Xorekis's Ogre Tutorials Pack 2

These tutorials are to help new people to Ogre get a fairly quick understanding of how to use Ogre. The tutorials are not setup to address questions like 'How do I create a game using Ogre'.

Each of these packs will build upon the previous pack unless stated otherwise.

Feedback on these tutorials would be appreciated, (that's feedback, not bitching).

Overview

Sorry to say that I will only be addressing the tutorials from a windows perspective using VC++6.

Please remember that Ogre is a ‘Object-Oriented Graphic Engine’, it is NOT a game engine, nor does it have any intention of being anything else but a ’Object-Oriented Graphic Engine’.

I will not be covering any of the OgreAddons.

These tutorials will show you how to create an Ogre project outside of samples/common.

Familiarity with the windows OS is assumed here. If you are not familiar with how to work with windows I would suggest searching for some windows tutorials first.

Familiarity with how to successfully use and compile with VC6++ is also assumed. Again, there are plenty of tutorials on the web about this that I would suggest you look at first.

An understanding of 3D and 3D components is also assumed. That means you know what a camera is, what a Vector is and what you would use a Matrix for.

The forum on the website is a great place to ask questions. The one thing I would encourage you to do is to search the forum for answers first before posting your questions. More than likely, you question was asked by someone else and there is an answer waiting for you to read.

It is also assumed that you know how to debug and step though code. Putting posts on the forums ‘My code does not work! Why?’ will not win you any popularity points.

Note: Even though this tutorial shows different ways of getting the Ogre source code, I always use the most current from CVS.

Please Note: I am writing these tutorials as I am writing the code, so by following the steps everything should work. I say should, as even I can miss type something.

Tutorial 6: Broken Rules and the sky is Falling

Everybody has their own rules for coding in a standard format. I unfortunately broke one of my own cardinal rules. I never put two classes in the same file. So while creating a new Tutorial6 I put the **TutorialFrameListener** into its own file. Follow the steps from the previous tutorials and create a new Tutorial Project, but instead, put **TutorialFrameListener** into its own file as I did.

You should now have a new Tutorial project called Tutorial6. This project should contain the following files.

Source
- **TutorialFrameListener.cpp**
- **TutorialApplication.cpp**
- **WinMain.cpp**

Headers
- **TutorialFrameListener.h**
- **TutorialApplication.h**

Ok, new project, new tutorial, now what? Well you may think we are creating a space scene as one of the other tutorials available shows. To be honest, we were, at least until I remembered that existing tutorial. So we are going to do something different.

On wards and upwards, but not too far up for the sky is falling.

Remember that cool looking skybox? Did you know Ogre Scene Managers also supports skyplanes and skydomes?

We will do something creative in this part of the tutorial: changing the skybox to a skydome.

Find the following piece of code in your **TutorialApplication::createScene** method.

```
// Create the SkyBox
mSceneMgr->setSkyBox(true, "Examples/SpaceSkyBox");
```

now delete that code and enter the following

```
// Create the SkyDome
mSceneMgr->setSkyDome(true, "Examples/CloudySky", 5, 8);
```

While we are here, move the **setAmbientLight** from the bottom of the method to the top, is this just so that the coding style looks good.

But what is the setSkyDome doing?

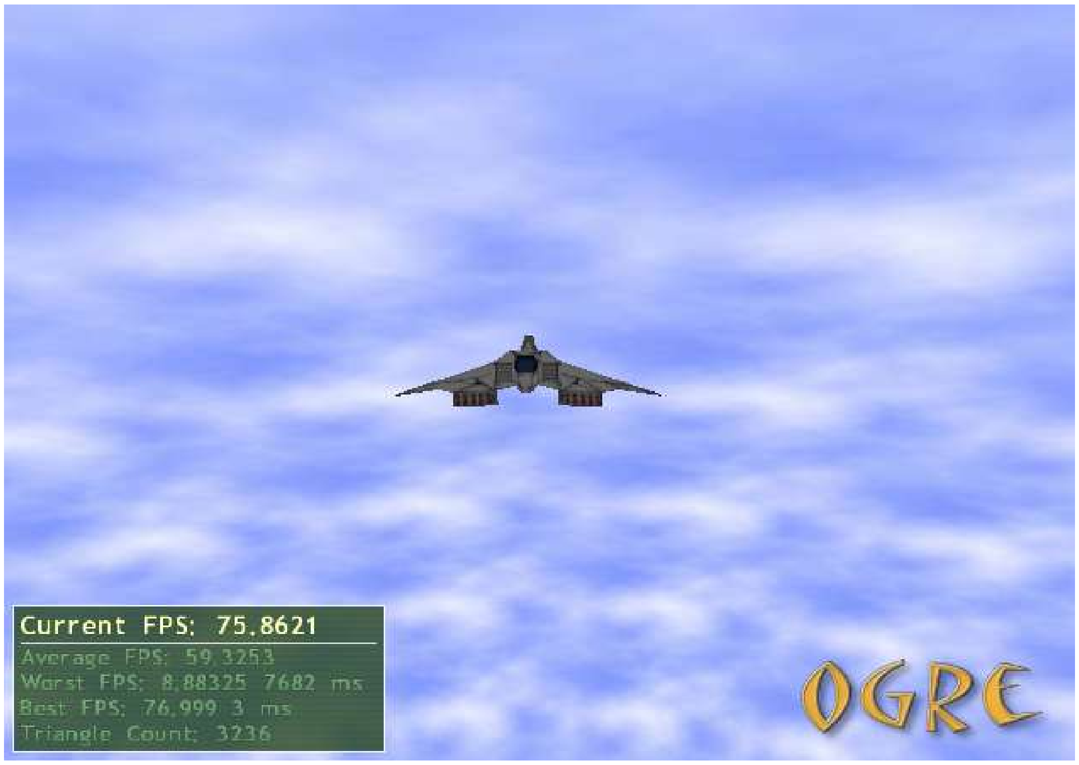
The value of true tells the scenemanger that the sky dome is enabled.

Examples/CloudySky is again, another material definition that already exists in the media directory.

Now 5 and 8 are interesting. 5 is the sky curvature and 8 is the tiling of the texture in the sky.

A more detailed description of the setSkyDome can be found in
Ogrenew\OgreMain\include\\`OgreSceneManager.h`

Now compile and see what we have created.



```
Current FPS: 75.8621
Average FPS: 59.3253
Worst FPS: 8.88325 7682 ms
Best FPS: 76.999 3 ms
Triangle Count: 3236
```

Ohh, that looks good. But it is still missing something. Let’s keep adding stuff shall we?

Tutorial 7: Land Ahoy

In a previous tutorial I said that we would come back to selecting different scenemanagers, now is that time.

Ogres default SceneManager is an OctreeSceneManager. This scene manager uses Octrees to help cull (Remove) sceneNodes from rendering.

Ogre also has a TerrainSceneManager that is based on OctreeSceneManager. This manager has the added feature of rendering a heightmap and texture into a 3D terrain.

By using the BSPSceneManager Ogre has the ability to display Quake3 BSP files. One thing to remember when using this scenemanger is that it does NOT implement all off the features in a Quake3 BSP.

Another scene manager in Ogre is the NatureSceneManager. This one renders terrain using patches.

Each of these SceneManagers can be retrieved from Ogre using the getSceneManager function.

For instance;

- To get the BSPSceneManager we would use a call like the following
  ```
  getSceneManager(ST_INTERIOR)
  ```
- To get the NatureSceneManager we would use a call like
  ```
  getSceneManager(ST_EXTERIOR_FAR)
  ```

When scenemanagers are added to Ogre, they are added via an enumerated value. There are currently 5 predefined values, each representing a scenemanager: ST_GENERIC, ST_EXTERIOR_CLOSE, ST_EXTERIOR_FAR, ST_EXTERIOR_REAL_FAR, and ST_INTERIOR.

We are going to change the scene manager in our TutorialApplication now. ExampleApplication has a virtual method that gets called internally that creates the default sceneManager. We will want to override that method in our TutorialApplication.

Add the following to the TutorialApplication class definition.

```
void chooseSceneManager();
```

Your class should now look like the following.

```
class TutorialApplication : public ExampleApplication
{
public:
     TutorialApplication();
     virtual ~TutorialApplication();
protected:
     void createScene();
     void createFrameListener();
     void chooseSceneManager();

     Entity* mShip;
     SceneNode* mShipNode;
};
```

Now add the following code to the end of the TutorialApplication.cpp file.

```
void TutorialApplication::chooseSceneManager(void)
{
    // Get the SceneManager, in this case the terrain one
    mSceneMgr = mRoot->getSceneManager( ST_EXTERIOR_CLOSE );
}
```

We are going to use the TerrainSceneManager from now on. When the TerrainSceneManager plugin was registered with Ogre, it was registered using the ST_EXTERIOR_CLOSE value. Because we want to use this manager, we have to pass ST_EXTERIOR_CLOSE to the getSceneManager.

If you compile and run now, you will not see any visible difference between now and the last time. The reason is that we have not told the new manager what file it should use for generating the terrain.

We have to setup a special file, which tells the sceneManager what heightmap we should use for generating the terrain.

Create a file in the **C:\OgreTutorials\Bin\Debug** called **terrain.cfg**

Now add the following lines to it

```
WorldTexture=terrain_texture.jpg
DetailTexture=terrain_detail.jpg
DetailTile=3
Terrain=terrain.png
TileSize=17
MaxPixelError=8
ScaleX=4
ScaleY=1
ScaleZ=4
MaxMipMapLevel=5
```

1. WorldTexture = The overall texture that will give the terrain its' colors, like white for snow, green for grass.

2. DetailTexture = This texture gives the ground detail. If this texture were not used, you would get large blocks of colour. It would not look nice.

3. DetailTile = The number of times the detail texture will tile in a terrain tile.

4. Terrain = This is the actual texture we want to use for the heightmap .

5. TileSize = The size of the tiles that the heightmap will get split into.

6. MaxPixelError = A value that is used for LOD. 8 is a good value to use.

7. ScaleX, ScaleY, ScaleZ = This is the scaling factor that is to be applied to the terrain. Remember Y is vertical scale.

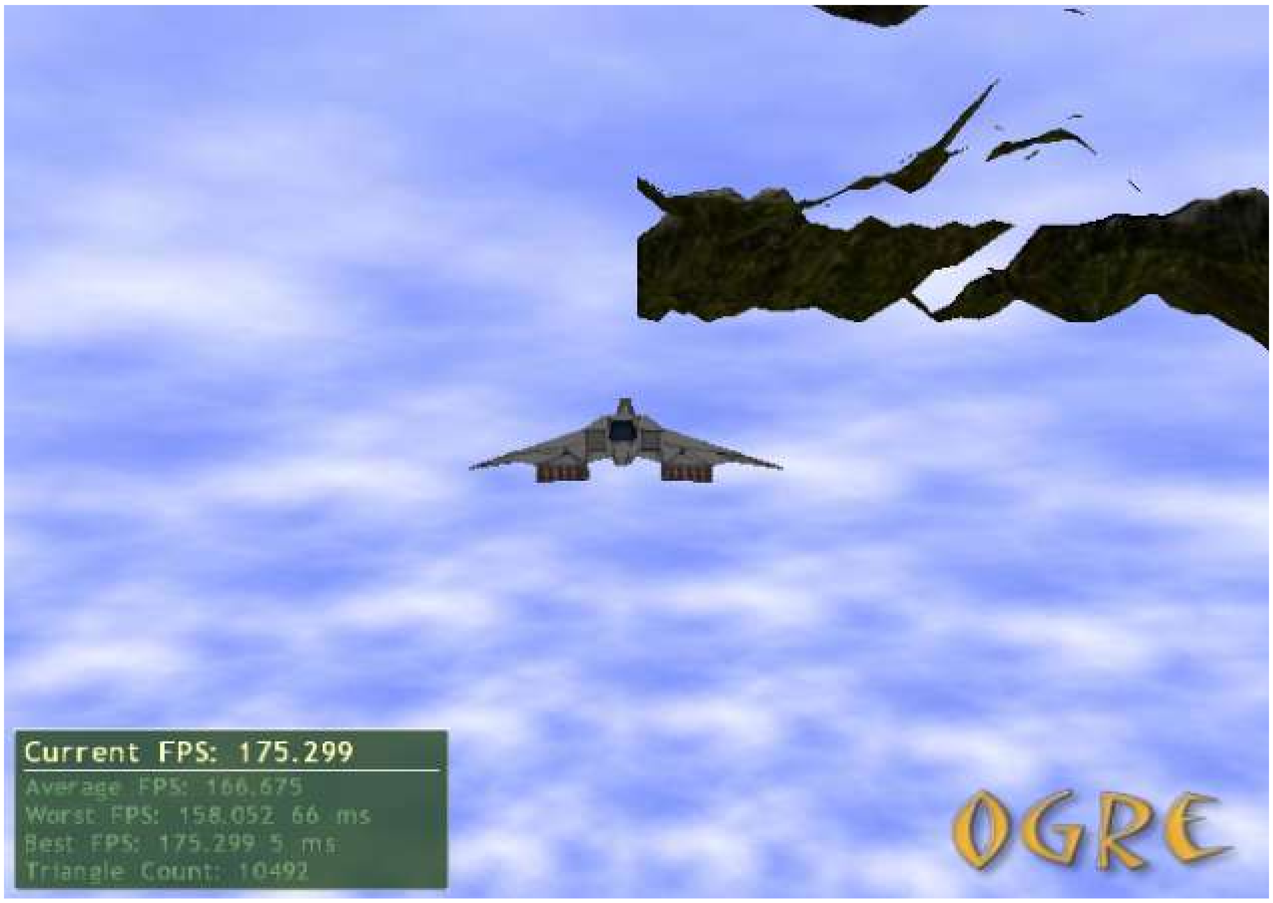
8. MaxMipMapLevel = The number of MipMaps to generate of the terrain.

In between **setSkyDome** and **createEntity** in **TutorialApplication::createScene** add the following code

```
// Create some terrain
mSceneMgr->setWorldGeometry( "terrain.cfg" );
```

This instructs our scenemanager to use terrain.cfg for creating the terrain. SetWorldGeometry is a common method for scene managers. Any special scene managers will override this method so that they can load static data. That is data that does not change like terrain or a BSP.

Now compile and run our Tutorial.



```
Current FPS: 175.299
Average FPS: 166.675
Worst FPS: 158.052 66 ms
Best FPS: 175.299 5 ms
Triangle Count: 10492
```

Well, there is terrain there, but it does not look good does it?

There are two contributing factors to this: back plane clipping and scale. Back plane clipping is the process of not rendering triangles that face away from us. This gives the impression of holes. The other factor was scale; this is something we can deal with. Edit the Terrain.cfg file and change to the following

```
ScaleX=32
ScaleY=4
ScaleZ=32
```

Yes we are scaling the terrain bigger not smaller. Hang with me and you will see why in the next tutorial.

Tutorial 8: If only I could see

In this tutorial we are going to be making some changes to existing code. Some are tricky changes, so please take your time.

Ogre uses cameras to view Scenes. So far we have been using the default camera created by ExampleApplication, you probably did not realize that. Don't you think it would be cool if we could attach our camera to our little razor?. But I just don't want to attach it that easily. Otherwise I would not need this tutorial. No. I want to do something far more interesting.

First off, we need to need to add some more scene nodes to the application.

Open up **TutorialApplication.h**, now add the following two scene nodes just below the existing one.

```
SceneNode* mControlNode;
SceneNode* mCameraNode;
```

ControlNode will become the node that is changed by the user's input.
CameraNode will manage the location of our camera.

Your class should look something like this now

```
class TutorialApplication : public ExampleApplication
{
public:
     TutorialApplication();
     virtual ~TutorialApplication();
protected:
     void createScene();
     void createFrameListener();
     void chooseSceneManager();

     Entity* mShip;
     SceneNode* mControlNode;
     SceneNode* mShipNode;
     SceneNode* mCameraNode;
};
```

Now that we have declared the variables for the new scene nodes, we may as well actually create them.

Open up **TutorialApplication.cpp**, and scroll down to the **TutorialApplication::createScene** method.

Find the piece of code that looks like

```
// Create our SceneNode
mShipNode = mSceneMgr->getRootSceneNode()->createChildSceneNode();
```

We don't want our ship being controlled directly anymore. So change mShipNode to mControlNode as shown below.

```
// Create our control SceneNode
mControlNode = mSceneMgr->getRootSceneNode()->createChildSceneNode();
```

But we still need the Ship Node, add the following line to create the ship node.

```
// Create a SceneNode for the ship
mShipNode = mControlNode->createChildSceneNode();
```

What this does is associate the mShipNode as a child of the mControlNode. When we move mControlNode, the mShipNode will automatically move as well.

Add the following line right after where we attach the Ship Entity to the mShipNode

```
mShipNode->setPosition( Vector3( 6, -10, 0) );
```

The razor is not actually centered in itself. What I mean is, if you could fold the razor in half, wing tip to wing top. The center is not where the center crease would be. So to center the razor on the crease and inside the body we change its position. If this node were our control node, we would constantly have to make this adjustment to keep it centered. By adjusting the ShipNode we have cut down on our coding.

Now we need to create our last node, the camera node. Add the following at the end of the method.

```
// Create a SceneNode for the Camera
mCameraNode = mControlNode->createChildSceneNode();
```

Again we have created another scene node as a child of the ControlNode. Another way to think about this, is that we can change the position of the camera without affecting the position of the ship or our control location. Nice hu?

Let's attach the existing camera now to the camera node. That is done this way

```
// Attached the camera to the SceneNode
mCameraNode->attachCamera( mCamera );
mCameraNode->setPosition( Vector3( 0, 50, -200) );
```

I am giving the camera a position behind and slightingly above the razor. Like a third person view.

Remember how I said that the ExampleApplication created our camera for us? Well we need to undo some of the things the ExampleApplication did. Mostly position and look at.

We want our camera at the center of our scenenode, and we want to look up the Z axis. The razor points up the Z axis, so we may as well do the same thing.

```
// Position it at 0
mCamera->setPosition(Vector3(0,0,0));
// Look back along Z
mCamera->lookAt(Vector3(0,0,100));
```

The last piece of code I want to add to the createScene method is a position. Because of the way we have created the scene nodes. We can set the overall position of the camera, ship and scene nodes by just setting the position of the mControlNode. Seeing as we have a large terrain, lets put ourselves somewhere near the center.

```
mControlNode->setPosition(Vector3(4000,500,4000));
```

Now we need to make some changes to TutorialFrameListener. There are a number of ways the FrameListener could have been setup. We created it as a separate class. We could of instead, also had the TutorialApplication inherit from ExampleFrameListener, and that would of saved us from having to pass the scene nodes. But, no, I did not do it the simple way. This time at least.

Go to our method **TutorialApplication::createFrameListener**, we need to add the two new scene node in the following way, so go ahead and make the change.

```
mFrameListener = new TutorialFrameListener(mWindow, mCamera,
     mControlNode, mShipNode, mCameraNode );
```

That does it for working in **TutorialApplication.cpp** and **TutorialApplication.h.**

Our next changes are in **TutorialFrameListener.cpp** and **TutorialFrameListener.h**. I would suggest you stand up walk around, stretch before continuing. That way I can get myself a drink.

<< INSERT MUSICAL INTERLUDE >>

Ahh. That's better. Now back to the tutorial. Open up **TutorialFrameListener.h**, we have a few changes to make.

We have to add two new SceneNode variables, a bool to indicate first person, and make changes to the constructor. I will just show you the end result; you should be able to see the changes you need to make.

```
class TutorialFrameListener : public ExampleFrameListener
{
protected:
     SceneNode* mControlNode;
     SceneNode* mShipNode;
     SceneNode* mCameraNode;

     bool mbFirstPerson;
public:
     TutorialFrameListener(RenderWindow* win, Camera* cam,
          SceneNode* controlNode,
          SceneNode* shipNode,
          SceneNode* cameraNode);
     bool frameStarted(const FrameEvent& evt);
};
```

That's right, I'm giving two views of the razor. A third person and first person view.

Now that we have done the changes to the class def, its is time to change the class methods.

Open up **TutorialFrameListener.cpp**.

The first thing we need to do is change every instance of **mShipNode** to **mControlNode**. So use search and replace on the CPP file and make the change.

Now make changes to the constructor to match the changes we just did in the header by adding the Control and Camera SceneNodes and re-adding the Ship SceneNode.

In the body of the constructor we need to assign the passed arguments to the proper variables. We also need to set the first person to false so that we start out in third person, add the following.

```
     mControlNode = controlNode;
     mShipNode = shipNode;
     mCameraNode = cameraNode;

     mbFirstPerson = false;
```

Now scroll down to the bottom of the **TutorialFrameListener::frameStarted** method.

Add the following code just before the return true.

```
if(mInputDevice->isKeyDown(Ogre::KC_F))
{
     while (mInputDevice->isKeyDown(Ogre::KC_F))
          mInputDevice->capture();

     if (mbFirstPerson)
     {
          // third person
          mCameraNode->setPosition( Vector3( 0, 50, -200) );
          mbFirstPerson = false;
     }
     else
     {
          // first person
          mCameraNode->setPosition( Vector3( 0, 5, 40) );
          mbFirstPerson = true;
     };
};
```

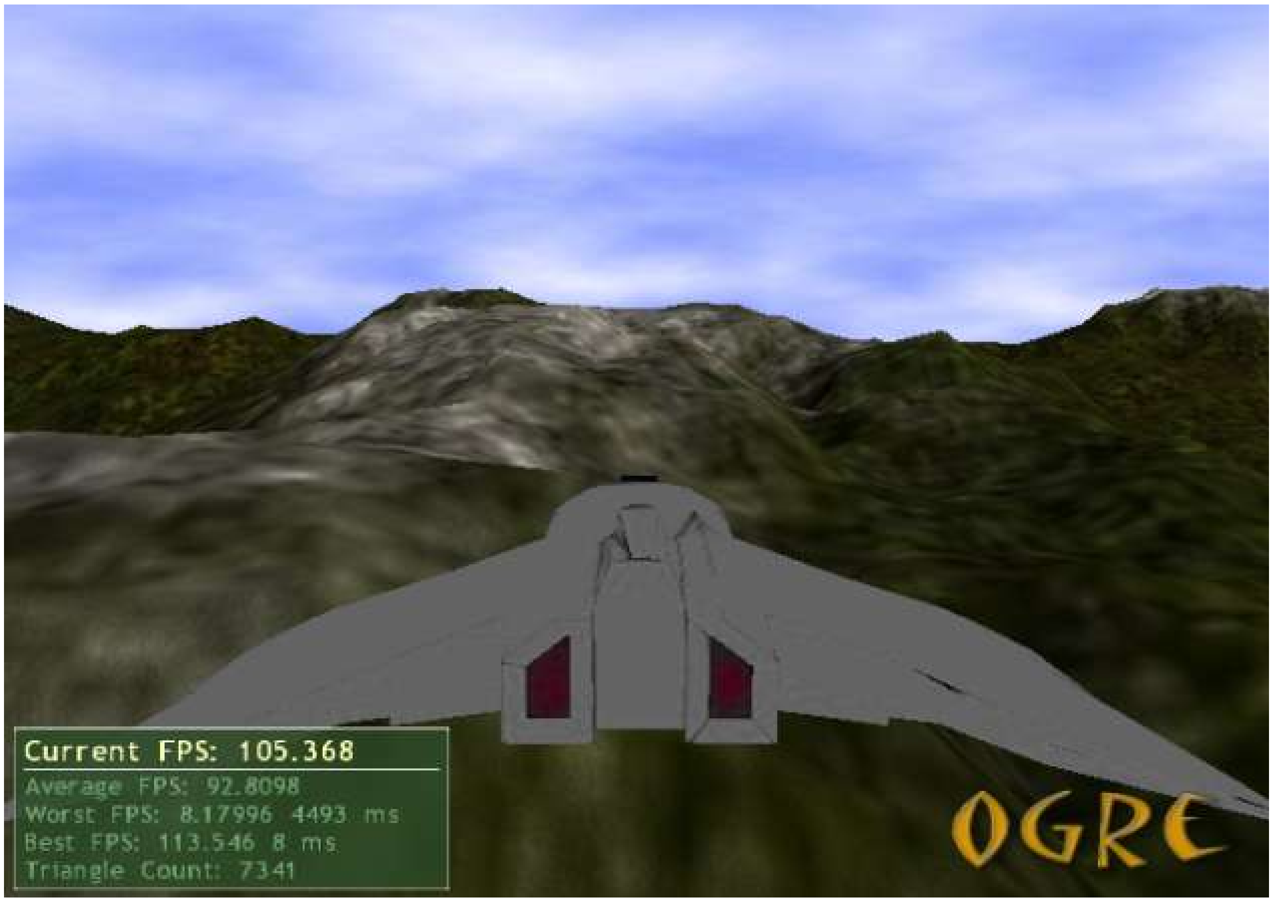
Ok, here is what this code does, it checks to see if the 'F' key has been pressed. It then sits in a loop waiting for the 'F' key to be no longer pressed. (Not the best way, but works for this tutorial.)

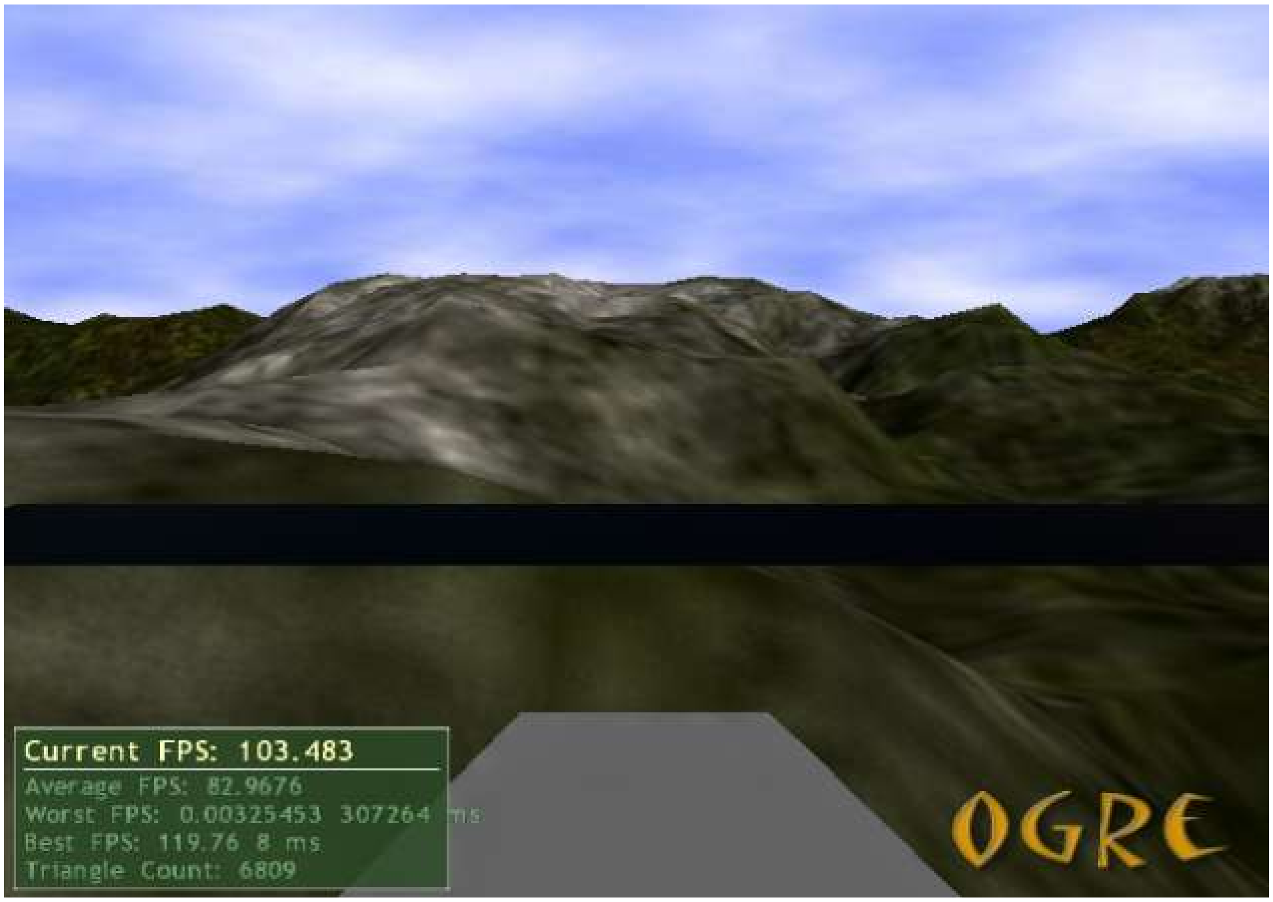
Once the 'F' key is released, we toggle first person depending on what it is currently set to. When we do that, we also change the location of the camera node. In third person, the camera is above and behind the Razor. In first person, the camera is close to where the cockpit should be.

Well believe it or not, we are done with the code changes in this tutorial. Wow, talk about sore fingers.

So the moment of truth, if you made the changes correctly, and I remembered to tell you all my changes, compile and run!!!!.

If everything worked, you should see a screen like the first image following, and after pressing 'f' the second image.


```
Current FPS: 105.368
Average FPS: 92.8098
Worst FPS: 8.17996 4493 ms
Best FPS: 113.546 8 ms
Triangle Count: 7341
```


Current FPS: 103.483
Average FPS: 82.9676
Worst FPS: 0.00325453 307264 ms
Best FPS: 119.76 8 ms
Triangle Count: 6809
OGRE

Tutorial 9: My mouse said move did it not?

Wow, this is fun. We have a sky, we have some terrain, and we even have a plane. But the controls suck, and flies like a rock.

Guess we better do something about that then.

We are going to be working exclusively with TutorialFrameListener.

First we need to add some variables to the header file. These variables need to be added below **mbFirstPerson**.

```
     Real mfAfterburner;
     Real mfSpeed;
     Real mfPitch;
     Real mfYaw;
     Real mfRoll;
     Real mfShipRoll;
     Real mfShipPitch;
```

Now we are going to change the code in **TutorialFrameListener::frameStarted**.

First, delete the code that is associated with the checks of KC_UP, KC_DOWN, KC_LEFT and KC_RIGHT.

All you should have left in the method is the following.

```
bool TutorialFrameListener::frameStarted(const FrameEvent& evt)
{
     Real MoveFactor = 80.0 * evt.timeSinceLastFrame;

     mInputDevice->capture();

     if(mInputDevice->isKeyDown(Ogre::KC_ESCAPE))
          return false;

     if(mInputDevice->isKeyDown(Ogre::KC_F))
     {
          while (mInputDevice->isKeyDown(Ogre::KC_F))
               mInputDevice->capture();

          if (mbFirstPerson)
          {
               // third person
               mCameraNode->setPosition( Vector3( 0, 50, -200) );
```

```
                mbFirstPerson = false;
            }
            else
            {
                // first person
                mCameraNode->setPosition( Vector3( 0, 5, 40) );
                mbFirstPerson = true;
            };
        }

        return true;
}
```

Now add the following code after the check for KC_ESCAPE and before the check for KC_F.

```
// Increase speed
if (mInputDevice->isKeyDown(KC_EQUALS) || mInputDevice->isKeyDown
(KC_ADD))
{
        mfSpeed += 10;
        if (mfSpeed > 200)
            mfSpeed = 200;
}

// Decrease speed
if (mInputDevice->isKeyDown(KC_MINUS) || mInputDevice->isKeyDown
(KC_SUBTRACT))
{
        mfSpeed -= 5;
        if (mfSpeed < 0)
            mfSpeed = 0;
}

// hit our afterburners
mfAfterburner = 0;
if (mInputDevice->isKeyDown(KC_TAB) )
{
        mfAfterburner = 500;
}
```

This code adjusts our speed, faster, slower, or afterburners.

```
rotX = -mInputDevice->getMouseRelativeX() * 0.5;
rotY = mInputDevice->getMouseRelativeY() * 0.5;

// process mfYaw (rudder function):
mfYaw += rotX;
mfShipRoll -= rotX / 5;

// process mfPitch (elevator function):
mfPitch += rotY;
mfShipPitch += rotY / 5;


if (mfPitch > 45) // (higher value, turn sharper)
     mfPitch = 45;
if (mfPitch < -45)
     mfPitch = -45;
if (mfYaw > 45)
     mfYaw = 45;
if (mfYaw < -45)
     mfYaw = -45;
if (mfRoll > 45)
     mfRoll = 45;
if (mfRoll < -45)
     mfRoll = -45;

if (mfShipPitch > 35)
     mfShipPitch = 35;
if (mfShipPitch < -35)
     mfShipPitch = -35;
if (mfShipRoll > 90)
     mfShipRoll = 90;
if (mfShipRoll < -90)
     mfShipRoll = -90;
```

We use our mouse to get yaw and pitch values. We do a valid values check so we don't get odd behavior.

```
     // dampen the changes
     mfPitch *= 0.9;
     mfYaw *= 0.9;
     mfRoll *= 0.9;

     mfShipPitch *= 0.95;
     mfShipRoll *= 0.95;
```

Let's remove the angle changes over time. Basically the ship will re-center.

```
// set angles for controlNode
if ((Math::Abs( mfPitch ) > 0.01) ||
     (Math::Abs( mfYaw ) > 0.01) ||
     (Math::Abs( mfRoll ) > 0.01))
{
     mControlNode->roll( mfRoll * evt.timeSinceLastFrame );
     mControlNode->pitch( mfPitch * evt.timeSinceLastFrame );
     mControlNode->yaw( mfYaw * evt.timeSinceLastFrame );
};
```

Adjust the controlNodes angles, this will change our view of the terrain.

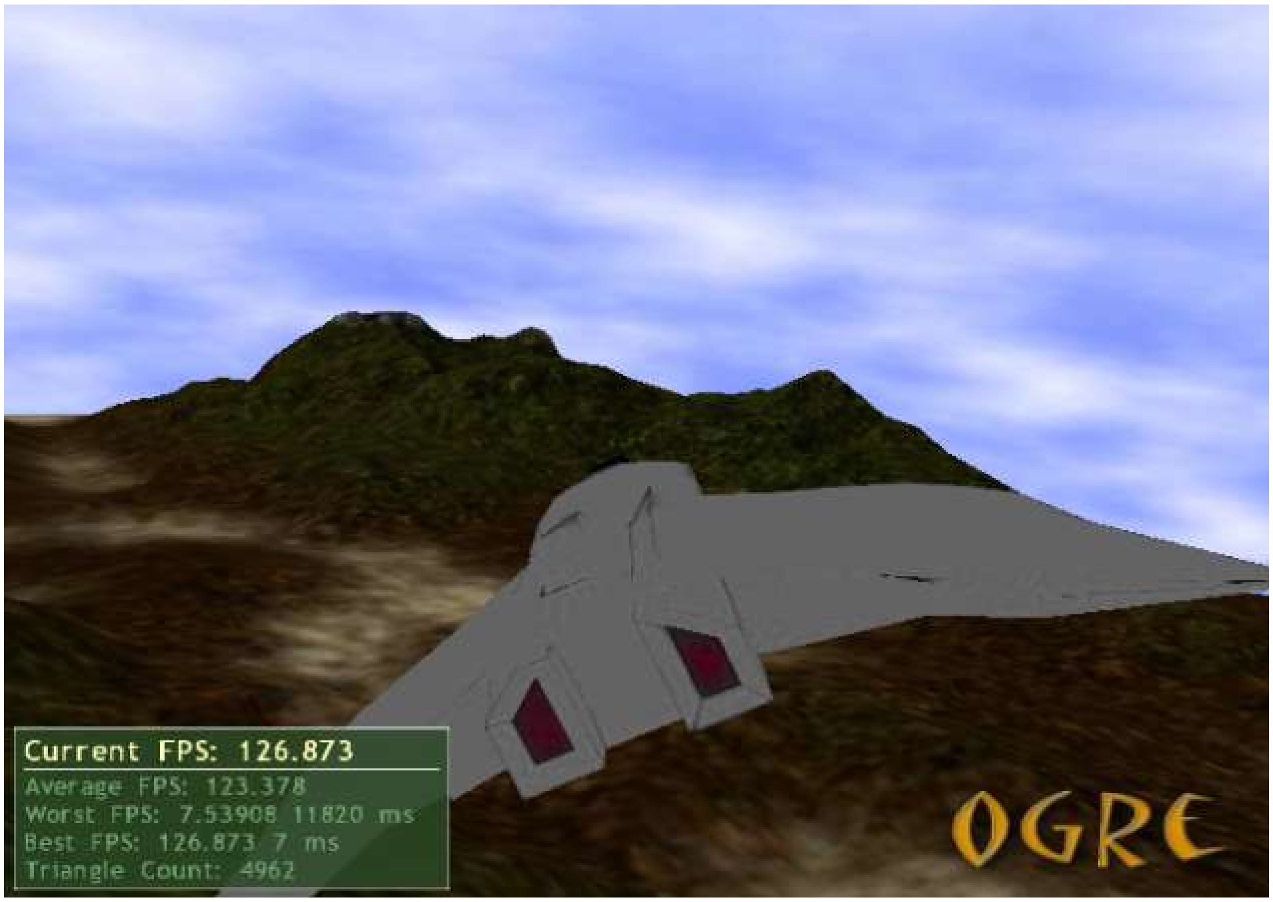
```
// set angles for ship node to make it looked banked in third person
if ((Math::Abs( mfShipRoll ) > 0.01) ||
     (Math::Abs( mfShipPitch ) > 0.01))
{
     Quaternion qRoll, qPitch;
     qRoll.FromAngleAxis(Math::AngleUnitsToRadians(mfShipRoll),
          Vector3::UNIT_Z);
     qPitch.FromAngleAxis(Math::AngleUnitsToRadians(mfShipPitch),
          Vector3::UNIT_X);
     mShipNode->setOrientation( qRoll * qPitch );
};
```

This is cool. In third person, the ship will roll doing a turn and pitch when going up and down.

```
static Vector3 vec;

vec = Vector3::ZERO;
vec.z = (mfSpeed + mfAfterburner) * evt.timeSinceLastFrame;

// Translate the controlNode by the speed
Vector3 trans = mControlNode->getOrientation() * vec;
mControlNode->translate(trans);
```

This little bit of code will move us around the terrain a number of units depending on the speed in the direction we are looking.

Ok, COMPILE AND RUN!!!!!!!!!



If everything worked as planned. You should have aircraft like controls now, and can fly your ship around.

If someone else comes up with a better way of doing aircraft controls, I would be interested in updating this tutorial.

Congratulation, You have created a Simple Flight Simulator With Ogre

Tutorial 10: Splish Splash I was taking a bath

Figured I would add one last thing into this tutorial set.

Go to the bottom of the TutorialApplication::createScene method. Now add the following code.

```
// Water
Entity *pWaterEntity;
Plane nWaterPlane;


// create a water plane/scene node
nWaterPlane.normal = Vector3::UNIT_Y;
nWaterPlane.d = -1.5;
MeshManager::getSingleton().createPlane(
    "WaterPlane",
    nWaterPlane,
    8000, 8000,
    20, 20,
    true, 1,
    10, 10,
    Vector3::UNIT_Z
);
```

To simulate water, we want to create a flat plane with a texture. Here we make a plane that is 8000 units by 8000 units. Each side of the plane has 20 control points.

Now we have to add the water to the scene. We do that like this:

```
pWaterEntity = mSceneMgr->createEntity("water", "WaterPlane");
pWaterEntity->setMaterialName("Examples/TextureEffect4");

SceneNode *waterNode =
    mSceneMgr->getRootSceneNode()->createChildSceneNode("WaterNode");
waterNode->attachObject(pWaterEntity);
waterNode->translate(4000, 50, 4000);
```

The material Examples/TextureEffect4 requires a light before you can see it. So we have to add a light as well.

```
// Create a light
Light* pLight = mSceneMgr->createLight("MainLight");
pLight->setType( Light::LT_DIRECTIONAL );
pLight->setDirection( 0, -100, 0 );
```

So we create a light and projects parallel rays of light in the downward direction.

Now go to the TutorialFrameListner.cpp and add the following two lines before the constructor.

```
#define FLOW_SPEED 0.4
#define FLOW_HEIGHT 10
```

Last bit of code. YEH!

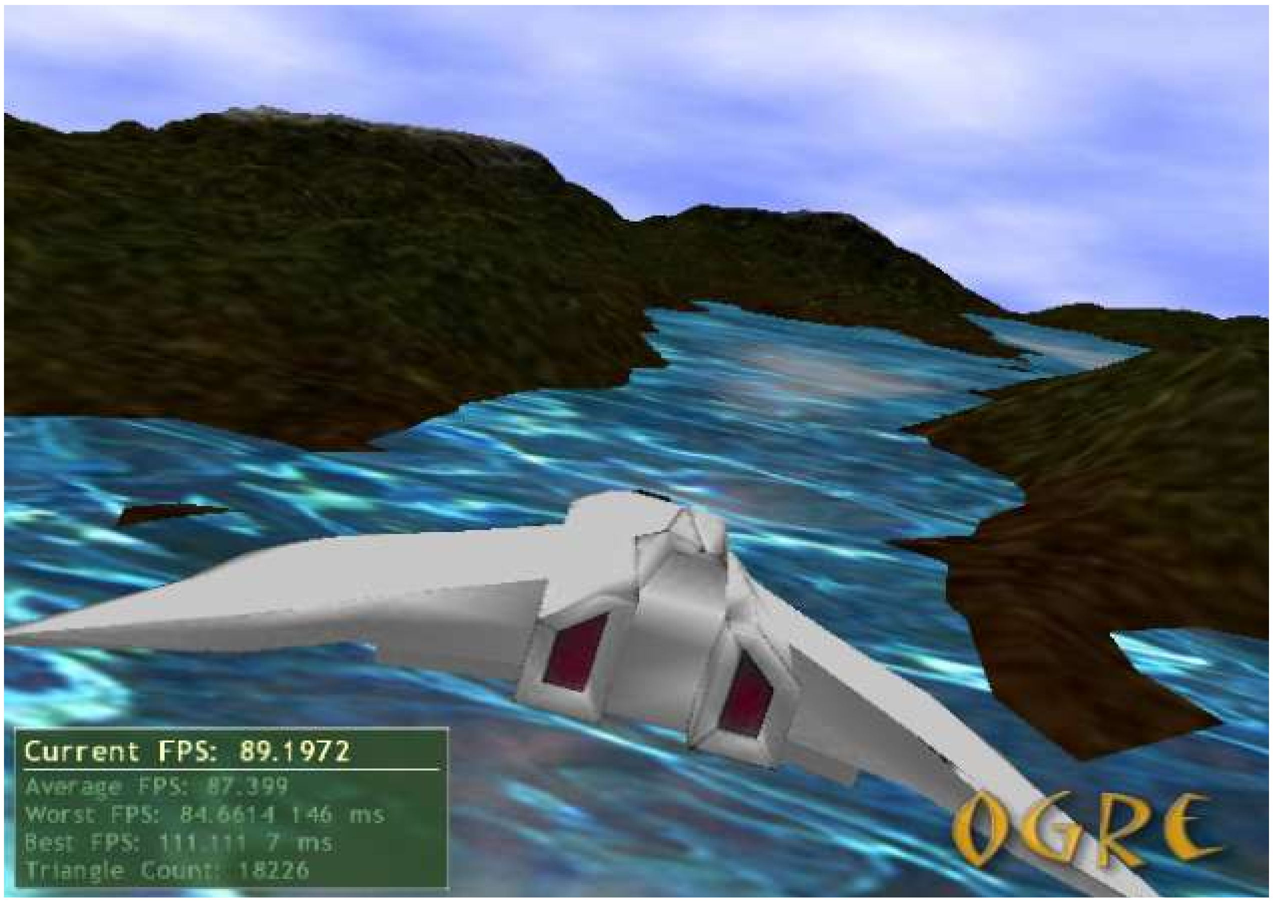
Go to the bottom of the TutorialFrameListener::frameStart method. Enter the following code before the return true.

```
float fWaterFlow = FLOW_SPEED * evt.timeSinceLastFrame;
static float fFlowAmount = 0.0f;
static bool fFlowUp = true;

SceneNode *pWaterNode = static_cast<SceneNode*>(
mCamera->getSceneManager()->getRootSceneNode()->getChild("WaterNode"));
if(pWaterNode)
{
     if(fFlowUp)
          fFlowAmount += fWaterFlow;
     else
          fFlowAmount -= fWaterFlow;

     if(fFlowAmount >= FLOW_HEIGHT)
          fFlowUp = false;
     else if(fFlowAmount <= 0.0f)
          fFlowUp = true;

     pWaterNode->translate(0, (fFlowUp ? fWaterFlow : -fWaterFlow), 0);
}
```

Some explanation is required here. Basically what this code does is gently moves the water up and down. Giving the illusion of a tide, or swell.

The other bit of interesting code, is how we get the Water SceneNode. Instead of passing the node in, you can query the scene manager for the node with a specific name. In this case 'WaterNode'. I would not suggest doing it this way. If you had passed the sceneNode in, then you don't have to run the query each frame, thus speeding up the frames.

So Compile and Run the code, you should see a screen something like…….



```
Current FPS: 89.1972
Average FPS: 87.399
Worst FPS: 84.6614 146 ms
Best FPS: 111.111 7 ms
Triangle Count: 18226
```

This completes this pack of tutorials. I have lots in mind I still want to cover. Anyway, play with what we have created. If you do something interesting, I would be interested in adding it to the tutorials.

Until next tutorial

Xorekis