3D GameStudio

Workshop
Vertex Space Flight

by Nicholas Chionilos / February 2001

Contents

Welcome to the Vertex Space Flight Workshop! I have produced this workshop to help people create their own 3D space games using 3D GameStudio. This workshop uses some features that became available with the recent (4.22) update.

This workshop, like the other workshops 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).

The **`spaceshp.wdl`** script that comes with the workshop was designed to simulate a ship flying in a zero-gravity environment. It comes with three different Camera views, four different control flags and 5 different parameters that can all be used to allow a great deal of flexibility in customizing the flight characteristics of your ship.

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.

In conclusion, I would like to offer my special thanks to:
- Remi, for introducing me to the concept of the Sky Sphere
- Kyodai, for his generous donation of the ship model used in this workshop
- Vivi, for the very cool skin on the ship
- JCL, for his donation of the Chase Cam script
- Doug, for his infinite patience and all around guidance.

I hope you enjoy this workshop! Any questions or comments can be sent to **Nicholas.Chionilos@Vertexconsulting.com** and I will do my best to answer them.

-Nick "WildCat" Chionilos.

Before we begin:

Get the latest version

Before you begin, make sure you have the latest version of 3DGameStudio (4.22 or greater) since we are going to make use of some of the new features that have been added.

Prepare your workspace

Create a folder called "**Space**" in your GStudio folder. This is the folder where all of your game elements will be stored.

Unzip the contents of **space.zip** into this folder. You folder should now contain the following files:

spaceshp.wdl
spaceshp.mdl
sphere.mdl
starmap.pcx
stars.mdl
engine.wav
thrustr.wav

Lets get started!

Ceating a space game is a little different from creating a shooter, a role playing game, or even a flight simulator. The biggest difference is that most conventional levels use a stationary "Sky Box" defined through WED to enclose the world, while a space level uses a model "Star Sphere" created in MED that moves as the ship moves to enclose the level.

We are going to create a space scene in the following steps:

1. Create a Star Sphere to define what your heavens look like
2. Create a large level in WED
3. Place a ship in the center of the world
4. Assign the **`player_ship`** action to it

Creating a Star Sphere

A Star sphere is simply a very large sphere model that will create the view of the stars that we see when we are in our space ship. We make it through the following steps:

1. Create a sphere model
2. Flip the sides on the polygons to effectively turn the sphere inside out.
3. Scale the sphere to a suitable size
4. Skin it with a space scene

While the theory of creating a star sphere is very simple. There are some practical considerations to take into account in order to create a professional looking star map.

- The more polygons used in your sphere, the less distortion you will see around the edges of your view as you turn in your game. I recommend between 500 and 1200 polygons.

- The larger your sphere, the farther away the polygons that make it up are from the camera, and thus the less likely you are to see the shapes of the individual polygons.

- The more uniform you keep the individual polygons the less likely you are to have anomalies in your star field.

MED is not necessarily the best place to start if you wish to build a sphere from scratch. The sphere primitive in MED has too few polygons, and if you increase the polygon count with the sub-divide tool, you get polygons of differing shapes.

For these reasons, I recommend that you either import a polygon from another model package like Milkshape, or use the **`sphere.mdl`** that I have included as a starting point for your star sphere.

Enough talk. Open your MED editor and let's create the heavens!

Step 1: Start with a Sphere in MED.

We will begin by loading the **sphere.mdl** included with this tutorial. This model has 720 polygons that are all the same shape, and is an excellent starting point for our sky sphere.

Open MED and select `file`→`open` and select **sphere.mdl** from your "**Space**" folder.

Your screen should look something like this:

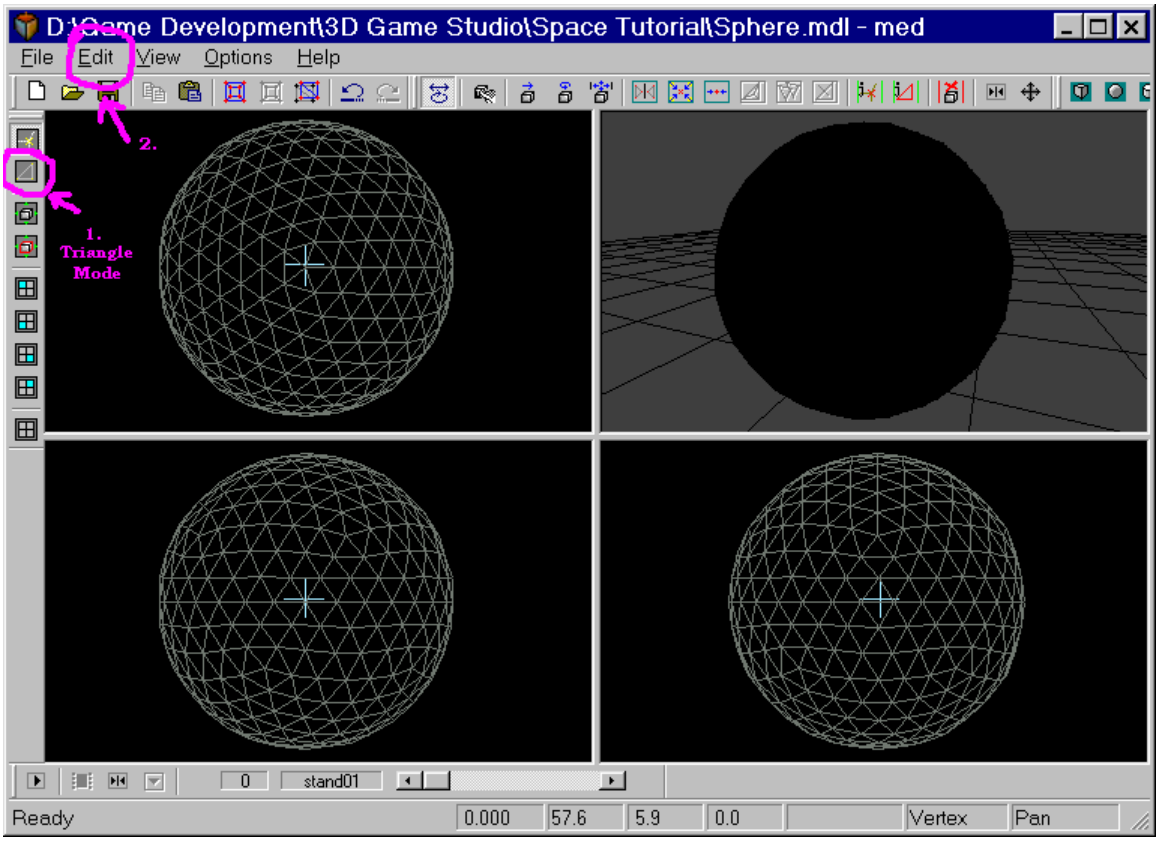

FIGURE 1: Sphere.mdl

Before we proceed, we want to be sure that we are working in Triangle Mode for our selections and manipulations for this model. Do this by clicking the triangle mode icon from the left tool bar. See Figure 1 for its location.

Go to the Edit menu and `select all`
You should now see all the polygons for the sphere highlighted in gold, as seen in Figure 2.

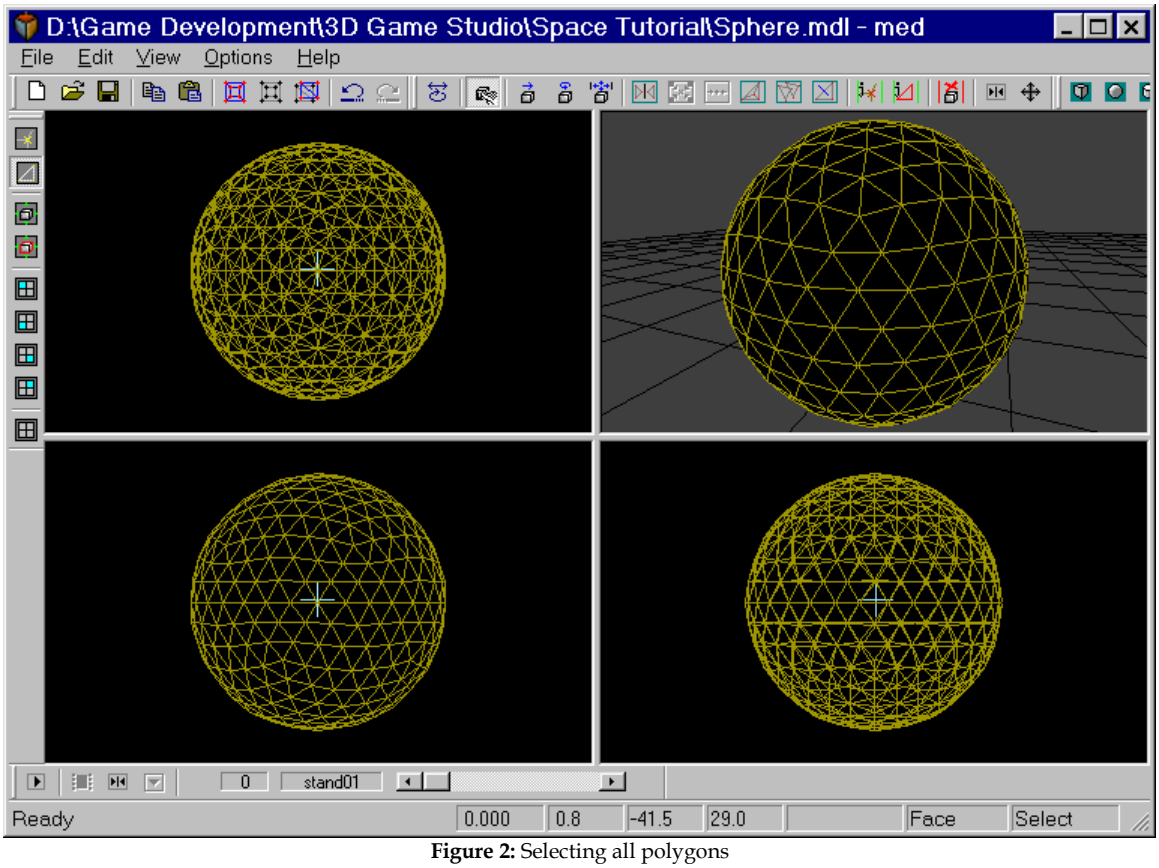

Figure 2: Selecting all polygons

Step 2: Flip Normals

A normal is simply the direction that the visible side of a polygon is facing. View the normals now by going to the `options` → `show normals` → `show all`. Your results should look like figure 3.

Notice that the model looks "prickly" now. This is because each polygon has a small line coming from its center and pointing away from the center of the model. This tells us that all polygons are currently pointed out.

The reason that we are so concerned with the direction that the normals are pointing to, is because the skin we apply to this sphere will only be visible from the direction that the normals are pointing to. If the normals point out, you can only see the skin of this object when looking at it from the outside. This would be fine if we wanted to make a planet that we would only view from the outside.

Since our ship will be traveling **inside** the sphere, we need to flip the normals so that they will be pointing into the center of the sphere. This way we will be able to see the skin that has our stars when our camera is inside the model. Do this by clicking the `flip normals` button on the tool bar, as shown in Figure 3.

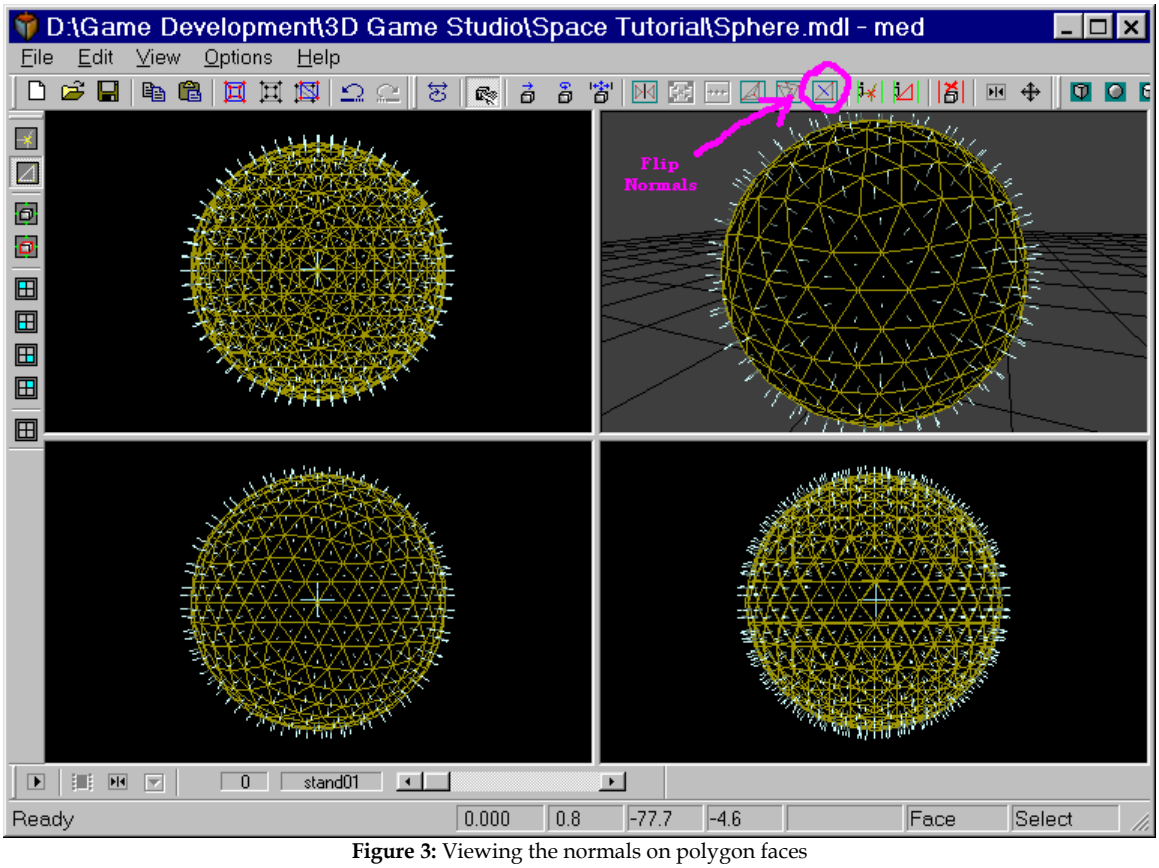

Figure 3: Viewing the normals on polygon faces

You should now have a sphere model with the normals all pointing inwards, as shown in figure 4.

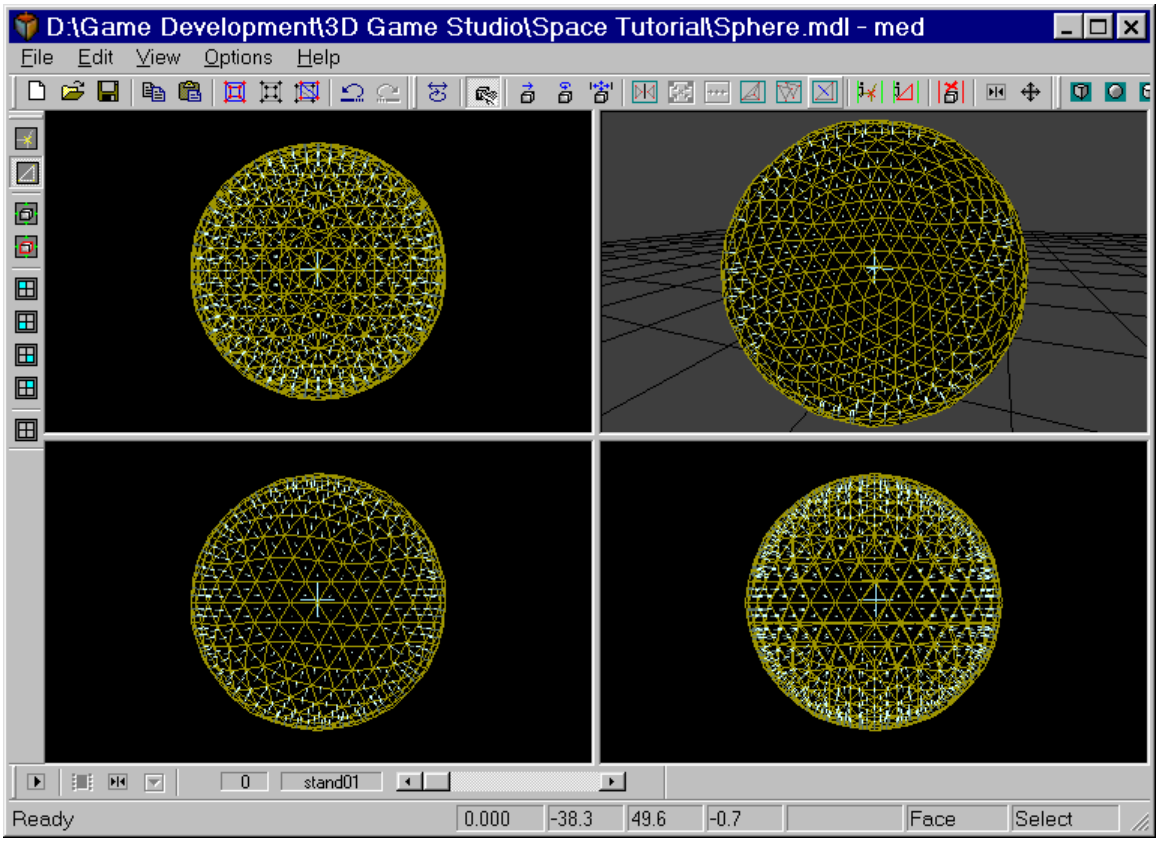

Figure 4: Sphere with Normals Flipped inward

You may now go back to `options` → `show normals` → `none`. This will get your regular view of the model back.

Step 3: Scale the model

Scale the model to make it big. I suggest about 12,000 units along each axis. Do this by selecting the Position tool from the tool bar as shown in figure 5. In the upper left window, right click and hold your mouse button and then drag your cursor down.

This will have the effect of pulling the camera far away from your sphere and making it look very small. (Test your distance by putting your cursor next to the right boundary of the top view. Make sure that the Y displacement is fairly large, around 12,000 as shown in figure 5)

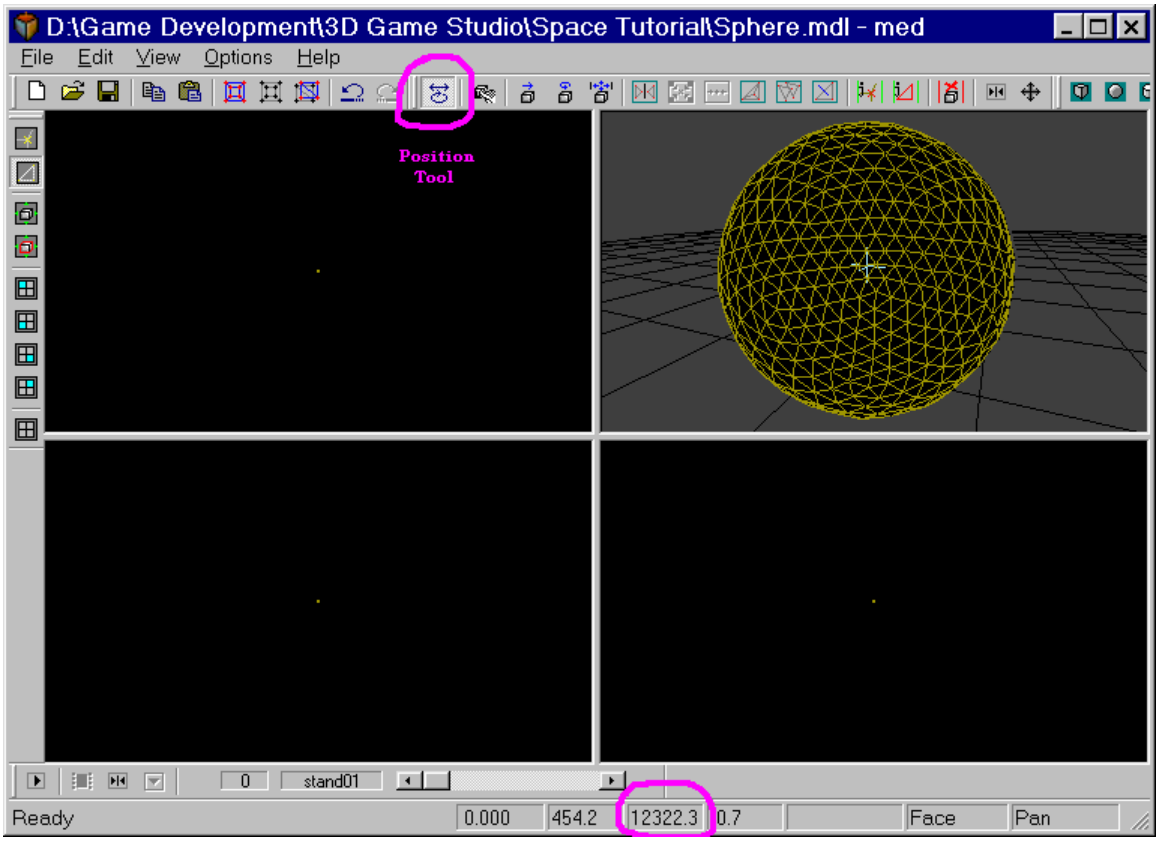

Figure 5: Camera positioned to scale the sphere

Next, select the scale tool, located 4 tools to the right of the position tool, as seen on figure 6. The entire sphere should still be selected, so simply left click and drag the mouse cursor down across one of the 2D views until it fills the view window.

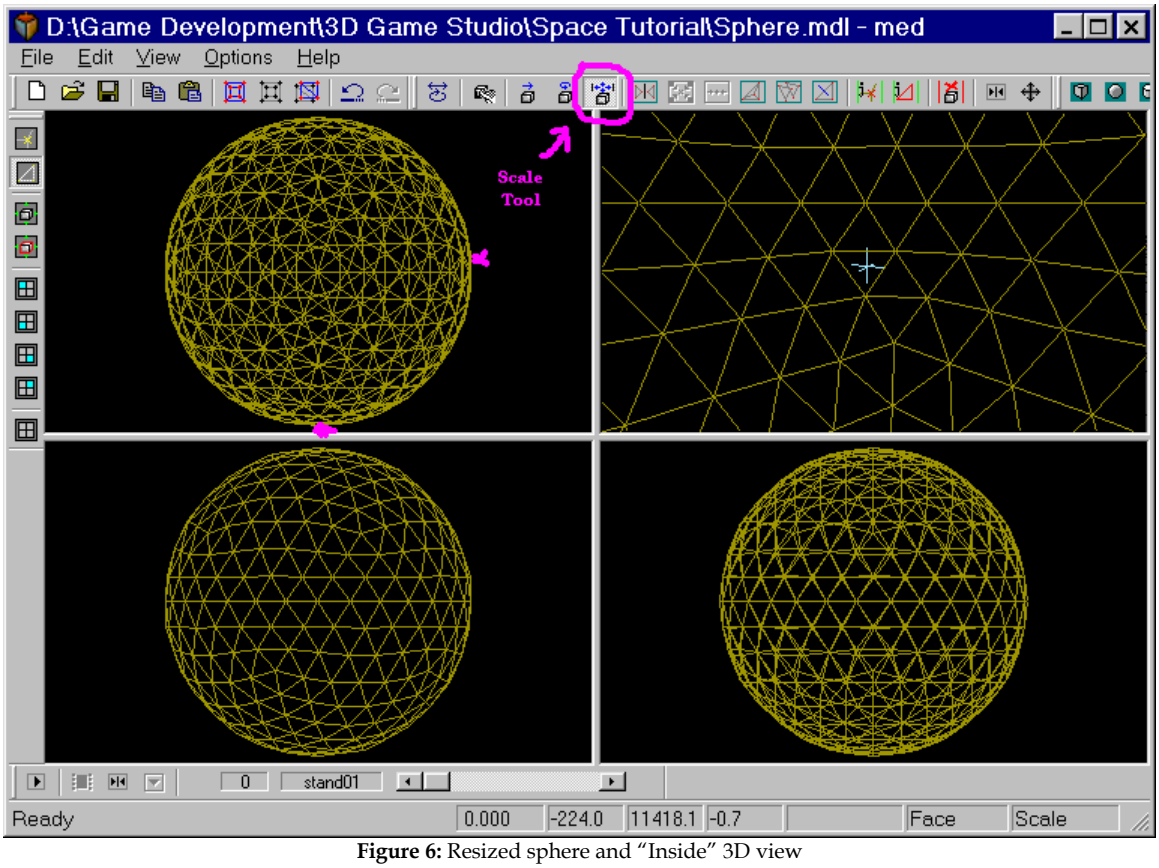

Figure 6: Resized sphere and "Inside" 3D view

Notice that the 3D view now appears to be inside the Sphere! Now we are getting somewhere!

Also notice the two splotches of color I put on the right and bottom edges of the sphere in the top view. When your cursor is near these positions, the displacement indicators on the bottom of the screen will give you a good estimate of how large you have made your sphere. The specific size is not too critical. Just make it relatively big. You can always go back and adjust it later after trying it out in your level.

Step 4: Skinning the Sphere!

The final step to creating the Star Sphere is to create a skin for it. In this example, I am going to use MDL style skinning to save time. Depending on how you lay out your heavens, this might produce a noticeable distortion of the stars at the seam between front and back. Using MD2 style skinning will fix this.

The space sphere that I have included with this tutorial, **`stars.mdl`** uses the MD2 style skin. To learn more about MD2 style skinning, I suggest you search on **MD2 Skin** on the Conitec user forum or check out some of the excellent skinning tutorials found on the net and through the Conitec links area.

To create our MDL style skin:

1) Click on the `view` menu, and select `skins` to bring up the skin editor.
2) In the skin editor, click on `edit`, then `resize skin`. In dialog box, set the size of the picture file that you will use for this skin. I have used 512 X 512 in this example. (Some older Video cards like the Voodo 2 may require a smaller bitmap of 256 X 256. Try this if you get an error about video memory)

3) Click on `edit`, then `create mdl mapping`. Select `front` and click `ok` in the dialog box. The polygon map of the front and back of the model sphere should now be laid out on the skin as seen in figure 7.

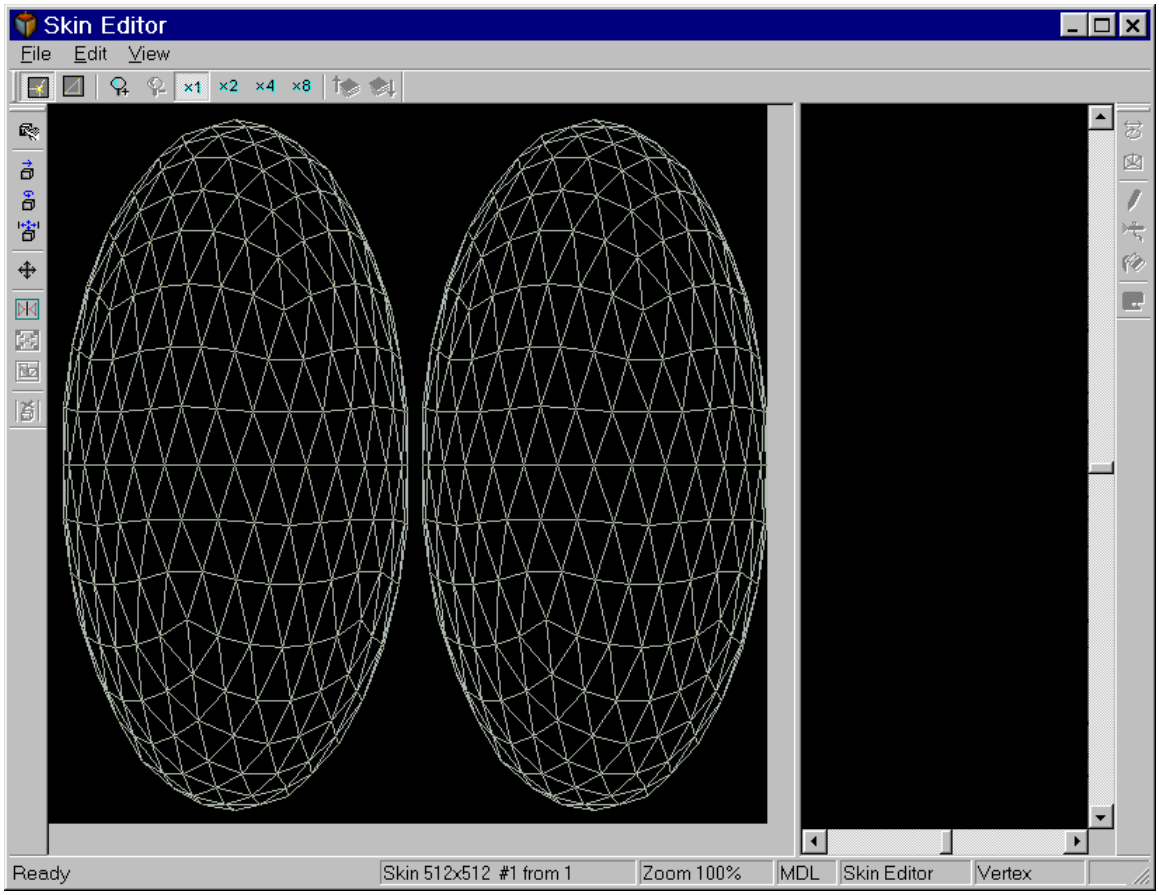

Figure 7: Exporting the MDL style skin map

4) Click on `file`, then `export` and then either `export to bmp` or `pcx`, depending on your preference.
5) Save your skin map
6) Open the skin map in the paint program of your choice. Keep it mostly black, with a sprinkling of stars and other Celestial phenomena. (I used the file **`starmap.pcx`** file that is included with this tutorial.)
7) Save the new skin map
8) Go back into the skin editor and click on File `import` → `skin image`. Use the skin image you just painted. You should now have something that looks like figure 8.

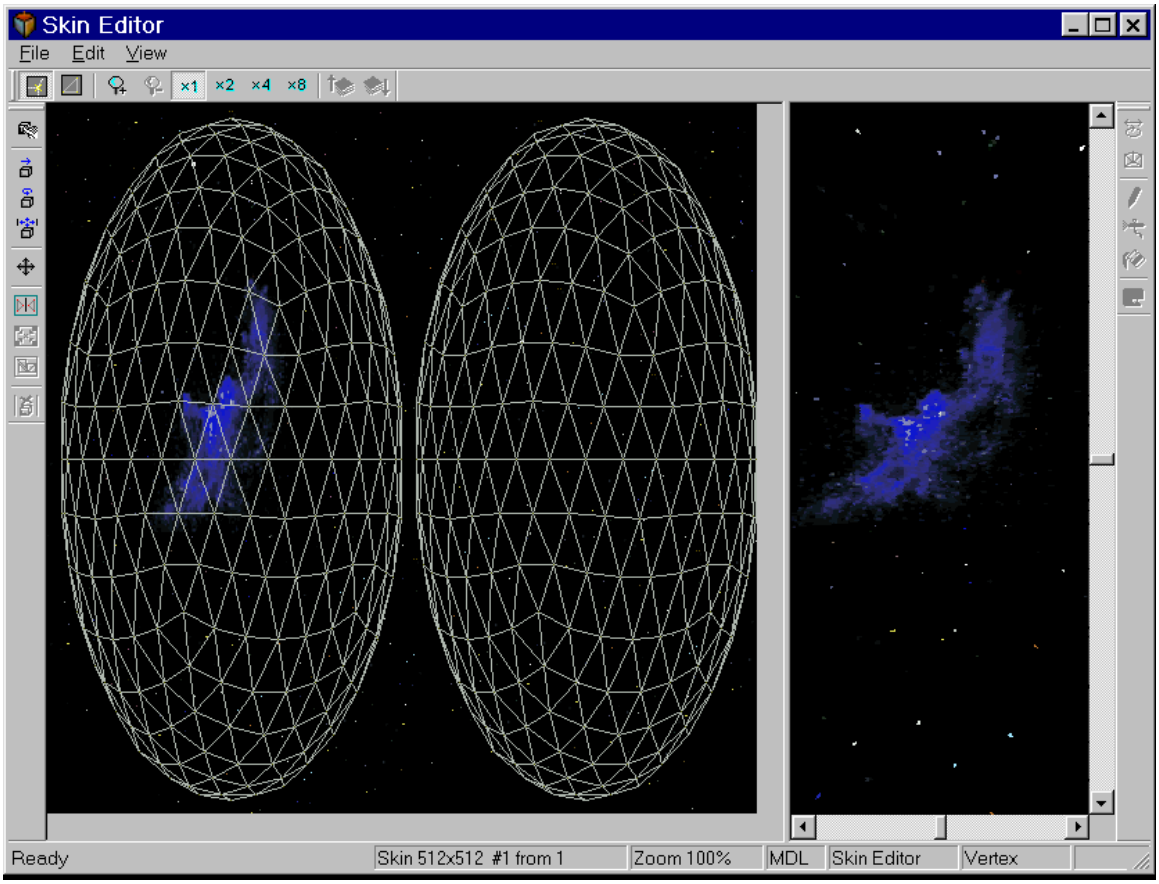

Figure 8: A completed Space Skin

Note that this is a 'quick and dirty' star sphere. The triangles of the mapping have obviously different sizes, and the stars in the flat image between them are not mapped onto the sphere. For a really good sphere we'd need more time to create a cylindrical mapping, like for the earth globe in an atlas.. However, space is dark and gracefully covers the holes and patches of our mapping. Close down the skin editor, and admire your work using the position tool in the 3D view!

Click on `file` → `save as` to save your completed Star model in your "**Space**" folder. Call it **`mystars.mdl`**

My heavens! You have just created… ***Your*** heavens!

To recap:

1) Create a sphere
2) Flip the normals
3) Scale it
4) Skin it!

Creating the Space Level

The space level is actually nothing more than an invisible bounding box with our space ship inside. Our view of the stars is tied to the position of our space ship. For this reason, our star sphere is created by the script of the spaceship, after the spaceship synonym is set. This way the sky will be taken care of automatically once we start the level and it will move as the ship moves, to create the impression of a vast empty space.

To create our space level, Open WED and select `file` →`new`.

Add a WAD to your level by selecting `texture` → `texture manager`

Click on the `add wad` button and select the WAD of your choice and then click the `open` button as seen in figure 9. I am using the **`standard.wad`** file.

Close the Texture Manager window when you are done.

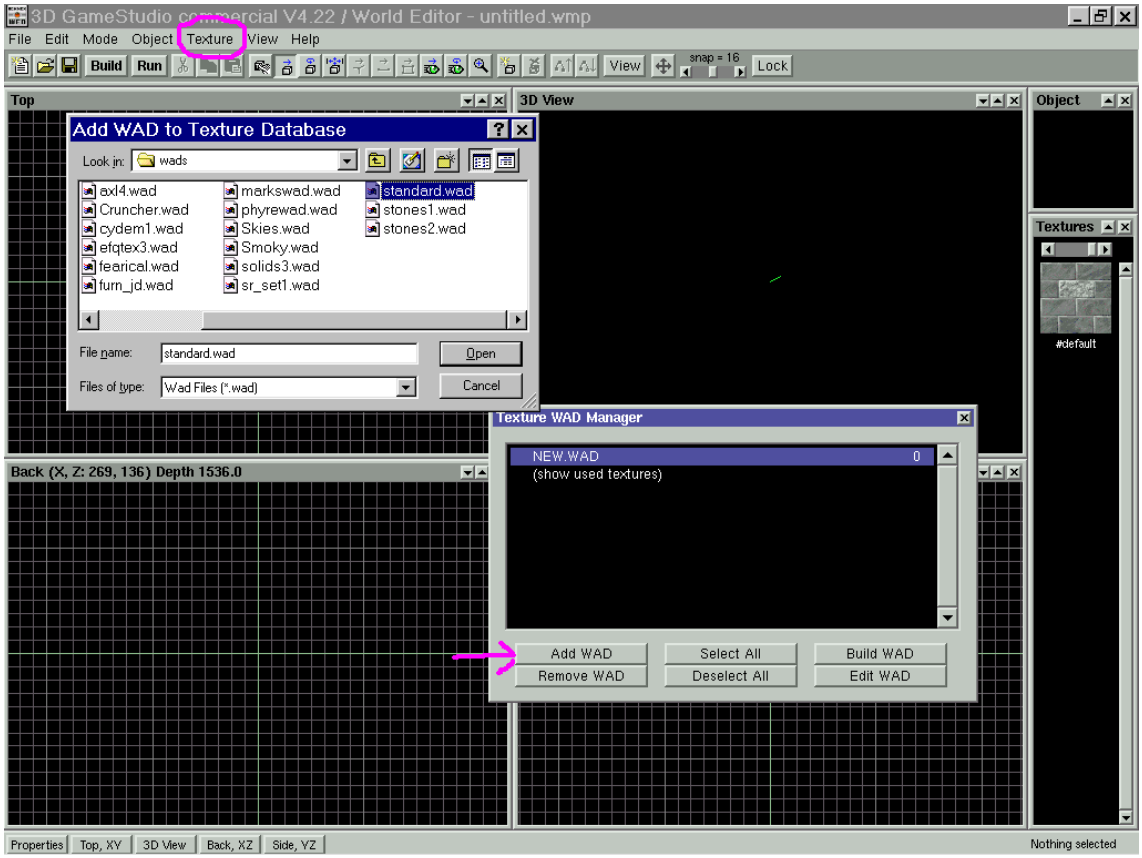

Figure 9: Adding a WAD to the level

Next, we will create our bounding box. Do this by Selecting `object` → `add primitive` →`cube large`

Scale the cube, making it approximately 30,000 quants on a side.

Do this by increasing the view depth for each window by using the [+]key. Expand each window's depth to the size that you are shooting for (In our case, 30080 as seen in figure 10.)

Next, select the `zoom eye` tool from the main tool bar, Left Click and drag the mouse down on one of the 2D screens. This will have the effect of zooming the camera **way** out so that we can see enough of our level to scale it correctly. (I zoomed out as far as it would let me.) Your 2D cubes will appear as tiny little dots at this point.

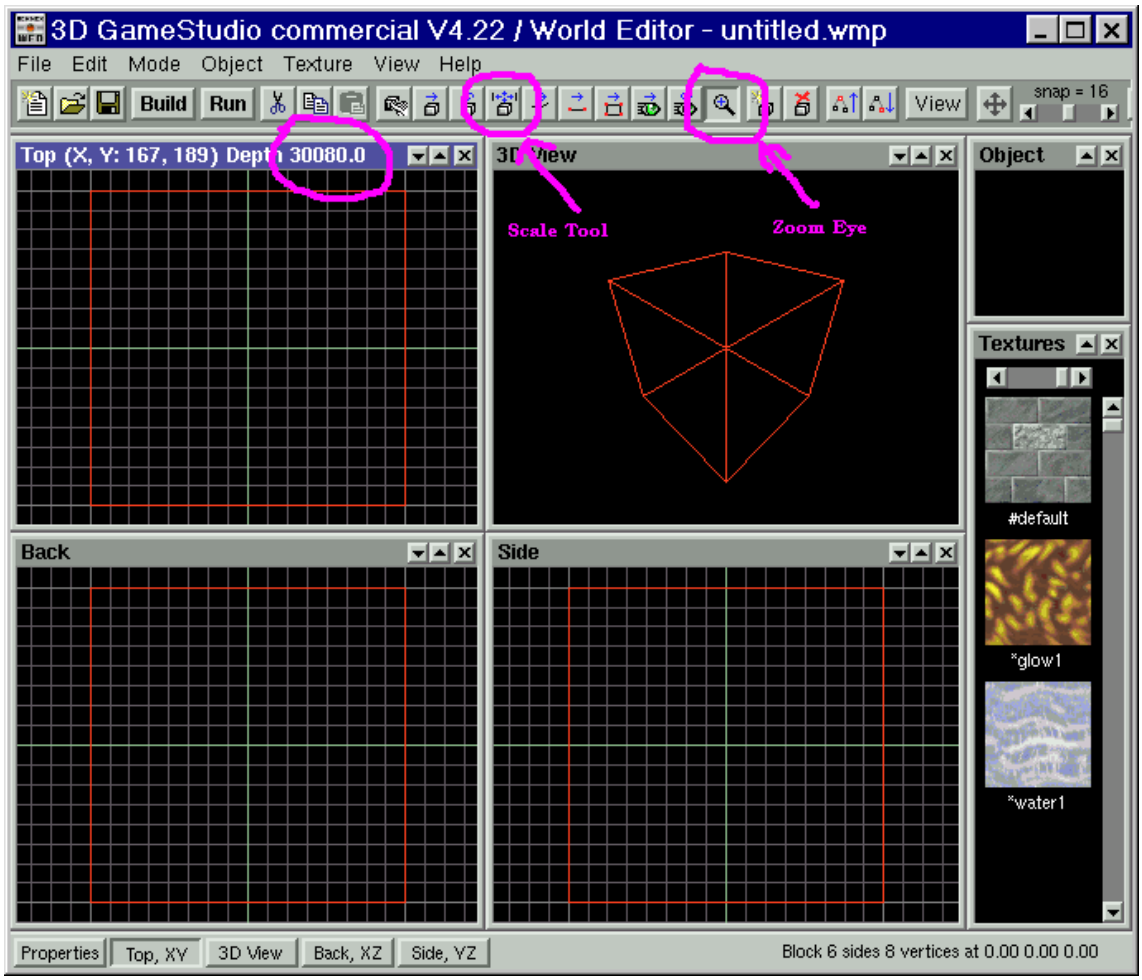

Figure 10: Setting the View depth

Now, select the `scale` tool and scale the cube in each window so that it is as large as it can be, and still be visible in the other views. It should look like Figure 11.

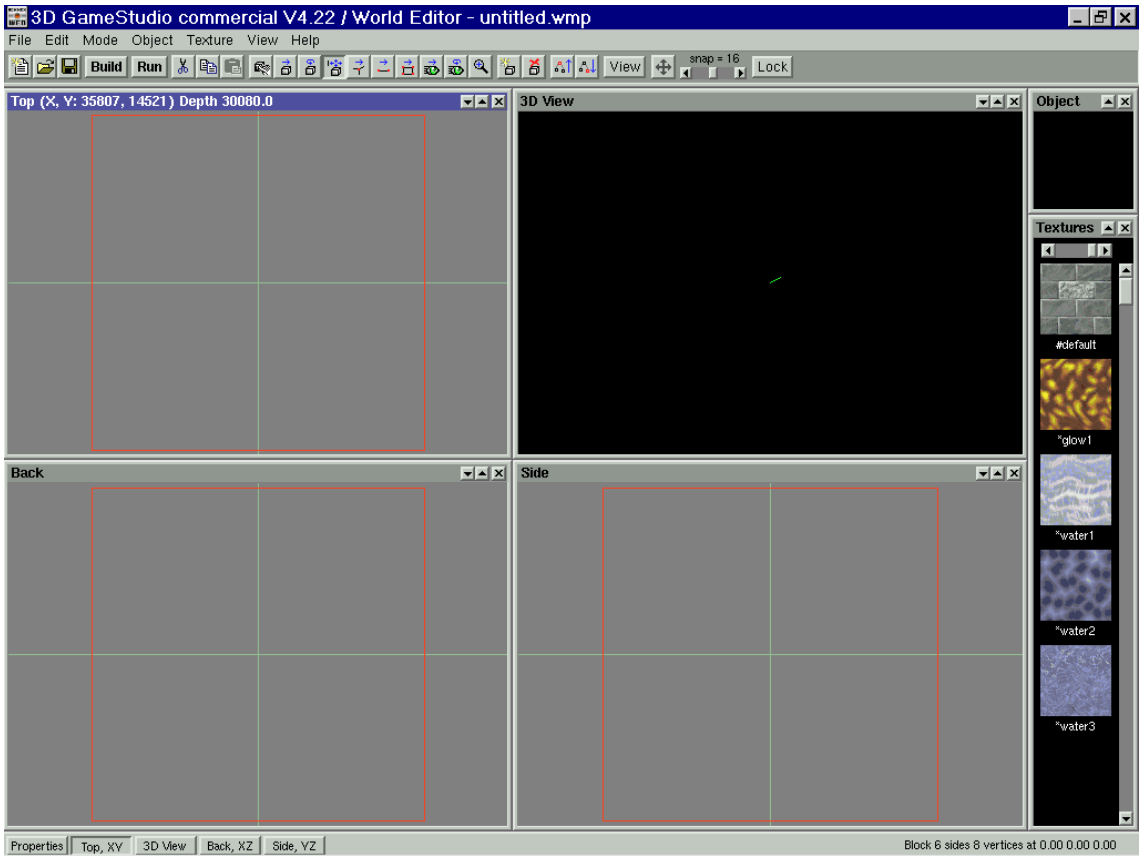

Figure 11: a box scaled to 30,000 quants

Next, hollow this cube with the [**Alt / H**]command. (Or go to `edit` → `hollow block` from the menus)

Note that we did not make our bounding box the max size of [f]50,000 quants. This is because our ship will always be positioned in the center of the star sphere, and the star sphere extends out about 12,000 quants.

If our ship is stopped at 30,000 quants, and our stars expand out for another 12,000 we have covered 42,000 quants... safely under our max level size. You want to make sure that parts of your star sphere don't drift past the magic 50,000 quant limit or you may have problems.

Apply the default texture to your bounding box by single Right clicking the texture with the mouse and then left clicking the `settings` selection. Be sure that the **`flat`** flag is set on the default texture before you apply it to the box as shown by figure 12.

f the latest edition of A5 comes with an increased limit of 100.000 quants

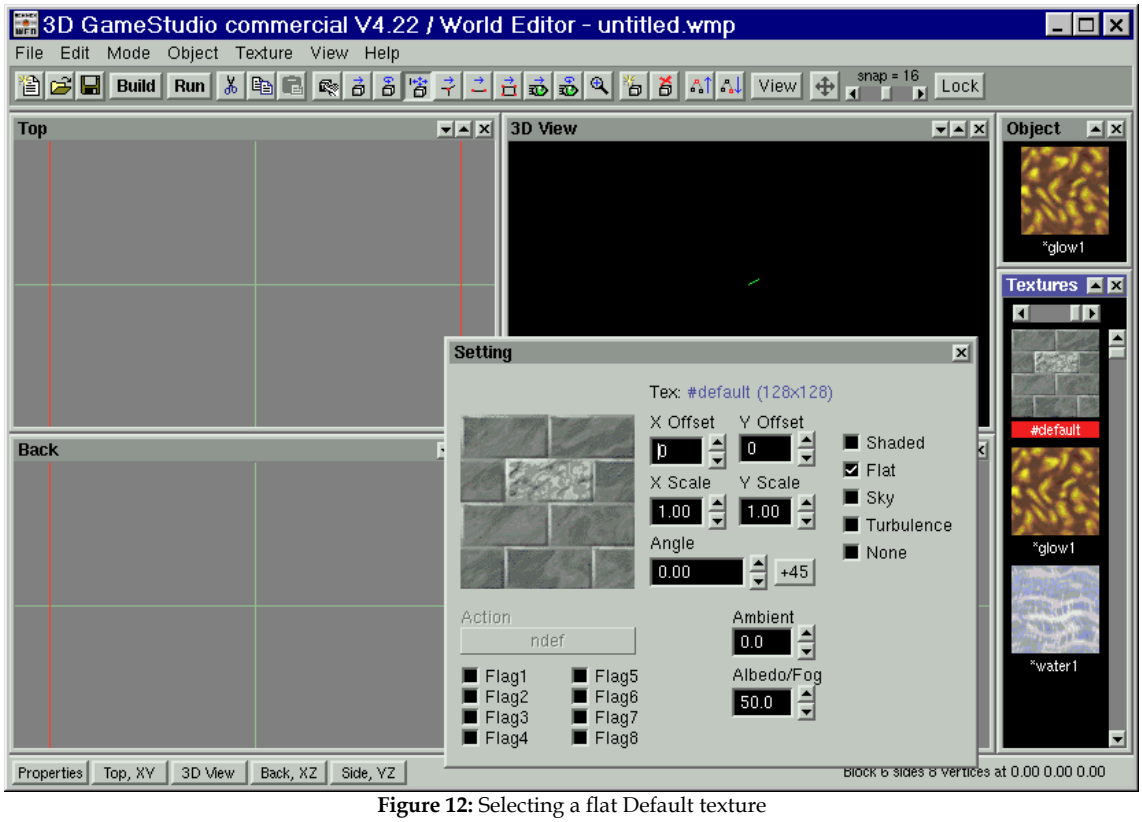

Figure 12: Selecting a flat Default texture

Now, that you have your box textured, Open the box's properties window by clicking on the properties button in the lower left corner of the screen. Make sure that the **`invisible`** flag is the only one set as seen in figure 13.

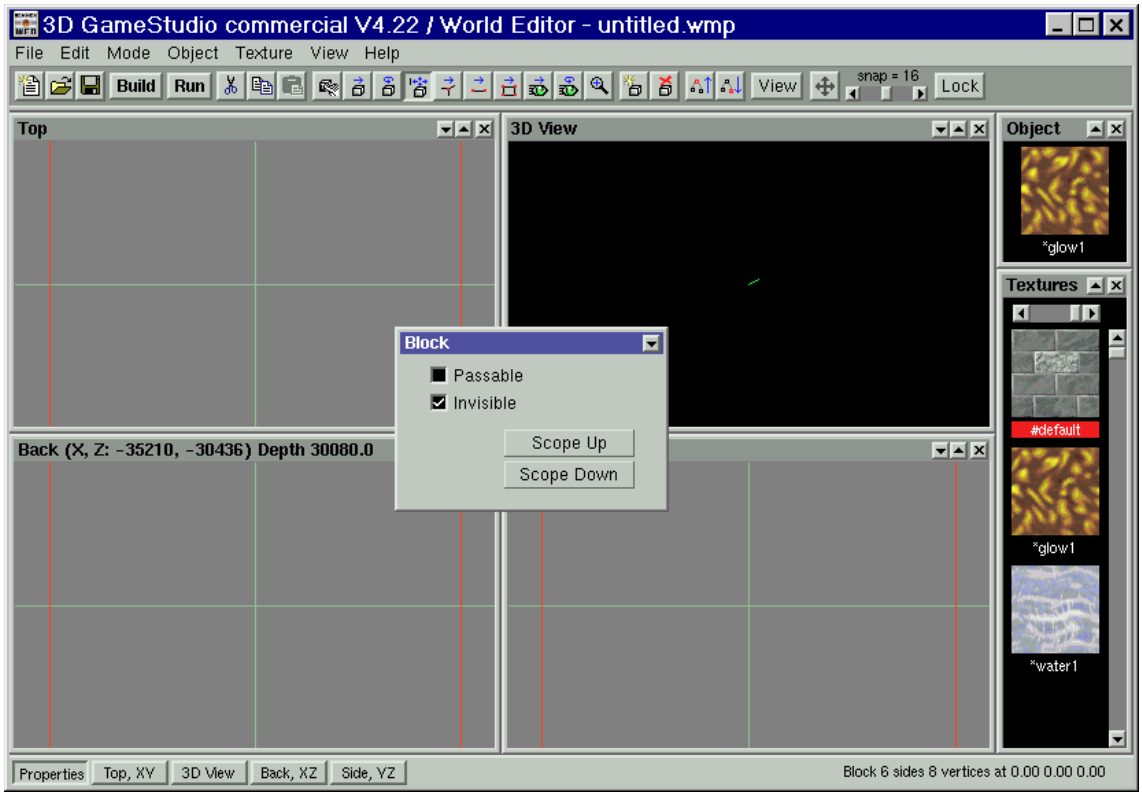

Figure 13: Setting the Invisible flag on the Bounding box

Congratulations! You have just created your bounding box, the edge of known space!

Now zoom down and add a number of primitives to your level by selecting `object` → `add primitive` and texture them as you wish.

The size, placement and textures on them are not particularly important, but make sure you have a number of them sprinkled about. It's especially important to have a few close to where you will put the ship so you can get a feel for how the ship is moving. One of the problems with space is that it's mostly empty. With no stationary reference points, it is impossible to tell if you are moving or not, and it is very easy to get lost.

Now, assign the **`spaceship.wdl`** Script to this level by Going into the `file` → `map properties` and assign the **`spaceship.wdl`** to this map as seen in figure 14 below.

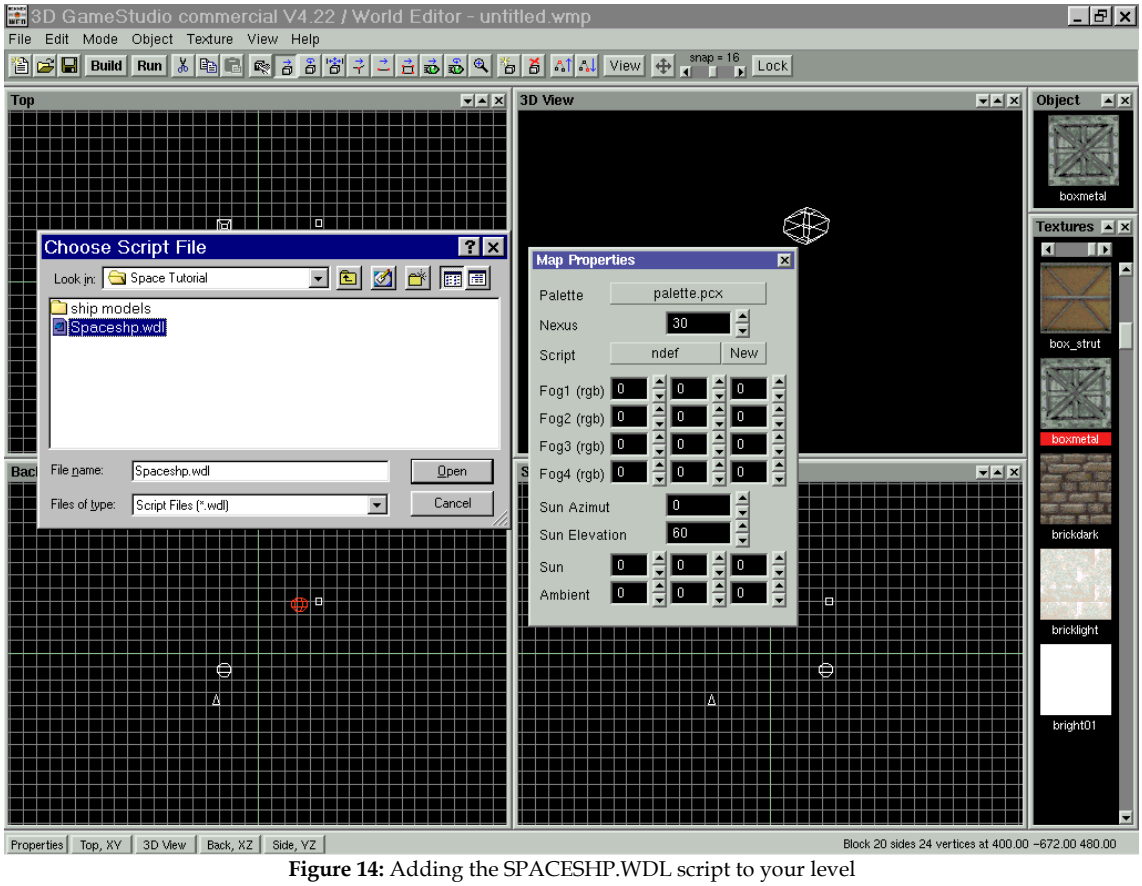

Figure 14: Adding the SPACESHP.WDL script to your level

We are in the home stretch now!

Just add a camera start position by going into `object` → `add start position`. Position the camera somewhere slightly off to the side but pointing towards the origin as seen in figure 15.

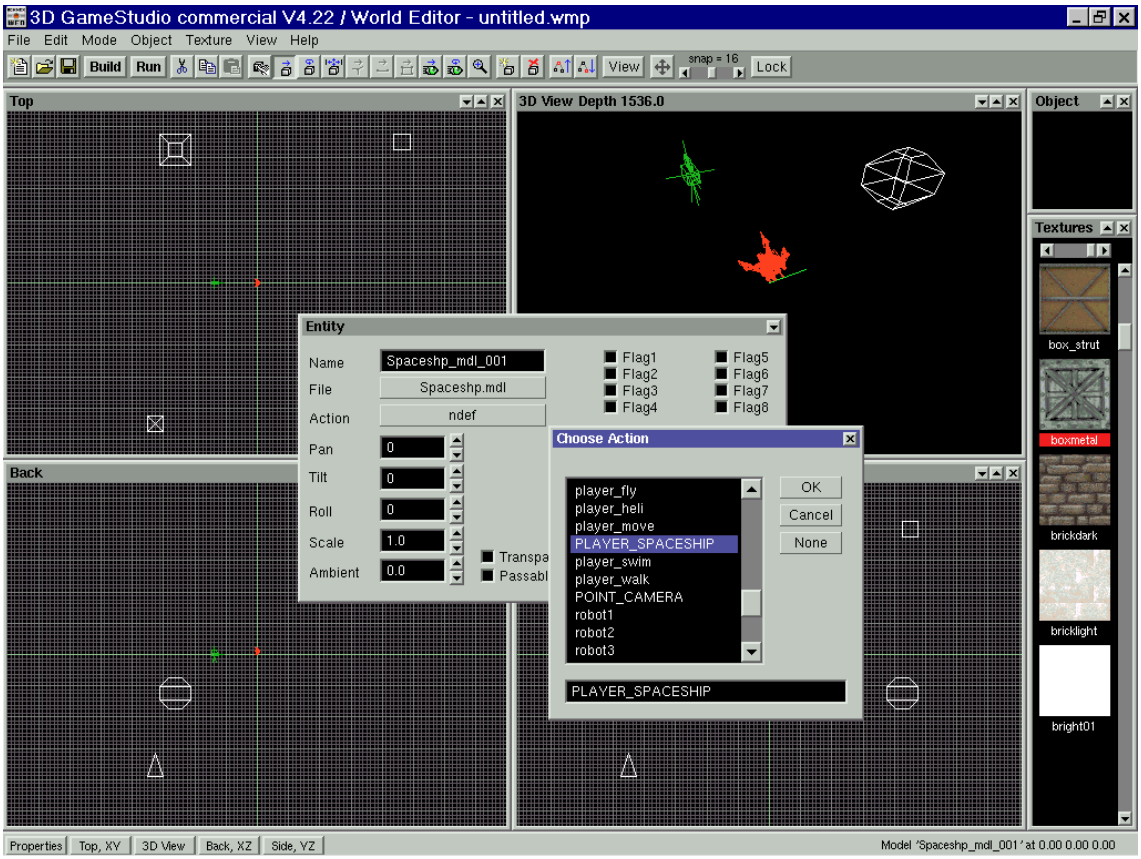

Figure 15: Assigning the camera and ship

The last thing that we have to do to complete our first space level is to add a ship!

Do this by Selecting `object→load entity` and use the file windows to find and add the **`spaceshp.mdl`** model to our level. Place it at the origin, and make sure that your camera position is point at it.

Then, right click on the ship, and bring up its properties box. Click the `action` button and then scroll through the list until you can select **`player_spaceship`**. Click `ok`.

Congratulations! You have just finished building your first space level!

Save the level in your "Space" folder and call it **`bigspace.wmp`**

`build` the level with the Level Map selected, and then run the level.

If you have done everything correctly, you Should see your spaceship floating in space!

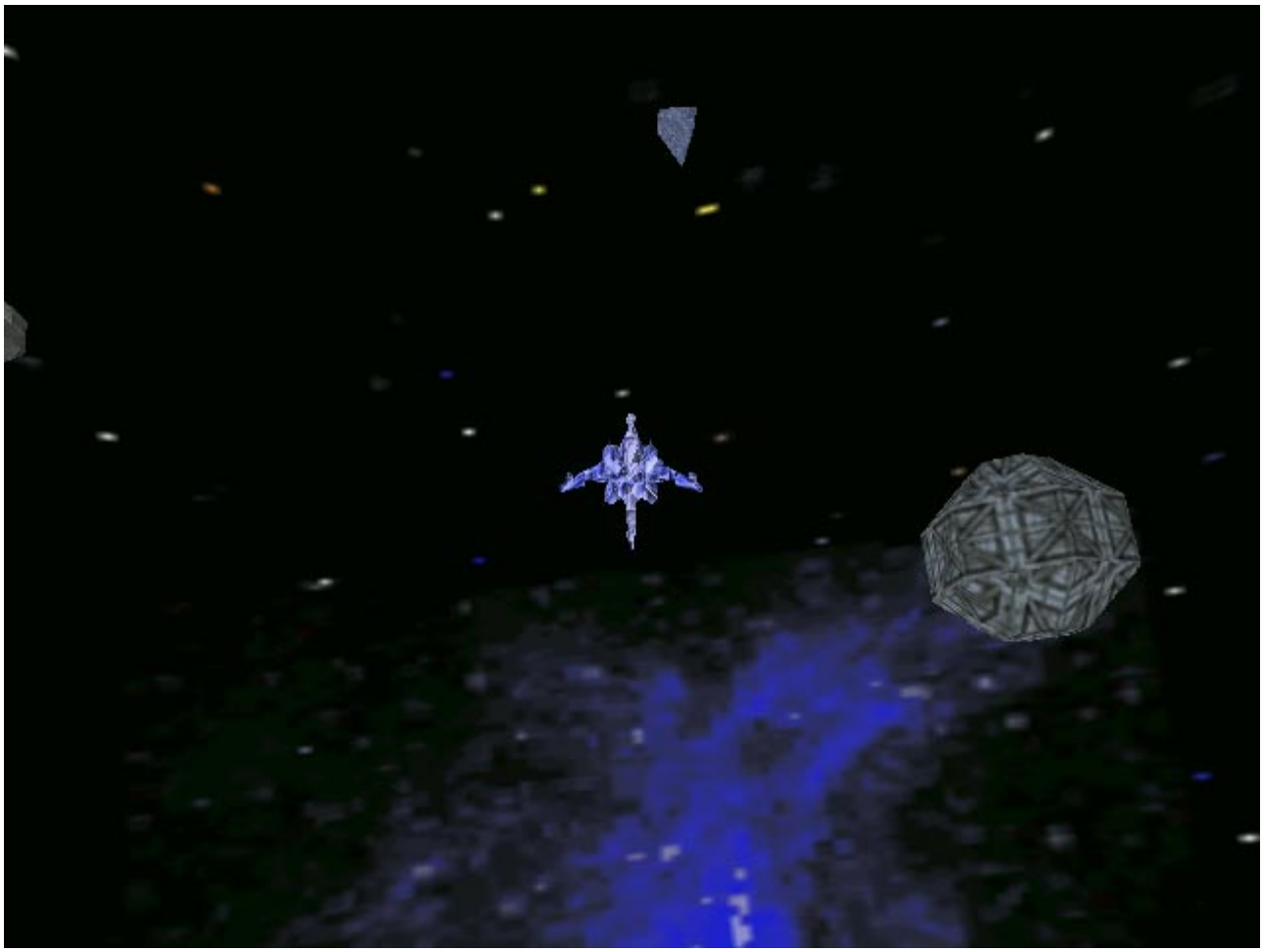

Figure 16: Our final Space level!

Play around with the ship for a while. The controls are simple:

- Use Left and Right arrows [🡐] / [🡒] to pan your ship
- Use Up and Down arrows [🡑] / [🡓] to control your ship's Tilt
- Use the [**Space**] bar to fire engines

Customizing the Space Flight

Once you have gotten familiar with the ship and how it handles, ****Backup your `spaceshp.wdl` script**** and then try making some changes to the parameters and variables set in the `player_spaceship` action of your `spaceshp.wdl` script.

Open the `spaceshp.wdl` script in your favorite editor and modify the variables listed below to suit your own needs. Remember to save your changes and then run your level to see your changes at work.

You may change the following variables within the script:

camera_type

Is initially set to **1**, the "**Chase Cam**" mode, where the ship looks like it is stationary, and the world seems to be in motion. This script was donated by JCL and will be improved in the near future. When set to **2** you will see the view from the cockpit and when set to **3** you can view your ship from the vantage of a fixed camera.

my.auto_spin_stop
Is initially set to **1**, which will automatically slow your spin when you are not actively applying thrusters. When set to **0**, the ship will spin until you manually apply thrusters for the opposite direction.

my.auto_decel
Is initially set to **1**, which will allow the ship to come to a stop. When set to **0**, the ship will continue to drift on it's current vector.

my.limit_turn_speed
Is initially set to **1**, which will limit how fast you spin based on the value of the **my.max_spin_speed** variable. When set to **0**, there is no turn limiting in effect.

my.limit_top_speed
When set to **1**, will impose a speed limit on each vector of the thrust based on the value of **my.max_ship_speed**. When set to **0**, there is no speed limit.

my.engine_thrust
Defines the effective power of the "**Main Engines**". Larger values make more thrust. The default value is **.5**.

my.spin_rate
Defines the effective power of the "**Turning Thrusters**". Larger values make a faster spin (and make it harder to control the ship!) The default value is **.15**.

my. decel_rate
Defines the effective power of "**Friction"** on the ship in space. I know, not horribly realistic, which is why you can turn it off by setting the **my.auto_decel** flag to **0**, but it does make the ship much easier to handle. The default value is **.04**.

my. max_spin_speed
Defines the effective top speed that the "**Turning Thrusters"** will still fire at. Larger values allow a faster top speed for the spin rate of the ship. You can turn this option off by setting the **my.limit_turn_speed** flag to **0**. The default value is **5**.

my. max_ship_speed
Defines the effective top speed for each vector of the ship's motion. Larger values allow the ship to achieve faster top speeds. You can turn this option off by setting the **my.limit_top_speed** flag to 0.The default value is **20**.

That's a wrap!

This concludes the Vertex Space Flight Workshop. I hope you enjoyed it! This workshop only scratches the surface of space based games. It is constructed on a fairly realistic physics model, and can be lots of fun to work with, but there is also a lot of room for improvement. While this is an excellent springboard to make a space simulator type game, you might want a little less realism and a little more control to make an Arcade-type space game! I'd love to see what people come up with.

Here are a few ideas for things that can be added to spice up your extra-terrestrial activities: adding a panel for a cockpit (when in that view!), adding weapons, adding commands to control the roll of your ship, prettier star maps, nicer features (including planets!) built in WED. Landing, docking, and take-off sequences...

Also, if you come up with a particularly interesting flight profile for a ship, post it on the Forum so other people can enjoy it too!

One last thing... Look for the "Vertex Space Tool" to be landing on the Conitec Download page in the near future. It is a panel based interface that allows you to change all of the flags and settings "On the fly"!

The sky is no longer the limit!

- Nick "WildCat" Chionilos

For those brave souls who wish to peek under the hood and see what makes our space ship tick, I have included **Appendix** for detailed comments on the script. It assumes a pretty solid understanding of WDL, and in some places, a bit of math unless you are willing to take my word on some things!

APPENDIX : Details on the SPACESHP.WDL script

```
#                       Welcome to the Space Template!
#
#       Contols are very simple:
#       Use Left and Right arrows to pan your ship
#       Use Up and Down arrows to control your ship's Tilt
#       Use the SPACE BAR to fire engines
#
#    NOTE: User Changeable fields are bold and blue - like this -
#
////////////////////////////////////////////////////////////////////////////
//      A4 main wdl
////////////////////////////////////////////////////////////////////////////
// The PATH keyword gives directories where game files can be found,
// relative to the level directory
path    "models";               // Path to model subdirectory - if any
path    "sounds";               // Path to sound subdirectory - if any
path    "bmaps";                // Path to graphics subdirectory - if any
path    "images";               // Path to images subdirectory
path    "..\\template";         // Path to WDL templates subdirectory


////////////////////////////////////////////////////////////////////////////
// The INCLUDE keyword can be used to include further WDL files,
// like those in the TEMPLATE subdirectory, with prefabricated actions
////////////////////////////////////////////////////////////////////////////

include <movement.wdl>;
include <messages.wdl>;
include <particle.wdl>;
include <doors.wdl>;
include <actors.wdl>;
include <weapons.wdl>;
include <war.wdl>;
include <menu.wdl>;

// Set the starting resolution for the system
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


// The MAIN function is called at game start and gets everything going

function  main()
{
        load_level <bigspace.wmb>;
        load_status();          // restore global variables
        wait 16;
}
```

```
////////////////////////////////////////////////////////////////////////////
//      SPACE SCRIPT BEGINS HERE
////////////////////////////////////////////////////////////////////////////

// Sounds:

        sound main_engines,<engine.wav>;
        sound thrusters,<thruster.wav>;
        var thrust_handle = 0;   // Sound handles let you control when sound
        var engine_handle = 0;   //      is playing

// Synonyms:

        synonym player_ship {type entity;}
        synonym star_sphere {type entity;}


//      The ACCEL_VECTOR tells the ship in which direction the thrust from the
//      engines is coming. Because the ship was built with it's front pointing
//      down the "X" axis, the thrust from the engine will always act on the "X"
//      component of the ACCEL_VECTOR, which I have renamed ENGINE_THRUST for
//      clarity.

        define accel_vector,skill1;
        define engine_thrust,skill1;


//      While the ACCEL_VECTOR applies thrust relative to the SHIP'S orientation,
//      the DRIFT_VECTOR applies the existing speed relative to the worlds
//      orientation.

        define drift_vector,skill4;
        define drift_vector_x,skill4;
        define drift_vector_y,skill5;
        define drift_vector_z,skill6;

        define ship_angles,skill7;       // Pan, Tilt and Roll speeds for ship
        define pan_speed,skill7;         // How fast you are panning
        define tilt_speed,skill8;        // How fast you are tilting
        define roll_speed,skill9;        // How fast you are rolling
        define spin_rate,skill10;        // The acceleration of your spin


// *** Note: While the coefficient of friction in space is for all practical
//      purposes **ZERO**, using a small value in DECEL_RATE or putting a cap on
//      the ships speed and spin can make it much easier to handle the ship or
//      tune your game play. Think of it as the Navigation computer automatically
//      taking the appropriate actions when the ship is not under active control.
//      These fields are used when their corresponding flag is turned on.

// How fast "Friction" will slow you down

        define decel_rate,skill11;

// Cap for turn speed if LIMIT_TURN_SPEED flag is set
        define max_spin_speed,skill12;

// Cap for ship speed if LIMIT_TOP_SPEED flag is set
        define max_ship_speed,skill13;

//*** Define Flags for space ship flight characteristics
        define auto_spin_stop,flag1;     //    These flags should all be off for
        define auto_decel,flag2;         //    maximum realism, but can be
        define limit_turn_speed,flag3;   //    useful to make a ship "Handle"
        define limit_top_speed,flag4;    //    better.


//define star_sky,<stars.mdl>;           // Default SPHERE MODEL from tutorial
define star_sky,<mystars.mdl>;                 // New SPHERE MODEL that users created
```

```
// ************ Working Variables start here ******************
var ship_angs[3];               // Used in angle computations for pan and tilt
var engine_work_speed[3];       // Used in speed and drift calculations


// Camera Variables
   var camera_type;
  var camera_spot[3];
  var workang[3];


// The space sphere simply moves with the player space ship. By changing the
//      spheres X, Y, and Z coordinates directly through it's POS variable, instead
//      of using the MOVE command, we eliminate all collision checking, which is
//      what we want. The Star Sphere should be able to go through the bounding box of
//      the level. The Ship should not!

action space_sphere
{
        star_sphere = me;

        while (1)       // Constantly keep the player at the center of the universe
        {
                vec_set(my.pos,player_ship.pos);
                wait 1;
        }
}


action player_spaceship
{
// Set Camera mode
        camera_type = 1;      //**** 1 = Chase Cam, 2 = Cockpit, 3 = Stationary ****

// Set Synonym for future use and Zero out all ship speeds
        player_ship = me;
        my.narrow = on;                         // Use the smaller collision hull
        vec_set(my.accel_vector,nullvector);    // No starting speed
        vec_set(my.drift_vector,nullvector);    // No starting momentum
        create star_sky,my.pos,space_sphere;    // Initialize background stars


//*********************************************************
// All special flight flags are on to start. Set them to 0 to turn them off.

        my.auto_spin_stop = 1;      // Let the ship stop spinning by itself
        my.auto_decel = 1;          // Let the ship slow down by using MY.DECEL_RATE
        my.limit_turn_speed = 1;    // Limit how fast you spin based on MY.MAX_SPIN_SPEED
        my.limit_top_speed = 1;     // Ship "speedlimit" based on MY.MAX_SHIP_SPEED

// Starting flight Characteristics for ship:

        my.engine_thrust = .5;              // Strength of the main engines
        my.spin_rate = .15;                 // Strength of the steering thrusters
        my.decel_rate = .04;                // "Drift" deceleration rate
        my.max_spin_speed = 5;              // Top speed for changes in ships orientation
        my.max_ship_speed = 20;             // Top speed the ship can go

//*********************************************************


        while (1) {

# *****Turning thruster logic here:
#
#       If you fire a thruster and you are using LIMIT_TURN_SPEEDS then check
#           to make sure you are still under the speed limit. Do this for the
```

```
#          Left (CUL), Right (CUR), Up (CUU), and Down (CUD) cursor keys

               if (key_cul == 1)
                   {if ((my.limit_turn_speed == 0) ||
                        (my.limit_turn_speed == 1) &&
                        (my.pan_speed < my.max_spin_speed))
                       {my.pan_speed += my.spin_rate * time;
                         if (thrust_handle == 0)               // no sound playing
                             {play_loop thrusters,15;          // Play the thruster sound
                               thrust_handle = result;}}}

               if (key_cur == 1)
                   {if ((my.limit_turn_speed == 0) ||
                          (my.limit_turn_speed == 1) &&
                          (my.pan_speed > -my.max_spin_speed))
                       {my.pan_speed -= my.spin_rate * time;
                         if (thrust_handle == 0)               // no sound playing
                             {play_loop thrusters,15;          // Play the thruster sound
                               thrust_handle = result;}}}

               if (key_cuu == 1)
                   {if ((my.limit_turn_speed == 0) ||
                          (my.limit_turn_speed == 1) &&
                          (my.tilt_speed < my.max_spin_speed))
                       {my.tilt_speed += my.spin_rate * time;
                         if (thrust_handle == 0)               // no sound playing
                             {play_loop thrusters,15;          // Play the thruster sound
                               thrust_handle = result;}}}

               if (key_cud == 1)
                   {if ((my.limit_turn_speed == 0) ||
                          (my.limit_turn_speed == 1) &&
                          (my.tilt_speed > -my.max_spin_speed))
                       {my.tilt_speed -= my.spin_rate * time;
                         if (thrust_handle == 0)               // If no sound playing
                             {play_loop thrusters,15;          // Play the thruster sound
                               thrust_handle = result;}}}

               if ((thrust_handle != 0) &&      // If sound is playing...
                       (key_cuu == 0) &&
                       (key_cud == 0) &&        //      ...and...
                       (key_cul == 0) &&
                       (key_cur == 0))          //  No arrow keys were pressed
                               {stop_sound thrust_handle;       // stop the sound
                                thrust_handle = 0;}

//******* If using Auto_Spin_Stop, Slow the ship's spin when not actively stearing.

               if (my.auto_spin_stop == 1)
                       {stop_rotation();}


//********Main Engine firing is here:

               if (key_space == 1)              // if the space bar is pressed
                        {if (engine_handle == 0)         // If no sound playing, then play
                           {play_loop main_engines,25;
                                 engine_handle = result;}

//         1) Set ENGINE_WORK_SPEED to the current thrust
//         2) Vec_Rotate will convert the thrust relative to the ship's orientation
//                (which is only the x axis) and convert it into the
//               X,Y,& Z components relative to the world's orientation.
//
// Note: Steps 1 and 2 could also have been done using trigonometry to convert
//      to the world orientation via the following lines:
//         X:   ENGINE_WORK_SPEED[0] += (MY.ENGINE_THRUST * COS(MY.TILT)
//                              * COS(MY.PAN));
```

```
//          Y:   ENGINE_WORK_SPEED[1] += (MY.ENGINE_THRUST * COS(MY.TILT)
//                                * SIN(MY.PAN));
//          Z:   ENGINE_WORK_SPEED[2] += (MY.ENGINE_THRUST * SIN(MY.TILT));
//
//      Using the VEC_ROTATE seemed a little easier on the eye.


                        vec_set(engine_work_speed,my.engine_thrust);
                        vec_rotate(engine_work_speed,my.pan);

//      If there is no speed limit, then apply the thrust, If there is a
//         speed limit, apply only the components of thrust that have not maxed out

                        if (my.limit_top_speed == 0)
                                {vec_add(my.drift_vector,engine_work_speed);}
                        else
                                {if (abs(engine_work_speed.x + my.drift_vector_x) < my.max_ship_speed)
                                     {my.drift_vector_x += engine_work_speed.x;}

                                 if (abs(engine_work_speed.y + my.drift_vector_y) < my.max_ship_speed)
                                     {my.drift_vector_y += engine_work_speed.y;}

                                 if (abs(engine_work_speed.z + my.drift_vector_z) < my.max_ship_speed)
                                     {my.drift_vector_z += engine_work_speed.z;}}}
                else
                        {stop_sound engine_handle ;       // stop playing sound
                         engine_handle = 0;}

//**** Set it all in motion
        my.roll_speed = 0;                          // We don't use ROLL as an active control
        ROTATE ME,MY.SHIP_ANGLESS,NULLVECTOR;    // Rotate the ship based on our pan/tilt rates

// If you are not actively firing the engines, and Auto_decel is active, slow down the ship
                if ((key_space != 1) && (my.auto_decel == 1))
                        {call decel_ship;}

// calculate a distance from the speed
                vec_set(temp,my.drift_vector);
                vec_scale(temp,time);
                move(me,nullvector,temp);  // Then move the ship according to Drift

                point_camera();  // Adjust the camera
                wait 1;}
}

function stop_rotation()
{
//   **** If you are still rotating, apply a rotation in the opposite direction to slow down
//

        if (player_ship.pan_speed == 0)
            {goto checktilt;}
        else
            {if ((key_cul == 0) && (key_cur == 0))
                {if (player_ship.pan_speed > 0)
                    {player_ship.pan_speed -= player_ship.spin_rate * time;}
                 else
                    {player_ship.pan_speed += player_ship.spin_rate * time;}}
                 if (abs(player_ship.pan_speed) < .02)
                        {player_ship.pan_speed = 0;}}
checktilt:

        if (player_ship.tilt_speed == 0)
            {return;}
        else
            {if ((key_cuu == 0) && (key_cud == 0))
                {if (player_ship.tilt_speed > 0)
                    {player_ship.tilt_speed -= player_ship.spin_rate * time;}
                 else
```

```
                   {player_ship.tilt_speed += player_ship.spin_rate * time;}}
                if (abs(player_ship.tilt_speed) < .02)
                       {player_ship.tilt_speed = 0;}}
}
}

# DECEL_SHIP works like this:
# *** If you have momentum:
#    1) Set ENGINE_WORK_SPEED to have the decel rate as it's "X" component of thrust
#               (It's negative because we are counter-thrusting to slow down,
#                not thrusting to speed up)
#    2) Figure out the pan and tilt angles that correspond to the existing drift
#               vector using Vec_to_angle.
#    3) Use Vec_Rotate to convert the thrust from the Ship's perspective to the real
#               world's perspective.
#    4) Use the Vec_Scale command to time correct these speeds based on frame rate
#    5) Add this new Deceleration Vector to the existing Drift Vector to slow down
#    6) If you are going really slow anyway ( < .02), set your speed to zero
# *** If you are stopped: do nothing and return

action decel_ship
{
        if (vec_length(player_ship.drift_vector) != 0)            // If I have momentum
                {vec_set(engine_work_speed,nullvector);
                 engine_work_speed.x = -player_ship.decel_rate;
                 vec_to_angle(ship_angs.pan,player_ship.drift_vector);
                 vec_rotate(engine_work_speed,ship_angs.pan);
                 vec_add(my.drift_vector,engine_work_speed);}
        else
                {return;}               // If DRIFT_VECTOR is Zero, we are done

# *** If DRIFT_VECTOR is close enough to zero, call it zero. This will reduce the number
#       of times we need to go through this loop before an entity has no more momentum.
#       When an entity is not drifting, we don't need to do any math for it, and so we can
#       reduce the overhead on our computer. This can help increase frame rates if many entities
#       are using this routine at the same time.

        if (camera_type == 3)
                {goto stationary;}

// Begin Chase Camera code:
//              (NOTE: The Chase Cam portion of this script was donated by JCL.
//               It is currently somewhat rough, and I don't understand enough
//               about how it works to attempt to explain it. Please direct
//               questions about this section of code to him.)

        camera.genius = null;                   // Camera won't ignore anyone
        vec_set(workang.pan,player_ship.pan);
        camera_spot.x = -200;                   // Relative displacement of camera to
        camera_spot.y = 0;                      //      ship when in chase mode
        camera_spot.z = 80;

        vec_set(camera.x,camera_spot.x);
        vec_rotate(camera.x,player_ship.pan);
        vec_add(camera.x,player_ship.x);

// Set Camera  PAN and TILT angles
        vec_set (tempa.x,player_ship.x);
        vec_sub (tempa.x,camera.x);
        vec_to_angle(camera.pan,tempa);


// Set Camera ROLL angles
        c1.x = tempa.y;
        c1.y = -tempa.x;
        c1.z = 0;

        c2.x = -camera_spot.y;
```

```
        c2.y = camera_spot.x;
        c2.z = 0;

        vec_rotate(c2,player_ship.pan);

// Get the angle difference between both vectors using dot product
        tempa = c1.x*c2.x +c1.y*c2.y;
        tempa /= vec_length(c1);
        tempa /= vec_length(c2);

        if (c2.z < 0)
                {camera.roll = acos(tempa);}
        else
                {camera.roll = -acos(tempa);}

        return;

//Cockpit Camera starts here
cockpit:
        camera.genius = player_ship;
        vec_set(camera.x,player_ship.x);          // Camera is where the ship is
        vec_set(camera.pan,player_ship.pan);      // Camera looks in the same direction as ship
        RETURN;

//Stationary Camera Starts here
stationary:
        camera.genius = null;
        vec_set(camera_work,player_ship.x);       // Point Camera at player
        vec_sub(camera_work,camera.x);
        vec_to_angle(camera.pan,camera_work);   // now camera looks at the player
        return;
}


//////////////  SPACE SCRIPT ENDS HERE ///////////////////////
```