TNM086 – VR-technology
1. Scene Graphs

December 20, 2019

1 Introduction

In this exercise you will try out both basic and more advanced scene graph concepts. After completing the exercise you will

1. understand the basic parts of a scene graph API,
2. be able to use a scene graph API to place objects in the world and control these,
3. understand how more advanced scene graph features can assist in building dynamic behaviour, and
4. should have enough general knowledge to easily understand any software that implements the same functionality and relay this to the software used in this exercise.

The API installed for this exercise is the OpenSceneGraph API (OSG). Other APIs may, however, be used instead. The support from supervisors can of course not be guaranteed for other APIs. Each task may also be replaced by an *equivalent* task with the API of choice.

OSG is a very feature rich scene graph system. Because of this, you will only have time to take a look at and use a basic subset of the capabilities of the software.

1.1 Documentation

- OpenSceneGraph Quick Start Guide (free in PDF format — in the models folder)
  http://www.lulu.com/shop/paul-martz/openscenegraph-quick-start-guide/ebook/product-17451155.html
- OpenSceneGraph Reference Manual
  http://weber.itn.liu.se/ karlu20/tmp/OpenSceneGraph/

1.2 Preparatory Reading

- OpenSceneGraph Quick Start Guide, sections 2.0–2.5, 3.2
- OpenSceneGraph Getting Started
  http://www.openscenegraph.org/index.php/documentation/getting-started

1.3 Tips

- To get more information about what OSG is doing, execute in your bash
  `export OSG_NOTIFY_LEVEL=DEBUG`

2 Getting Started

In this part the most basic things are done with the scene graph. Start with the provided stub, make sure you understand what is happening and add the code required for the tasks below. Most of the information for this part is available in the quick start guide for OSG.

Task 1 — Create the ground:
The `osg::HeightField` is a shape node that provides a means for manual specification of a ground patch. Use this node to create a ground. Also load an image and apply this as texture for the ground. You may specify the height of the ground as some kind of function, for example `sin` and `cos`.

The minimum set of functions you'll need to use to perform this task are `allocate` to allocate the nodes for the grid, `setXInterval` and `setYInterval` to set the grid intervals, and `setHeight` to set the height of each node in the grid. You may also want to use the function `setOrigin`.

Task 2 — Load objects:
Load at least two other objects (using `osgDB::readNodeFile` found in `osgDB/ReadFile`) and position them in the world.

Various models have been made available in OSG format.

Since the OSG system uses OpenGL as a back-end interface towards the graphics hardware, the lighting capabilities are closely related to the behaviour of lights in legacy OpenGL. A light is positioned through a `osg::LightSource` node in the scene graph, and its properties are controlled through an associated `osg::Light`, but is activated by setting a state in the root of the sub graph where the light should be active. This can be done in two ways: manually by calling the root's state set's `setMode`, or by letting the light source do that through a call to `setStateSetModes` with the root's state set as argument.

Task 3 — Lighting:
Add lighting effects to the scene. Create at least two lights that affect the shading of objects in your world, with different colour. The rendering of shadows is not required.

3 Advanced Features

Now that you have familiarized with both the OSG structure and the documentation it is time to use some more advanced features of the system. In this part of the exercise you will need to make use of features that are more sparsely found in scene graph APIs and systems. This is still just a small selection of the various effects and tools available in OSG.

Task 4 — Use LoD:
Use the `osg::LOD` node to make the system show a model of different level-of-detail at different distances. Use at least three different levels and use the `osgUtil::Simplifier` to create low level models from the high level version of the model.

Since the simplifier tool makes polygon reduction on a selected model, make sure that you first copy the model. That can be done using the `clone` function, but make sure that you do a deep copy. Also, observe that version 1, 2 and 3 of OSG have slightly different interfaces towards the simplifier tool. Depending on version you can use `traverse`, `apply` or `accept`, respectively.

You may, to make the examination quicker, adjust the range for the different levels-of-details so that it is easy to see the LoD change. In a real application the change is of course not supposed to be perceptible.

Task 5 — Moving object:
Use the `osg::AnimationPath` class together with the `osg::AnimationPathCallback` class to update the transform of an object or light source, thereby making it move through your world.

The animation path functionality is not described in the quick start guide, but the use is straightforward. A animation callback object (`osg::AnimationPathCallback`) is created and added as callback function for a transform node using the `setUpdateCallback` function available in the super class `osg::Node`. When this callback object is created it should be set to use a configured animation path object (`osg::AnimationPath`). Look up the respective class in the reference manual for details.

The animation path callback described above is a convenient pre-implemented callback to update a transform based on an animation path. However, it is straightforward to implement your own callback that updates nodes in any other way, by providing your own sub class of the `osg::NodeCallback` class, implementing the *function call operator* (`operator()`).

OpenSceneGraph and other scene graph APIs include means to trigger events and interact with objects. This can be used to encode a more dynamic world. In the following task you will use intersection visitor (employing the Visitor design pattern) and an intersector. An example of how these are used can be found in osgintersection.cpp[1]. You will also have to identify which of multiple objects that are currently being intersected. This can be done using the nodePath, which is an attribute of each intersection detected by the `Intersector`.

Task 6 — Trip Wire:
Enable the line drawing code in the stub and implement a trip wire that changes things in the scene, such as the color of light sources or the shapes of objects, when moving objects cross the line. Let at least two objects be affected by the trip wire. Use a node callback to have the intersection checked for every frame in the execution, and attach the callback object to the root of the scene graph or to the objects that you want to update upon intersection. Adjust the line end points if needed.

The following task is optional, but can be quite nice to try out. It is possible to add a particle source to a moving object.

Task 7 — Particle systems: (optional)
OSG has a package for generating particle systems, `osgParticle`. Use selected effects with the particle system, for example to make some objects explode or smoke.

4 Final Remarks

Your programming tasks are now all over and all you have to do is take a look at what you've done. You should now have an understanding of both the OSG API and its capabilities, and the functionality of a scene graph system in general. Now, just one last task to demonstrate your new skills.

Task 8 — Understand the scene graph:
Draw the resulting scene graph of your application either on paper or in some drawing program on a computer. Make sure that you include all level-of-detail models, textures and light sources.

[1] http://www.itn.liu.se/ karlu20/tmp/OpenSceneGraph/osgintersection.cpp