TUTORIAL

OGRE

HOW-TO…

SUMMARY

1 Introduction

1.1 How-to list

Written by Oogst, last update on 25-05-2004, check the Oogst-site for more Oogst-stuff.
Completed by Sphinx on 03-08-2004.

1.2 About this list

This is a list of how to do simple things in Ogre. It contains mainly simple code examples, which is something I myself really missed while learning the basics of Ogre. The explanations provided here are all kept brief. For more details on all the things mentioned, see:
-The official manual for the Ogre-basics and the script-definitions;
-The API for descriptions of all classes and functions in Ogre;
-The tutorials provided with Ogre;
-The search-function of the Ogre-forum.

1.3 Why I made this list

I made this list for four reasons:
-I am working on a group project at school and the rest of the team would like to be able to find code examples quickly;
-I really missed this kind of thing when learning Ogre myself and it would have saved me a lot of time;
-I keep forgetting function-names and this way I can look them up quickly;
-I wanted to do something in return for the fantastic engine the Ogre-team has provided and which I can use for free without any expectation from the creators of getting something back for it.

1.4 ExampleApplication

This list works from the *ExampleApplication* and supposes the user uses *ExampleApplication* and *ExampleFrameListener*. Some variables, like *mSceneMgr*, are only available with this name through *ExampleApplication*. If you are not using *ExampleApplication*, you will have to fill in the different variables yourself.

Furthermore, all examples use arbitrary names for variables and such; of course you should change these to what you find appropriate for your own application.

2 How to …

2.1 How to set up an application using *ExampleApplication*

If you do not know how this works already, you MUST read more about it in Ogre's tutorial-section: it is really important to understand this properly. In short it requires creating a class that is derived from *ExampleApplication*. This class can implement the functions *createScene*() and *createFrameListener*(). It will probably look something like this:

```
#include "ExampleApplication.h"

class MyClass: public ExampleApplication,
{
public:
  MyClass(void);
  ~MyClass(void);
protected:
  void createScene(void);
  void createFrameListener(void);
};
```

You will also need a .cpp-file that creates one instance of MyClass and runs it. In general it will look something like this (taken from the "Guide To Setting Up Application Project Files"-tutorial):

```
#include "Ogre.h"
#include "MyClass.h"

#if OGRE_PLATFORM == PLATFORM_WIN32
#define WIN32_LEAN_AND_MEAN
#include "windows.h"

INT WINAPI WinMain( HINSTANCE hInst, HINSTANCE, LPSTR strCmdLine, INT )
#else
int main(int argc, char **argv)
#endif
{
  MyClass app;

  try
  {
    app.go();
  }
  catch( Exception& e )
  {
#if OGRE_PLATFORM == PLATFORM_WIN32
    MessageBox( NULL, e.getFullDescription().c_str(), "An exception has occured!",
                MB_OK | MB_ICONERROR | MB_TASKMODAL);
#else
    fprintf(stderr, "An exception has occured: %s\n", e.getFullDescription().c_str());
#endif
  }
  return 0;
}
```

2.2 How to put a 3D-model in the scene

A 3D-model is an Entity and it must be attached to a *SceneNode* to be placed in the scene. The *SceneNode* can be taken from the *rootSceneNode* in the *SceneManager*. Creating an Entity and *SceneNode* and attaching the Entity to the *SceneNode* will look something like this:

```
Entity* thisEntity = mSceneMgr->createEntity("nameInTheScene", "FileName.mesh");
SceneNode* thisSceneNode =
            static_cast<SceneNode*>(mSceneMgr->getRootSceneNode()->createChild());
thisSceneNode->attachObject(thisEntity);
```

2.3 How to remove a 3D-model from the scene

To remove a 3D-model, you must detach the Entity from its *SceneNode*, delete it and if needed destroy the *SceneNode* as well, which must be done using the *SceneManager*. If you have a *SceneNode**

called *myNode* you can completely destroy all its contents (both *MovableObjects* and child *SceneNodes*) and the node itself using the following code:

```
while(myNode->numAttachedObjects() > 0)

{
  MovableObject* thisObject = myNode->detachObject(static_cast<unsigned short>(0));
  delete thisObject;
}
myNode->removeAndDestroyAllChildren();
mSceneMgr->getRootSceneNode()->removeAndDestroyChild(myNode->getName());
```

2.4 How to move, reposition, scale and rotate a *SceneNode*

If you have a *SceneNode* with some *MovableObjects*, like Entities, Lights and Cameras, attached to it, you can move it using a lot of different functions, see the API for all of them. The following functions respectively move it, reposition it, scale it and rotate it over its X-, Y- and Z-axis:

```
thisSceneNode->translate(10, 20, 30);
thisSceneNode->setPosition(1.8, 20.1, 10.5);
thisSceneNode->scale(0.5, 0.8, 1.3);
thisSceneNode->pitch(45);
thisSceneNode->yaw(90);thisSceneNode->roll(180);
```

2.5 How to put a light in the scene

To add a light to the scene, you must ask the *SceneManager* for one. You can then set its settings, of which some examples are given below:

```
Light* myLight = mSceneMgr->createLight("nameOfTheLight");
myLight->setType(Light::LT_POINT);
myLight->setPosition(200, 300, 400);
myLight->setDiffuseColour(1, 1, 0.7);
myLight->setSpecularColour(1, 1, 0.7);
```

You can also attach a light to a *SceneNode*. The following code creates a *SceneNode* and attaches *myLight* to it:

```
SceneNode* thisSceneNode = static_cast<SceneNode*>
                                (mSceneMgr->getRootSceneNode()->createChild());
 thisSceneNode->attachObject(myLight);
```

2.6 How to set the ambient lighting

The ambient lighting is controlled by the Scenemanager, so that is where you can set it:

```
mSceneMgr->setAmbientLight(ColourValue(0.2, 0.2, 0.2));
```

2.7 How to control the camera

The standard camera in *ExampleApplication* is called *mCamera* and is available in the class that is derived from *ExampleApplication*. The following code changes its position, changes the point it looks at, creates a *SceneNode* and attaches the camera to it:

```
mCamera->setPosition(0, 130, -400);
mCamera->lookAt(0, 40, 0);
SceneNode* thisSceneNode = static_cast<SceneNode*>
                                (mSceneMgr->getRootSceneNode()->createChild());
thisSceneNode->attachObject(mCamera);
```

2.8 How to add a billboard/sprite to the scene

A Billboard is a square polygon that is always pointed at the camera. It is also known as a sprite. To make one, you must first make a *BillboardSet*. Then the billboard can be added to it on a given position. The *BillboardSet* is a *MovableObject* en should therefore be added to a *SceneNode*. The whole procedure is as follows:

```
SceneNode* myNode = static_cast(mSceneMgr->getRootSceneNode()->createChild());
BillboardSet* mySet = mSceneMgr->createBillboardSet("mySet");
Billboard* myBillboard = mySet->createBillboard(Vector3(100, 0, 200));
myNode->attachObject(mySet);
```

2.9 How to create a basic *FrameListener* using *ExampleFrameListener*

A *FrameListener* gives you the opportunity to do something at the start and the end of every frame. You must first create a class that is derived from *ExampleFrameListener* and in this class you can implement *frameStarted()* and *frameEnded().* This will look something like this:

```
#include "ExampleFrameListener.h"

class myFrameListener: public ExampleFrameListener
{
public:
  myFrameListener(RenderWindow* win, Camera* cam);
  ~myFrameListener(void);
  bool frameStarted(const FrameEvent& evt);
  bool frameEnded(const FrameEvent& evt);
};
```

The constructor should call its parent-constructor, which will look something like this:

```
myFrameListener::myFrameListener
                    (RenderWindow* win, Camera* cam): ExampleFrameListener(win, cam){}
```

You must also register your *FrameListener* to the Root. This can be done in the *createFrameListener ()*-function of the class that is derived from *ExampleApplication*. You can register as many *FrameListeners* to the root as you want. It will look something like this:

```
void createFrameListener(void)
{
  MyFrameListener listener = new MyFrameListener(mWindow, mCamera);
  mRoot->addFrameListener(listener);
}
```

2.10 How to control some object from your scene in a *FrameListener*

If you want to control some object you have in your scene in a *FrameListener*, the *FrameListener* must have access to it. An easy way to do this, is by providing a pointer to it in the constructor of the *FrameListener*. Its constructor will now look something like this:

```
myFrameListener(RenderWindow* win, Camera* cam, Car* car);
```

2.11 How to get the time since the previous frame

In the functions *frameEnded()* and *frameStarted()* of a *FrameListener* you can get the time in seconds (this is a float) in the following way:

```
bool frameStarted(const FrameEvent& evt)
{
  float time = evt.timeSinceLastFrame;
  return true;
}
```

You can now for instance multiply the speed per second with this float in order to get the movement since the last frame. This will make the pace of the game framerate-independent.

2.12 How to react to key-presses

In the functions *frameEnded()* and *frameStarted()* of a *FrameListener* you can react to key-presses by first ordering Ogre to capture them and then checking which key is being pressed. You only have to capture the *InputDevice* once per frame. This will look something like this:

```
mInputDevice->capture();
if (mInputDevice->isKeyDown(Ogre::KC_DOWN))
{
  //react however you like
}
if (mInputDevice->isKeyDown(Ogre::KC_UP))
{
  //react however you like
}
```

2.13 How to make sure key-presses are not reacted to too shortly after each other

If you implement reactions to key-presses in the above way, they will happen each frame. If you want them to happen, for instance, at most twice per second, you can achieve this by setting a timer. This timer must be kept in the class itself and not in the function in order to be able to access it through different calls of the function. Implementing this will look something like this:

```
class myFrameListener: public ExampleFrameListener
{
protected:
  float buttonTimer;
public:
  myFrameListener(RenderWindow* win, Camera* cam): ExampleFrameListener(win, cam)
  {
    buttonTimer = 0;
  }
  bool frameStarted(const FrameEvent& evt)
  {
    float time = evt.timeSinceLastFrame;
    buttonTimer -= time;
    mInputDevice->capture();
    if (mInputDevice->isKeyDown(Ogre::KC_DOWN) && buttonTimer <= 0)
    {
      buttonTimer = 0.5;
      //react however you like
    }
    return true;
  }
};
```

2.14 How to quit your application

You can quit you application in the *frameStarted()* or *frameEnded()*-function of the *FrameListener* by returning false. If you want to tie this to pressing the escape-button on the keyboard, this will look as follows:

```
bool frameStarted(const FrameEvent& evt)
{
  mInputDevice->capture();
  if (mInputDevice->isKeyDown(Ogre::KC_ESCAPE))
    return false;
  return true;
}
```

2.15 How to efficiently add and remove 3D-objects from the scene during run-time (like rockets being fired)

In the *createScene()*-function you can load meshes from a file, but this is too slow to do if new objects must be added in run-time (though in a very simple application you will not see the difference in speed). To add objects faster, you can load a lot of meshes in the *createScene()*-function and then take them from a stack in run-time. After the objects are not used anymore, you can put them back on the stack for later use. A good example of the usefulness of this is a canon that fires rockets: these must be created when fired and removed when exploded.

In order to always have access to this stack of rockets, you can make it a global variable (outside any class) or access it through a singleton. The latter is much nicer, but takes more code so I will not do so in this example. So you keep a stack of Entities somewhere:

```
stack rocketEntities;In createScene()
```

you fill this stack with a lot of rockets. Each Entity is set invisible for now and must have a unique name, which is achieved using the sprintf()-function. All in all it looks like this:

```
for (unsigned int t = 0; t < 100; t++)
{
  char tmp[20];
  sprintf(tmp, "rocket_%d", t);
  Entity* newRocketEntity = mSceneMgr->createEntity(tmp, "Rocket.mesh");
  newRocketEntity->setVisible(false);
  rocketEntities.push(newRocketEntity);
}
```

Now when creating a new Rocket we can take a mesh from *rocketEntities*. In this example I do so in the constructor of the Rocket and put it pack in the destructor of the Rocket. In order to get the Entity back from the *SceneNode* in the destructor, I store its name in *rocketEntityName*. I also position the rocket correctly in the constructor. To make this all work, the *SceneManager* must be available throughout your program; in this case this is achieved by having it be a global variable (outside any class). I also handle creation and destruction of the *SceneNode* in the constructor and destructor. The Rocket-class will now look like this:

```
class Rocket
{
protected:
  SceneNode* rocketNode;
  string rocketEntityName;
public:
  Rocket(const Vector3& position, const Quaternion& direction)
  {
    rocketNode = static_cast<SceneNode*>(sceneMgr->getRootSceneNode()->createChild());
    Entity* rocketEntity = rocketEntities.top();
    rocketEntities.pop();
    rocketEntity->setVisible(true);
    rocketEntityName = rocketEntity->getName();
    rocketNode->attachObject(rocketEntity);
    rocketNode->setOrientation(direction);
    rocketNode->setPosition(position);
  }
  ~Rocket()
  {
                Entity*  rocketEntity  =  static_cast<Entity*>(rocketNode->detachObject
(rocketEntityName));
    rocketEntity->setVisible(false);
    rocketEntities.push(rocketEntity);
    sceneMgr->getRootSceneNode()->removeAndDestroyChild(rocketNode->getName());
  }
};
```

If you use this construction, you should be aware of the fact that this crashes if no rockets are left. You can solve this by checking whether the stack of Entities is empty and if it is, load new meshes to add to the stack.

2.16 How to show an Overlay (and hide it again)

To show and hide an Overlay, you must first get a pointer to it using the *OverlayManager*, which is a singleton. If you have an Overlay that is defined in a .overlay-script with the name "*myOverlay*" you can get it, show it and hide it again using the following code:

```
    Overlay* thisOverlay = static_cast<Overlay*>
                                (OverlayManager::getSingleton().getByName("myOverlay"));
    thisOverlay->show();
    thisOverlay->hide();
```

If you want to know if the overlay is displayed (or not) on the screen, you can use:

```
    Overlay* thisOverlay;
    bool visible;

    thisOverlay = mCamera->getSceneManager()->getOverlay ("myOverlay");
    visible = thisOverlay->isVisible();
```

2.17 How to change the text in an Overlay

You can change all parameters of Containers and Elements in the GUI in runtime. In order to do so, you must first get a pointer to the Element or Container you want and, if needed for what you want to do, cast it to the type it is. Getting a pointer to a *TextArea* that is defined in a .overlay-script as "*myTextArea*" and changing its caption will look something like this:

```
    GuiElement* thisTextArea;
    thisTextArea = GuiManager::getSingleton().getGuiElement("myTextArea");
    thisTextArea->setCaption("text to print");
```

Note: the overlay file should be in a directory referenced by resources.cfg.
Note: these files are case-sensitive.

In this case no casting was needed, as every *GuiElement* has a caption. If you want to set a setting that is specific for one type of *GuiElement*, you will have to cast it to that type. Changing the font-name of a *TextArea* will look something like this:

```
    TextAreaGuiElement* thisTextArea;
    thisTextArea =  static_cast<TextAreaGuiElement*>
                                                (GuiManager::getSingleton().getGuiElement
("myTextArea"));
    thisTextArea->setFontName("RealCoolFont");
```

2.18 How to show the mouse-cursor

If you want to show the mouse-cursor on the screen, you have to do two things: set it to be shown and tell a *FrameListener* to track it. Telling it to be shown can be done as follows:

```
    GuiContainer* cursor = OverlayManager::getSingleton().getCursorGui();
    cursor->setMaterialName("Cursor/default");
    cursor->setDimensions(32.0/640.0, 32.0/480.0);
    cursor->show();
```

Telling a FrameListener to track it should be done in its parents' constructor by setting its last boolean-parameter to true. The constructor will now look something like this:

```
    myFrameListener::myFrameListener
              (RenderWindow* win, Camera* cam): ExampleFrameListener(win, cam, false, true)
              {}
```

Beware though that after doing this, this specific *FrameListener* will not react to button-presses anymore as the *capture()*- and *isKeyPressed()*-functions of *mInputDevice* now do not work properly anymore. A different *FrameListener* should be made to handle keyboard input now.

2.19 How to create a working button

You can define a button in an Overlay-script using something like the following syntax, where it is important to set all these materials:

```
container Button(myButton)
{
  metrics_mode relative
  horz_align left
  vert_align top
  top 0.1
  left 0.1
  width 0.18
  height 0.1
  material NCG/GuiSession/RedMaterial
  button_down_material NCG/GuiSession/RedMaterial
  button_up_material NCG/GuiSession/RedMaterial
  button_hilite_down_material NCG/GuiSession/RedMaterial
  button_hilite_up_material NCG/GuiSession/RedMaterial
  button_disabled_material NCG/GuiSession/RedMaterial
}
```

Buttons are not standard in Ogre and therefore you must make sure they are found by the compiler in the folder *ogrenew\PlugIns\GuiElements\Include*. Now you can include it:

```
#include "OgreButtonGuiElement.h"
```

To make the button work, an *ActionListener* must be registered to it, which can be done as follows:

```
ActionTarget* thisButton = static_cast<ButtonGuiElement*>
                                (GuiManager::getSingleton().getGuiElement("myButton"));
thisButton->addActionListener(this);
```

This examples supposes 'this' is an *ActionListener*, which is not true by default. Of course any *ActionListener* will do, it does not have to be 'this'. You can make a class an *ActionListener* by deriving it from *ActionListener* and implementing *actionPerformed()*. This will look something like this:

```
class Listener: public ActionListener
{
public:
  void actionPerformed(ActionEvent *e);
};
```

2.20 How to find out which button was pressed

If you register one *ActionListener* to several buttons, they will all call the same function *actionPerformed*(). You can find out which button was actually being pressed by comparing its name with the name in the *ActionEvent* e. This will look something like this:

```
#include

void actionPerformed(ActionEvent *e)
{
  std::string action = e->getActionCommand();
  if (action == "myButton")
  {
    //handle the button-press
  }
}
```

2.21 How to quit the application using an ActionListener

Quitting an application can be done in the *frameStarted()* and *frameEnded()* functions of a *FrameListener*, not in the *actionPerformed()* function. So how do we get there? For this we can introduce a simple *FrameListener* that does nothing more than quit if asked to. This *FrameListener* will look something like this (from the GUI-demo provided with Ogre):

```
#include "ExampleFrameListener.h"

class QuitListener: public ExampleFrameListener
{
public:
  QuitListener(RenderWindow* win, Camera* cam): ExampleFrameListener
                                                    (win, cam, false, false)
  {
```

```
      quit = false;
    }
    bool frameStarted(const FrameEvent& evt)
    {
      if (quit)
        return false;
      return true;
    }
    void scheduleQuit(void)
    {
    quit = true;
    }
  protected:
    bool quit;
  };
```

Now if your *ActionListener* has a pointer to this *QuitListener*, it can simply schedule a quit by calling *scheduleQuit()* and the *QuitListener* will perform it before the next frame starts.

2.22 How to get a different *SceneManager*

The *SceneManager* is chosen automatically using an identifier that tells what kind of scene it is. This is done in *ExampleApplication*, so to change this, you will have to change *ExampleApplication* itself. Normally, *ExampleApplication* contains the following piece of code:

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

You can change ST_GENERIC to any of the following identifiers to get a *SceneManager* that is appropriate for your specific scene: ST_GENERIC ST_EXTERIOR_CLOSE ST_EXTERIOR_FAR ST_EXTERIOR_REAL_FAR ST_INTERIOR

2.23 How to efficiently get a list of all possible collisions

Ogre can provide you with a list of all objects that are possible colliding. Objects that are close but not in a collision might be in this list as well, so you will have to make sure which one is a collision yourself afterwards. You can ask for this list of collisions using an *IntersectionSceneQuery*, which you can get using the following code:

Prepare the collision Request:

```
  IntersectionSceneQuery* intersectionQuery = sceneMgr->createIntersectionQuery();
```

Now you can ask it for a list of all possible collisions:

```
  IntersectionSceneQueryResult& queryResult = intersectionQuery->execute();
```

If you intend to get this list several times before updating your scene, there is no need to have Ogre calculate it again over and over. You can get back the same list again without a new calculation using the following line:

```
  IntersectionSceneQueryResult& queryResult = intersectionQuery->getLastResults();
```

After use, you can store the *IntersectionSceneQuery* for later use or remove it. If you remove it, you must tell the *SceneManager* to do so using the following code:

```
  mSceneMgr->destroyQuery(intersectionQuery);
```

The *IntersectionSceneQueryResult* is a list of pairs of *MovableObjects*. An example of its use is getting the name of the first of the two colliding objects:

```
  queryResult.movables2movables.begin()->first->getName();
```

See the Ogre-API for more details on the types of the results.

2.24 How to find out to which of your own objects a *MovableObject* belongs

The list of colliding objects *IntersectionSceneQuery* delivers is of *MovableObjects*, but in general you will want to relate this to some custom object in your own scene. To do so, you can attach a pointer to your own object to a *MovableObject* and get this pointer back later on. If you have an Entity (which is a *MovableObject*) and a custom object (like *myRocket* here below), you can attach your custom object to the Entity as follows:

```
Entity* myEntity = sceneMgr->createEntity("myEntity", "Rocket.mesh");
Rocket* myRocket = new Rocket();
myEntity->setUserObject(myRocket);
```

Now you can get a pointer to myRocket back from the Entity using the following code:

```
myEntity->getUserobject();
```

To make this work, your own custom class must be derived from *UserDefinedObject*. This also allows you to find out what kind of class the returned object is, so that you can cast it back. The definition for Rocket could now look something like this:

```
class Rocket: public UserDefinedObject
{
public:
  const string& getTypeName(void)
  {
    const string& typeName = "Rocket";
    return typeName;
  }
};
```

Now after having detected which *MovableObjects* are involved in a collision, you can also find out which of your own objects are involved in this collision.

You can also use:

```
String name = movable->getName();
```

2.25 How to exclude objects from collision detection

You can exclude certain objects from collision detection by using flags. You can add a flag to any *MovableObject*, for instance 100 in the following example:

```
Entity* ground = mSceneMgr->createEntity("ground", "Ground.mesh");
ground->setQueryFlags(100);
```

Now you can tell your *IntersectionSceneQuery* to exclude objects with this flag from its list of intersections by setting its mask. This will look like this:

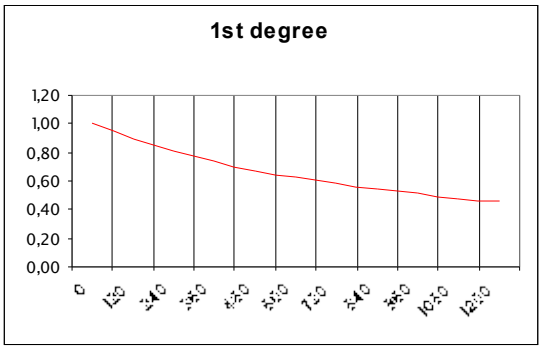
```
IntersectionSceneQuery* intersectionQuery = mSceneMgr->createIntersectionQuery();
intersectionQuery->setQueryMask(~100);
```

2.26 How to set the range / attenuation of a light

The attenuation of a light follows this rule: $A = p_0 + p_1 x R + p_2 x R^2$. So the luminosity of the light is: $L = 1/A$.

Example1:

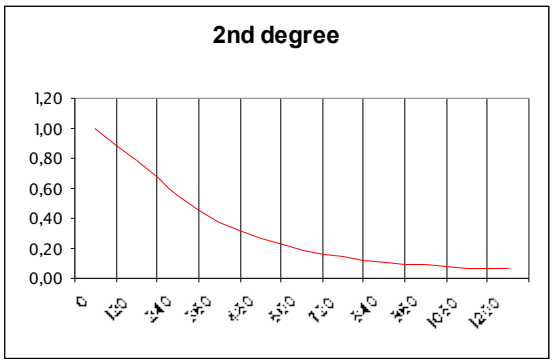
If you want a light, which lost half of its luminosity at 1000 units, the parameters are: $L=\frac{1}{1+\frac{R}{1000}}=\frac{1}{1+R\times 0.001}$




```
mLight->setAttenuation(1000, 1, 0.001, 0) ;
```

Example 2:

An attenuation of 2nd degree will fade more rapidly. For instance with these parameters $L=\frac{1}{1+\frac{R}{1000}+\frac{R^2}{1000^2}}=\frac{1}{1+R\times 0.001+R^2\times 0.00001}$, you will have a third of his luminosity at R.




```
mLight->setAttenuation(1000, 1, 0.001, 0.00001) ;
```

2.27 How to choose your SceneManager

Change or override the *chooseSceneManager()* function, and replace the ST_GENERIC by the type of scene you want:

```
Virtual void chooseSceneManager(void)
{
    mSceneMgr = mRoot->getSceneManager(ST_GENERIC);
}
```

You have the choice between the following types of scene:

ST_GENERIC

This scene is based on a 3D world with objects, but no ground. World is rendered based on octree (far objects are not rendered). This scene fits for outerspace, etc.

ST_INTERIOR

The ground is based on a Quake III BSP-file.

ST_EXTERIOR_CLOSE

This scene is based on a world with 3D objects and a basic terrain for ground. The scene has a simple LOD. The ground is based on 1 description file, and 3 images:

- 1 image for terrain heightmap (dimension: 2^n+1 x 2^n+1)

- 1 image for terrain texture (dimension: 2^n x 2^n)
- 1 image for detailed terrain texture (when close to the camera)

This scene fits for small map (257x257 or 513x513). You load the description file with the **setWorldGeometry("terrainfile.cfg")** function. See TerrainSceneManager. See demo_terrain.exe

ST_EXTERIOR_FAR

This scene is based on a world with 3D objects and ground. The ground is based on images:

- NxN heightmaps
- Several tiled textures

The heightmap is repeated N times, with a different texture on each.
This scene fits wide terrain.
See NatureSceneManager. See demo_nature.exe

ST_EXTERIOR_REAL FAR (plug-in)

This scene is based on a world with 3D objects and paged terrain for ground. The scene uses a good LOD. The ground is divided into pages. Each page is based on images:

- heighmap
- Several textures

In Version 2 of this scene, the textures can be applied by height.
This scene fits for very wide terrain.
Note: you have to install the paging landscape plug-in and compile it, and adjust the plugin.cfg file.
See also demo_pagingLandscape.exe

ST_DOT_SCENE (plug-in)

A scene based on a mesh (created with 3DS, for example) and converted into a level.

- Mesh file
- Textures

This scene fits for urban world with a lot of buildings
Note: you have to install the st_dot_scene plug-in (and compile it), and adjust the plugin.cfg file.

DISPLACEMENT MAPPING (plug-in)

A scene based on a mesh (created with 3DS, for example) and converted into a level.

2.28 How to have debug information

For real time debug information, you can have one line displayed on the main window, with the command:

```
mWindow->SetDebugText("debug information")

int value;
mWindow->SetDebugText("Value = " + StringConverter::toString(value));
```

For differed time debug information, you can log the event in the default log file (ogre.log):

```
LogManager::getSingleton().logMessage(LML_NORMAL, "text 1",  "text 2");
```

If you want to use a specific logfile (instead of the default ogre.log file):

```
Log* mylogfile, ogrelogfile;

ogrelogfile = LogManager::getSingleton().getDefaultLog();
mylogfile   = LogManager::getSingleton().createLog("mylogfile.log");

LogManager::getSingleton().setDefaultLog(mylogfile);
LogManager::getSingleton().logMessage(LML_NORMAL, "write in mylogfile.log");

LogManager::getSingleton().setDefaultLog(ogrelogfile);
LogManager::getSingleton().logMessage(LML_NORMAL, "write in ogre.log");
```

2.29 How to maintain the camera above the ground

You can use different ways, according to the nature of the “floor”. The floor can be a terrain, a mesh object, a planeEntity, etc…

First, send a Ray from your camera towards the ground:

```
Vector3 pos = mCamera->getPosition();

float  floorD = 0;

RaySceneQuery* mRayQuery;
SceneManager*  mSceneMgr;

mSceneMgr = mCamera->getSceneManager();
mRayQuery = mSceneMgr->createRayQuery(Ray(pos, Vector3::NEGATIVE_UNIT_Y));

RaySceneQueryResult& queryResult = mRayQuery->execute();
```

You get the list of the objects crossed by the ray. Take the index of the first and last item of the list.

```
RaySceneQueryResult::iterator i;
  RaySceneQueryResult::iterator i_final;

  i_final = queryResult.end();
i = queryResult.begin();
```

If the floor is a terrain, you sould search a world fragment in the list.

```
if ( i!=queryResult.end() && i->worldFragment )
    {
      SceneQuery::WorldFragment* wf = i->worldFragment;
   mWindow->setDebugText("Detected: " + StringConverter::toString(i->worldFragment->
                singleIntersection.y));
    }
```

If the floor is an object, you should search for the first object found, or for a specified object. Note that this way is a bit inaccurate, if objects are complex or rescaled; and it does not work if objects are hollow or convex. In this example, we are looking for a plane entity named « floor ».

```
for ( i=queryResult.begin(); i!=i_final; ++i)
   {
  MovableObject* wf = i->movable;
Real distance                          = i->distance;
String detected ;
  detected  = "movable found " + wf->getName();
  detected += " at "+ StringConverter::toString(distance);

  if (wf->getName() == "floor")
     {
     mWindow->setDebugText(detected);
     floorD=distance;
     }
  }
```

Finally, destroy the query.

```
mCamera->getSceneManager()->destroyQuery(mRayQuery);
```

And adjust the Y position of your camera (this is a very poor example).

```
// keep camera not too close of the ground
if (floorD < MIN_Y_ALLOWED)
   {
   pos.y = floorY+ MIN_Y_ALLOWED;
   mCamera->setPosition(pos);
   }

// make the camera falling if too far of the ground
if (floorD > MIN_Y_ALLOWED)
   {
   pos.y = pos.y - 4;
   mCamera->setPosition(pos);
   }
```

2.30 How to get a viewer to face a target

(Code not checked yet)

```
Vector3 lookAt = target->getPosition() - viewer->getPosition();
Vector3 axes[3];

viewer->getOrientation().ToAxes(axes);
Quaternion rotQuat;

if (lookAt == axes[2])
{
   rotQuat.FromAngleAxis(Math::PI, axes[1]);
}
else
{
   rotQuat = axes[2].getRotationTo(lookAt);
}
viewer->setOrientation(rotQuat* viewer->getOrientation());
```

2.31 How to make a screenshot

There is two ways of naming your screenshot files, in the *frameStarted()* function.

If you want to have files named *screenshot_1.png, screenshot_2.png*, you can use:

```
unsigned int indice = 1;
char filename[30] ;
sprintf (filename, "screenshot_%d.png", ++indice);
mWindow->writeContentsToFile(filename);
```

Or, if you want to have files named *screenshot_MMDDYYY_HHMMSSmmm.jpg* in the current application directory, you can just use:

```
mWindow->writeContentsToTimestampedFile("screenshot",".jpg");
```

Note: MM is from 0 to 11. YYY is (year-1900) (ie: 2004=104)
Note: The extension you specify (".png", or ".bmp", or ".jpg", etc) will determine the type of the generated file.

2.32 How to get some fog

```
mSceneMgr->setFog (FOG_LINEAR, ColourValue(0.25,1.1,0.2), 0.5, 200, 500);
```

The parameters are:

- Type of fog: FOG_LINEAR, FOG_EXP, FOG_EXP2, FOG_NONE
- Color
- Exp_density 0..1 (small number = large zone without fog)
- Initial density (for linear fog)
- Final density (for linear fog)

Note that transparent objects in the fog will become opaque. You should exclude them from the fog, with the instruction **fog_override true** in the **pass{}** section of the material file.

2.33 How to render the world in wire-frame

To render the scene in solid or in wire-frame, use the following commands:

```
mCamera->setDetailLevel (SDL_WIREFRAME);
mCamera->setDetailLevel (SDL_SOLID);
```

You can do this only with the whole scene, not with a single object.

2.34 How to get and set parameters in a configuration file

You can read configuration parameters in a configuration file. The configuration file looks like this (for separators, you can use ";", or "=", or "TAB"):

```
language=french
```

And the code to use it, (eg: read the language parameter):

```
ConfigFile setupfile;
setupfile.load("myconfig.cfg");

String Lang = setupfile.getSetting("language");
```

2.35 How to have a separate configuration application

You can make a separate configuration exe with the configuration dialog box.

```
mRoot->showConfigDialog();
```

This single function will show the configuration dialog box, and store the configuration in an ogre.cfg file.

In you main program, you just have to call the functions:

```
mRoot->restoreConfig();
mWindow = mRoot->initialise(true);
```

And it will automatically load and apply the configuration (if the file exists).

For setting other configuration parameters (audio, controls, etc), I did not manage to call the *showConfigDialog()* function in a standard MFC application, so I had to write a second configuration exe. (if somebody has an idea to how to do it…)

2.36 How to manage world-units

Knowing the value of your world units is useful when you create your 3D objects with a modeler. You can then set the same scale to all your objects, and you will not have to rescale individually each object in your code with the *setScale()* function.

To determine the world-units of your scene, you don't have a Ogre-function: you will have to walk in your scene, and visually estimate the distance from one point to another, and check the coordinates of these to points.
So, in the Ogre demo programs, you fill find 15m=1.500wu. That is: 1wu=1cm. And, in the ExampleFrameListener, you will find the speed set at 100 wu per second (= 1 m/s = the speed of a walking man).

If you want to have a different setting, ie: 1wu = 1m, you just have to adjust the speed of your camera in the FrameListener (lets say to 1 wu per second), and eventually adjust the size of your 3D objects. You can also adjust the focal angle of your camera of a few degree (between 45 and 60) with the function *setFOVy(angle)*.

Note:
You better choose this very early in your project, otherwise, you will have to rescale all the objects, camera, particle, etc (dimension and speed) that you already defined in your code and scripts.

2.37 How to add a Plug-in or a Library in your project

In the Visual C++ environment, adjust:

```
Project Setting
C++
```

Category “preprocessor“
Line “additional include directory”
Add the directory where the header files (.h) are located (the separator is ‘,’).

Then adjust:
Project Setting
Link
Category “input“
Line “additional library path”
Add the directory where the library file (.lib) is located (the separator is ‘,’).

Then adjust:
Project Setting
Link
Category “input“
Line “objects/library module”
Add the name of library file (.lib) (the separator is ‘ ’).

If this is an Ogre plug-in, you should also adjust the plugin.cfg file.

2.38 How to manage N objects moving

The usual way is to write two classes for each objects: for instance a class **Rocket** derived from **UserDefinedObject**, and a class **RocketListener** derived from **FrameListener**.

```
class Rocket  : public UserDefinedObject{
public:
     Rocket ();
     virtual ~ Rocket ();
…
}

class RocketListener  : public FrameListener
{
public:
     RocketListener ();
     virtual ~ RocketListener ();
     bool                        frameStarted(const FrameEvent& evt);
…
}
```

In the Rocket code, you will call once (in the constructor function, or somewhere else):
```
mRoot->addFrameListener(…);
```
and in the destructor function (or elsewhere):
```
mRoot->removeFrameListener(…);
```
Then the function *RocketListener::frameStarted* will be called by Ogre at the beginning of every frame.

With this method, you will have N "*Rocket*" objects, and 1 "*RocketListener*" object. This *RocketListener* will make move the rockets, that have registered to him. (You write the registration function).

Other method:

An other way is to have only one class derived from **UserDefinedObject**, and from **FrameListener**.

```
class Robot  : public UserDefinedObject, public FrameListener
{
public:
     Robot();
     virtual ~Robot();
     bool                        frameStarted(const FrameEvent& evt);
…
}
```

In the Robot code, I call once:
```
mRoot->addFrameListener(this);
```
and when the object don’t need to move anymore:

```
mRoot->removeFrameListener(this);
```

The function frameStarted() is called by Ogre at the beginning of every frame. The frameStarted() function has also a direct access to all member variables.
Note that this method can affect performance if you have many objects of this class (because Ogre will make N calls to *frameStarted()* at every frame, instead of 1). In that case, it is better to use the first method.

2.39 How to have transparency

In the .material file, you can set the **scene_blend** parameter to **add, modulate** or **alpha_blend**.

scene_blend add

You can use a JPEG or PNG image file for the texture.
The dark parts of the picture will become transparent (black = fully transparent).
Use it for explosion, flare, lights, etc

scene_blend modulate

You can use a JPEG or PNG image file for the texture.
The texture will multiply with the background. Usually, it colours and darken the picture. The dark part of the texture will become transparent.
Use it for smoked glass, semi transparent objects, etc

scene_blend alpha_blend

You should use a PNG image file for the texture.
To have transparency in your PNG file, you should prepare the file with photoshop, by sending the background layer to the trashcan, and use the eraser on the other layers.
The erased parts will become transparent.

2.40 How to get some shadows in your scene

There two kinds of shadows: the shadowed side of an object, and the projected (=casted) shadow of an object.

The shadowed side

To see the shadowed sides of your objects, you have to set the appropriate parameters in the material description.

- In the createScene function, don't use an ambient light (or very low light).
- In the createScene function, create a light (cf the above chapter)
- In the object.material file, set the following parameters in the pass{} section.
  - if dynamic **lighting** is OFF, object is fully lit, so use ON
  - note: for colors, 000 is black and 111 is white.
  - **ambient** is the global ambient light. It allows to have a strong or a soft contrast between the lit side and the shadowed side.
  - **diffuse** is the reflected colour for the lit side. You can imagine it like a coloured filter hold in front of the Light Object.
  - **Specular** is the reflected colour for the most lit part of the lit side. It can give a metallic aspect to the object. To see this, you have to lower the ambient light, and create a *setSpecularColor* for your Light.

```
                                                    lighting on
                                    ambient  1 1 1
                                    diffuse  1 1 1
                                // specular 1 1 1 12
```

Note that transparent textures, and billboards cannot receive a shadow.

The casted shadows:

An entity can cast (=send) a shadow. By default, lights don't cast shadows, you have to enable it.

There are four techniques of shadows rendering: STENCIL or TEXTURE, and ADDITIVE or MODULATIVE. (The more "cpu friendly" is SHADOWTYPE_TEXTURE_MODULATIVE).

STENCIL SHADOWS:

- This technique uses the CPU for rendering.
- You should set a max distance (with setShadowFarDistance) in order to keep a good framerate.
- The mesh (which cast shadows) should be completely manifold (ie: closed / no holes).
- Avoid long shadows (ie: dawn)
- Avoid objects too close of the light.
- This technique allows self-shadowing.
- The shadows have hard edges (hard transition between shadow and light).

TEXTURE SHADOWS:

- This technique uses the graphic card for rendering.
- This technique works only with directional or spotlight.
- This technique does not allow self-shadowing.
- The meshes should be defined as "shadow caster" or "shadows receiver".
- Avoid long shadows (ie dawn)
- Avoid objects too close of the light.
- The shadows have smooth edges.
- Color and definition of the shadow can be defined.

For more details, see demo_shadows.exe, and also see the chapter 7 "Shadows" of the manual: http://ogre.sourceforge.net/docs/manual.

Example:

```
mSceneMgr->setShadowsTechnique(SHADOWTYPE_TEXTURE_MODULATIVE);
mSceneMgr->setShadowsColor(0.5, 0.5, 0.5);
…
mLight->setCastShadows(true);
```

```
..
mCastingObject->setCastShadows(true);
mReceivingObject->setCastShadows(false);
..
myMaterial->setReceiveShadows(true);
..
```

Note that the instruction `myMaterial->setReceiveShadows(true)` can be replaced by `receive_shadows on` in the material file, at top level, or be omitted, as the default setting for material is **ON**. You can use this instruction to set the value to **OFF** for materials that should not receive shadows. This can save some framerate.

2.41 How to play an animated mesh

First, you need to have:

- a **mesh file** (which contains the description of the mesh, and the association between the vertices and the bones)
- a **skeleton file** (which contain the description of the bones, and the movements of the bones during the animation).

These two files are generated by the ogre-Exporter utility.
You can find how to make them in my other tutorial "Aide-mémoire 3DS Ogre" (in French: sorry).

To load the animation:

```
Animation::setDefaultInterpolationMode(Animation::IM_SPLINE);
mAnimState = mEntity->getAnimationState("move");
mAnimState->setEnabled(true);
mAnimationSpeed = 1;
```

To play the animation (in the *frameStarted()* function) :

```
mAnimState->addTime(evt.timeSinceLastFrame * mAnimationSpeed);
```

To stop the animation:

```
if (mAnimState->getTimePosition() >= mAnimState->getLength())
{
        // animation is over
}
```

Or , when you load the animation, you can set:

```
mAnimState->setLoop(false);
```

To play the animation backwards:

```
mAnimState->addTime(evt.timeSinceLastFrame * (-mAnimationSpeed));
```

3 More Q&A to come

3.1 How to use the CollideCamera

3.2 How to use LOD

3.3 How to create a texture during runtime