Xorekis's Ogre Tutorials Pack 1

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'.

Overview..2
Tutorial 1: How to setup and compile Ogre...3
Directory Setup...3
Getting Ogre...3
Compiling Ogre...4
Tutorial 2: How to create the Tutorial Workspace...5
Tutorial 3: How to create the Tutorial Project...6
Tutorial 4: Creating the tutorial Application..11
WinMain..11
TutorialApplication..12
Your first compile...13
Your First Run..14
Tutorial 5: Adding something exciting...16
Skybox..16
A ship..17
Space is dark...19
Who is flying my ship?..21
Xorekis...24

Overview

Sorry to say that I will only be addressing the tutorials from a windows perspective using VC++6. This is because I personally only work in windows, and I don't have access to a linux box let alone a Mac,

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, at least not yet. I may change my mind as I do the tutorials.

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 on 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.

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 like '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.

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.

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

Tutorial 1: How to setup and compile Ogre

This tutorial will cover the basic setup and compilation of Ogre for the rest of the Tutorials.

Directory Setup

For these Tutorials we will be creating our own directory structure. Create the following directories using dos or file explorer.

- ***C:\OgreTutorials***
- ***C:\OgreTutorials\Bin\Debug***
- ***C:\OgreTutorials\Bin\Release***
- ***C:\OgreTutorials\Bin|Media***
- ***C:\OgreTutorials\Common***

Getting Ogre

There are three ways you can get Ogre.

1. Download a stable version of Ogre from http://www.ogre3d.org/modules.php?op=modload&name=Downloads&file=index&req=viewsdownload&sid=1

2. Download an unstable version of Ogre from http://www.ogre3d.org/modules.php?op=modload&name=Downloads&file=index&req=viewsdownload&sid=2. (This version is a nightly zip of CVS)

3. Download Ogre directly from CVS. The instructions for doing this are found at http://www.ogre3d.org/modules.php?op=modload&name=Sections&file=index&req=viewarticle&page=1&arttitle=Getting+Ogre+from+CVS

After downloading Ogre, you will need to also download the Dependencies called '3rd-part libraries for MS Visual C++'. These are found at http://www.ogre3d.org/modules.php?op=modload&name=Downloads&file=index&req=viewsdownload&sid=3

Compiling Ogre

If you downloaded a ZIP you can now unzip Ogre into our directory. If you used CVS and put it someplace else, move or copy the OgreNew directory to ours. In the end you should have ***C:\OgreTutorials\OgreNew***

Now unzip the dependencies into ***C:\OgreTutorials***. If it was done right, there now should be a directory at ***C:\OgreTutorials\OgreNew\Dependencies***

The last thing you will need to do is to setup stlport. There are plenty of instructions in the forums on how to do this and will NOT be covered here. These tutorials are also going to make the assumption that you are setting your Project Paths 'include' and 'libraries' to point to where stlport is found.

Using VC++6, Open up the Ogre workspace found in ***C:\OgreTutorials\OgreNew***. If everything was setup properly, at this time you should be able to compile the whole workspace in debug and release modes.

Once compiled, you can go to ***C:\OgreTutorials\OgreNew\Samples\common\bin\debug*** or ***C:\OgreTutorials\OgreNew\Samples\common\bin\release*** and you will find a whole load of demo executables. Run them, look though them.

The last part of this tutorial is copying the appropriate files for the rest of the tutorials.

Using file explorer we need to copy the DLLs and Plugins.cfg from

C:\OgreTutorials\OgreNew\Samples\common\bin\debug to
C:\OgreTutorials\bin\debug

And

C:\OgreTutorials\OgreNew\Samples\common\bin\release to
C:\OgreTutorials \bin\release

Using file explorer copy the following files to ***C:\OgreTutorials\Common***

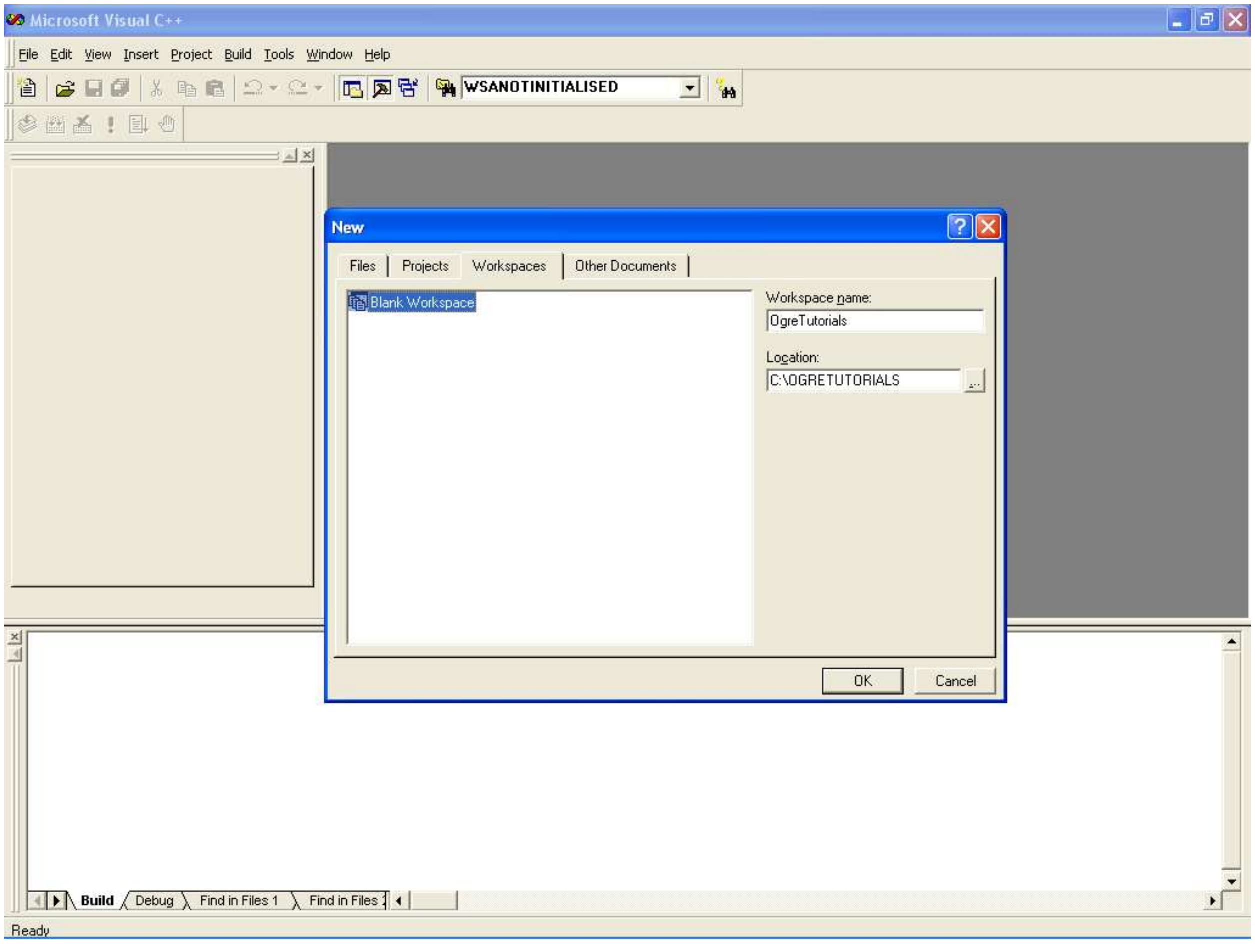
- ***C:\OgreTutorials\OgreNew\Samples\Common\Include\ExampleApplication.h***
- ***C:\OgreTutorials\OgreNew\Samples\Common\Include\ExampleFrameListener.h***

Using file explorer copy the DLLs from ***C:\OgreTutorials\Ogrenew\dependancies\Devil\libs*** to ***C:\OgreTutorials\bin\debug***.

Tutorial 2: How to create the Tutorial Workspace

If you don't already have MS Dev Studio studio open, please open it at this time.

Using the File/New Command, create a workspace called OgreTutorials in the **C:\OgreTutorials** directory as shown in the image below.

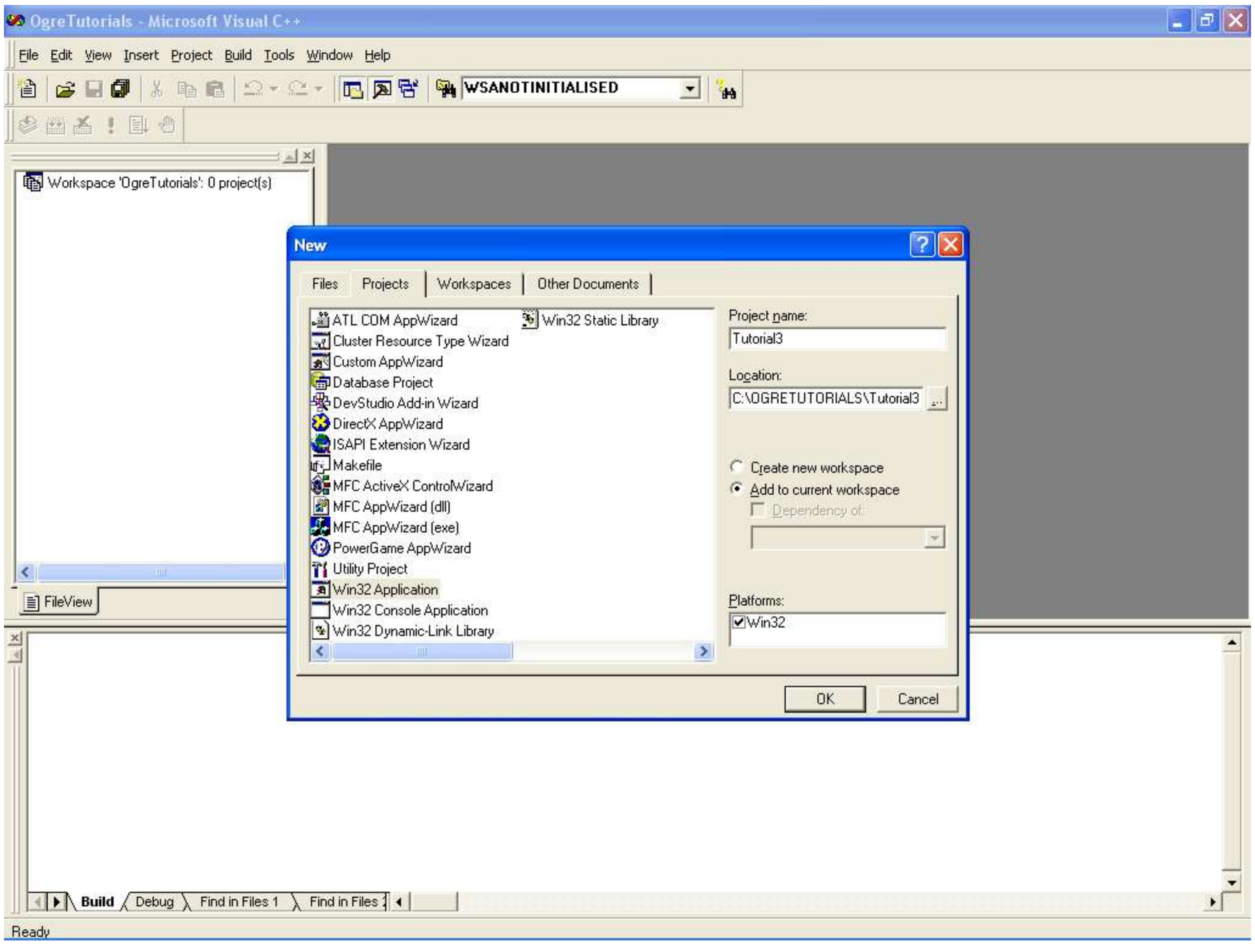


Workspace name: OgreTutorials

Location: C:\OGRETUTORIALS

Tutorial 3: How to create the Tutorial Project

Using the File/New command, create a windows project called in this case ‘Tutorial3’. This name could change from tutorial to tutorial. Make sure that ‘add to current workspace’ is selected as shown below.



Workspace 'OgreTutorials': 0 project(s)

Project name: Tutorial3

Location: C:\OGRETUTORIALS\Tutorial3

Add to current workspace

Platforms: ✓ Win32

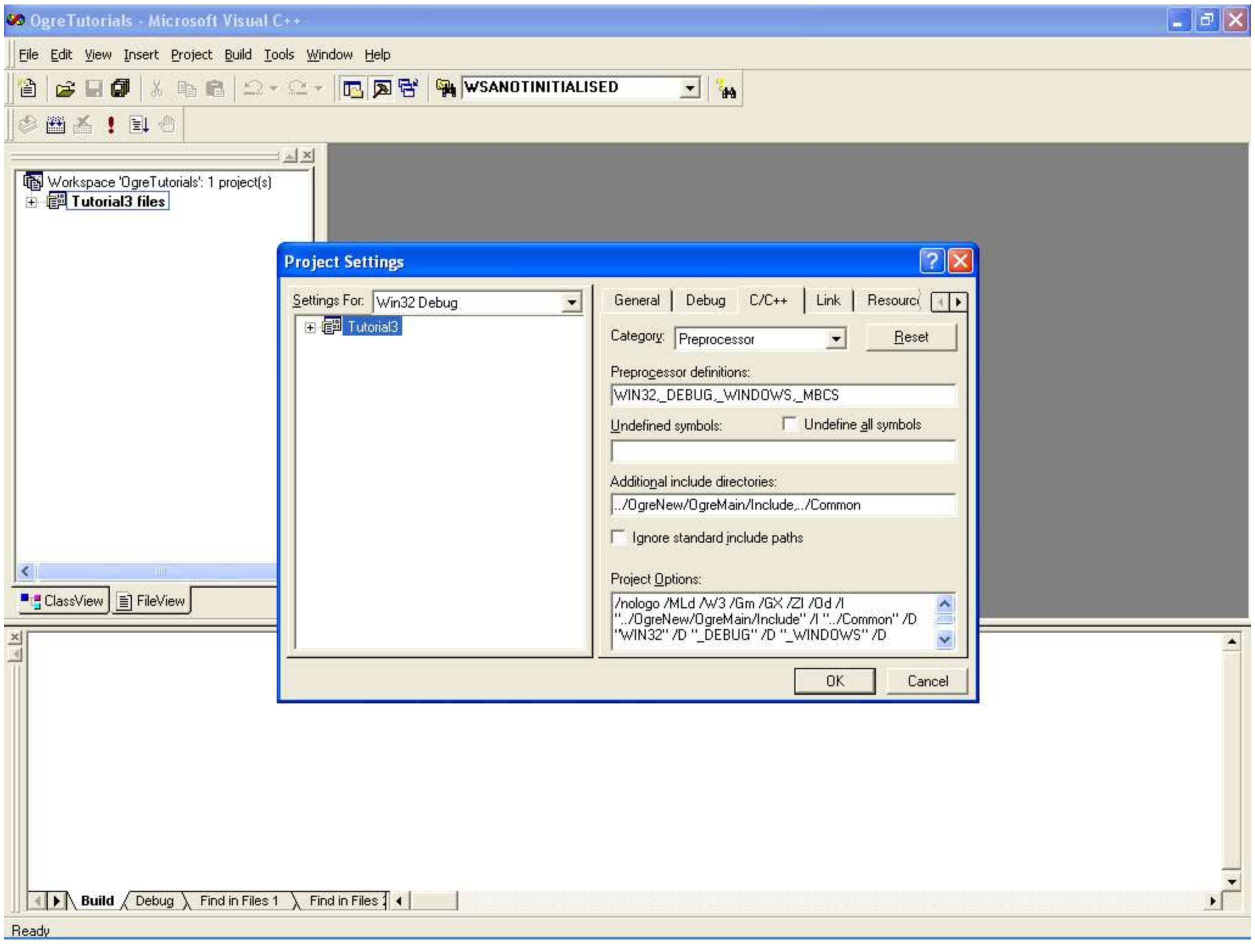
In the next popup you will be asked ‘What kind of windows application would you like to create?’, make sure that ‘An empty Project’ is selected.

Now that we have a project, we need to setup some includes and libraries.

First let's add the additional include directories for Ogre and the tutorial.

Open the settings for our Tutorial Project.

Go to the C++ Tab, and select Preprocessor from the Category drop down.
Now add ***../OgreNew/OgreMain/Include;../Common*** to the Additional Include Directories as shown below.



Settings For: Win32 Debug

Category: Preprocessor

Preprocessor definitions: WIN32,_DEBUG,_WINDOWS,_MBCS

Undefined symbols:

Undefine all symbols [ ]

Additional include directories: ../OgreNew/OgreMain/Include,../Common

Ignore standard include paths [ ]

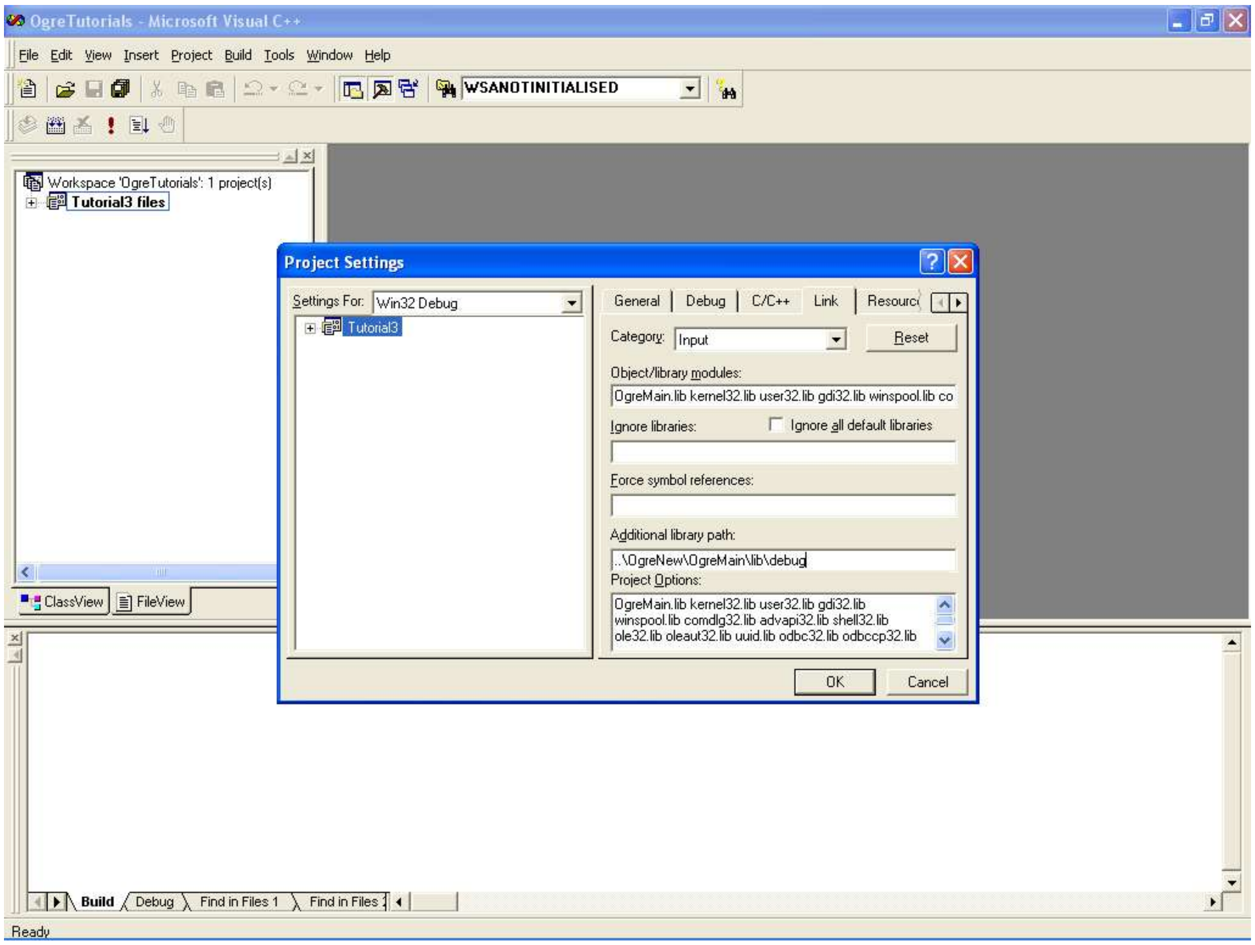
Project Options:

```
/nologo /MLd /W3 /Gm /GX /ZI /Od /I
"../OgreNew/OgreMain/Include" /I "../Common" /D
"WIN32" /D "_DEBUG" /D "_WINDOWS" /D
```

Next, go to the link tab and select Input from the category dropdown.

Now add **OgreMain.lib** to the Objects/Library Modules.

Enter the following into Additional Library Paths field ***..\OgreNew\OgreMain\lib\debug***



Settings For: Win32 Debug

Tutorial3

Category: Input

Object/library modules:
OgreMain.lib kernel32.lib user32.lib gdi32.lib winspool.lib co

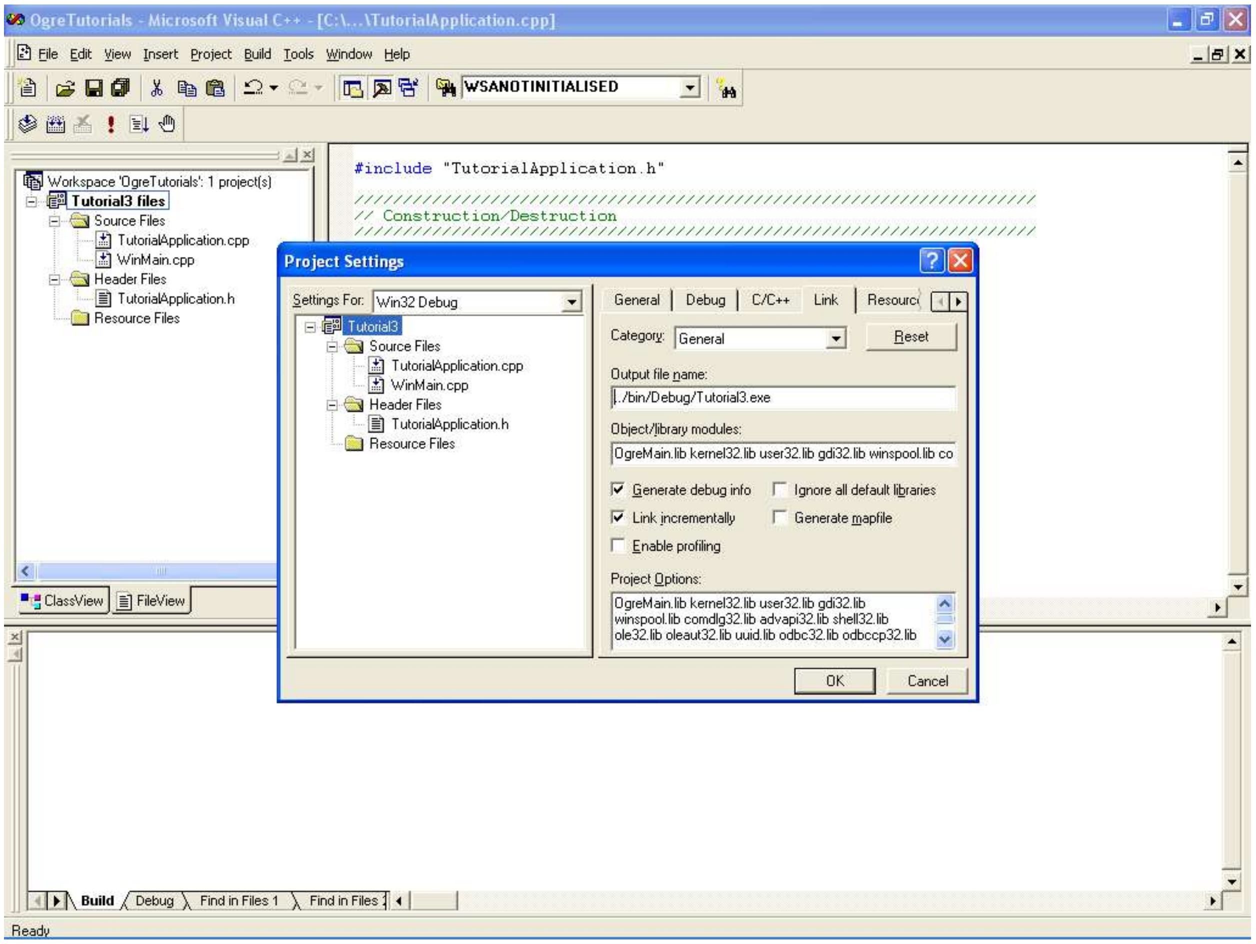
Ignore libraries: ☐ Ignore all default libraries

Force symbol references:

Additional library path:
..\OgreNew\OgreMain\lib\debug

Project Options:
OgreMain.lib kernel32.lib user32.lib gdi32.lib
winspool.lib comdlg32.lib advapi32.lib shell32.lib
ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib

Change the Category to General, add the path ***../bin*** to the Output file name as shown below. As you can see from the background image, I thought about this after I got into doing the tutorial.



```
#include "TutorialApplication.h"

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////
```

Category: General

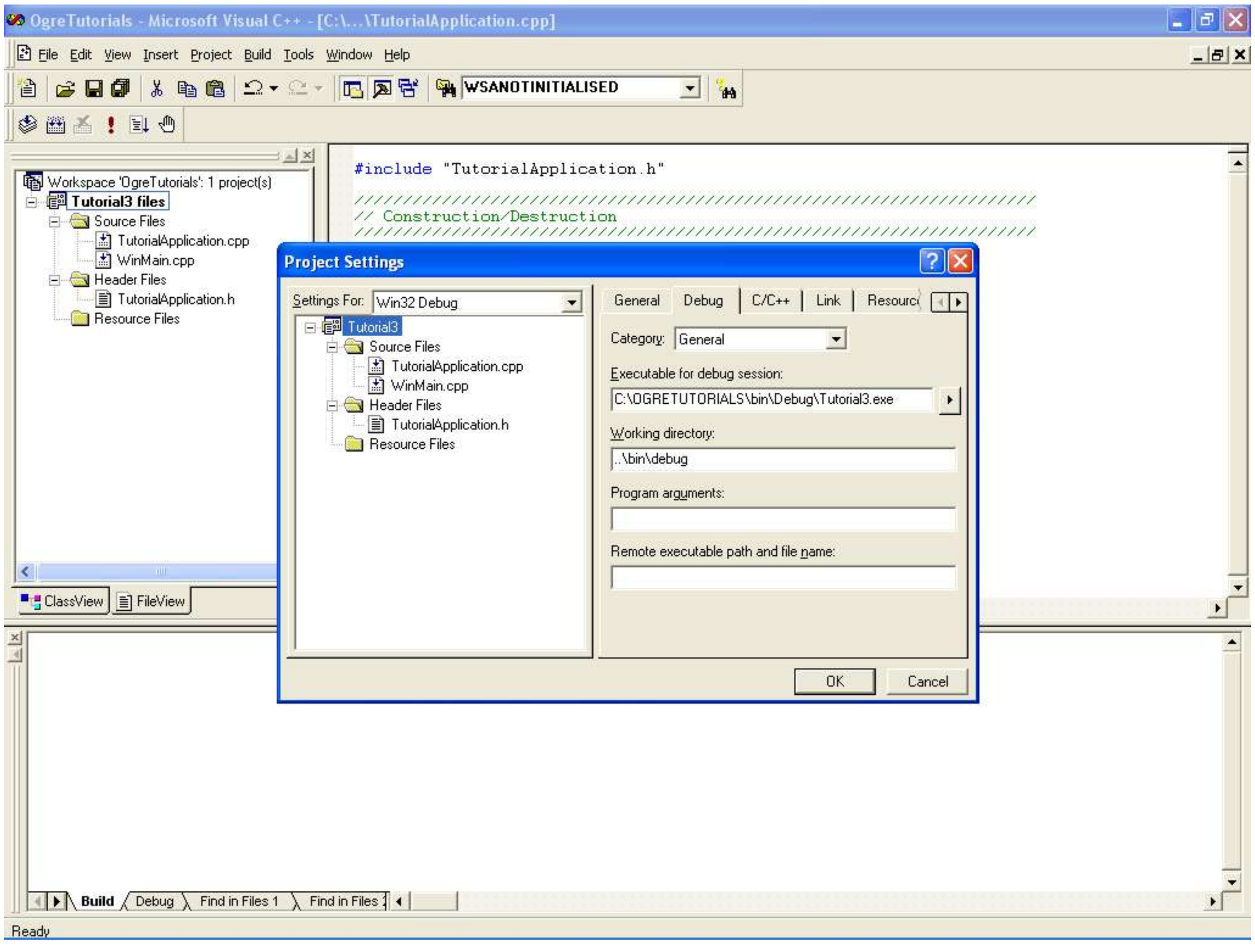
Output file name:
../bin/Debug/Tutorial3.exe

Object/library modules:
OgreMain.lib kernel32.lib user32.lib gdi32.lib winspool.lib co

[✓] Generate debug info
[ ] Ignore all default libraries
[✓] Link incrementally
[ ] Generate mapfile
[ ] Enable profiling

Project Options:
OgreMain.lib kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib

Click on the debug tab, and enter ***..\bin\debug*** to the Working Directory field as shown below. Make sure that the executable for debug session is pointing to C:\OgreTutorials\bin\debug\Tutorial3.exe.


OgreTutorials - Microsoft Visual C++ - [C:\...\TutorialApplication.cpp]

```
#include "TutorialApplication.h"

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////
```

Project Settings

Settings For: Win32 Debug

Tabs: General | Debug | C/C++ | Link | Resource

Category: General

Executable for debug session: C:\OGRETUTORIALS\bin\Debug\Tutorial3.exe

Working directory: ..\bin\debug

Program arguments:

Remote executable path and file name:

We are now in a state that we can start creating our application.. On to the next Tutorial

Tutorial 4: Creating the tutorial Application

This will create a simple application, nothing fancy, yet.

WinMain

Before we start creating the actual application, we need to handle WinMain. WinMain is called when starting an executable.

Add a file called WinMain.cpp to the Tutorial3 project. If you don't know how to do this, then look at the MS Dev Studio Users Guide.

Enter the following code into WinMain.cpp

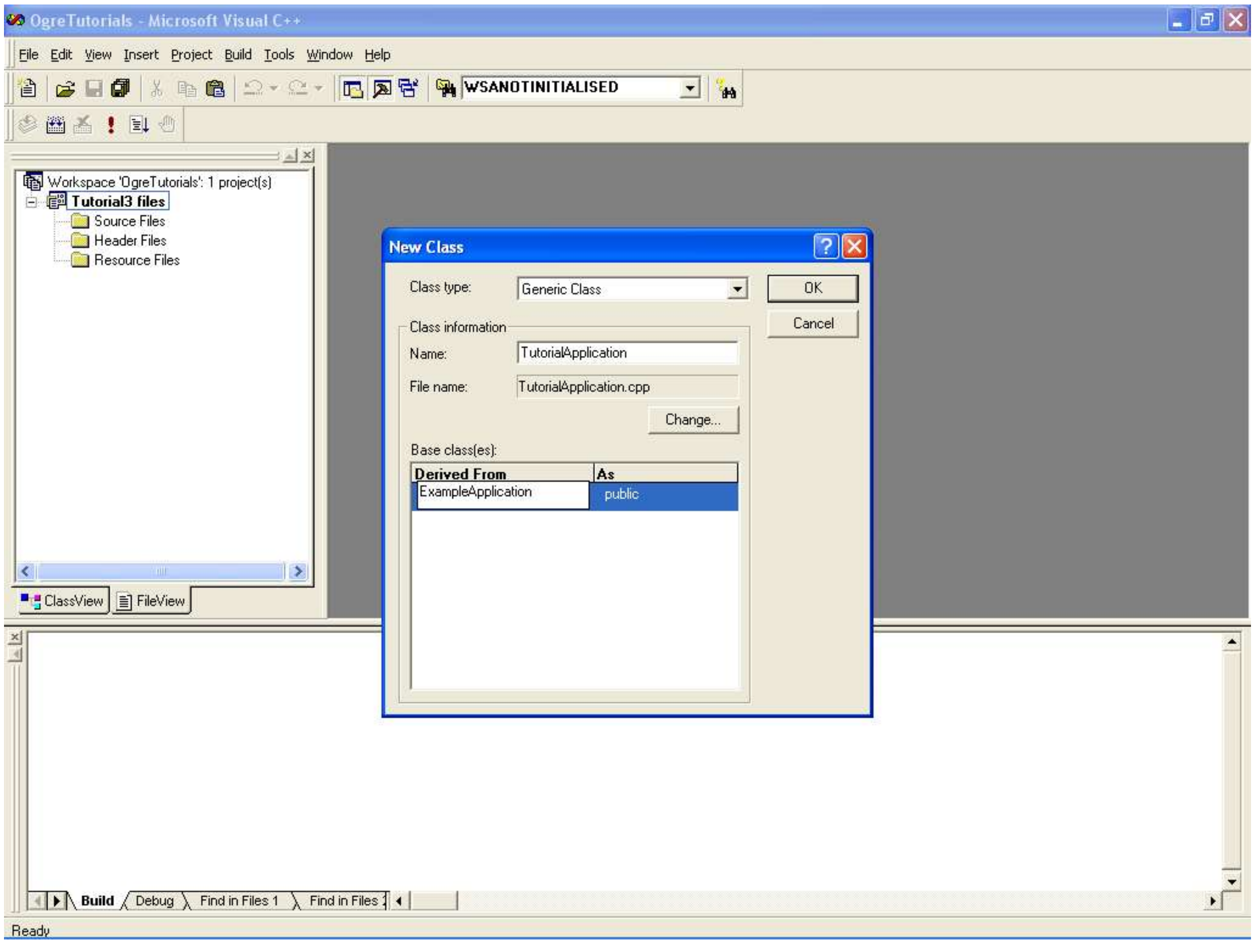
```
// Include the TutorialApplication header
#include "TutorialApplication.h"

// We are always windows, no need for checks
#define WIN32_LEAN_AND_MEAN
#include <windows.h>

// The mean and nasty WinMain
INT WINAPI WinMain( HINSTANCE hInst, HINSTANCE, LPSTR
   strCmdLine, INT )
{
   // Start the application
   try
   {
      // Create an instance of our application
      TutorialApplication TutorialApp;

      // Run the Application
      TutorialApp.go();
   }
   catch ( Ogre::Exception &error )
   {
      // An exception has occured
      MessageBox( NULL, error.getFullDescription().c_str(),
         "An exception has occured!",
         MB_OK | MB_ICONERROR | MB_TASKMODAL );
   }
   return 0;
}
```

What this code does is create the TutorialApplication, calls the go method and provides a nice trap for any exceptions that could occur.

TutorialApplication

Let's now create the actual TutorialApplication. There are two ways you can do this. You can either do this by hand, or you can use the Insert/New Class functionality. We will use the latter for this Tutorial.

Create a new class called TutorialApplication, have it derived from the ExampleApplication, as shown below.



Class type: Generic Class

Class information

Name: TutorialApplication

File name: TutorialApplication.cpp

Base class(es):

Derived From	As
ExampleApplication	public

When you click ok, it will complain that it cannot find the appropriate headers to include of the base class. Just ignore it and click ok anyway.

You have now created two new files. TutorialApplication.h and TutorialApplication.cpp.

Our next step is to modify the generated files of TutorialApplication.

Open up TutorialApplication.h in DevStudio.

Remember when you created this class, DevStudio complained about not finding a header? We will now manually add the required include line.

Above the **class TutorialApplication** line add the following

```
#include "ExampleApplication.h"
```

Unfortunately we cannot compile our Tutorial yet. ExampleApplication has a virtual function defined that needs to be added to our TutorialApplication.

Add the following two lines to the class

```
protected:
   void createScene();
```

Now your class definition should look like the following.

```
#include "ExampleApplication.h"

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

Let's complete the changes required. We need to add a body to the method createScene. Open up TutorialApplication.cpp and add the following.

```
void TutorialApplication::createScene()
{
// empty body for now
}
```

Your first compile

If everything was done correctly, you can now compile. I ran into a little problem when I compiled mine. I tend to use multithreaded code, and had not set the project to be multithreaded, so I received the following load of errors.

```
LIBCMT.lib(osfinfo.obj) : error LNK2005: __alloc_osfhnd already defined in LIBCD.lib
(osfinfo.obj)
LIBCMT.lib(osfinfo.obj) : error LNK2005: __set_osfhnd already defined in LIBCD.lib
(osfinfo.obj)
LIBCMT.lib(osfinfo.obj) : error LNK2005: __free_osfhnd already defined in LIBCD.lib
(osfinfo.obj)
LIBCMT.lib(osfinfo.obj) : error LNK2005: __get_osfhandle already defined in
LIBCD.lib(osfinfo.obj)
LIBCMT.lib(osfinfo.obj) : error LNK2005: __open_osfhandle already defined in
LIBCD.lib(osfinfo.obj)
```

The way to fix this if you get the same errors, is to open the settings for the project, and select the C++ tab. Select Code Generation from the Category combobox. Change the Use run-time library from singlethread debug to multithread debug.

Recompile. If everything worked this time, then you get a valid compile.

So now what? Lets Run this application.

Your First Run

Let me guess, you got an exception about missing resources.cfg. Guess what, lets add in the resources now.

Using File Explorer copy all of the directories and files in

C:\OgreTutorials\OgreNew\Samples\Media to our media directory at ***C:\OgreTutorials\bin\media***

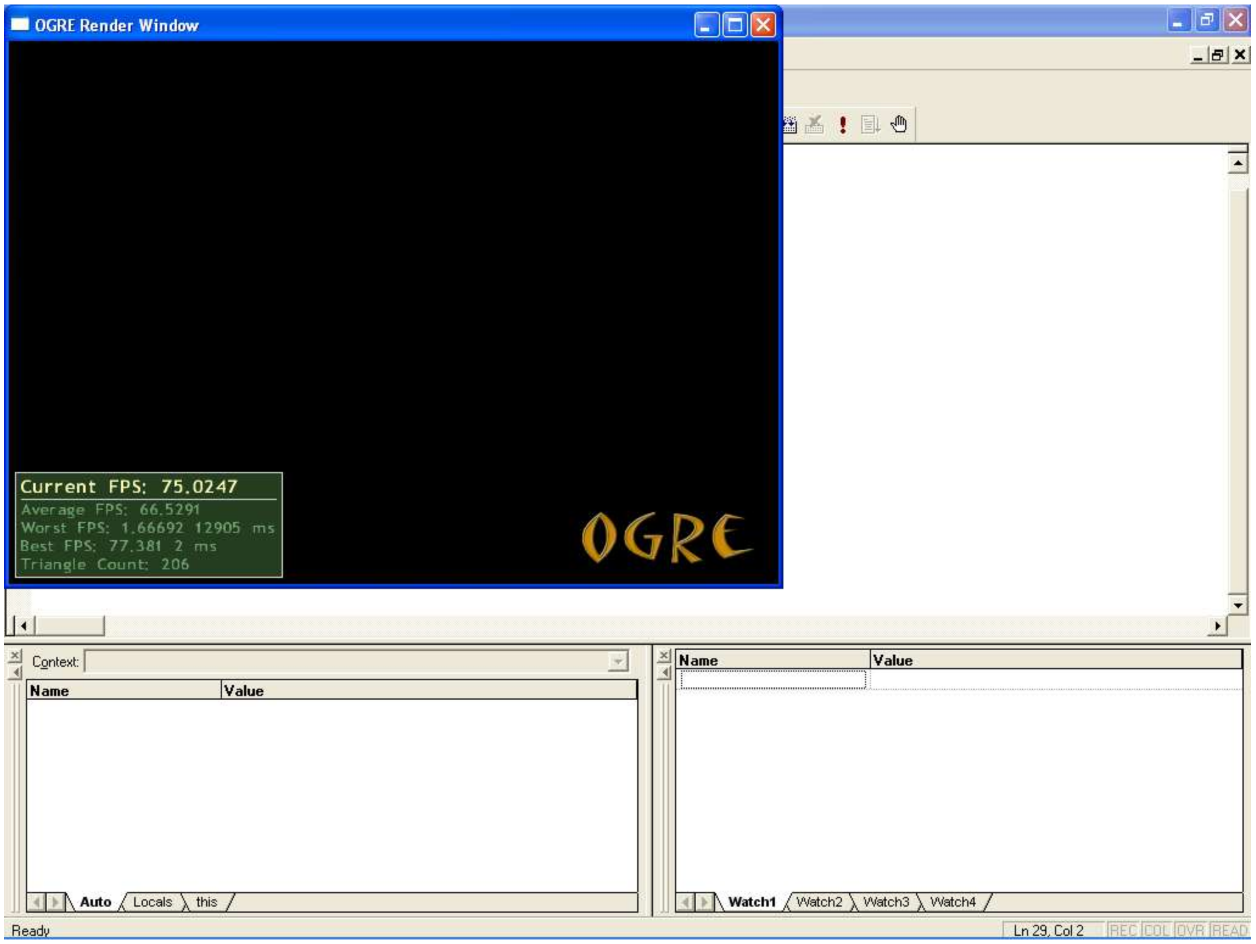
Now copy the file resources.cfg from ***C:\OgreTutorials\OgreNew\Samples\Common\Bin\Debug*** to ***C:\OgreTutorials\Bin\Debug***

We need to edit the resources.cfg for it to work with our resources. Open up resources.cfg in notepad or write. Now change the ***=../../../Media*** to ***=../Media***.

Re-run the Application, if you set the resource file correctly, you should now see the Ogre's Driver Selection Dialog. Selection your driver and hit ok.

You will now see a black screen, with the Ogre Logo in the bottom right and the FPS display in the bottom left.

Here I am running the TutorialApplication in a window using OpenGL.



```
Current FPS: 75.0247
Average FPS: 66.5291
Worst FPS: 1.66692 12905 ms
Best FPS: 77.381 2 ms
Triangle Count: 206
```

Congratulations, you have successfully created and ran your first Ogre Application.

The rest of the tutorials will be building upon this application.

Tutorial 5: Adding something exciting

Wow, Tutorial 5, already, are we having fun yet? From this point on we will be building upon the TutorialApplication we created in the previous steps.

Skybox

In our last Tutorial we had a black background, let's make it more interesting by adding a skybox.

Remember the TutorialApplication::createScene method we had to add? Well this method has been setup to be called once at the start of the application. It is in here, that we add most of the code for setting up what we want to see on the screen.

Let us add our first bit of fancy code.

```
void TutorialApplication::createScene()
{
   // Add a skybox
   mSceneMgr->setSkyBox(true, "Examples/SpaceSkyBox");
}
```

Your probably thinking, whoa, what's this? Where did mSceneMgr come from, what is Examples/SpaceSkyBox ?

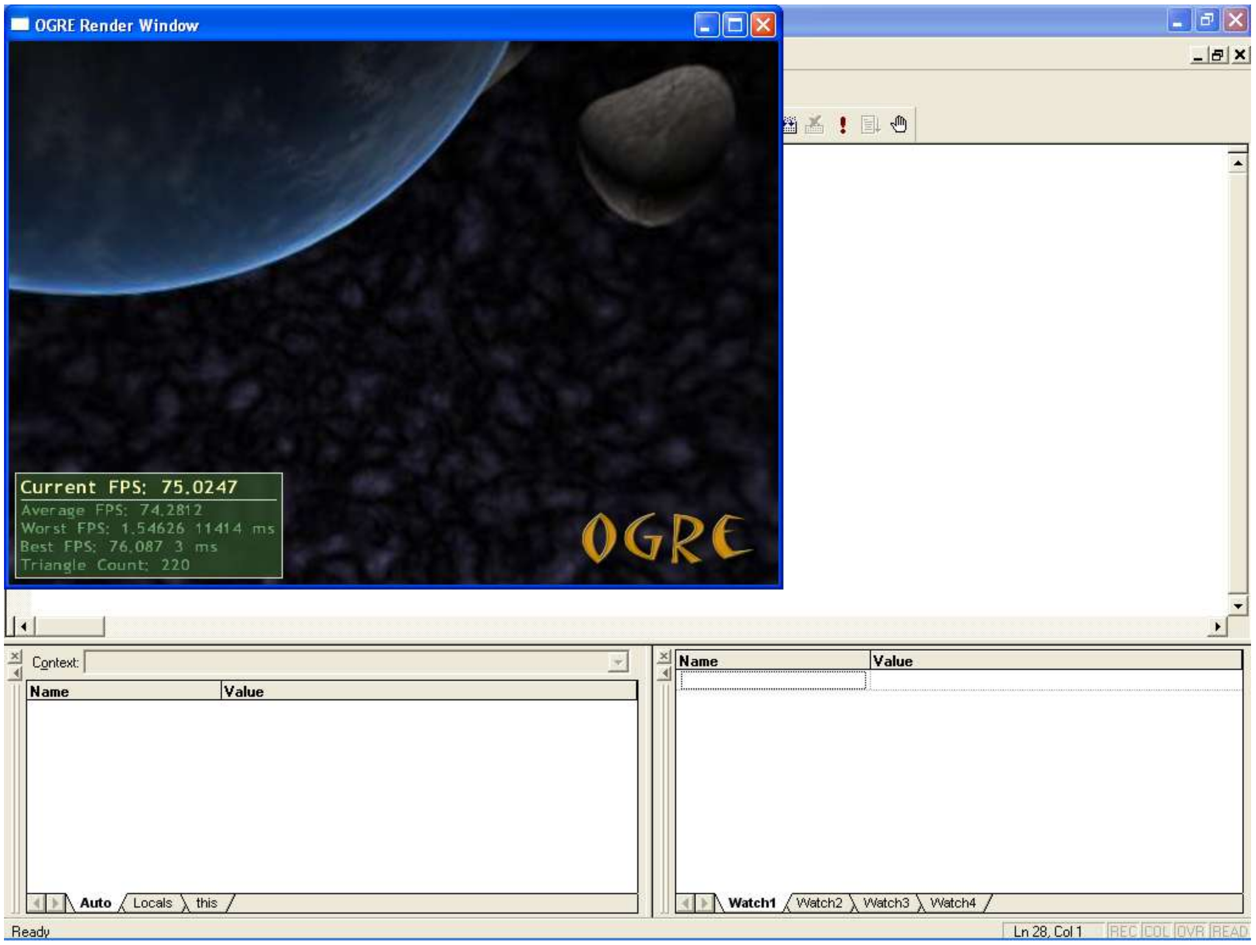
Well, do not panic. mSceneMgr is created by the class we inherited TutorialApplication from. If you look in the ExampleApplication.h file you will see the following code, this is what created the mSceneMgr for us.

```
    virtual void chooseSceneManager(void)
    {
        // Get the SceneManager, in this case a generic one
        mSceneMgr = mRoot->getSceneManager(ST_GENERIC);
    }
```

Later in a different tutorial we will cover choosing different scenemanagers, but for now rest assured that we have a scenemanager.

As for the Examples/SpaceSkyBox, this refers to a material definition. These definitions are found in .material files in the media directory. Materials will be covered in a different tutorial. So let's just go with SpaceSkyBox at the moment.

Now compile the code and run it. You should see something close to the following



```
Current FPS: 75.0247
Average FPS: 74.2812
Worst FPS: 1.54626 11414 ms
Best FPS: 76.087 3 ms
Triangle Count: 220
```

Hey cool, you have a skybox. Did you notice that you could spin around using the mouse? Again, this is default code that is supplied by ExampleApplication.

A ship

As cool as this is, what now? I want to do some more. How about adding a user controlled ship?

So how do we go about doing this? Well one of the first things to understand is how Ogre works with scenemanagers.

Mesh based objects are not part of the scenegraph, they are in fact objects attached to SceneNodes in the scenegraph. SceneNodes are what gives the entities their position, rotation, scale etc in the scene.

For us to create our ship we need to create an entity and a SceneNode to attach the entity to.

Open up TutorialApplication.h again, as we need to add some member variables.

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

// ADD THE FOLLOWING TWO LINES TO YOUR CODE
   Entity* mShip;
   SceneNode* mShipNode;
};
```

mShipNode will be the scenegraph node we will attach our ship entity mShip to.

Now open up the TutorialApplication.cpp, we need to add code to the createScene method.

```
void TutorialApplication::createScene()
{
   // Create the SkyBox
   mSceneMgr->setSkyBox(true, "Examples/SpaceSkyBox");

   // Create our ship entity
   mShip = mSceneMgr->createEntity("razor", "razor.mesh");
```

The first entry is still our skybox. The second entry is a new one that creates our ship. Ogre comes with a plane/ship model that is called razor. For this tutorial we will be using the Ogre supplied model. Later I may do another tutorial for adding in our own models from milkshape.

createEntry requires two parameters, the first being a general but unique name we want to call our entity, the second parameter is the name of the mesh that is to be displayed. The mesh must be in our media resource directory.

Add the following two lines below the createEntity line.

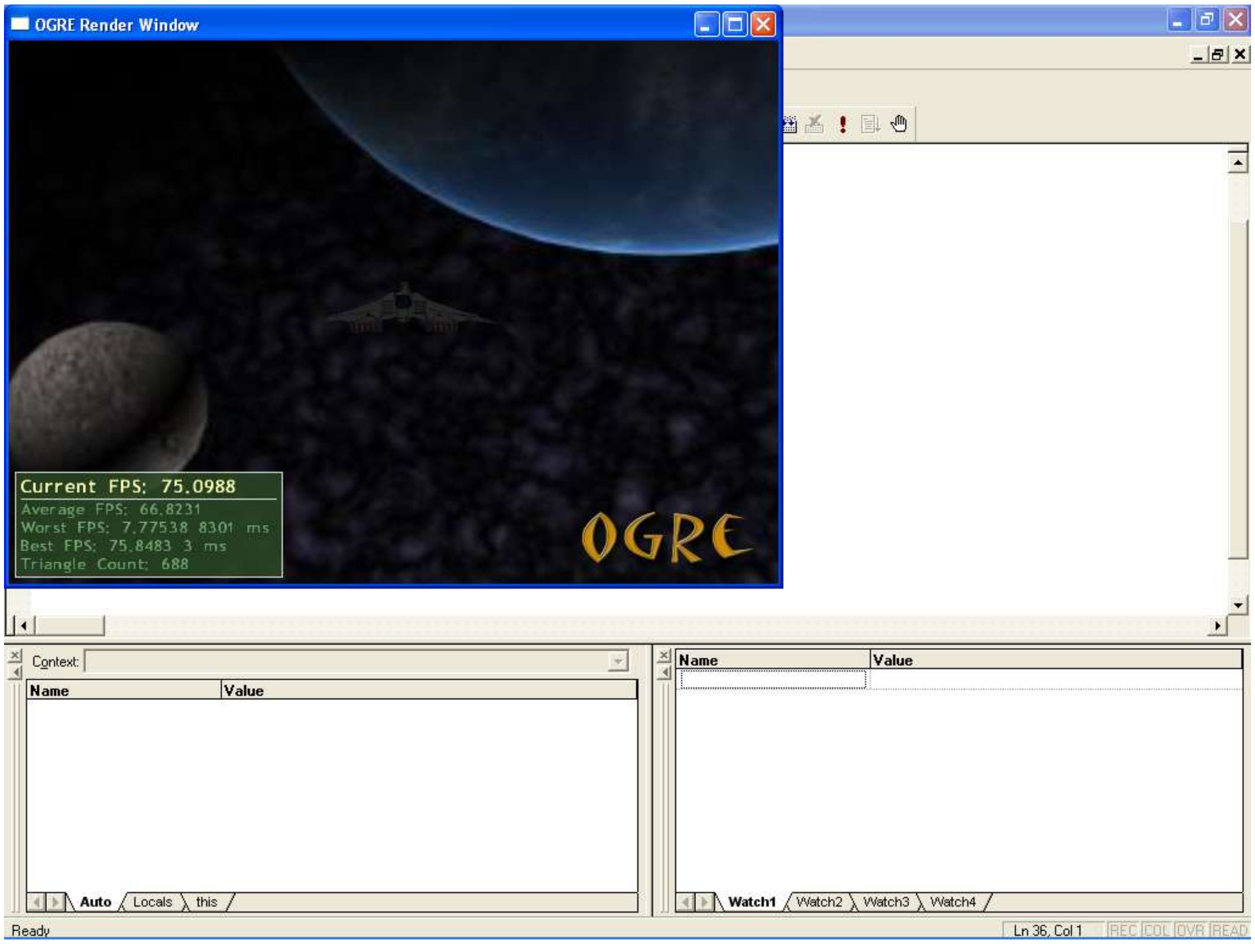
```
mShipNode = mSceneMgr->getRootSceneNode()->createChildSceneNode
();
mShipNode->attachObject(mShip);
```

When the SceneManager is created, a root scene node is created. This root node is used for the rendering the scene. So to have something render, we need to create a child SceneNode from the RootSceneNode.

Once we have our node, we then attach our ship entity to it.

Space is dark

Let's compile and run, we should see a ship in our screen now.



```
Current FPS: 75.0988
Average FPS: 66.8231
Worst FPS: 7.77538 8301 ms
Best FPS: 75.8483 3 ms
Triangle Count: 688
```

If yours like mine, then you're wondering why the ship is so dark. The razor material has been setup to reflect to lights. Since no lights have been added to the scene it is using the ambient light and the ambient light is set very low.

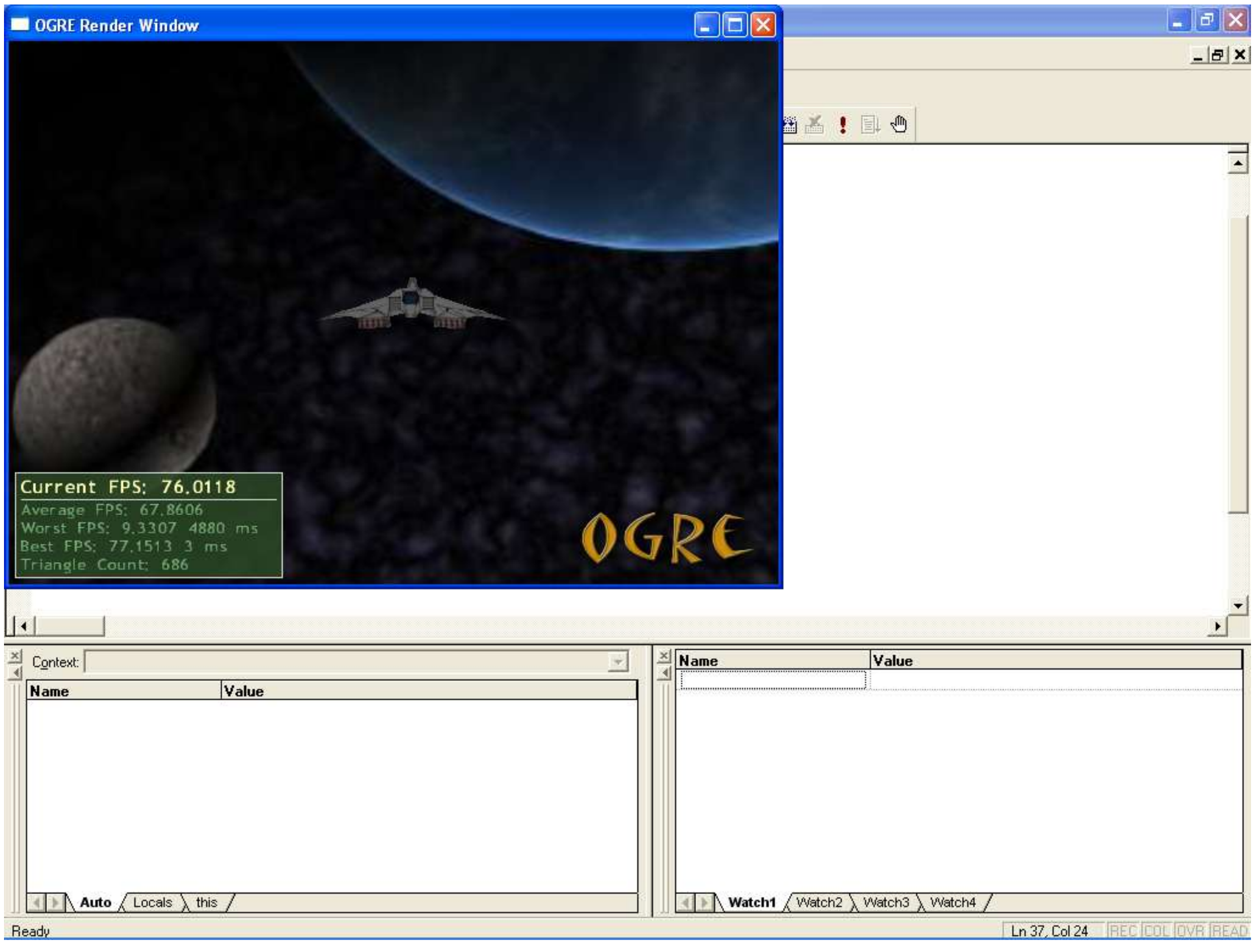
For now, we will just adjust the ambient light. Add the following line below the attachObject line in our createScene Method

```
mSceneMgr->setAmbientLight(ColourValue(0.5, 0.5, 0.5));
```

Ogre Colour elements are in the range of 0.f to 1.0f. Some other 3D engines use 0 to 255.

Black = ColourValue(0.f, 0.f, 0.f)
White = ColourValue(1.f, 1.f, 1.f)
Red = ColourValue(1.f, 0.f, 0.f)

Recompile and rerun the TutorialApplication



```
Current FPS: 76.0118
Average FPS: 67.8606
Worst FPS: 9.3307 4880 ms
Best FPS: 77.1513 3 ms
Triangle Count: 686
```

Yup, the ship is brighter, now using the mouse and the keyboard, you can move around the ship.

As you can see, if it fairly simple to add entities to the scene graph. Try adding a two more ships one either side of the original one.

Hint: SceneNode has a function called setPosition;

Who is flying my ship?

Well I did promise a user controlled ship, so this tutorial is not over yet.

Ogre uses a built in render loop. It is possible to create and manage your own render loop, but that is out of scope on these initial tutorials. Anyway, we need a way of plugging in code into this render loop. Ogre provides us a way by using what are called FrameListeners. These FrameListeners are called at the beginning of the render loop and at the end of the render loop. That way, we can plug in code, that needs to be executed before rendering and after rendering.

In this case we will only be interested in plugging in code before the render starts. For our example we need to create our own version of a FrameListener.

Open up the TutorialApplication.cpp file.

We will be adding code after the include but before the TutorialApplication constructor.

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

public:
 TutorialFrameListener (RenderWindow* win, Camera* cam,
SceneNode* shipNode) :
        ExampleFrameListener(win, cam)
    {
        mShipNode = shipNode;
    };
    bool frameStarted(const FrameEvent& evt);
};
```

Why did we inherit from ExampleFrameListener ? Well ExampleFrameListener has some need stuff already added, so it just made sense for now.

Here we have a custom constructor, RenderWindow and Camera are needed by the FrameListener. But SceneNode is not, we pass it because we want to move our ship around. If we did not pass the node, we would have no idea where it was, unless we created it globally.

The last line of the above class definition is an important one. FrameListerner has two methods that get called. FrameStarted is called before rendering frameEnded is called after rendering. FrameEvent is a simple structure that will hold the delta time between frames and between events.

Considering we want to add code before the render starts, we add the frameStarted method override.

Let's add some body to the frame started. Add the following code after the class definition.

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

   mInputDevice->capture();

   if(mInputDevice->isKeyDown(Ogre::KC_UP))
      mShipNode->translate(0.0, MoveFactor, 0.0);

   if(mInputDevice->isKeyDown(Ogre::KC_DOWN))
      mShipNode->translate(0.0, -MoveFactor, 0.0);

   if(mInputDevice->isKeyDown(Ogre::KC_LEFT))
      mShipNode->translate(-MoveFactor, 0.0, 0.0);

   if(mInputDevice->isKeyDown(Ogre::KC_RIGHT))
      mShipNode->translate(MoveFactor, 0.0, 0.0);

   return true;
}
```

Remember I said we could plug in code into the render loop. One of the things you can do is react to user input. This is the reason we are adding code. We want to be able to control the ship.

Ogre uses arbitrary units for its positions/rotations/scale. So moving one unit could mean one Millimeter or one Kilometer or one Parsec. It all depends on your scale. In our tutorial we will assume the ship can move 80 units per second.

But how do we move 80 units when we have a frame rate display of say 100? Well that is where **evt.timeSinceLastFrame** comes in. This is the time between frames as a percent of a second. So for us to figure out how far we move the ship in this frame, we multiply 80 by evt.timeSinceLastFrame giving us the MoveFactor as in the code.

OK cool, we have the move distance, but now we need user input. Luckily ExampleApplication comes to our rescue again. Ogre uses a PlatformPlugin that gets automatically linked. In windows, the plugin uses DirectInput to query the Keyboard and Mouse only. I personally have expanded my plugin to work with my gamecontroller (Joystick).

ExampleApplication provides us with an instance of this plugin though the variable mInputDevice.
For us to get information from the DirectInput we first need to capture the input. This is done by calling **mInputDevice->capture();**

Now the input device has captured the state of the keyboard and mouse. As we are only interested in the keyboard, we can query the input device to see if certain keys are pressed. This is done by the command **mInputDevice->IsKeyDown(Ogre::KC_UP).** Ogre has a number of enumerated keys, KC_UP is just one of them.

Now we have the Move Rate, and we know a certain key is down, we have all we need to move our ship. Remember how I said that SceneNode gives an entity position and rotation?

Now we need to apply the move rate to the scenenode.

This is done via SceneNodes::translate method. This method excepts three parameters. X, Y and Z. I should explain that Ogre uses a right handed coordinate system with Y being vertical and Z being depth. So to move our ship up and down on the screen we need to enter the move rate as Y.

The code above will move our ship up and down and left and right on the screen.

Hey, don't compile yet. We have to hook the TutorialFrameListener in to our TutorialApplication yet.

While we are still in the TutorialApplication.cpp file, go to the bottom of the file and add the following code.

```
void TutorialApplication::createFrameListener()
{
    mFrameListener= new TutorialFrameListener (mWindow, mCamera,
mShipNode);
    mRoot->addFrameListener(mFrameListener);
}
```

ExampleApplication has a virtual method called createFrameListener. We override this method to stop it from creating the default frameListener and instead create our new TutorialFrameListener.

Once created we actually have to add the framelistener to mRoot. What is mRoot? Ogre has a base class that helps to link and manage the different components of Ogre. This is called Root. mRoot is an instance of Root. A different tutorial will go into Root more.

We are almost done.

Now open TutorialApplication.h.

Add the following line before createScene in your class.

```
    void createFrameListener();
```

Your class def should look like the following now

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


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

STOP!! Don't run your application yet. To save you the embarrassment that I had when I ran the app, we need to add a little bit more code.

Reopen TutorialApplication.cpp and add the following to the TutorialFrameListener::frameStarted just before **return true**.

```
   if( mInputDevice->isKeyDown( KC_ESCAPE) )
   {
      return false;
   }
```

By overriding the frameStarted method the way we did, we could no longer press escape to exit the application. This code allows us to continue doing this. FrameStarted returns a Boolean, this bool tells the Ogre render loop whether to breakout of the loop or to continue. If True is returned then the render loop will keep going.

Compile and run your app. You can now move your ship up/down and left/right.

Congratulations you have completed your second Ogre Application.

This concludes the first set of tutorial I will be creating. Look forward to more in the future covering different aspects of Ogre.

Xorekis