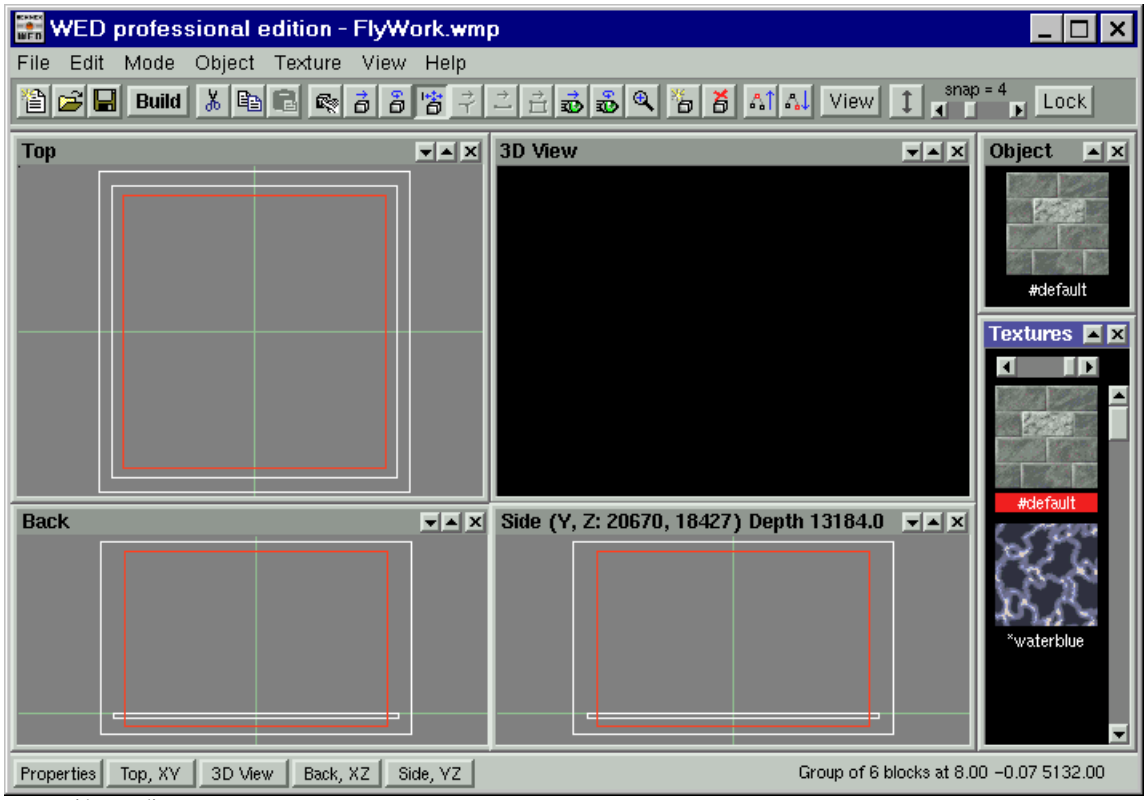
WED: Our Ground Plane

Add another cube and scale it up so it just surrounds the ground plane. It is okay if the ground plane extend slightly outside this box, but make sure to leave plenty of space above the ground. Hollow this block (Alt+H) and give it the 'skyblue' texture. This is our sky-box.



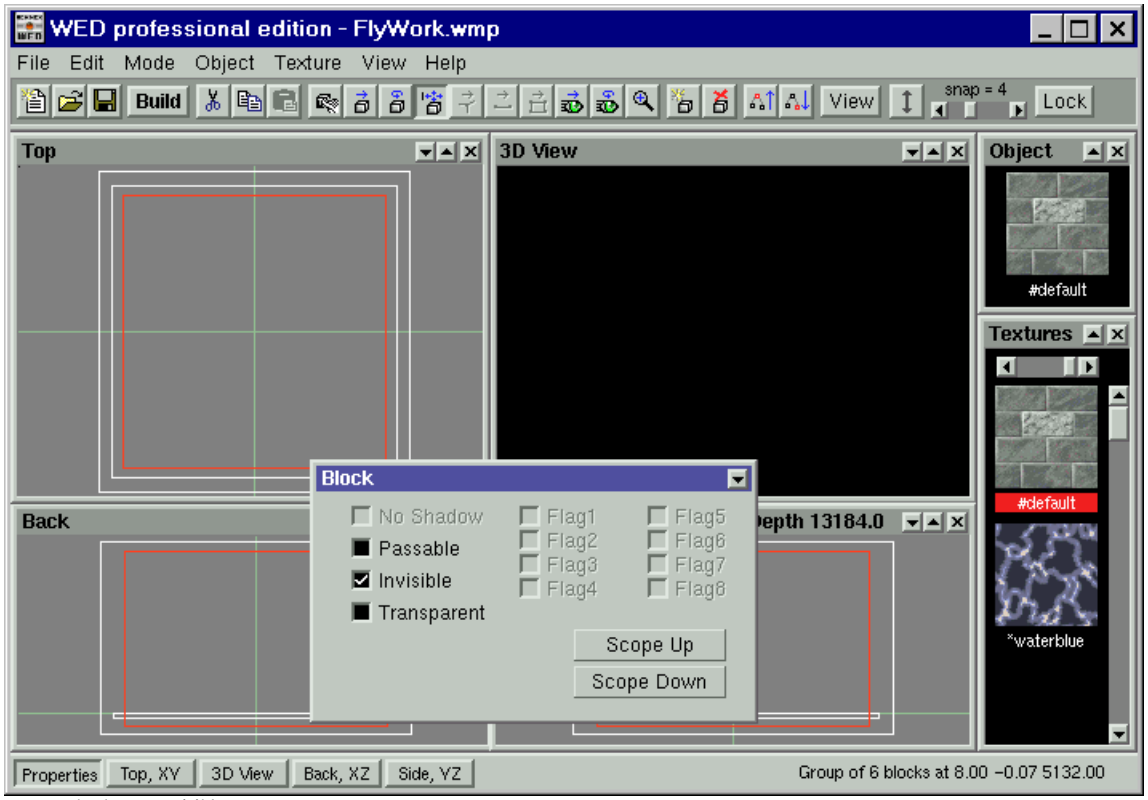
WED: Our Sky-Box

With the sky-box selected, use the duplicate command (Ctl+D) to duplicate this box and scale it down so it fits inside the sky-box. The box should still have plenty of space above the ground plane, extend beneath the plane, and intersect the ground plane along its edges.



WED: Inside Bounding Box

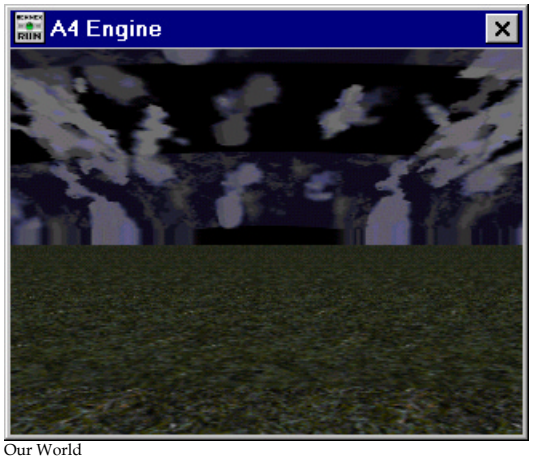
Check the 'Flat' flag of the 'default' texture and apply it to the block. Then open up the block's Properties window. Check the 'Invisible' flag and make sure the other options are unchecked. This will act as our "bounding box", making sure that the player can't fly outside our world. We can later add mountains to the border of the world, but for the moment the bounding box will do.



WED: Make the Box Invisible

Save the level in your Flight Sim Workshop folder (call it "FlyWork"), 'Build' (with Level Map selected), and Run the level.

You should be able to see a nice grass lawn that stretches out for miles.



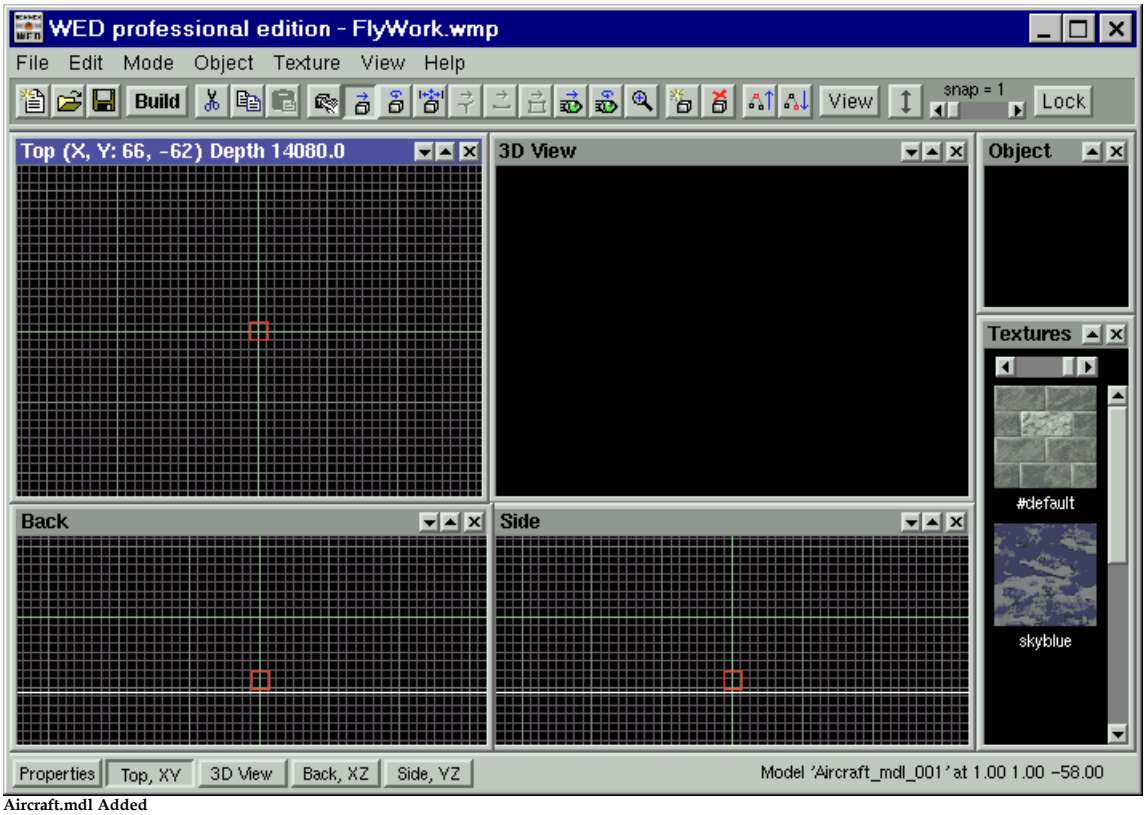
Our World

This is the minimum map for a flight sim; just a space to get us started. Later you should feel free to add other structures to the map (houses, mountains, etc.) to make things more interesting.

Adding the Plane

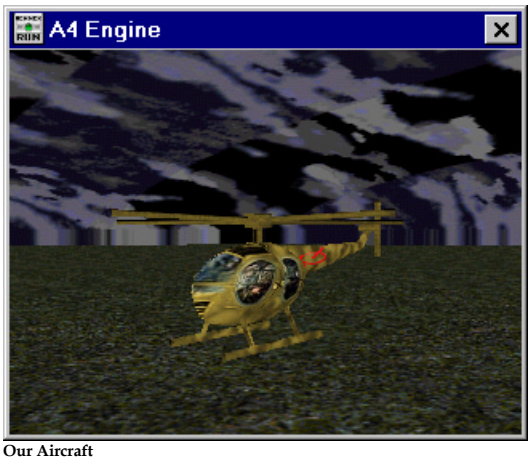
Now we are going to add our aircraft model. For our flight sim we will make due with the helicopter model (although we will still use airplane physics to control it).

Select "Object->Load Entity..." and use the file navigation windows to find the "aircraft.mdl" model. The model should appear in your world. Place it near the center of your ground plane so that it is resting on the surface.



Aircraft.mdl Added

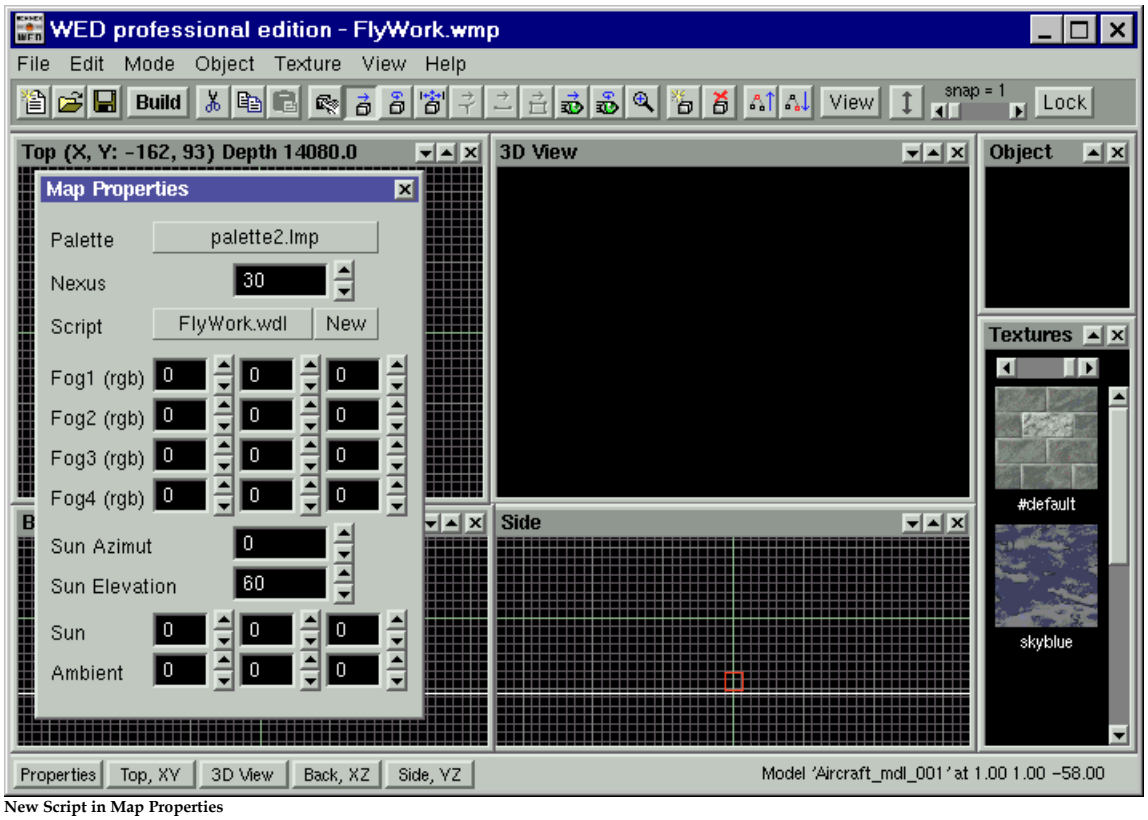
Save, rebuild and run the level. Look for our airplane model. Make sure it is on, or just slightly above, the ground level. Adjust the model placement until you are satisfied.



Our Aircraft

Creating your script

Create a script for your level. Open your Map Properties window ("File->Map Properties..") and press the 'New' button. The button next to Script should change from "ndef" to "flywork.wdl".



New Script in Map Properties

Open your level folder and select and open (double click) the "flywork.wdl" file. If Windows asks what application to use to open the file, select "notepad" (or any other plain text editor).

You should notice that the typical game 'template' has been created for you. This is fine for most projects but we want to do something more advanced this time. Go ahead and select everything (in Microsoft Notepad use "Edit->Select All") and hit the delete key. Now you're starting from ground zero!

Adding Paths

Lets start by defining the paths our program is going to use. Paths are used to tell the engine where it can locate the files used in our project (images, sounds, other scripts, etc.). The folder we are in ("Flight Sim Workshop") is already included so, in our case, we only need to add the "template" and "image" folders.

Type in the following line:

```
PATH    "..\\template"; // Path to WDL templates subdirectory
```

Note that all paths are relative to our level folder. The line above says the following, "Go 'up' one directory ("...") and go into the template folder ("\\template")".

Now we want to include our "images" folder the same way. Add this line:

```
PATH    "images";       // Path to our images subdirectory
```

Since the "images" folder is in the same folder as our level all we need to do is tell it to do is: "look in the folder called IMAGES in this folder".

Note: The order these two lines appear in determines which folder gets checked first. If you use more then one file with the same name, your level only uses the first file it finds. For example, if you wanted to change the arrow cursor ("ARROW.PCX") so it was white instead of red, you could create your new white arrow and save it as "ARROW.PCX" in your level folder. Now your level would use your new cursor and not the default cursor in template. But, if you put that cursor in IMAGES, you would still be using the template cursor (since you told your level to look there first). You can change the order you look in by changing the order the PATH lines appear. For example, change the two lines you entered to this:

```
PATH    "images";               // Path to our images subdirectory
PATH    "..\\template"; // Path to WDL templates subdirectory
```

Now the engine will look in your level folder, then your "images" folder, and lastly the template folder.

Adding includes

After we set up our paths we should add our include files. Type in the following under PATHs:

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

The 'INCLUDE' command tells A4 to replace this line with the contents of the file between the '<>'. It is as if you went to the file in question and copied all the code from it and pasted it into you script.

This is a powerful tool because it allows us to reuse code from other projects and take advantage of updates in the included files without having to rewrite your code. For

example, the 4.19 update included code to allow the player to swim in water. So now any project that includes "movement.wdl" can use this new swimming code.

Just like paths, the order of the INCLUDE lines are important. Since some scripts use values that are declared in other scripts. For example: "actors.wdl" uses the variable 'force' which is declared in "movement.wdl". If we put "actors.wdl" before "movement.wdl" we would get errors.

It's okay to INCLUDE scripts even if you don't use features from them in you code. Most of the files in template are interdependent on each other so if you plan on using one of them, you should INCLUDE all of them just to be safe. The exception to this rule is the "Venture.wdl" script, which is not used by any of the other scripts, but uses many of them.

Starting Engine Values

Now we are going to set some important values that will help determine how the simulator will be displayed. These values effect the resolution, color depth, frame rate, and lighting. Add the following lines beneath the 'INCLUDE' lines:

```
// Starting engine values
IFDEF LORES;
var video_mode = 4;             // 320x240
IFELSE;
var video_mode = 6;             // 640x480
ENDIF;
var video_depth = 16;   // D3D, 16 bit resolution
var fps_max = 40;       // 40 fps max
var floor_range = 200;  // prevent sudden brightness changes on aircraft
```

If you've written WDL scripts with previous versions of 3D GameStudio, the first thing you'll notice is that we're using 'var' instead of 'SKILL'. We do this because 'var' is a little shorter, and it's also compatible with Javascript. Which means the old way of "`SKILL VIDEO_MODE { VAL 6; }`" has been replaced with the more elegant, javascript-style "`var video_mode = 6;`".

'fps_max' and 'floor_range' are new to 4.19. You can read about them in the WDL Manual. Briefly, 'fps_max' limits the frame rate so it does not exceed that value. By setting 'fps_max' to 40 we are limiting ourselves to 40 frames per second (even if the system we are running this on can display more).

'floor_range' was created with flight simulators in mind. It limits the range within which the entity's light depends on the brightness and shadows of the floor area below. Which means that if our plane is flying high over a city and it passes over a shadow, its lighting is not changed.

Note that 'video_mode' and 'video_depth' are "read only" values, which means that they can not be set directly once the program is running they can only be changed by calling the instruction 'SWITCH_VIDEO'.

The Main Function
In any project you need a 'main' function. This is the first function to be called when the program starts. In most cases the main function is very simple, our main is no exception. Please enter the following lines (under your last line):

```
// Desc: our MAIN function called at game start
function  main()
{
IFNDEF NOTEX; // disable d3d_texreserved with -d notex on weak cards
        D3D_TEXRESERVED = min(12000,D3D_TEXMEMORY/2);
ENDIF;

        // set sky maps before level loading to give them priority
        init_environment();     // set up our 'environment' (sky, clouds, etc.)
        LOAD_LEVEL <flywork.WMB>;
        load_status();  // restore global skills
}
```

We'll explore each of these lines in turn.

Functions vs. ACTIONs
You'll notice that we've changed from "`ACTION main {..}`" to "`function main() {..}`". Functions are new to 4.19. They behave almost identically to ACTIONs except that they do not show up in the WED pop-up list as attachable entity actions. So, if you are creating a function that is going to be directly attached to an entity (i.e. my_player) make it an ACTION, if not, make it a function.

D3D_TEXRESERVED (and the min(x,y) function)
The first thing we are going to do in main is try and reserve a section of memory on the user's video card to preload our textures. This reduces the 'first load lag' that can occur when a new texture is seen for the first time while playing.

I said 'try' because not all video cards will allow us to reserve memory. To handle these cards we have put the code inside an "IFNDEF NOTEX; ... ENDIF;" block. If the user can't use D3D_TEXRESERVED (i.e. they have a "weak" card) they need to set the '-d' flag to "-d notex" before running this program so A4 will know to skip this block of code.

We don't want to use all our video memory for preloaded textures, if we did A4 wouldn't have any room to load other textures while the level is running (causing it to abort with an error message). A4 automatically reserves 2MB of memory for these texture but, to be on the safe side, we should allocate no more then half of the available video memory. The amount of memory available for you varies from system to system, but you can always find out the total amount of memory available by checking the value in D3D_TEXMEMORY.

So to reserve 12MB of video memory on systems with 24MB or more of video memory, or half the available memory on cards with less then 24MB we would take half the video memory and compare it to 12MB, taking which ever is less.

We could do it like this:

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

Or we can do it all in one line by using the "min(x,y)" which takes two values and returns the smallest:

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

init_environment() (the new function call)
Our first function call shows the 'new way' to call functions. The old way "CALL init_environment;" has been replaced with the new "init_environment();". Both these methods work the same way, but the new way lets us use our own functions and actions the same way we would use predefined functions (like 'min' and 'max').

Lets type in the 'init_environment' function (below the main function):

```
// our environment bitmaps
BMAP my_sky = <sky1.pcx>,0,0,128,128;
BMAP my_clouds = <sky1.pcx>,128,0,128,128;
BMAP my_mountains = <mount16.pcx>;

// Desc: set up the sky, cloud, and scene maps
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;
}
```

The first three lines define our three sky bitmap images. The first two bitmaps ('my_sky' and 'my_clouds') load from the left and right half of our sky image ("sky1.pcx") while 'my_mountains' loads from the mountain image bitmap. These bitmaps are then loaded into the three bitmaps that make up our sky ('sky_map', 'cloud_map', and 'scene_map').

More information on these bitmaps and other values that are set here can be found in the WDL Manual (Chapter 7: Engine Variables).

Save your work and run the level. You should notice our rendered sky with clouds moving overhead and a mountain range in the distance. You'll also notice that as you move to the edge of our world you can see past the bottom of those mountains. You can fix this to some degree by making the 'scene_map' taller and moving it down (by adjusting the 'scene_angle.tilt' value). You can also hide the edge of the map by adding buildings, trees, or mountains along the edges.

Creating our Aircraft

What makes our project a flight simulator and not a car racing game, tank simulator, or some other type of game is the airplane flight model. We are going to explore how to create a simple aircraft ACTION to attach to our airplane model to help get it off the ground.

Basic flight model

What sets flight simulators apart from each other (besides the graphics and the choice of aircrafts to fly) is how the flight model works. Some flight models are very complex; they can use real flight data and perform complex mathematical calculations to figure out exactly what would happen when you increase your flap angle by 3 degrees.

Other flight simulators take a different approach; they set some simple rules and values to produce results that approximate the physics of a plane. Besides being easier to write, these 'simple' flight sims can also be a lot of fun (especially if you don't want to sit down and read a textbook before flying one).

Our workshop is going to go the 'simple' route. The flight model presented here covers the basic four forces acting on the plane (lift, thrust, gravity, and drag), creates a few rules and limits, and calculates how the plane will behave.

Creating the variables

Lets start by setting the values used by our airplane model. Enter these lines at the end of the script file:

```
// Aircraft DEFINEs and values
DEFINE _MODE_PLANE,16;
DEFINE _RPM,SKILL31;           // motor speed
```

```
// airplane values
var stallspeed = 8;     // plane stalls below this speed
var climbrate = 1.5;    // maximum climb rate
var climbfactor = 0.1;  // wing profile
var height_max = 750;   // maximum height
var speed_max = 25;     // max airspeed
var max_rpm = 7;        // maximum engine speed

// sound vales
var enghandle = 0;
SOUND engsound = <jet.wav>;     // engine sound

// animation string
string anim_fly_str = "fly";

define _FLYSCALE 0.3;   // scaling down the aircraft
```

The first DEFINE sets one of the two MOVEMODEs we are going to check for in our flight model (the second being _MODE_DRIVING which is defined in "movement.wdl"). The second DEFINE tells us to use SKILL31 as the model's RPM value.

The airplane values are constant values that will be used to calculate how different forces will effect the airplane (and the limits to which those forces can act). We'll cover these values as they come up in the code. When you have completed this workshop feel free to adjust these values and see how they change the behavior of the plane.

The next values handle the sound produced by the engine ('enghandle' and 'engsound') the name of the prop animation frames in our airplane model ('anim_fly_str'), and the scale our airplane is going to get in order to make the world appear bigger.

Coding the Action

Since this function will be attached to the airplane model inside the level we will make it an ACTION. This is a large action that is broken up into several sections. To avoid confusion I've included the entire separately in this archive.

Initilizing Aircraft Values

The first thing we do is initialize our entity:

```
// Desc: our aircraft action
ACTION player_aircraft
{
        // Init values
        if (MY.client == 0) { player = MY; } // created on the server?

        MY._TYPE = _TYPE_PLAYER;
        MY.ENABLE_SCAN = on;    // so that enemies can detect me

        IF(MY._FORCE == 0) {  MY._FORCE = 1.5; }
        IF(MY._MOVEMODE == 0) { MY._MOVEMODE = _MODE_PLANE; }
```

```
// scale the aircraft model down to increase the world size
        MY.scale_x *= _FLYSCALE;
        MY.scale_y *= _FLYSCALE;
        MY.scale_z *= _FLYSCALE;

        drop_shadow(); // needs _movemode to be set before
```

So far this is similar to the 'player_move' action in "movement.wdl". We check to see if we are on a server, set our type to player, make the entity sensitive to scan events, set its force value (if not already set in the model itself), and set it's 'MOVEMODE' to that of a plane.

After all the values are set we call 'drop_shadow' so our plane will produce a shadow on the ground.

The Aircraft's WHILE Loop
Now we start to enter the core of our action:

```
        // while we are in a valid MOVEMODE
        WHILE(MY._MOVEMODE == _MODE_PLANE
                || MY._MOVEMODE == _MODE_DRIVING)
        {

                _player_force();  // set force and aforce values from player input
                scan_floor();     // set floor_normal, my_height, and floor_speed
```

We'll stay in this loop until the entity is no longer flying as a plane or driving on the ground (in our case, we should always be in one of these two modes).

The first thing we do is gather input from the player and the environment. With a simple call to '_player_force' we gather all the movement information from the user's keyboard, mouse, and joystick input and store the results in the 'force' and 'aforce' variables. 'scan_floor' tells us how far above the closest surface we are (and the speed and tilt of that surface).

Engine (Thrust)
Next we calculate the engine speed:

```
                // Calculate Engine speed (accelerate motor with [HOME], [END])
                // MY._RPM = rpm / 1000
                MY._RPM += 0.0025*force.Z*TIME;
                // limit _RPM between 0 and 7000 rpm max
                MY._RPM = max(0,min(max_rpm,MY._RPM));
```

The engine speed is set by the user pressing the 'HOME' and 'END' keys, this is meant to simulate the player adjusting the aircraft's throttle. '_player_force' converts the 'HOME/END' key presses into a positive/negative 'force.Z' value. By multiplying the 'force.Z' value by the amount of time that has passed since the last frame (TIME) and a

constant value (used to tune how fast we change our engine speed) we can calculate the change in the RPM of the engine.

We want to limit the speed of the engine so that it doesn't exceed the 'max_rpm' and stays positive (above zero). We could do this by using a couple of 'IF' statements, but its quicker to use nested min/max functions. The statement "max(0,min(max_rpm,MY.RPM))" reads as follows: "take the minimum (smallest) value of max_rpm and MY.RPM, now take the maximum (biggest) value of that value and zero".

Engine Sound

Now lets handle the engine sound. We want the engine sound to play continuously in the background whenever the engine is running. I would also be nice to change the sound so the player has some sort of audio clue to how fast the engine is running.

```
// start, stop or tune the engine sound
if(MY == player)
{
        // if the engine is running...
        if(MY._RPM > 0)
        {
                if(enghandle == 0)  // no sound is playing
                {
                        // start playing engine sound
                        PLAY_LOOP engsound,25;
                        enghandle = RESULT;
                }
                // tune the sound depending on speed_ahead
                temp = (MY._RPM + MY._SPEED_X * 0.2) * 60;
                TUNE_SOUND enghandle,25,temp;
        }
        else  // engine is stopped...
        {
                if(enghandle != 0)      // sound is playing
                {
                        // stop playing engine sound
                        STOP_SOUND enghandle;
                        enghandle = 0;
                }
        }
} // END if(MY == player)
```

The first thing we do is check to make sure this action is being called by the player (if we attached this action to another entity, we don't want it to effect our sound). If so we then check to see if the engine is running (MY._RPM > 0). If the engine is running we then check if the engine sound is already playing by checking to see if a value is stored in 'enghandle'. If we are not currently playing the engine sound, the value in 'enghandle' will be set to zero.

To start the engine sound and play it constantly in the background you can use the PLAY_LOOP instruction. This will loop the sound in the background until we call the

STOP_SOUND instruction. PLAY_LOOP will return a value in the RESULT field. This value is known as the sound's handle. It is important that we save this handle because it is the only way we can change or stop this sound once it starts playing. Therefore we will save this sound handle in the variable 'enghandle'.

The next thing we do while the engine is running is to adjust the frequency of the engine sound to reflect how fast the engine is running. We do this by using the formula (MY._RPM + MY._SPEED_X * 0.2) * 60 to calculate the percent of the normal frequency to use in playback. A higher frequency makes it sound like the engine is running faster, a lower frequency makes it sound slower.

Calculating Lift

Next we are going to calculate how the current speed and angle of the aircraft effects the lift of our plane.

```
// Calculate lift force, dependent on speed, tilt and roll angle
force.z = (climbfactor*MY._SPEED_X)
         * (0.05 * (90 + ang(MY.tilt)- (0.5*abs(ang(MY.roll))) ));
```

In this calculate we calculate lift (force.z) by combining three factors.

The lift of an aircraft comes from air moving rapidly over it's wings. The faster the aircraft is moving through the air the more lift is created. This "basic lift" force is calculated by ("climbfactor*MY._SPEED_X") where 'MY._SPEED_X' is the forward speed of the aircraft and 'climbfactor' is a constant value that represents the shape of the wing and its ability to produce lift (it's 'airfoil effect').

The lift value is then modified by the tilt and roll of the aircraft (know in avionics as the "angle-of-attack"). If the nose of the aircraft is pointing up, the aircraft is more likely to climb. Pointing the nose of the aircraft down makes it fall down to earth. Rolling the aircraft from side to side reduces the amount of lift that can be created by the wings. The combination of these two angles is calculated by taking 1/20 of the tilt and subtracting half of the absolute roll value from it. These values are far from actual physical data, but they produce reasonable results in most situations.

Planes have to maintain a certain amount of speed to stay in the air. When a plane falls below this speed the enter into what pilots call a 'stall'. When the plane is stalled it's wings no longer produce any lift. We can simulate this by adding this simple rule:

```
// if our speed is less then the stall speed...
if(MY._SPEED_X < stallspeed)
{
        // stall (no lift)
        force.z = 0;
}
```

The force that lift is trying to overcome is gravity. The next thing we do is reduce any lift effect by the force of gravity:

```
// airplane is effected by gravity
force.z -= gravity;
```

To make sure we don't suddenly shoot up into the air we are going to limit the amount of which we can climb to our maximum 'climbrate':

```
// limit the lift force
force.z = min(force.z,climbrate);
```

Movement on the Ground

Airplanes spend a great deal of their time on the ground. We use the following steps to handle movement while we are not in the air.

The first thing we do is to test if we are on the ground (my_height < 5). If so we set our '_MOVEMODE' to '_MODE_DRIVING':

```
// if on the ground, drive around
IF(my_height < 5)
{
        MY._MOVEMODE = _MODE_DRIVING;
```

Then we calculate the forward force (force.x) by taking half the engine speed minus 1 (using the max value to make sure we never go backwards):

```
// the ahead force depends on the motor speed
force.X = max(0,0.5*(MY._RPM - 1));
```

Since steering on the ground is handled by both the stick (or left/right arrow keys) and pedals we check to see if the player is using the stick to steer. If they are, the aforce.pan value will already be set, if not we use any force coming from the pedals (which is stored in force.Y):

```
// steer with both stick and pedals [<] [>]
if (aforce.pan == 0)
{
        aforce.pan = force.Y;
}
```

To keep the plane level on the ground, we adjust the roll angle by setting the roll force (aforce.roll) equal to a fraction of the current roll angle of the aircraft (ang(MY.roll)):

```
// If the roll angle was not zero,
// apply a roll force to set the angle back
aforce.roll = -0.2*ang(MY.roll);
```

Now we use the angular force (aforce) values to set the angular speed of the aircraft:

```
// Now accelerate the angular speed, and set the angles
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);
```

And then we use the angular speed of the aircraft times a fraction of the forward speed (MY._SPEED_X) to calculate the pan, tilt, and roll of the aircraft as it rolls across the ground:

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

We use the same sort of calculations to calculate the forward speed of the aircraft:

```
friction = min(1,TIME*gnd_fric*0.3);
// accelerate the entity relative speed by the force
MY._SPEED_X += TIME*force.X - friction*MY._SPEED_X;
dist.X = TIME * MY._SPEED_X;
dist.Y = 0;
dist.Z = 0;
```

Now we use any lift force (force.z) gained from the earlier lift calculations to lift the plane off the ground:

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

Now if we have scaled the aircraft down for gaining a bigger relative world size, we have to scale it's speed – the distance per frame – by the same amount. An aircraft of toy size moves slower than the real thing:

```
// if the aircraft model is scaled, we need to scale its speed too
        dist.x *= _FLYSCALE;
        dist.y *= _FLYSCALE;
        dist.z *= _FLYSCALE;
        absdist.x *= _FLYSCALE;
        absdist.y *= _FLYSCALE;
        absdist.z *= _FLYSCALE;
```

The absolute vertical distance of the plane is adjusted (absdist.Z) by the lift (which, on the ground, is never less then zero) plus the distance it is underground (this keeps the plane above the surface of the ground). This is a true distance that must not be scaled down!

```
// Add the speed given by the ground elasticity
        absdist.Z = max(0,absdist.Z) - min(0,max(my_height,-10));
```

The last thing we do is add any horizontal speed gathered from a moving platform (this would be useful if your plane was resting on the back of an aircraft carrier):

```
        // If the plane is on a moving platform add floorspeed
        absdist.X += my_floorspeed.X;
        absdist.Y += my_floorspeed.Y;
}
```

Movement in the Air

This is the type of movement in a flight simulator that we are most interested in. Here is where we control the plane in the air.

The first thing we do is set the '_MOVEMODE' to '_MODE_PLANE':

```
ELSE // airborne
{
        MY._MOVEMODE = _MODE_PLANE;
```

Then we use the 'force' and 'aforce' values set in '_player_force' to calculate the tilt, roll, and pan forces on the aircraft. To simulate the plane's natural tendency to center itself when no input is give (i.e. the joystick is centered), we take a fraction of the current angle (pan, tilt, or roll) and subtract it from each of the angular forces:

```
        // tilting is done by the up/down keys,
        // and swings back after releasing
        aforce.tilt = (-0.1*force.X) - (0.01*ang(MY.tilt));
        // rolling is done by the left/right keys,
        // and swings back after releasing
        aforce.roll = (-0.1*aforce.pan) - (0.02*ang(MY.roll));
        // the pan force depends on the rudder padals
        // and the roll angle
        aforce.pan = (0.01*force.Y) - (0.03*ang(MY.roll));
```

The forward speed (thrust) is directly related to the engine speed. This value is then modified by the tilt angle (reducing speed if we tilt up, gaining speed if we tilt down) and, to a lesser extent, the absolute value of the roll:

```
        // the ahead force depends on the motor speed
        // and the tilt and roll angle
        force.X = (0.5*MY._RPM)   // motor speed
                  -(0.01*ang(MY.tilt))// minus speed due to tilt
                  -(0.005*abs(ang(MY.roll)));// minus speed due to roll
```

Now we use the angular forces that we caluculated above to change the angular speed of the aircraft:

```
        // Now accelerate the angular speed, and change the angles
        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);
```

These angular speeds are then used to change the angles of the aircraft:

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

To keep our aircraft from flying too high or too fast we reduce the lift (force.Z) and thrust (force.X) values by a fraction the current height and speed minus their max values:

```
// Artificial limits
// prevent climbing out of the world
        force.Z -= max(0, 0.01*(my_height - height_max));
// prevent over speed
        force.X -= max(0, 0.1*(MY._SPEED_X - speed_max));
```

Now we use the thrust (force.X) to calculate the forward speed of the aircraft. We will use this value to calculate the relative distance the plane will travel this frame:

```
// accelerate the entity relative speed by the force
        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;     // lift force controls absolute speed only
```

We do the same thing with the lifting force (force.Z). The only difference from the forward motion calculated above, is that lift always acts opposite to the gravitation pull of the earth (absolute Z). Finally we scale down the resulting speeds again:

```
// Add the lift and gravity force
        MY._SPEED_Z += (TIME*force.z) - (friction*MY._SPEED_Z);
        absdist.X = 0;
        absdist.Y = 0;
        absdist.Z = TIME * MY._SPEED_Z;

// if the aircraft model is scaled, we need to scale its speed too
        dist.x *= _FLYSCALE;
        dist.y *= _FLYSCALE;
        dist.z *= _FLYSCALE;
        absdist.x *= _FLYSCALE;
        absdist.y *= _FLYSCALE;
        absdist.z *= _FLYSCALE;
}  // END 'airborne'
```

Finishing the 'player_aircraft' Action

Now that we have calculated both the relative and absolute distances the aircraft is going to move for this frame (whether it be by ground or by air) we can move the plane

by that distance, update it's animation ('aircraft_anim'), move the camera view ('move_view'), and release control to other functions for a frame ('wait(1)'):

```
        // Now move ME by the relative and the absolute distance
                YOU = NULL;     // YOU entity is considered passable by MOVE
                MOVE ME,dist,absdist;

                aircraft_anim(); // animate the aircraft

        // If I'm the only player, draw the camera and weapon with ME
                if (client_moving == 0) { move_view(); }

        // Wait one tick, then repeat
                wait(1);
        }
}
```

The aircraft_anim function

Now all we need to do is define the 'aircraft_anim' function. 'aircraft_anim' rotates the plane's propeller by the engine's current RPM. We do this by incrementing the aircraft's '_ANIMDIST' value by it's '_RPM' times the time it took to animate this frame. This value is then 'wrapped' to a value between 0 and 100 (by use of a WHILE loop) then that value is used to select the proper animation frame using 'SET_CYCLE'. The function is as follows:

```
function aircraft_anim()
{
        MY._ANIMDIST += MY._RPM * TIME;
        // wrap animation TIME to a value between zero and 100 percent
        while(MY._ANIMDIST > 100) { MY._ANIMDIST -= 100; }
        // set the frame from the percentage
        SET_CYCLE MY,anim_fly_str,MY._ANIMDIST;
}
```

Attaching the Action

Save your script file and go back into WED. Find your airplane model and make sure it is selected. Use its properties window to change its action to 'player_aircraft'. Save the level and build it (since the only change we made was to a map entity you can save time by selecting "Update Entities").

Select "Run Level..." from the "File" menu and enjoy flying your very first flight sim.

Finished Level Running

Wrap up

We have covered a lot of ground with this tutorial. We've written a script file from scratch, learned some of the new 4.19 commands and syntax, and created a simple flight model that is fun to fly.

This workshop only cover the very basics of flight sims. Some things you can improve on include: improving the flight model (by adding more realism), modifying the terrain in WED (adding things like mountains, trees, etc), using panels to create a cockpit with gauges, compass, and so on. The second part of this workshop will cover air combat.