FROM THE BOOK:
„Ogre 3D 1.7 - Beginner's Guide"
by Felix Kerger, Packt Publishing, 2010

9
The Ogre 3D Startup Sequence

We have covered a lot of ground in the progress of this book. This chapter is going to cover one of the few topics left: how to create our own application without relying on the `ExampleApplication`. *After we have covered this topic, this chapter is going to repeat some of the topics from the previous chapters to make a demo of the things we have learned using our new open application class.*

In this chapter, we will:

- Learn how to start Ogre 3D ourselves
- Parse `resources.cfg` to load the models we need
- Combine things from the previous chapters to make a small demo application showing off the things we have learned

So let's get on with it...

Starting Ogre 3D

Up until now, the `ExampleApplication` class has started and initialized Ogre 3D for us; now we are going to do it ourselves.

Time for action – starting Ogre 3D

This time we are working on a blank sheet.

1. Start with an empty code file, include `Ogre3d.h`, and create an empty main function:

```
#include "Ogre\Ogre.h"

int main (void)
{

  return 0;
}
```

2. Create an instance of the Ogre 3D `Root` class; this class needs the name of the `"plugin.cfg"`:

```
Ogre::Root* root = new Ogre::Root("plugins_d.cfg");
```

3. If the config dialog can't be shown or the user cancels it, close the application:

```
if(!root->showConfigDialog())
{
  return -1;
}
```

4. Create a render window:

```
Ogre::RenderWindow* window = root->initialise(true,"Ogre3D
Beginners Guide");
```

5. Next create a new scene manager:

```
Ogre::SceneManager* sceneManager = root-
>createSceneManager(Ogre::ST_GENERIC);
```

6. Create a camera and name it `camera`:

```
Ogre::Camera* camera = sceneManager->createCamera("Camera");
camera->setPosition(Ogre::Vector3(0,0,50));
camera->lookAt(Ogre::Vector3(0,0,0));
camera->setNearClipDistance(5);
```

7. With this camera, create a `viewport` and set the background color to black:

```
Ogre::Viewport* viewport = window->addViewport(camera);
viewport->setBackgroundColour(Ogre::ColourValue(0.0,0.0,0.0));
```

8. Now, use this viewport to set the aspect ratio of the camera:

```
camera->setAspectRatio(Ogre::Real(viewport->getActualWidth())/
Ogre::Real(viewport->getActualHeight()));
```

9. Finally, tell the root to start rendering:

```
root->startRendering();
```

10. Compile and run the application; you should see the normal config dialog and then a black window. This window can't be closed by pressing *Escape* because we haven't added key handling yet. You can close the application by pressing *CTRL+C* in the console the application has been started from.

What just happened?

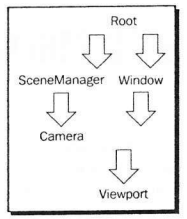
We created our first Ogre 3D application without the help of the `ExampleApplication`. Because we aren't using the `ExampleApplication` any longer, we had to include `Ogre3D.h`, which was previously included by `ExampleApplication.h`. Before we can do anything with Ogre 3D, we need a `root` instance. The `root` class is a class that manages the higher levels of Ogre 3D, creates and saves the factories used for creating other objects, loads and unloads the needed plugins, and a lot more. We gave the `root` instance one parameter: the name of the file that defines which plugins to load. The following is the complete signature of the constructor:

```
Root(const String & pluginFileName = "plugins.cfg",const String &
configFileName = "ogre.cfg",const String & logFileName = "Ogre.log")
```

Besides the name for the plugin configuration file, the function also needs the name of the Ogre configuration and the log file. We needed to change the first file name because we are using the debug version of our application and therefore want to load the debug plugins. The default value is `plugins.cfg`, which is true for the release folder of the Ogre 3D SDK, but our application is running in the debug folder where the filename is `plugins_d.cfg`.

`ogre.cfg` contains the settings for starting the Ogre application that we selected in the config dialog. This saves the user from making the same changes every time he/she starts our application. With this file Ogre 3D can remember his choices and use them as defaults for the next start. This file is created if it didn't exist, so we don't append an `_d` to the filename and can use the default; the same is true for the log file.

Using the `root` instance, we let Ogre 3D show the config dialog to the user in step 3. When the user cancels the dialog or anything goes wrong, we return `-1` and with this the application closes. Otherwise, we created a new render window and a new scene manager in step 4. Using the scene manager, we created a camera, and with the camera we created the viewport; then, using the viewport, we calculated the aspect ratio for the camera. The creation of camera and viewport shouldn't be anything new; we have already done that in Chapter 3, *Camera, Light, and Shadow*. After creating all requirements, we told the `root` instance to start rendering, so our result would be visible. Following is a diagram showing which object was needed to create the other:



Adding resources

We have now created our first Ogre 3D application, which doesn't need the `ExampleApplication`. But one important thing is missing: we haven't loaded and rendered a model yet.

Time for action – loading the Sinbad mesh

We have our application, now let's add a model.

1. After setting the aspect ratio and before starting the rendering, add the `zip` archive containing the Sinbad model to our resources:

```
Ogre::ResourceGroupManager::getSingleton().
addResourceLocation("../../Media/packs/Sinbad.zip","Zip");
```

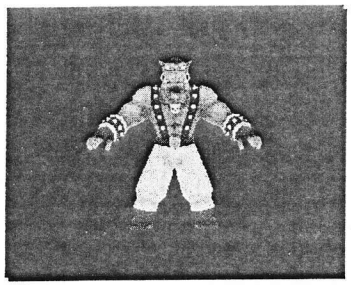
2. We don't want to index more resources at the moment, so index all added resources now:

```
Ogre::ResourceGroupManager::getSingleton().
initialiseAllResourceGroups();
```

3. Now create an instance of the Sinbad mesh and add it to the scene:

```
Ogre::Entity* ent = sceneManager->createEntity("Sinbad.mesh");
sceneManager->getRootSceneNode()->attachObject(ent);
```

4. Compile and run the application; you should see Sinbad in the middle of the screen:



What just happened?

We used the `ResourceGroupManager` to index the `zip` archive containing the Sinbad mesh and texture files, and after this was done, we told it to load the data with the `createEntity()` call in step 3.

Using resources.cfg

Adding a new line of code for each `zip` archive or folder we want to load is a tedious task and we should try to avoid it. The `ExampleApplication` used a configuration file called `resources.cfg` in which each folder or `zip` archive was listed, and all the content was loaded using this file. Let's replicate this behavior.

(3) Time for action – using resources.cfg to load our models

Using our previous application, we are now going to parse the `resources.cfg`.

1. Replace the loading of the `zip` archive with an instance of a config file pointing at the `resources_d.cfg`:

```
Ogre::ConfigFile cf;
cf.load("resources_d.cfg");
```


2. First get the iterator, which goes over each section of the config file:

```
Ogre::ConfigFile::SectionIterator sectionIter =
cf.getSectionIterator();
```

3. Define three strings to save the data we are going to extract from the config file and iterate over each section:

```
Ogre::String sectionName, typeName, dataname;
while (sectionIter.hasMoreElements())
{
```

4. Get the name of the section:

```
sectionName = sectionIter.peekNextKey();
```

5. Get the settings contained in the section and, at the same time, advance the section iterator; also create an iterator for the settings itself:

```
Ogre::ConfigFile::SettingsMultiMap *settings = sectionIter.
getNext();
Ogre::ConfigFile::SettingsMultiMap::iterator i;
```

6. Iterate over each setting in the section:

```
for (i = settings->begin(); i != settings->end(); ++i)
{
```

7. Use the iterator to get the name and the type of the resources:

```
typeName = i->first;
dataname = i->second;
```

8. Use the resource name, type, and section name to add it to the resource index:

```
Ogre::ResourceGroupManager::getSingleton().
addResourceLocation(dataname, typeName, sectionName);
```

9. Compile and run the application, and you should see the same scene as before.

What just happened?

In the first step, we used another helper class of Ogre 3D, called `ConfigFile`. This class is used to easily load and parse simple configuration files, which consist of name-value pairs. By using an instance of the `ConfigFile` class, we loaded the `resources_d.cfg`. We hardcoded the filename with the debug postfix; this isn't good practice and in a production application we would use `#ifdef` to change the filename depending on the debug or release mode. `ExampleApplication` does this; let's take a look at `ExampleApplication.h` line 384:

```
#if OGRE_DEBUG_MODE
        cf.load(mResourcePath + "resources_d.cfg");
#else
    cf.load(mResourcePath + "resources.cfg");
#endif
```

Structure of a configuration file

The configuration file loaded by the helper class follows a simple structure; here is an example from `resource.cfg`. Of course your `resource.cfg` will consist of different paths:

```
[General]
FileSystem=D:/programming/ogre/ogre_trunk_1_7/Samples/Media
```

`[General]` starts a section, which goes on until another `[sectionname]` occurs in the file. Each configuration file can contain a lot of sections; in step 2 we created an iterator to iterate over all the sections in the file and in step 3 we used a while loop, which runs until we have processed each section.

A section consists of several settings and each setting assigns a key a value. We assign the key `FileSystem` the value `D:/programming/ogre/ogre_trunk_1_7/Samples/Media`. In step 4, we created an iterator so we can iterate over each setting. The settings are internally called name-value pairs. We iterate over this map and for each entry we use the map key as the type of the resource and the data we use as the path. Using the section name as resource group, we added the resource using the resource group manager in step 8. Once we had parsed the complete file, we indexed all the files.

Creating an application class

We now have the basis for our own Ogre 3D application, but all the code is in the main function, which isn't really desirable for reusing the code.

Time for action – creating a class

Using the previously applied code we are now going to create a class to separate the Ogre code from the main function.

1. Create the class `MyApplication`, which has two private pointers, one to a Ogre 3D `SceneManager` and the other to the `Root` class:

```
class MyApplication
{
private:
  Ogre::SceneManager* _sceneManager;
Ogre::Root* _root;
```

2. The rest of this class should be public:

```
public:
```

3. Create a `loadResources()` function, which loads the `resources.cfg` configuration file:

```
void loadResources()
{
  Ogre::ConfigFile cf;
  cf.load("resources_d.cfg");
```

4. Iterate over the sections of the configuration file:

```
Ogre::ConfigFile::SectionIterator sectionIter =
cf.getSectionIterator();
Ogre::String sectionName, typeName, dataname;
while (sectionIter.hasMoreElements())
{
```

5. Get the section name and the iterator for the settings:

```
sectionName = sectionIter.peekNextKey();
Ogre::ConfigFile::SettingsMultiMap *settings = sectionIter.
getNext();
Ogre::ConfigFile::SettingsMultiMap::iterator i;
```

6. Iterate over the settings and add each resource:

```
    for (i = settings->begin(); i != settings->end(); ++i)
    {
      typeName = i->first;
      dataname = i->second;
      Ogre::ResourceGroupManager::getSingleton().
addResourceLocation(
      dataname, typeName, sectionName);
```

```
    }
  }

  Ogre::ResourceGroupManager::getSingleton().
initialiseAllResourceGroups();
}
```

7. Also create a `startup()` function, which creates an Ogre 3D `root` class instance using the `plugins.cfg`:

```
int startup()
{
  _root = new Ogre::Root("plugins_d.cfg");
```

8. Show the config dialog and when the user quits it, return `-1` to close the application:

```
if(!_root->showConfigDialog())
{
  return -1;
}
```

9. Create the `RenderWindow` and the `SceneManager`:

```
Ogre::RenderWindow* window = _root->initialise(true,"Ogre3D
Beginners Guide");
_sceneManager = root->createSceneManager(Ogre::ST_GENERIC);
```

10. Create a `camera` and a `viewport`:

```
Ogre::Camera* camera = _sceneManager->createCamera("Camera");
camera->setPosition(Ogre::Vector3(0,0,50));
camera->lookAt(Ogre::Vector3(0,0,0));
camera->setNearClipDistance(5);

Ogre::Viewport* viewport = window->addViewport(camera);
viewport->setBackgroundColour(Ogre::ColourValue(0.0,0.0,0.0));
camera->setAspectRatio(Ogre::Real(viewport->getActualWidth())/
Ogre::Real(viewport->getActualHeight()));
```

11. Call the function to load our resources and then a function to create a scene; after that, Ogre 3D starts rendering:

```
loadResources();
createScene();
_root->startRendering();
return 0;
```


12. Then create the `createScene()` function, which contains the code for creating the `SceneNode` and the `Entity`:

```
void createScene()
{
  Ogre::Entity* ent = _sceneManager->createEntity("Sinbad.mesh");
  _sceneManager->getRootSceneNode()->attachObject(ent);
}
```

13. We need the constructor to set both the pointers to NULL so we can delete it even if it hasn't been assigned a value:

```
MyApplication()
{
  _sceneManager = NULL;
_root = NULL;
}
```

14. We need to delete the `root` instance when our application instance is destroyed, so implement a destructor which does this:

```
~MyApplication()
{
  delete _root;
}
```

15. The only thing left to do is to adjust the main function:

```
int main (void)
{
  MyApplication app;
  app.startup();
  return 0;
}
```

16. Compile and run the application; the scene should be unchanged.

What just happened?

We refactored our starting codebase so that different functionalities are better organized. We also added a destructor so our created instances would be deleted when our application is closed. One problem is that our destructor won't be called; because `startup()` never returns, there is no way to close our application. We need to add a `FrameListener` to tell Ogre 3D to stop rendering.

Adding a FrameListener

We have already used the `ExampleFrameListener`; this time we are going to use our own implementation of the interface.

Time for action – adding a FrameListener

Using the code from before we are going to add our own `FrameListener` implementation

1. Create a new class called `MyFrameListener` exposing three publicly visible event handler functions:

```
class MyFrameListener : public Ogre::FrameListener
{
public:
```

2. First, implement the `frameStarted` function, which for now returns `false` to close the application:

```
bool frameStarted(const Ogre::FrameEvent& evt)
{
  return false;
}
```

3. We also need a `frameEnded` function, which also returns `false`:

```
bool frameEnded(const Ogre::FrameEvent& evt)
{
  return false;
}
```

4. The last function we implement is the `frameRenderingQueued` function, which also returns `false`:

```
bool frameRenderingQueued(const Ogre::FrameEvent& evt)
{
  return false;
}
```

5. The main class needs a point to store the `FrameListener`:

```
MyFrameListener* _listener;
```

6. Remember that the constructor needs to set the initial value of the listener to `NULL`:

```
_listener = NULL;
```


7. Let the destructor delete the instance:

```
delete _listener;
```

8. At last, create a new instance of the `FrameListener` and add it to the root object; this should happen in the `startup()` function:

```
_listener = new MyFrameListener();
_root->addFrameListener(_listener);
```

9. Compile and run the application; it should be closed directly.

What just happened?

We created our own `FrameListener` class, which didn't rely on the `ExampleFrameListener` implementation. This time we inherited directly from the `FrameListener` interface. This interface consists of three virtual functions, which we implemented. We already knew the `frameStarted` function, but the other two are new. All three functions return `false`, which is an indicator to Ogre 3D to stop rendering and close the application. Using our implementation, we added a `FrameListener` to the `root` instance and started the application; not surprisingly, it closed directly.

Investigating the FrameListener functionality

Our `FrameListener` implementation has three functions; each is called at a different point in time. We are going to investigate in which sequence they are called.

Time for action – experimenting with the FrameListener implementation

Using the console printing we are going to inspect when the `FrameListener` is called.

1. First let each function print a message to the console when it is called:

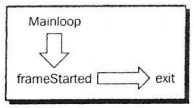
```
bool frameStarted(const Ogre::FrameEvent& evt)
{
  std::cout << «Frame started» << std::endl;
  return false;
}
bool frameEnded(const Ogre::FrameEvent& evt)
{
  std::cout << «Frame ended» << std::endl;
  return false;
}
bool frameRenderingQueued(const Ogre::FrameEvent& evt)
```

```
{
  std::cout << «Frame queued» << std::endl;
  return false;
}
```

2. Compile and run the application; in the console you should find the first string—**Frame started**.

What just happened?

We added a "debug" output to each of the `FrameListener` functions to see which function is getting called. Running the application, we noticed that only the first debug message is printed. The reason is that the `frameStarted` function returns `false`, which is a signal for the `root` instance to close the application.



Now that we know what happens when `frameStarted()` returns `false`, let's see what happens when `frameStarted()` returns `true`.

Time for action – returning true in the frameStarted function

Now we are going to modify the behavior of our `FrameListener` to see how this changed its behavior.

1. Change `frameStarted` to return `true`:

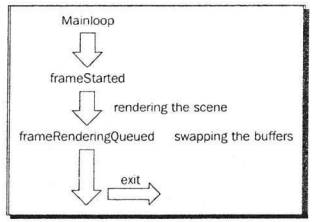
```
bool frameStarted(const Ogre::FrameEvent& evt)
{
  std::cout << «Frame started» << std::endl;
  return true;
}
```

2. Compile and run the application. Before the application closes directly, you will see a short glimpse of the rendered scene and there should be the two following lines in the output:

```
Frame started
Frame queued
```


What just happened?

Now, the `frameStarted` function returns `true` and this lets Ogre 3D continue to render until `false` is returned by the `frameRenderingQueued` function. We see a scene this time because directly after the `frameRenderingQueued` function is called, the render buffers are swapped before the application gets the possibility to close itself.



Double buffering

When a scene is rendered, it isn't normally rendered directly to the buffer, which is displayed on the monitor. Normally, the scene is rendered to a second buffer and when the rendering is finished, the buffers are swapped. This is done to prevent some artifacts, which can be created if we render to the same buffer, which is displayed on the monitor. The `FrameListener` function, `frameRenderingQueued`, is called after the scene has been rendered to the back buffer, the buffer which isn't displayed at the moment. Before the buffers are swapped, the rendering result is already created but not yet displayed. Directly after the `frameRenderingQueued` function is called, the buffers get swapped and then the application gets the return value and closes itself. That's the reason why we see an image this time.

Now, we will see what happens when `frameRenderingQueued` also returns `true`.

Time for action – returning true in the frameRenderingQueued function

Once again we modify the code to test the behavior of the Frame Listener.

1. Change `frameRenderingQueued` to return `true`:

```
bool frameRenderingQueued (const Ogre::FrameEvent& evt)
{
  std::cout << «Frame queued» << std::endl;
  return true;
}
```

2. Compile and run the application. You should see Sinbad for a short period of time before the application closes, and the following three lines should be in the console output:

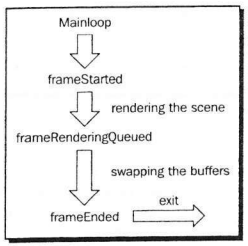
   Frame started

   Frame queued

   Frame ended

What just happened?

Now that the `frameRenderingQueued` handler returns `true`, it will let Ogre 3D continue to render until the `frameEnded` handler returns `false`.



Like in the last example, the render buffers were swapped, so we saw the scene for a short period of time. After the frame was rendered, the `frameEnded` function returned `false`, which closes the application and, in this case, doesn't change anything from our perspective.

Time for action – returning true in the frameEnded function

Now let's test the last of three possibilities.

1. Change `frameRenderingQueued` to return `true`:

```
bool frameEnded (const Ogre::FrameEvent& evt)
{
  std::cout << «Frame ended» << std::endl;
  return true;
}
```


2. Compile and run the application. You should see the scene with Sinbad and an endless repetition of the following three lines:

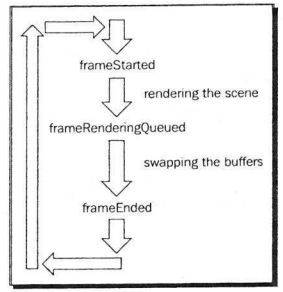
   Frame started

   Frame queued

   Frame ended

What just happened?

Now, all event handlers returned `true` and, therefore, the application will never be closed; it would run forever as long as we aren't going to close the application ourselves.



Adding input

We have an application running forever and have to force it to close; that's not neat. Let's add input and the possibility to close the application by pressing *Escape*.

Time for action – adding input

Now that we know how the `FrameListener` works, let's add some input.

1. We need to include the `OIS` header file to use `OIS`:

```
#include "OIS\OIS.h"
```

2. Remove all functions from the `FrameListener` and add two private members to store the `InputManager` and the `Keyboard`:

```
OIS::InputManager* _InputManager;
```

```
OIS::Keyboard* _Keyboard;
```

3. The `FrameListener` needs a pointer to the `RenderWindow` to initialize `OIS`, so we need a constructor, which takes the window as a parameter:

```
MyFrameListener(Ogre::RenderWindow* win)
{
```

4. `OIS` will be initialized using a list of parameters, we also need a window handle in string form for the parameter list; create the three needed variables to store the data:

```
OIS::ParamList parameters;
unsigned int windowHandle = 0;
std::ostringstream windowHandleString;
```

5. Get the handle of the `RenderWindow` and convert it into a string:

```
win->getCustomAttribute("WINDOW", &windowHandle);
windowHandleString << windowHandle;
```

6. Add the string containing the window handle to the parameter list using the key `"WINDOW"`:

```
parameters.insert(std::make_pair("WINDOW", windowHandleString.
str()));
```

7. Use the parameter list to create the `InputManager`:

```
_InputManager = OIS::InputManager::createInputSystem(parameters);
```

8. With the manager create the keyboard:

```
_Keyboard = static_cast<OIS::Keyboard*>(_InputManager-
>createInputObject( OIS::OISKeyboard, false ));
```

9. What we created in the constructor, we need to destroy in the destructor:

```
~MyFrameListener()
{
  _InputManager->destroyInputObject(_Keyboard);
  OIS::InputManager::destroyInputSystem(_InputManager);
}
```

10. Create a new `frameStarted` function, which captures the current state of the keyboard, and if *Escape* is pressed, it returns `false`; otherwise, it returns `true`:

```
bool frameStarted(const Ogre::FrameEvent& evt)
{
  _Keyboard->capture();
  if(_Keyboard->isKeyDown(OIS::KC_ESCAPE))
```



```
  {
    return false;
  }
  return true;
}
```

11. The last thing to do is to change the instantiation of the `FrameListener` to use a pointer to the render window in the `startup` function:

```
_listener = new MyFrameListener(window);
_root->addFrameListener(_listener);
```

12. Compile and run the application. You should see the scene and now be able to close it by pressing the *Escape* key.

What just happened?

We added input processing capabilities to our `FrameListener` the same way we did in *Chapter 4, Getting User Input and using the Frame Listener*. The only difference is that this time, we didn't use any example classes, but our own versions.

Pop quiz – the three event handlers

Which three functions offer the `FrameListener` interface and at which point is each of these functions called?

Our own main loop

We have used the `startRendering` function to fire up our application. After this, the only way we knew when a frame was rendered was by relying on the `FrameListener`. But sometimes it is not possible or desirable to give up the control over the main loop; for such cases, Ogre 3D provides another method, which doesn't require us to give up the control over the main loop.

Time for action – using our own rendering loop

Using the code from before we are now going to use our own rendering loop.

1. Our application needs to know if it should keep running or not; add a `Boolean` as a private member of the application to remember the state:

```
bool _keepRunning;
```

2. Remove the `startRendering` function call in the `startup` function.

3. Add a new function called `renderOneFrame`, which calls the `renderOneFrame` function of the `root` instance and saves the return value in the `_keepRunning` member variable. Before this call, add a function to process all window events:

```
void renderOneFrame()
{
  Ogre::WindowEventUtilities::messagePump();
  _keepRunning = _root->renderOneFrame();
}
```

4. Add a getter for the `_keepRunning` member variable:

```
bool keepRunning()
{
  return _keepRunning;
}
```

5. Add a `while` loop to the main function, which keeps running as long as the `keepRunning` function returns `true`. In the body of the loop, call the `renderOneFrame` function of the application.

```
while(app.keepRunning())
{
  app.renderOneFrame();
}
```

6. Compile and run the application. There shouldn't be any noticeable difference to the last example.

What just happened?

We moved the control of the main loop from Ogre 3D to our application. Before this change, Ogre 3D used an internal main loop over which we hadn't any control and had to rely on the `FrameListener` to get notified if a frame was rendered.

Now we have our own main loop. To get there, we needed a `Boolean` member variable, which signals if the application wishes to keep running or not; this variable was added in step 1. Step 2 removed the `startRendering` function call so we wouldn't hand over the control to Ogre 3D. In step 3, we created a function, which first calls a helper function of Ogre 3D, which processes all window events we might have gotten from the operating system. It then sends all messages we might have created since the last frame, and therefore makes the application "well-behaved" in the context of the host windowing system.

After this we call the Ogre 3D function `renderOneFrame`, which does exactly what the name suggests: it renders the frame and also calls the `frameStarted`, `frameRenderingQueued`, and `frameEnded` event handler of each registered `FrameListener` and returns `false` if any of these functions returned `false`. Since we assign the return value of the function to the `_keepRunning` member variable, we can use this variable to check if the application should keep running. When `renderOneFrame` returns a `false`, we know some `FrameListener` wants to close the application and we set the `_keepRunning` variable to `false`. The fourth step just added a getter for the `_keepRunning` member variable.

In step 5, we used the `_keepRunning` variable as the condition for the `while` loop. This means the `while` loop will run as long as `_keepRunning` is `true`, which will be the case until one `FrameListener` returns `false`, which then will result in the `while` loop to exit and with this the whole application will be closed. Inside the `while` loop we call the `renderOneFrame` function of the application to update the render window with the newest render result. This is all we needed to create our own main loop.

Adding a camera (again)

We have already implemented a camera in *Chapter 4, Getting User Input and Using the Frame Listener*, but, nevertheless, we want a controllable camera in our own implementation of the frame listener, so here we go.

Time for action – adding a frame listener

Using our `FrameListener` we are going to add a user controlled camera.

1. To control the camera we need a mouse interface, a pointer to the camera, and a variable defining the speed at which our camera should move as a member variable of our `FrameListener`:

```
OIS::Mouse* _Mouse;
Ogre::Camera* _Cam;
float _movementspeed;
```

2. Adjust the constructor and add the camera pointer as the new parameter and set the movement speed to `50`:

```
MyFrameListener(Ogre::RenderWindow* win,Ogre::Camera* cam)
{
  _Cam = cam;
  _movementspeed = 50.0f;
```

3. Init the mouse using the `InputManager`:

```
_Mouse = static_cast<OIS::Mouse*>(_InputManager-
>createInputObject( OIS::OISMouse, false ));
```

4. And remember to destroy it in the destructor:

```
_InputManager->destroyInputObject(_Mouse);
```

5. Add the code to move the camera using the *W*, *A*, *S*, *D* keys and the movement speed to the `frameStarted` event handler:

```
Ogre::Vector3 translate(0,0,0);
if(_Keyboard->isKeyDown(OIS::KC_W))
{
  translate += Ogre::Vector3(0,0,-1);
}
if(_Keyboard->isKeyDown(OIS::KC_S))
{
  translate += Ogre::Vector3(0,0,1);
}
if(_Keyboard->isKeyDown(OIS::KC_A))
{
  translate += Ogre::Vector3(-1,0,0);
}
if(_Keyboard->isKeyDown(OIS::KC_D))
{
  translate += Ogre::Vector3(1,0,0);
}
_Cam->moveRelative(translate*evt.timeSinceLastFrame * _
movementspeed);
```

6. Now do the same for the mouse control:

```
_Mouse->capture();
float rotX = _Mouse->getMouseState().X.rel * evt.
timeSinceLastFrame* -1;
float rotY = _Mouse->getMouseState().Y.rel * evt.
timeSinceLastFrame * -1;
_Cam->yaw(Ogre::Radian(rotX));
_Cam->pitch(Ogre::Radian(rotY));
```

7. The last thing to do is to change the instantiation of the `FrameListener`:

```
_listener = new MyFrameListener(window,camera);
```


8. Compile and run the application. The scene should be unchanged but now we can control the camera:

What just happened?

We used our knowledge from the previous chapters to add a user-controlled camera. The next step will be to add compositors and other features to make our application more interesting and to leverage some of the techniques we learned along the way.

Adding compositors

Previously, we have created three compositors, which we are now going to add to our application with the capability to turn each one off and on using keyboard input.

SKIP

Time for action – adding compositors

Having almost finished our application, we are going to add compositors to make the application more interesting.

1. We are going to use compositors in our `FrameListener`, so we need a member variable containing the viewport:

```
Ogre::Viewport* _viewport;
```

2. We also are going to need to save which compositor is turned on; add three `Booleans` for this task:

```
bool _comp1, _comp2, _comp3;
```

3. We are going to use keyboard input to switch the compositors on and off. To be able to differentiate between key presses, we need to know the previous state of the key:

```
bool _down1, _down2, _down3;
```

4. Change the constructor of the `FrameListener` to take the viewport as a parameter:

```
MyFrameListener(Ogre::RenderWindow* win,Ogre::Camera*
cam,Ogre::Viewport* viewport)
```

5. Assign the viewport pointer to the member and assign the Boolean value their starting value:

```
_viewport = viewport;

_comp1 = false;
_comp2 = false;
_comp3 = false;

_down1 = false;
_down2 = false;
_down3 = false;
```

6. If the key number *1* is pressed and it wasn't pressed before, change the state of the key to pressed, flip the state of the compositor, and use the flipped value to enable or disable the compositor. This code goes into the `frameStarted` function:

```
if(_Keyboard->isKeyDown(OIS::KC_1) && ! _down1)
{
  _down1 = true;
  _comp1 = !comp1;
  Ogre::CompositorManager::getSingleton().setCompositorEnabled(_
viewport, "Compositor2", _comp1);
}
```

7. Do the same for the other two compositors we are going to have:

```
if(_Keyboard->isKeyDown(OIS::KC_2) && ! _down2)
{
  _down2 = true;
  _comp2 = !comp2;
  Ogre::CompositorManager::getSingleton().setCompositorEnabled(_
viewport, "Compositor3", _comp2);
}
if(_Keyboard->isKeyDown(OIS::KC_3) && ! _down3)
{
  _down3 = true;
  _comp3 = !comp3;
```



```
  Ogre::CompositorManager::getSingleton().setCompositorEnabled(_
viewport, "Compositor7", _comp3);
}
```

8. If a key is no longer pressed, we need to change the state of the key:

```
if(!_Keyboard->isKeyDown(OIS::KC_1))
{
  _down1 = false;
}
if(!_Keyboard->isKeyDown(OIS::KC_2))
{
  _down2 = false;
}
if(!_Keyboard->isKeyDown(OIS::KC_3))
{
  _down3 = false;
}
```

9. In the `startup()` function, add the three compositors to the viewport to the end of the function:

```
Ogre::CompositorManager::getSingleton().addCompositor(viewport,
"Compositor2");
Ogre::CompositorManager::getSingleton().addCompositor(viewport,
"Compositor3");
Ogre::CompositorManager::getSingleton().addCompositor(viewport,
"Compositor7");
```

10. Remember to change the instantiation of the `FrameListener` to add the viewport pointer as parameter:

```
_listener = new MyFrameListener(window,camera,viewport);
```

11. Compile and run the application. Using the *1*, *2*, *3* keys, you should be able to turn different compositors on and off. The *1* key is for making the image black and white, the *2* key inverts the image, and the *3* key makes the image look like it has a smaller resolution; you can combine all of the effect the way you like:

3. We are going to use keyboard input to switch the compositors on and off. To be able to differentiate between key presses, we need to know the previous state of the key:

```
bool _down1, _down2, _down3;
```

4. Change the constructor of the `FrameListener` to take the viewport as a parameter:

```
MyFrameListener(Ogre::RenderWindow* win,Ogre::Camera*
cam,Ogre::Viewport* viewport)
```

5. Assign the viewport pointer to the member and assign the Boolean value their starting value:

```
_viewport = viewport;

_comp1 = false;
_comp2 = false;
_comp3 = false;

_down1 = false;
_down2 = false;
_down3 = false;
```

6. If the key number *1* is pressed and it wasn't pressed before, change the state of the key to pressed, flip the state of the compositor, and use the flipped value to enable or disable the compositor. This code goes into the `frameStarted` function:

```
if(_Keyboard->isKeyDown(OIS::KC_1) && ! _down1)
{
  _down1 = true;
  _comp1 = !comp1;
  Ogre::CompositorManager::getSingleton().setCompositorEnabled(_
viewport, "Compositor2", _comp1);
}
```

7. Do the same for the other two compositors we are going to have:

```
if(_Keyboard->isKeyDown(OIS::KC_2) && ! _down2)
{
  _down2 = true;
  _comp2 = !comp2;
  Ogre::CompositorManager::getSingleton().setCompositorEnabled(_
viewport, "Compositor3", _comp2);
}
if(_Keyboard->isKeyDown(OIS::KC_3) && ! _down3)
{
  _down3 = true;
  _comp3 = !comp3;
```



```
  Ogre::CompositorManager::getSingleton().setCompositorEnabled(_
viewport, "Compositor7", _comp3);
}
```

8. If a key is no longer pressed, we need to change the state of the key:

```
if(!_Keyboard->isKeyDown(OIS::KC_1))
{
  _down1 = false;
}
if(!_Keyboard->isKeyDown(OIS::KC_2))
{
  _down2 = false;
}
if(!_Keyboard->isKeyDown(OIS::KC_3))
{
  _down3 = false;
}
```

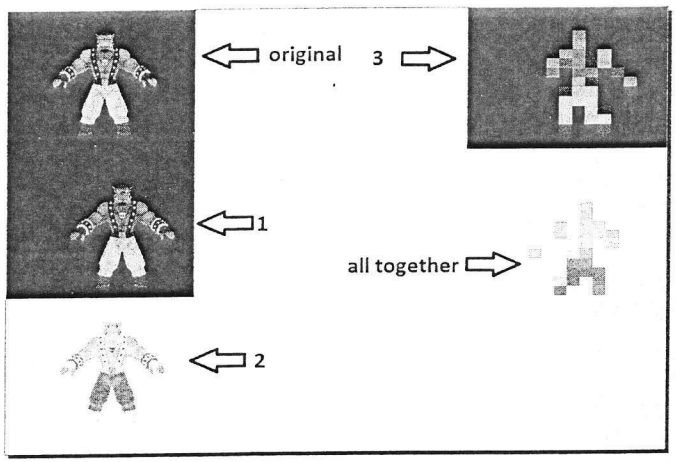
9. In the `startup()` function, add the three compositors to the viewport to the end of the function:

```
Ogre::CompositorManager::getSingleton().addCompositor(viewport,
"Compositor2");
Ogre::CompositorManager::getSingleton().addCompositor(viewport,
"Compositor3");
Ogre::CompositorManager::getSingleton().addCompositor(viewport,
"Compositor7");
```

10. Remember to change the instantiation of the `FrameListener` to add the viewport pointer as parameter:

```
_listener = new MyFrameListener(window,camera,viewport);
```

11. Compile and run the application. Using the *1*, *2*, *3* keys, you should be able to turn different compositors on and off. The *1* key is for making the image black and white, the *2* key inverts the image, and the *3* key makes the image look like it has a smaller resolution; you can combine all of the effect the way you like:



What just happened?

We added the compositors we wrote in the chapter about and made it possible to turn them on and off using the *1*, *2*, and *3* keys. To combine the compositors, we used the fact that Ogre 3D automatically chains compositors if more than one is enabled.

Adding a plane

Without a reference to where the ground is, navigation in 3D space is difficult, so once again let's add a floor plane.

Time for action – adding a plane and a light

Everything we are going to add this time is going in the `createScene()` function:

1. As we already know we need a plane definition, so add one:

```
Ogre::Plane plane(Ogre::Vector3::UNIT_Y, -5);
Ogre::MeshManager::getSingleton().createPlane("plane",
  Ogre::ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME, plane,
  1500,1500,200,200,true,1,5,5,Ogre::Vector3::UNIT_Z);
```

2. Then create an instance of this plane, add it to the scene, and change the material:

```
Ogre::Entity* ground= _sceneManager->createEntity("LightPlaneEnti
ty", "plane");
_sceneManager->getRootSceneNode()->createChildSceneNode()-
>attachObject(ground);
ground->setMaterialName("Examples/BeachStones");
```

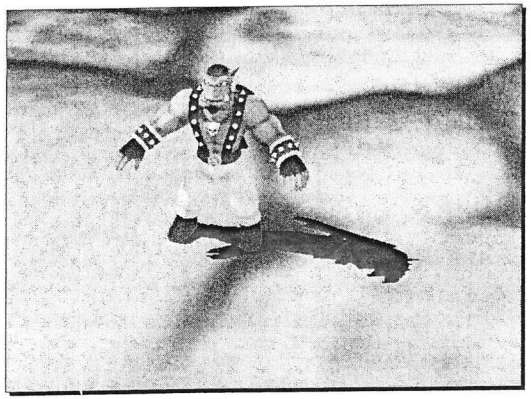
3. Also we would like to have some light in the scene; add one directional light:

```
Ogre::Light* light = _sceneManager->createLight("Light1");
light->setType(Ogre::Light::LT_DIRECTIONAL);
light->setDirection(Ogre::Vector3(1,-1,0));
```

4. And some shadows would be nice:

```
_sceneManager->setShadowTechnique(Ogre::SHADOWTYPE_STENCIL_
ADDITIVE);
```

5. Compile and run the application. You should see a plane with a stone texture and on top the Sinbad instance throwing a shadow on the plane.



What just happened?

Again, we used our previously gained knowledge to create a plane, light, and add shadows to the scene.

Adding user control

We have our model instance on a plane, but we can't move it yet; let's change this now.

10
OPTIONAL

Time for action – controlling the model with the arrow keys

Now we are going to add interactivity to the scene by adding the user control to the movements of the model.

1. The `FrameListener` needs two new members: one pointer to the node we want to move, and one float indicating the movement speed:

```
float _WalkingSpeed;
Ogre::SceneNode* _node;
```

2. The pointer to the node is passed to us in the constructor:

```
MyFrameListener(Ogre::RenderWindow* win,Ogre::Camera*
cam,Ogre::Viewport* viewport,Ogre::SceneNode* node)
```


3. Assign the node pointer to the member variable and set the walking speed to `50`:

```
_WalkingSpeed = 50.0f;
_node = node;
```

4. In the `frameStarted` function we need two new variables, which will hold the rotation and the translation the user wants to apply to the node:

```
Ogre::Vector3 SinbadTranslate(0,0,0);
float _rotation = 0.0f;
```

5. Then we need code to calculate the translation and rotation depending on which arrow key the user has pressed:

```
if(_Keyboard->isKeyDown(OIS::KC_UP))
{
  SinbadTranslate += Ogre::Vector3(0,0,-1);
  _rotation = 3.14f;
}
if(_Keyboard->isKeyDown(OIS::KC_DOWN))
{
  SinbadTranslate += Ogre::Vector3(0,0,1);
  _rotation = 0.0f;
}
if(_Keyboard->isKeyDown(OIS::KC_LEFT))
{
  SinbadTranslate += Ogre::Vector3(-1,0,0);
  _rotation = -1.57f;
}
if(_Keyboard->isKeyDown(OIS::KC_RIGHT))
{
  SinbadTranslate += Ogre::Vector3(1,0,0);
  _rotation = 1.57f;
}
```

6. Then we need to apply the translation and rotation to the node:

```
_node->translate(SinbadTranslate * evt.timeSinceLastFrame * _
WalkingSpeed);
_node->resetOrientation();
_node->yaw(Ogre::Radian(_rotation));
```

7. The application itself also needs to store the node pointer of the entity we want to control:

```
Ogre::SceneNode* _SinbadNode;
```

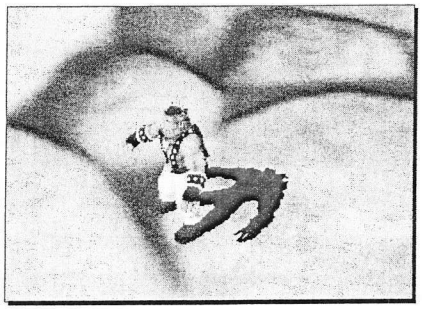
8. The `FrameListener` instantiation needs this pointer:

```
_listener = new MyFrameListener(window,camera,viewport,_
SinbadNode);
```

9. And the `createScene` function needs to use this pointer to create and store the node of the entity we want to move; modify the code in the function accordingly:

```
_SinbadNode = _sceneManager->getRootSceneNode()-
>createChildSceneNode();
_SinbadNode->attachObject(sinbadEnt);
```

10. Compile and run the application. You should be able to move the entity with the arrow keys:



What just happened?

We added entity movement using the arrow keys in the `FrameListener`. Now our entity floats over the plane like a wizard.

Adding animation

Floating isn't exactly what we wanted; let's add some animation.

Time for action – adding animation

Our model can move but it isn't animated yet, let's change this.

1. The `FrameListener` needs two animation states:

```
Ogre::AnimationState* _aniState;
Ogre::AnimationState* _aniStateTop;
```

2. To get the animation states in the constructor, we need a pointer to the entity:

```
MyFrameListener(Ogre::RenderWindow* win,Ogre::Camera*
cam,Ogre::Viewport* viewport,Ogre::SceneNode* node,Ogre::Entity*
ent)
```

3. With this pointer we can retrieve the `AnimationState` and save them for later use:

```
_aniState = ent->getAnimationState("RunBase");
_aniState->setLoop(false);

_aniStateTop = ent->getAnimationState(«RunTop»);
_aniStateTop->setLoop(false);
```

4. Now that we have the `AnimationState`, we need to have a flag in the `frameStarted` function, which tells us whether or not the entity walked this frame. We add this flag into the `if` conditions that query the keyboard state:

```
bool walked = false;
if(_Keyboard->isKeyDown(OIS::KC_UP))
{
  SinbadTranslate += Ogre::Vector3(0,0,-1);
  _rotation = 3.14f;
  walked = true;
}
if(_Keyboard->isKeyDown(OIS::KC_DOWN))
{
  SinbadTranslate += Ogre::Vector3(0,0,1);
  _rotation = 0.0f;
  walked = true;
}
if(_Keyboard->isKeyDown(OIS::KC_LEFT))
{
  SinbadTranslate += Ogre::Vector3(-1,0,0);
  _rotation = -1.57f;
  walked = true;
}
if(_Keyboard->isKeyDown(OIS::KC_RIGHT))
```

```
{
  SinbadTranslate += Ogre::Vector3(1,0,0);
  _rotation = 1.57f;
  walked = true;
}
```

5. If the model moves, we enable the animation; if the animation has ended, we loop it:

```
if(walked)
{
  _aniState->setEnabled(true);
  _aniStateTop->setEnabled(true);
  if(_aniState->hasEnded())
  {
    _aniState->setTimePosition(0.0f);
  }
  if(_aniStateTop->hasEnded())
  {
    _aniStateTop->setTimePosition(0.0f);
  }
}
```

6. If the model didn't move, we disable the animation and set it to the start position:

```
else
{
  _aniState->setTimePosition(0.0f);
  _aniState->setEnabled(false);
  _aniStateTop->setTimePosition(0.0f);
  _aniStateTop->setEnabled(false);
}
```

7. In each frame, we need to add the passed time to the animation; otherwise, it wouldn't move:

```
_aniState->addTime(evt.timeSinceLastFrame);
_aniStateTop->addTime(evt.timeSinceLastFrame);
```

8. The application now also needs a pointer to the entity:

```
Ogre::Entity* _SinbadEnt;
```

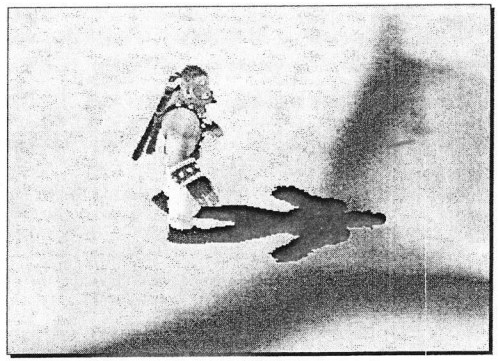
9. We use this pointer while instantiating the `FrameListener`:

```
_listener = new MyFrameListener(window,camera,viewport,_
SinbadNode,_SinbadEnt);
```


10. And, of course, while creating the entity:

```
_SinbadEnt = _sceneManager->createEntity("Sinbad.mesh");
```

11. Compile and run the application. Now the model should be animated when it moves:



What just happened?

We added animation to our model, which is only enabled when the model is moved.

Have a go hero – looking up what we used

Look up the chapters where we discussed the techniques we used for the last examples.

Summary

We learned a lot in this chapter about creating our own application to start and run Ogre 3D.

Specifically, we covered the following:

- How the Ogre 3D startup process works
- How to make our own main loop
- Writing our own implementation of an application and `FrameListener`

Some topics we have already covered, but this time we combined them to create a more complex application.

We have now learned everything needed to create our own Ogre 3D applications. The next chapter will focus on extending Ogre 3D with other libraries or additional features to make better and prettier applications.