OpenSceneGraph

Alexander Birkner
(3070106)

Programming of Graphics Shaders

Prof. Dr.-Ing. C. Schiedermeier

Technische Hochschule Nürnberg Georg Simon Ohm

Fakultät Informatik

WS 20/21

12.01.2021

Table of Contents

List of Figures

1 Introduction

OpenSceneGraph is established as the world´s leading scene graph technology [1]. Its history began as Don Burns developed a scene graph component of a hang-gliding simulator for Silicon Graphics International (SGI) in 1998. He wrote a scene graph application programmable Interface (API), runnable on Linux systems. This API was later used as the prototype of OpenSceneGraph. In 1999 Robert Ostfield started to collaborate on the project. Together they took the developed scene graph API, made a standalone open-source project out of it and ported it to Windows. After that Robert Ostfield named the project OpenSceneGraph [2].

In the following years, the project received a growing interest. In response to that, Robert Osfield set up the company OpenSceneGraph Professional Services and provided both commercial and free OpenSceneGraph services. Like Robert Osfield, Don Burns also formed his own company, namely Andes Computer Engineering. He pursued to support the further development of the OpenSceneGraph project. After many libraries and packages, like osgText for text rendering or osgFX for special effects, were added to the project, OpenSceneGraph 1.0 was announced in 2005. Two years later OpenSceneGraph 2.0 was released [2].

Till the release of OpenSceneGraph 3.0 which came with the support of OpenGL ES and OpenGL 3.0 in 2010, many further packages like osgWidget for the integration of user interface (UI) components, osgVolume to enable the rendering of volumes or osgAnimation to animate objects of the scene graph were included in the project [2]. In 2019 the most recent version, OpenSceneGraph 3.6.5, was released and the team behind OpenSceneGraph also started further projects like VulkanSceneGraph, an attempt combining Vulkan and C++ 17 to create a next-generation scene graph [1].

This report is intended to provide an introduction and an overview of OpenSceneGraph and its use. For this purpose, OpenSceneGraph is described in more detail in the second chapter. Afterwards, chapter three describes the use and the most important components of OpenSceneGraph. To demonstrate the effect of different nodes and settings within a scene graph, an interactive editor was developed. It is described in chapter 4. Subsequently, the features and advantages of using OpenSceneGraph are listed in chapter 5. The report concludes with the personal experiences that were made while using OpenSceneGraph and tries to clarify the question of when OpenSceneGraph should be used for the development of rendering applications.

2 OpenSceneGraph

This chapter provides an overview of OpenSceneGraph. It is an open-source high-performance 3D graphics toolkit, that is used in fields such as visual simulation, games and scientific visualization [1]. As rendering middleware, it raises the level of abstraction of the usage of the low-level graphics API OpenGL [3]. It has been developed using Standard C ++ [1]. OpenSceneGraphs current version (3.6.5) supports OpenGL 1.0 up to OpenGL 4.2 as well as OpenGL ES. This makes it possible to develop applications that can run on both rather older and newer hardware [4]. As the name suggests, OpenSceneGraph is based on the idea of scene graphs [1]. So, a short explanation of scene graphs should be given in the following section 2.1.

Since OpenSceneGraph is a retained rendering system [3], the term should also be described in more detail in section 2.2.

2.1 Scene Graphs

A scene graph is a general data structure that is used to define the spatial and logical relationships within a graphical scene to efficiently manage and render the scene. Often it is represented as a hierarchical graph that contains a collection of different types of graphical nodes, including a top-level root node, several group nodes that can have any number of child nodes and a set of leaf nodes that have zero child nodes. The grouping of child nodes under a parent node makes it possible to share the information of the parent node among all children. Besides, it allows to treat them as one unit [5].

2.2 Retained Rendering

In general, graphic APIs can be distinguished in retained mode APIs and intermediate mode APIs [6]. Figure 1 displays a side-by-side comparison of both forms.

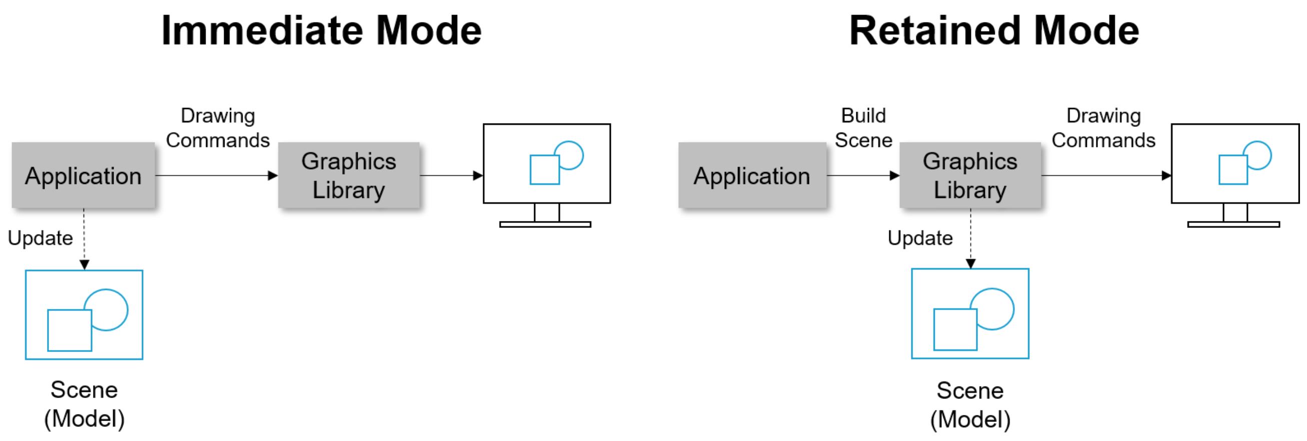


Figure 1: Comparison of immediate and retained mode rendering (based on figures of [6])

When using an immediate mode rendering API (shown on the left in figure 1) the application must store and keep track of its scene itself. Each time a new frame should be drawn on the screen, the application directly calls the drawing commands of the graphics library in a procedural way [6]. The graphics library then uses the hardware to draw the scene on the screen.

In contrary to immediate mode rendering APIs, retained mode APIs (shown on the right in figure 1) are declarative. Here the graphics library stores a model of the scene that is constructed by the application out of graphics primitives. To draw a new frame on the screen, the graphics library transforms the scene in a set of drawing commands and draws the scene on the screen. The application uses commands of the graphics library to update the stored scene (e.g. adding a new shape) in order to change the rendered frames. The library then updates the called drawing commands depending on the changes within the scene [6].

On the one hand, retained rendering APIs simplify use, as the API does more work, such as initialization, status management and clean up. On the other hand, they often offer less flexibility than immediate mode rendering APIs because the API imposes its own scene model [6].

When using OpenSceneGraph, it stores the scene that is build up by an application in a scene graph representation. In doing so, it records rendering commands and rendering data in a buffer. This enables the OpenSceneGraph toolkit to perform various optimization before the rendering of the scene. For the actual rendering, the OpenGL API is used by the toolkit [3]. The application does not call any OpenGL drawing commands itself.

3 Use of OpenSceneGraph

After the overview given in chapter 2, this section explains the use of OpenSceneGraph. First, the source code of a simple OpenSceneGraph application is shown. Afterwards, the most important libraries provided by OpenSceneGraph are listed and described. Then the classes for representing graphical nodes, how scene graphs are traversed and the concept of smart pointers are explained in more detail. The last section of this chapter deals with the `osg::StateSet` class, which is used to encapsulate the OpenGL state machine [7].

3.1 „Hello World“

Listing 1 shows the source code of the probably simplest possible OpenSceneGraph application.

```
 1  #include <osgDB/ReadFile>
 2  #include <osgViewer/Viewer>
 3
 4  int main(int argc, char** argv) {
 5          osg::ref_ptr<osg::Node> root =
              osgDB::readNodeFile("OpenSceneGraphData\\cow.osg");
 6
 7          osgViewer::Viewer viewer;
 8          viewer.setSceneData(root.get());
 9
10          viewer.run();
11  }
```

Listing 1: A minimalistic OpenSceneGraph sample (strongly inspired by [8])

In line 5, a model, that is built with a scene graph, is loaded from a file and the reference to the root node of the loaded model is stored in an instance of the `osg::ref_ptr<>` class template. This is a smart pointer provided by OpenSceneGraph for additional features in terms of efficient memory management (further explanation in section 3.5). The stored reference is of the `osg::Node` type. It represents basic nodes of a scene graph (further explanation in section 3.3). In line 7 an `osgViewer::Viewer` object, which provides a simulation loop for the application [8] and allows the rendering of a scene graph on the screen, is generated. With the call of the method `setSceneData()`, it is possible to inform the viewer about which node is the root of the scene graph to be rendered. By calling the `run()` method of the viewer in line 10, a window will be created and the rendering of the scene graph is. A screenshot of the result of the rendering can be found in figure 6 in appendix II.

3.2 Libraries

The OpenSceneGraph functionalities are provided via several libraries. These are divided into the standard core libraries and a set of additional modular libraries, the so-called node kits [9].

The libraries either can be built from source (available at [10]) using CMake and a C++ Compiler or the pre-build binaries (only available for Windows) from [11] are used [12].

The core libraries are [9]:

- **osg library:** It contains basic methods and elements to build a scene graph including nodes, geometries and rendering states as well as its own set of math classes to implement vector and matrix operations [9].
- **osgDB library**: This library provides support for reading and writing a wide variety of image types (e.g. .jpg, .png and .bmp [4]) and 3D database formats (e.g. COLLADA, LightWave and Alias Wavefront [4]) via a dynamic plugin mechanism [9].
- **OpenThreads library:** The OpenThreads library gives programmers a minimal and object-oriented thread interface [9].
- **osgUtil library:** It is designed to be used to implement OpenSceneGraph rendering backends, that traverse the scene graph, perform culling operations and convert the information in the scene into a series of OpenGL calls. Additionally, functionalities for polygon modification algorithms and user interaction are included [9].

By providing node kits, OpenSceneGraph offers a wide range of rendering functionality that can be compiled within the application or be loaded at runtime [9]. Because of the scope of this report, only two of the currently available node kits are exemplarily described here:

- **osgManipulator library:** This library enables developers to add interactive controls to the scene [9].
- **osgViewer library**: It offers a viewer class that can render a scene and provides a simulation loop for applications (it was already used in the sample in section 3.1). The library also contains further viewer-related classes to integrate OpenSceneGraph scenes in a wide variety of windowing systems [9].

Most of the applications that are based on OpenSceneGraph use the osg, osgDB, osgUtil and osgViewer libraries [9]. Additional libraries are used depending on the requirements of the respective application. Figure 5 in appendix I shows all libraries provided by OpenSceneGraph, including all node kits available within the current distribution (Version 3.6.5).

3.3 Classes to Represent Nodes

The `osg::Node` class, which is also used in the example from section 3.1, represents a basic element in a scene graph [8]. To provide different functionality through different types of nodes, OpenSceneGraph provides many classes that inherit from the `osg::Node` class.

The `osg::Group` class is used to group nodes within a scene graph. Instances of this class can have any number of child nodes. Since the `osg::Group` class represents a node, it is derived from the `osg::Node` class [13]. OpenSceneGraph offers different derivatives of the `osg::Group` class to be able to pass specific properties and functionalities to child nodes. Figure 2 displays the `osg::Geode` class and a subset of classes that derive from it. For the sake of clarity, only a subset of the classes is shown. A full list of all classes that are derived from the `osg::Group` class can be found in [14].

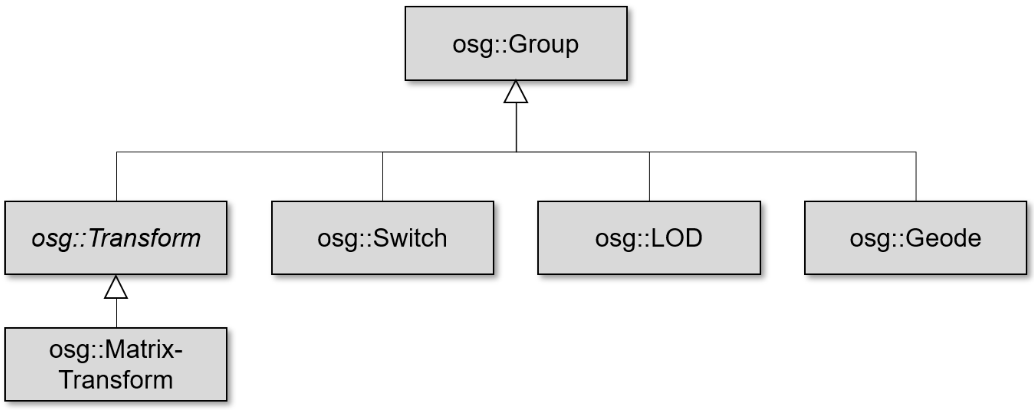


Figure 2: Simplified class diagram of the osg::Geode class including some derived classes

The derived classes that are displayed in figure 2 are described in more detail below:

- **osg::Transform class:** This class is used to apply traversal-concatenated transformations to geometries of child nodes. When a scene graph is traversed down, `osg::Transform` nodes add their own transformation to the currently used transformation matrix (the OpenGL model-view matrix). It is not possible to instantiate an `osg::Transform` node directly. Instead, OpenSceneGraph provides a variety of subclasses of the `osg::Transform` class, like the `osg::MatrixTransform` class, to realize different transformations [15].
- **osg::MatrixTransform class:** As shown in figure 2, this class is derived from `osg::Transform` class. It is used to apply 4x4 double type matrix transformation and uses an `osg::Matrix` variable internally to do so. The `osg::Matrix` class offers methods to create matrices for different types of transformations.
- **osg::Switch:** This type of node can be used to render or skip specific children depending on a given condition. To achieve this, it attaches a Boolean value, that indicates if the child node (and nodes that are below it) should be rendered or not [17].
- **osg::LOD:** Nodes of this class can be used to represent the same object with different levels of detail. This allows it, to render the object with the appropriate detail level according to the distance to the viewer. All child nodes of `osg::LOD` nodes should be the same object at varying levels of detail, while they are ordered from the highest to the lowest level. For each level (and therefore also for each child) a minimum and maximum visible range need to be specified. This allows the renderer to automatically decide which child nodes need to be rendered depending on the viewer´s current distance to the `osg::LOD` node [18].
- **osg::Geode:** The `osg::Geode` nodes represent leaf nodes of a scene graph. Nodes of this class have no children but they contain and manage geometry information for the actual rendering [19].

Geometry data that should be drawn on the screen are stored in `osg::Drawable` objects. Those can be attached to `osg::Geode` nodes. The `osg::Drawable` class is a pure virtual class and therefore it cannot be instantiated. It has a variety of subclasses to render models, images and texts to the OpenGL pipeline [19]. For example, to render basic geometry shapes, the `osg::ShapeDrawable` class can be instantiated and attached to an `osg::Geode` node [20].

3.4 Traversing the Scene Graph

In scene graphs, various update and rendering operations are carried out on nodes during a traversal. It is one of the key functions of a scene graph. The traversal of a scene graph typically consists of repeating steps. Starting from a certain node (at the root node if the whole graph should be traversed) the graph is traversed recursively down until a leaf node is reached. The graph is then traced back to the first node that has not yet been fully explored. The steps are repeated till all nodes have been explored. This type of traversal is also known as a depth-first search [21].

In OpenSceneGraph, four different types of traversals should be performed per frame one after another. First, all mouse and keyboard inputs as well as other user events should be processed in an event traversal. In the next traversal, the update traversal (or application traversal), the application can modify the scene graph (e.g. setting node and geometry properties). Then, during the cull traversal, checks are made to determine which nodes of the scene graph are within the viewport and are therefore worth rendering. During the last traversal, the draw traversal (or rendering traversal), the low-level OpenGL API is called to actually render the scene. For systems with multiple processors and graphic cards, OpenSceneGraph supports processing of the different traversals in parallel to improve rendering efficiency. By extending the `osg::NodeVisitor` class, it is possible to implement further rendering traversals [21].

3.5 Smart Pointer

Typically, when programming scene graphs, a pointer is created to the root node. This node directly or indirectly manages all further child nodes of the scene graph. If a node is no longer needed for rendering, the application needs to traverse the scene graph and delete its internal data carefully. This can quickly lead to errors or memory leaks, as it can be difficult for developers to evaluate if and how many other objects are still holding a pointer to the object that should be deleted. Some of the modern programming languages, like for example C# or Java, use a garbage collector to free any memory blocks that cannot be reached from any variables of a program. In Standard C++ this behavior can be mimicked utilizing smart pointers [22].

OpenSceneGraph comes with its own native smart pointer, the class template `osg::ref_ptr<>`, to enable automatic garbage collection and deallocation. For this to work, OpenSceneGraph provides the `osg::Referenced` class to manage the reference-counted memory blocks. This class is used as the base class of all classes that can be used as template arguments [23]. The classes for representing nodes and geometries mentioned in section 3.3 are also indirectly derived from `osg::Referenced`.

An integer counter is kept within the `osg::Reference` class to handle the number of allocated memory blocks. This reference count is initialized with 0 when the constructor is called. If the object is referenced by an `osg::ref_ptr<>`, the count is increased by 1. If it is removed from a smart pointer, the count is reduced by 1. If the object is no longer referenced by a smart pointer, i.e. the count falls back to 0, the object will be automatically destroyed. To save resources, to simplify debugging and to avoid bugs, OpenSceneGraph always recommends using smart pointers to manage a scene [23].

3.6 State Sets

To keep track of states that are related to the rendering, like attributes such as scene lights, materials, textures and modes that can be turned on or off using the OpenGL functions `glEnable()` or `glDisable()`, OpenGL typically uses a state machine. This may not be suitable for direct use in a scene graph structure. Therefore OpenSceneGraph uses the `osg::StateSet` class to encapsulate the OpenGL state machine. This enables developers to create realistic rendering effects [7].

To record rendering state attributes, OpenSceneGraph defines the virtual class `osg::StateAttribute`. It is inherited by different classes to implement various rendering attributes like lights, materials or blend functions [24]. Figure 3 shows the relationship between the `osg::Node`, the `osg::StateSet`, the `osg::StateAttribute` and derived classes of the `osg::StateAttribute` like `osg::BlendFunc`.

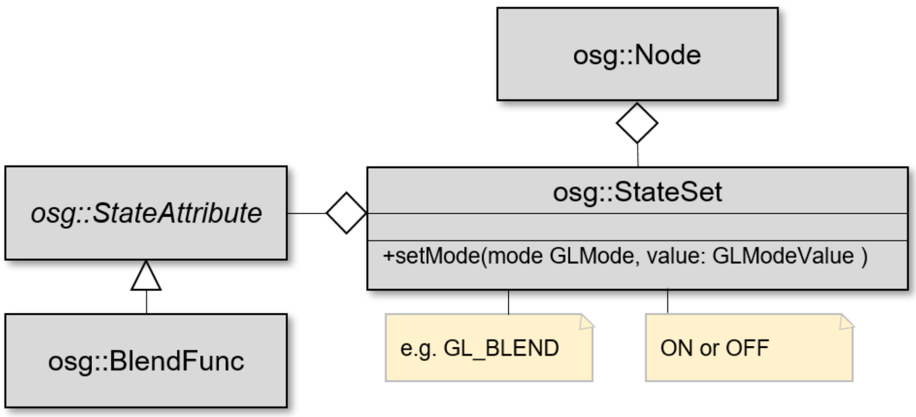


Figure 3: Simplified class diagram of the osg::StateSet class including related classes

An `osg::StateSet` object can be applied to a `osg::Node` [7]. Instances of classes that are derived from the `osg::StateAttribute` class can be added to an `osg::StateSet` to adjust the rendering behavior of objects to which the state set is applied to. It is also possible to turn on or off different rendering modes off an `osg::StateSet` by calling the method `setMode()`. The OpenGL rendering mode to be set (e.g. `GL_BLEND`) and a corresponding value (on or off are the possible values) are expected as parameters for calling this method. Sometimes a specific rendering mode is associated with an attribute [24]. For example, the mode `GL_BLEND` determines if specific blend functions, that can be set in a `osg::StateSet` via instances of the `osg::BlendFunc` class, are used or not.

The state of a node not only affects itself but also its children. However, child nodes can overwrite the state set of their parent. To change the inheritance of state sets, the three flags `osg::StateAttribute::OVERRIDE`, `osg::StateAttribute::PROTECTED` and `osg::StateAt-tribute::INHERITAT` can be used while setting or adding the values of state sets [25]. State sets also offer the possibility of adding vertex, fragment and geometry shaders [26], as well as uniforms that should be used within the shaders [27].

4 Playground Project

To show the effects of different nodes, attributes and rendering modes, an application was developed that enables the user to dynamically add or remove nodes to a scene graph via an UI that is created with Dear ImGui. All the node classes mentioned in section 3.3 are addable within the application. It also supports the functionality to add nodes from .osg files.

Besides, the application provides the functionality to adapt the state set of each node. This enables the user to select lighting settings and the polygon rasterization mode that should be used for the rendering. The creation of transparent rendering effects using blend functions and the addition of various shaders and associated uniforms is also possible.

To make the structure of the current scene graph recognizable for the user, the current scene graph is always displayed in a tree view in the UI (see screenshot figure 9 in appendix III). The user can select a node via this tree view (see figure 10 in appendix III). Depending on which type of node is selected, options to add or delete nodes and to adjust the state set, that are allowed on this type of node (e.g. it is not allowed to add further group nodes to nodes of the type `osg::Geode`), are displayed.

The following sections of this chapter give an overview of the architecture of the application and exemplary describe how some of the available functionalities were implemented.

4.1 Architecture

When the architecture of the developed application was created, an attempt was made to ensure that actions that can be carried out on nodes are designed as modularly as possible, so that they can be displayed or not depending on the type of node selected. Figure 4 shows a reduced class diagram of the architecture of the application.

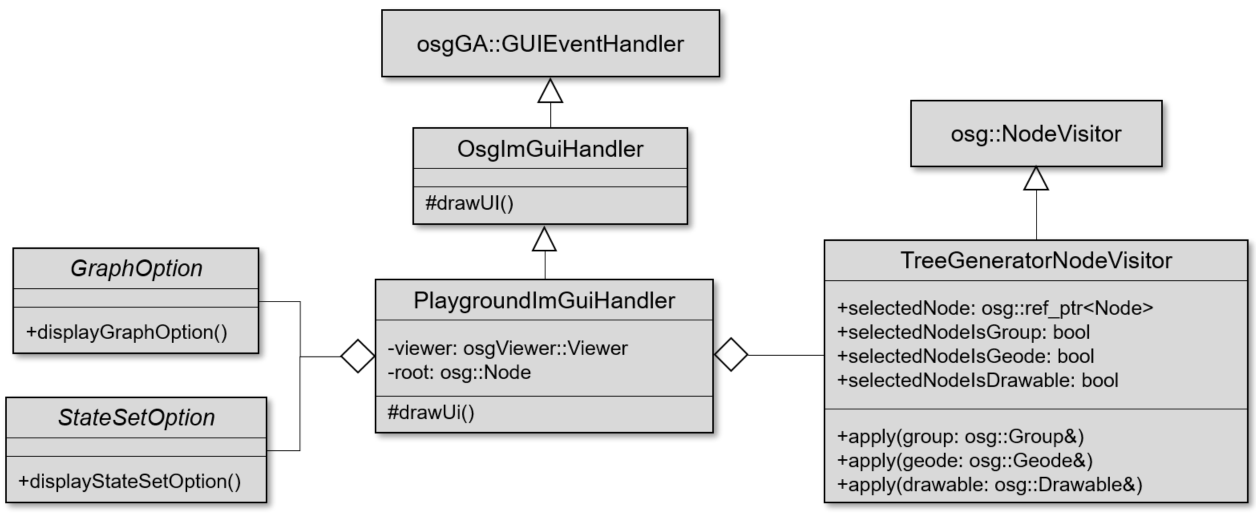

osgGA::GUIEventHandler
OsgImGuiHandler
#drawUI()
osg::NodeVisitor
GraphOption
+displayGraphOption()
PlaygroundImGuiHandler
-viewer: osgViewer::Viewer
-root: osg::Node
#drawUi()
TreeGeneratorNodeVisitor
+selectedNode: osg::ref_ptr<Node>
+selectedNodeIsGroup: bool
+selectedNodeIsGeode: bool
+selectedNodeIsDrawable: bool
+apply(group: osg::Group&)
+apply(geode: osg::Geode&)
+apply(drawable: osg::Drawable&)
StateSetOption
+displayStateSetOption()

Figure 4: Simplified class diagram of the architecture of the playground project

The start point of the application is the `main()` method as usual. First, an `osg::Viewer` object and an `osg::Group` node are created. This node is used as the root node of the scene graph of the application. Then the axes.osg file (it is delivered with OpenSceneGraph and contains a scene graph representing the coordinate axes) is read out using the osgDB library. This results in a group node, that contains coordinate axes as children. The axes should serve as an orientation for the user. The group that contains the axes is afterwards added to the root node.

Drawing the Dear ImGui UI is within the responsibility of the `PlaygroundImGuiHandler` class. An instance of this class is created in the `main()` method. Instances of the viewer and the root node of the `main()` method are passed to the constructor. To be able to use Dear ImGui in combination with OpenSceneGraph, the `PlaygroundImGuIHandler` extends the `OsgImGuiHandler` class. It is derived from the `osgGA::GUIEventHandler` class. By doing this, the inputs and events within the window are passed to the Dear ImGui elements. The `OsgImGuiHandler` class was not created as part of this work. It is imported from [28]. The `PlaygroundImGuiHandler` uses it as base class and overwrites the `drawUI()` method. The `osgGA::GUIEventHandler` and the `OsgImGuiHandler` class also ensure that the `drawUI()` method of the `PlaygroundImGuiHandler` is called in every frame. These calls draw the graphical user interface (GUI) of the application.

The `TreeGeneratorNodeVisitor` class is used to create the tree view, which shows the current state of the scene graph. Derived classes of the `osg::NodeVisitor` class can traverse scene graphs and carry out self-defined operations for each node. To define these operations, the `apply()` method must be implemented. It can be overloaded with several types of nodes. Depending on which node type is currently available during the traversal, the corresponding method is called within the traversal [29]. If the `apply()` method is not overwritten for a specific node class, the method for the next base class in the class hierarchy is called instead.

Within the `apply()` methods of the `TreeGeneratorNodeVisitor` class an entry in a tree view, to represent the current node, is created. Additionally, the currently selected node is set according to the selected tree view element. To display the possible options on this node, the type of the selected node is also stored. The `TreeGeneratorNodeVisitior` class overloads the `apply()` method to accept the three types `osg::Group`, `osg::Geode` and `osg::Drawable` since different operations should be possible for these three types of nodes (and their derivatives). Besides, leaf nodes should be displayed differently in the tree view (see figure 9 in appendix III).

The various options are implemented using derivatives of the two abstract classes `GraphOption` and `StateSetOption`. All child classes overwrite either the `displayGraphOption()` or the `displayStateSetOption()` method, depending on their base class. For each option, a fold-out label is created in the overwritten method (implemented via a tree). The child classes of the `GraphOption` class offer options for adding or removing nodes. The derivatives of the `StateSetOption` class provide the functionality to adapt the state sets of the nodes.

In the `DrawUI()` method of the `PlaygroundImGuiHandler`, the UI is drawn in each frame. First, the tree view is created using the `TreeGeneratorNodeVisitor` as described above. Then, depending on the selected type of node, the permitted options are displayed using the derivatives of the `GraphOption` and the `StateSetOption` class. Also, UI elements are created for adjusting the background color and for updating the camera control. The implementation of these functionalities is located directly in the `PlaygroundImGuiHandler`, as they only refer indirectly to the scene graph.

To enable the `PlaygroundImGuiHandler` to draw the UI and to receive user input within the `osg::Viewer`, an instance of it is added to the viewer in the `main()` method. Then the `osg::Viewer` is started by calling its `run()` method. This creates a window and the rendering loop is executed until the user exits the program. In this loop the various traverses (see section 3.4), are run through.

4.2 Implementation

This section explains the source code of one `displayGraphOption()` and one `displayStateSetOption()` method, as the implementations of those methods contain the essential code for adapting the scene graph within the application. As mentioned before, these methods are defined within the child classes of the `GraphOption` and the `StateSetOption` class. For each possible option within the project one class that derives either from the `GraphOption` or the `StateSetOption` class was created. Since the explanation of all those classes would exceed the scope of this report, only two implementations are shown.

4.2.1 Add of Group Nodes

The `AddGroupOption` class is a child class of the `GraphOption` class. Listing 2 shows the implementation of its `displayGraphOption()` method.

```
1   void AddGroupOption::displayGraphOption(TreeGeneratorVisitor*
        currentTreeGeneratorVisitor)
2   {
3           if (ImGui::TreeNode(“Add Group”)) {
4                   if (ImGui::Button(“Add Group”)) {
5                           Osg::ref_ptr<osg::Group> groupToAdd = new osg::Group;
6                           currentTreeGeneratorVisitor->selectedNode->asGroup()
                                ->addChild(groupToAdd);
7                   }
8                   ImGui::TreePop();
9           }
10  }
```

Listing 2: Implementation of the method displayGraphOption( ) of the AddGroupOption class

The reference to the `TreeGeneratorVisitor` instance that stores the selected node, is transferred as a parameter within a smart pointer. With the condition of the if statement in line 3, an expandable tree node with the label "Add Group" is drawn in the UI. If this tree node is expanded, the condition of the if statement is considered valid. A button is then added to the UI via the if statement in line 4. If this button is clicked, the code block of the if statement will be executed. First, a new group node is created, and its reference is assigned to a smart pointer. The currently selected node is then cast into a group node, because adding nodes using a reference to an `osg::Node` object is not permitted. Since this option is only displayed if the selected node is a group node, it is ensured, that the node stored in the `TreeGeneratorVisitor`, is of the type `osg::Group`. The newly created group node is then added as a child to the selected node.

The other implementations of the `displayGraphOption()` method are similar to the one shown in listing 2. Depending on the option they provide, additional input fields are displayed and used for the creation of the new node. For example, the option to add an `osg::MatrixTransform` node offers the selection of the type of transformation and allows to input values like the rotation angle.

4.2.2 Adjust Polygon Mode

To exemplarily show how the options to adjust the state sets of nodes are implemented, the major parts of the `displayStateSetOption()` method of the `AdjustPolygonModeOption` class is shown in listing 3. The class inherits from the `StateSetOption` class. It offers the user the possibility to change the polygon rasterization mode of an object.

```
1   void AdjustPolygonModeOption::displayStateSetOption(osg::ref_ptr<osg::StateSet>
2            stateSetOfCurrentlySelectedNode)
3   {
4            …
5            ImGui::Text(“Select Polygon Mode:”);
6            UtilityFunctions::displayImGuiComboBox("P-Face", polygonModeFaces,
                     faceSelectionIndex);
7            UtilityFunctions::displayImGuiComboBox(“P-Mode”, polygonModeModes,
                     modeSelectionIndex);
8            UtilityFunctions::displayImGuiComboBox(“P-Flag”, availableFlags,
                     flagSelectionIndex);
9
10           if (ImGui::Button("Set Polygon Mode")) {
11                   currentPolygonMode = new osg::PolygonMode;
12                   currentPolygonMode->setMode(
                             selectionIdToPolygonModeFace[faceSelectionIndex],
                             selectionIdToPolygonModeMode[modeSelectionIndex]);

13                   stateSetOfCurrentlySelectedNode->setAttribute(currentPolygonMode,
                             selectionIdToStateAttributeFlag[flagSelectionIndex]);
14           }
15  }
```

Listing 3: Major part of the implementation of the displayStateSetOption( ) method of the Adjust-PolygonMode class

A smart pointer (described in section 3.5) to the `StateSet` object (described in 3.6) of the selected node is transferred as a parameter. The calls of the method `displayImGuiComboBox()` in lines 6 to 8 display Dear ImGui combo boxes. The second parameters of the method calls contain the available selection options. The third parameters store the id of the selected elements. The combo boxes that are displayed with the calls in lines 6 and 7 allow the user to decide on which face which polygon mode should be applied. The possible selection options for the face correspond to the OpenGL enumerations `GL_FRONT`, `GL_BACK` and `GL_FRONT_AND_BACK`. The selection of the polygon mode to be applied corresponds to `GL_POINT`, `GL_LINE` and `GL_FILL`. To manage these values within OpenSceneGraph, it provides the enumerations `osg::PolygonMode::Face` and `osg::PolygonMode::Mode`. The index of the selected items within the combo boxes is mapped to values of these enumerations within the maps `selctionIdToPolygonModeFace` and `selectionIdToPolygonModeMode`. In line 8 an additional combo box is displayed, to allow the user to define the flag that should be used for this operation. With the flag he can control the inheritance of the created rasterization mode. The flags are described in section 3.6.

Below the combo boxes, a button is displayed by calling the `ImGui::Button()` method within the if condition line 10. If the button is pressed, the code block between lines 11 and 14 will be executed. To apply the polygon mode to a state set, the `osg::PolygonMode` class needs to be used. It is derived from the `osg::StateAttribute` class, which is described in section 3.6. An instance of the `osg::PolygonMode` class is created in line 11. By calling the `setMode()` method in line 12, the mode which is selected by the user within the combo box, will be applied to the `osg::PolygonMode` state attribute. The mode that is set here, is not the mode that is described in section 3.6. It only affects the attribute of the state set. After the configuration of the `osg::PolygonMode` instance, it is added to the state set of the selected node in line 13 with the call of the `setAttribute()` method of the `osg::StateSet` object. The flags are used as the second parameter.

The explanation of the `displayStateSetOption()` methods of further classes would exceed the scope of this report. The calls within the respective classes of the source code of the Playground project show how `osg::StateAttribute` objects can be read out as well as how various rendering modes can be adapted.

5 Features and Advantages of OpenSceneGraph

The basic functionality that the toolkit offers is the API, which allows the definition and rendering of a scene graph. One advantage that results from this is, that the productivity of developers can be increased compared to the pure use of a low-level API such as OpenGL. The developers no longer have to worry about low-level coding but can instead focus on developing the content of the application. The performance that is achieved by the support of view-frustum culling, occlusion culling, small feature culling, Level Of Detail (LOD) nodes, OpenGL state sorting, vertex arrays, vertex buffer objects, the OpenGL Shader Language and display lists are further advantages that OpenSceneGraph offers [4].

Via the osgDB library, which was already mentioned in section 3.2, OpenSceneGraph also offers support for reading and writing many 3D databases and 2D image file formats. Besides, OpenSceneGraph provides a large number of different NodeKits, which offer numerous additional functionalities such as the rendering of texts, volumes or particles. Since OpenSceneGraph only requires Standard C ++ and OpenGL and it is designed in a way that the dependencies on the various platforms are minimal, it can be executed on various platforms such as Windows, Android, MacOs and iOS. Further features are the support of additional programming languages like Java or Python through community projects and the support of multiple graphic contexts. This allows the execution of OpenSceneGraph based applications on multi-threaded and multi-GPU systems [4].

6 Conclusion

This report started with a brief introduction to the 3D graphics toolkit OpenSceneGraph and its history. Afterwards, the toolkit itself, scene graphs and retained rendering systems were briefly described. In the third chapter, the use of OpenSceneGraph was explained using a minimal example, followed by a description of some of the central elements for usage such as smart pointers and classes for representing nodes. Then the playground project, which offers an interactive editor for a scene graph, was presented in chapter 4. In the fifth chapter, the features and advantages of OpenSceneGraph were listed. In addition to the features, the object-oriented concept, the use of smart pointers and the flat learning curve, were positive points that were experienced during the development with OpenSceneGraph.

Especially due to the positive experience and the many features that OpenSceneGraph offers, the question of why it is not widely used in large projects arises. There are of course projects, like the flight simulator Flight Gear [30], that use it. But during the research for this report no large and popular project, like for example a AAA game (a classification that is used for games that are produced and distributed by a mid-sized or major publisher [31]) was found, that is based on OpenSceneGraph. One possible reason for this is, that in large applications, it is often necessary to map different types of relationships (e.g. spatial, semantic or rendering order) of objects. It seems quite difficult to map all these relationships using a single scene graph. It may only be achieved with further properties or regulations. Additionally, in projects with very large and complex scenes the mapping of the scene using a single scene graph, will likely result in either a root node with a very large number of single child nodes or in a graph with an unmanageable number of branches [32].

In conclusion, the decision about whether OpenSceneGraph should be used in a software system probably depends on the type of application that should be developed. In applications that cover a limited field that can be controlled by a scene graph, such as the Playground project, the use of OpenSceneGraph has certainly great advantages. In applications that require multiple complex scenes, the use of OpenSceneGraph may be more of an obstacle for developers than it brings advantages.

References

[1] OpenSceneGraph, *The OpenSceneGraph Project Website.* [Online]. Available: http://www.openscenegraph.org/ (accessed: Jan. 11 2021).
[2] R. Wang and X. Qian, *Openscenegraph 3.0 Beginner's Guide, Chapter 1, Section: The Birth and development of OSG*. Birmingham, Mumbai: Gardners Books, 2010. [Online]. Available: http://www.openscenegraph.org/index.php/documentation/books/10-openscenegraph-beginners-guide-published
[3] R. Wang and X. Qian, *Openscenegraph 3.0 Beginner's Guide, Chapter 1, Section: A quick overview of rendering middleware*. Birmingham, Mumbai: Gardners Books, 2010.
[4] OpenSceneGraph, *Features.* [Online]. Available: http://www.openscenegraph.org/index.php/about/features (accessed: Dec. 22 2020).
[5] R. Wang and X. Qian, *Openscenegraph 3.0 Beginner's Guide, Chapter 1, Section: Scene graphs*. Birmingham, Mumbai: Gardners Books, 2010.
[6] Quinn Radich and Michael Satran, *Retained Mode Versus Immediate Mode.* [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/learnwin32/retained-mode-versus-immediate-mode
[7] R. Wang and X. Qian, *Openscenegraph 3.0 Beginner's Guide, Chapter 6, Section: Encapsulationg the OpenGL state machine*. Birmingham, Mumbai: Gardners Books, 2010.
[8] R. Wang and X. Qian, *Openscenegraph 3.0 Beginner's Guide, Chapter 1, Section: Have a quick taste*. Birmingham, Mumbai: Gardners Books, 2010.
[9] R. Wang and X. Qian, *Openscenegraph 3.0 Beginner's Guide, Chapter 1, Section: Components*. Birmingham, Mumbai: Gardners Books, 2010.
[10] OpenSceneGraph, *Github Repository: OpenSceneGraph.* [Online]. Available: https://github.com/openscenegraph/OpenSceneGraph/ (accessed: Dec. 23 2020).
[11] OBJEXX Engineering, *OpenSceneGraph.* [Online]. Available: https://objexx.com/OpenSceneGraph.html
[12] OpenSceneGraph, *Stable Releases.* [Online]. Available: http://www.openscenegraph.org/index.php/download-section/stable-releases
[13] R. Wang and X. Qian, *Openscenegraph 3.0 Beginner's Guide, Chapter 5, Section: The Group interface*. Birmingham, Mumbai: Gardners Books, 2010.
[14] OpenSceneGraph, *Reference Guide - osg::Group Class Reference.* [Online]. Available: https://codedocs.xyz/openscenegraph/OpenSceneGraph/a02438.html (accessed: Dec. 23 2020).
[15] R. Wang and X. Qian, *Openscenegraph 3.0 Beginner's Guide, Chapter 5, Section: Transformation nodes*. Birmingham, Mumbai: Gardners Books, 2010.
[16] R. Wang and X. Qian, *Openscenegraph 3.0 Beginner's Guide, Chapter 5, Section: The MatrixTransform class*. Birmingham, Mumbai: Gardners Books, 2010.
[17] R. Wang and X. Qian, *Openscenegraph 3.0 Beginner's Guide, Chapter 5, Section: Switch nodes*. Birmingham, Mumbai: Gardners Books, 2010.
[18] R. Wang and X. Qian, *Openscenegraph 3.0 Beginner's Guide, Chapter 5, Section: Level-of-detail-nodes*. Birmingham, Mumbai: Gardners Books, 2010.
[19] R. Wang and X. Qian, *Openscenegraph 3.0 Beginner's Guide, Chapter 4, Section: Geode and Drawable classes*. Birmingham, Mumbai: Gardners Books, 2010.
[20] R. Wang and X. Qian, *Openscenegraph 3.0 Beginner's Guide, Chapter 4, Section: Rendering basic shapes*. Birmingham, Mumbai: Gardners Books, 2010.
[21] R. Wang and X. Qian, *Openscenegraph 3.0 Beginner's Guide, Chapter 5, Section: Traversing the scene graph*. Birmingham, Mumbai: Gardners Books, 2010.
[22] R. Wang and X. Qian, *Openscenegraph 3.0 Beginner's Guide, Chapter 3, Section: Understanding memory management*. Birmingham, Mumbai: Gardners Books, 2010.

[23] R. Wang and X. Qian, *Openscenegraph 3.0 Beginner's Guide, Chapter 3, Section: ref_ptr<> and Referenced classes*. Birmingham, Mumbai: Gardners Books, 2010.
[24] R. Wang and X. Qian, *Openscenegraph 3.0 Beginner's Guide, Chapter 6, Section: Attributes and modes*. Birmingham, Mumbai: Gardners Books, 2010.
[25] R. Wang and X. Qian, *Openscenegraph 3.0 Beginner's Guide, Chapter 6, Section: Inheriting render states*. Birmingham, Mumbai: Gardners Books, 2010.
[26] R. Wang and X. Qian, *Openscenegraph 3.0 Beginner's Guide, Chapter 6, Section: Understanding graphics shaders*. Birmingham, Mumbai: Gardners Books, 2010.
[27] R. Wang and X. Qian, *Openscenegraph 3.0 Beginner's Guide, Chapter 6, Section: Using uniforms*. Birmingham, Mumbai: Gardners Books, 2010.
[28] Alexey Galitsyn, *Github Repository: imgui-osg.* [Online]. Available: https://github.com/Tordan/imgui-osg (accessed: Jan. 11 2021).
[29] R. Wang and X. Qian, *Openscenegraph 3.0 Beginner's Guide, Chapter 5, Section: Visiting scene graph structures*. Birmingham, Mumbai: Gardners Books, 2010.
[30] OpenSceneGraph, *Use Cases - FlightGear.* [Online]. Available: http://www.openscenegraph.org/index.php/gallery/use-cases/81-flightgear
[31] Wikipedia - The Free Encyclopedia, *AAA (video game industry).* [Online]. Available: https://en.wikipedia.org/wiki/AAA_(video_game_industry) (accessed: Jan. 11 2021).
[32] Tom Forsyth, *Scene Graphs - just say no.* [Online]. Available: http://tomforsyth1000.github.io/blog.wiki.html#%5B%5BScene%20Graphs%20-%20just%20say%20no%5D%5D (accessed: Jan. 11 2021).
[33] R. Wang and X. Qian, *Openscenegraph 3.0 Beginner's Guide, Chapter 6, Section: Time for action - implementing a cartoon cow*. Birmingham, Mumbai: Gardners Books, 2010.

Appendix I: Image of all OpenSceneGraph Libraries

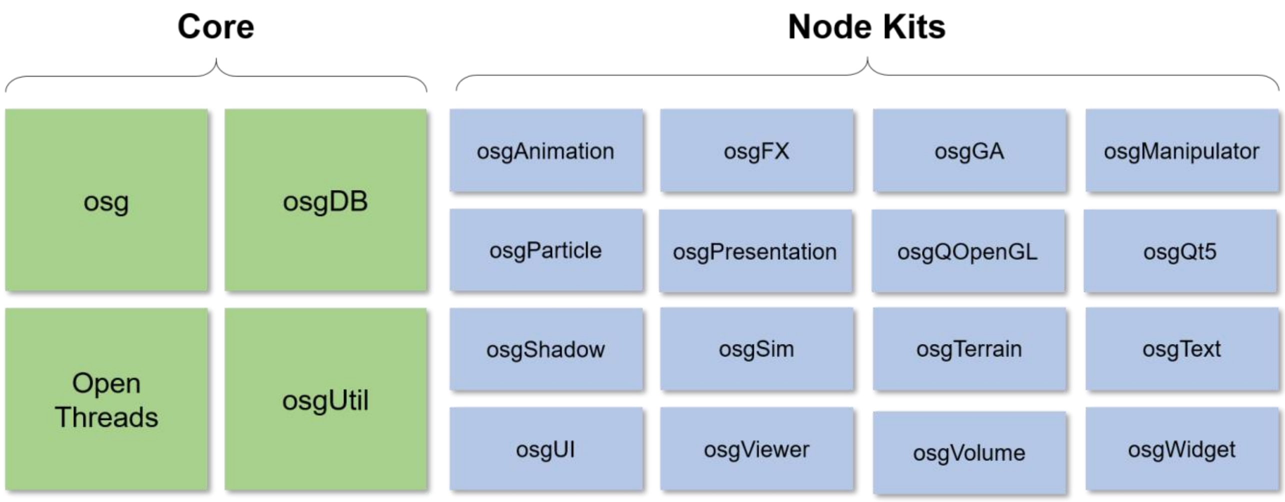


Figure 5: List of all libraries provided by OpenSceneGraph Version 3.6.5

Appendix II: Screenshot of the "Hello World" Project

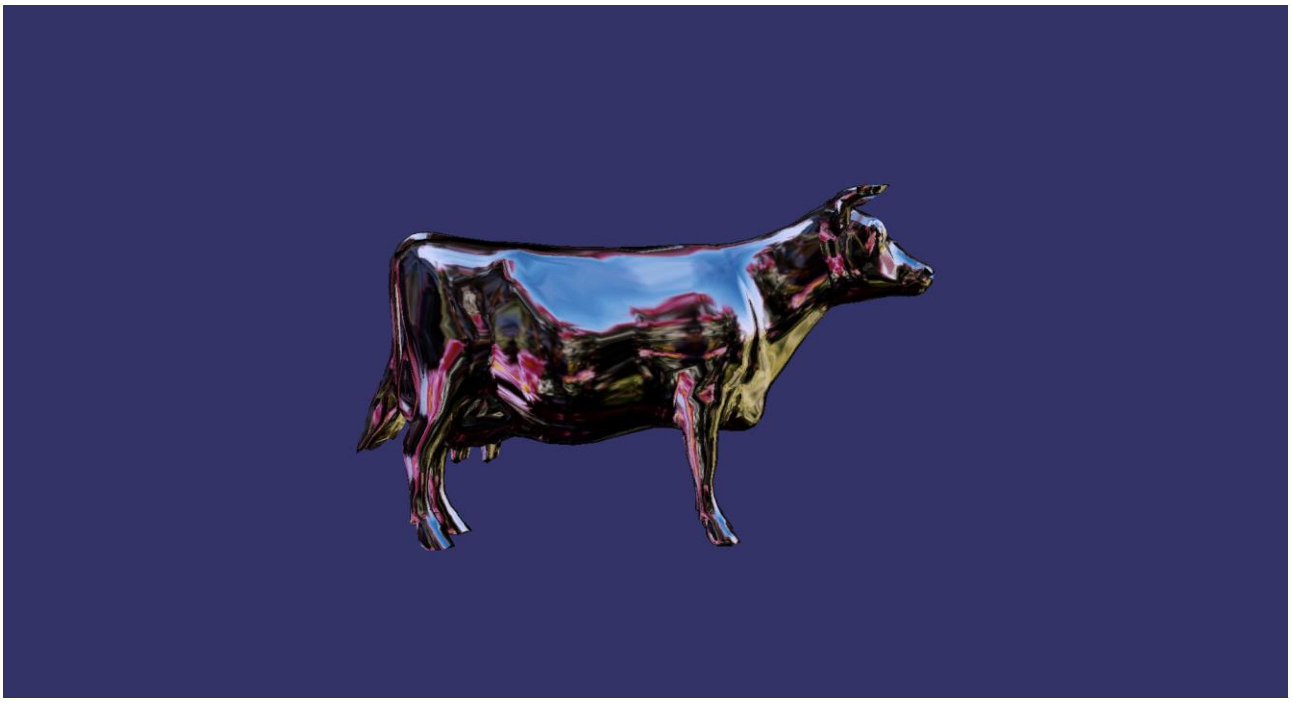


Figure 6: Screenshot of the "Hello World" Project

Appendix III: Screenshots of the Playground Project

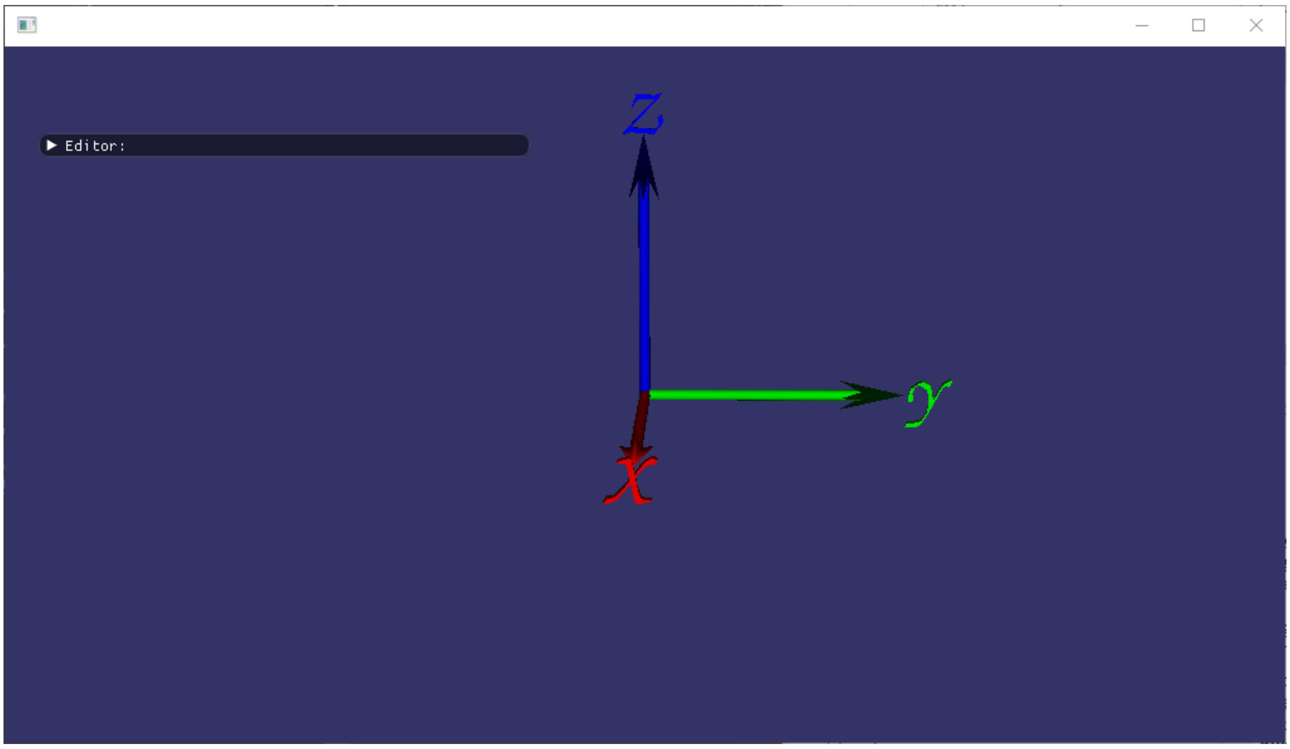


Figure 7: Closed editor

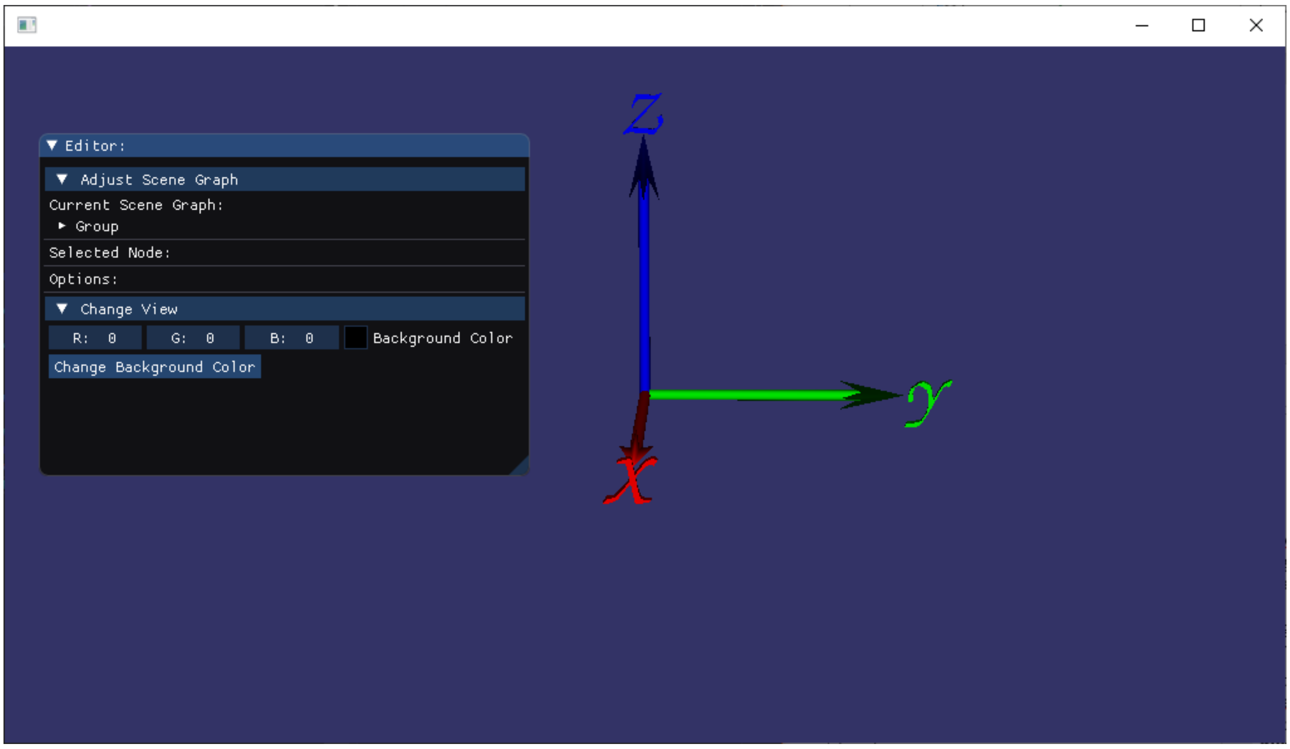


Figure 8: Opened editor

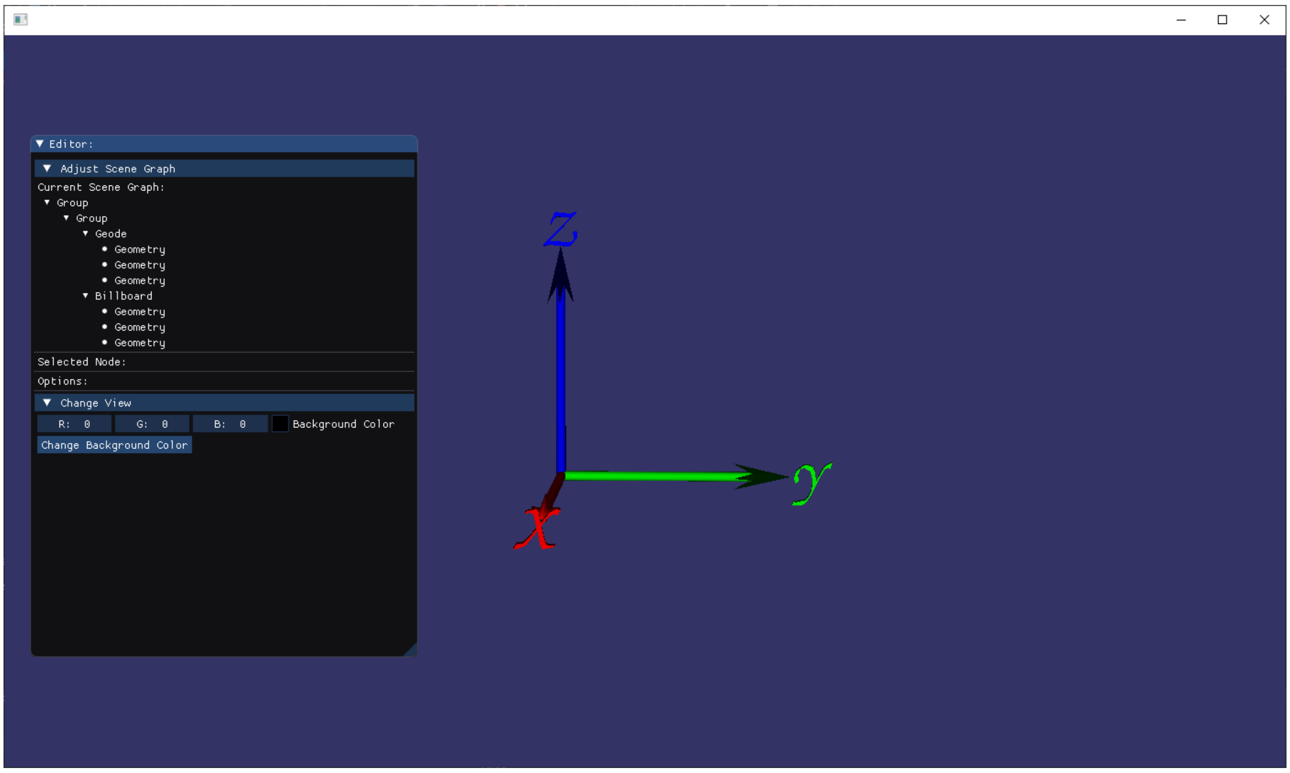


Figure 9: Opened tree view

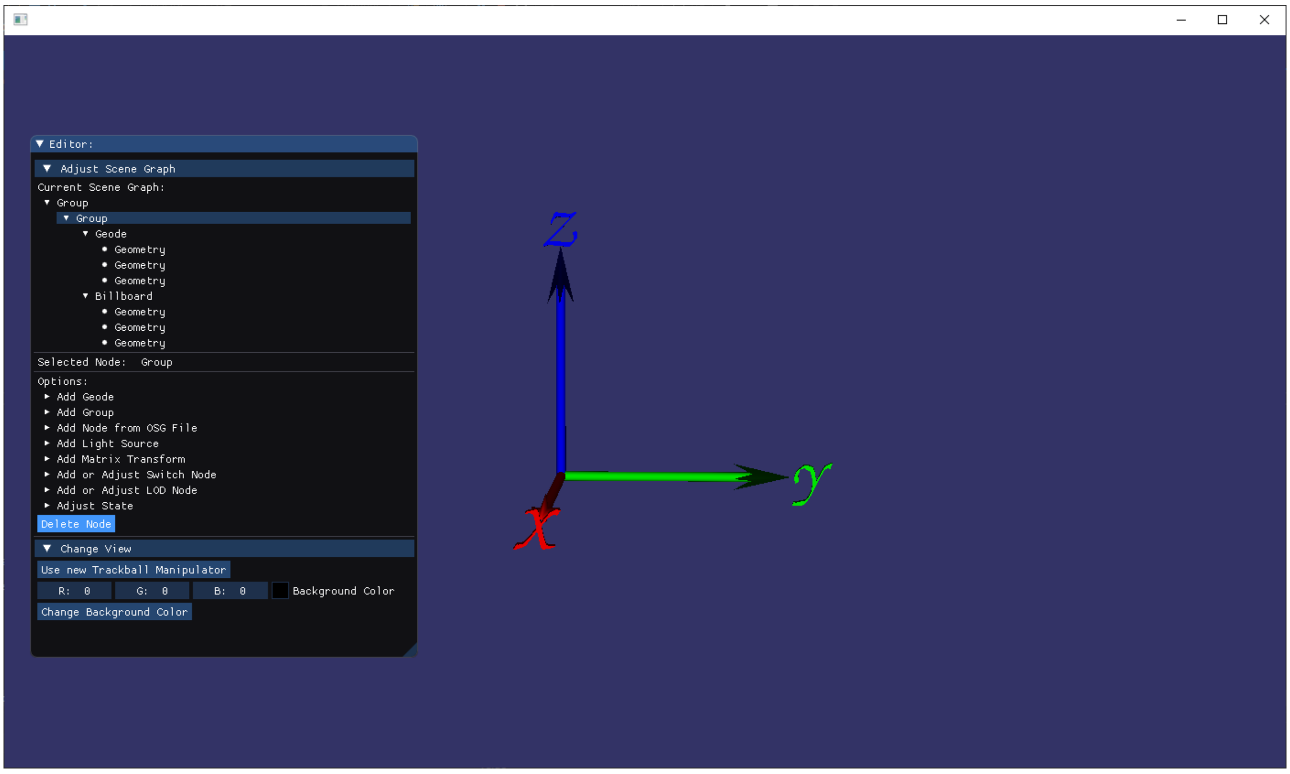
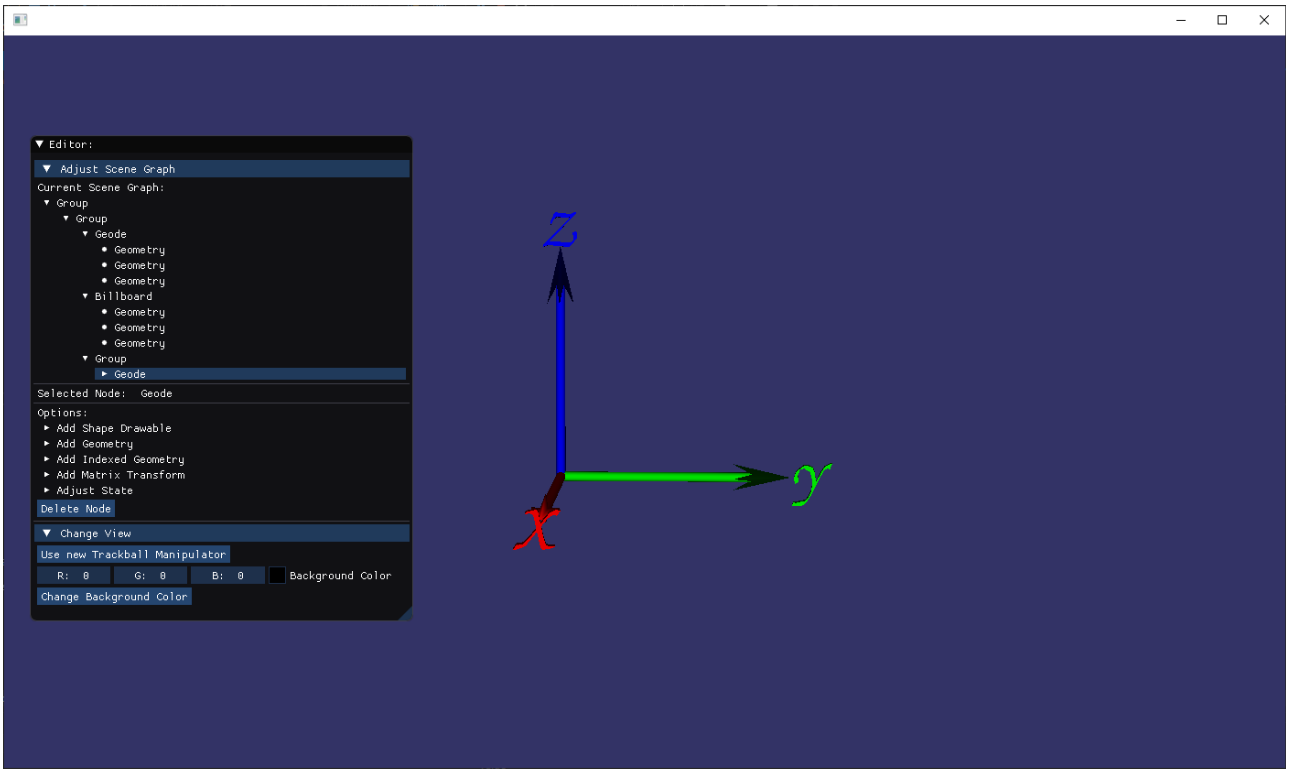
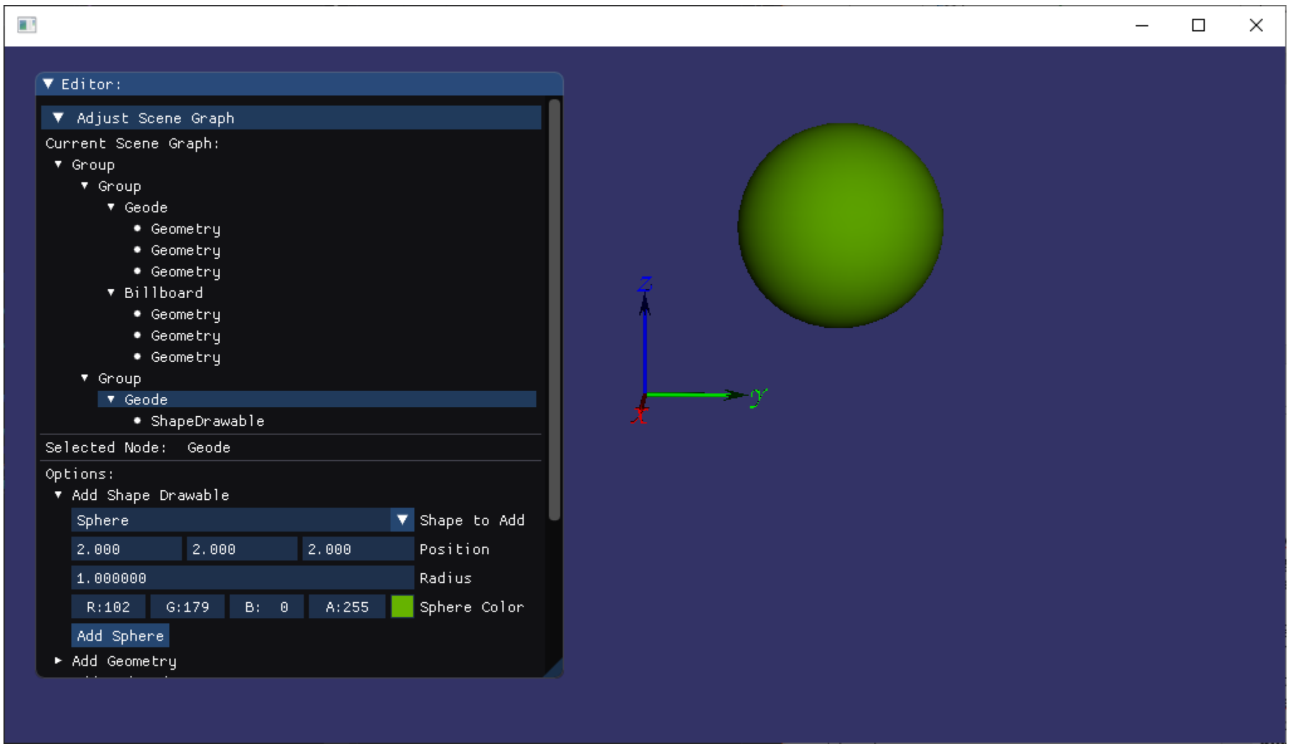


Figure 10: Group node selected



```
▼ Editor:
 ▼ Adjust Scene Graph
 Current Scene Graph:
  ▼ Group
    ▼ Group
      ▼ Geode
        • Geometry
        • Geometry
        • Geometry
      ▼ Billboard
        • Geometry
        • Geometry
        • Geometry
      ▼ Group
        ▸ Geode
 Selected Node:   Geode
 Options:
  ▸ Add Shape Drawable
  ▸ Add Geometry
  ▸ Add Indexed Geometry
  ▸ Add Matrix Transform
  ▸ Adjust State
 Delete Node
 ▼ Change View
 Use new Trackball Manipulator
 R:  0   G:  0   B:  0   ■ Background Color
 Change Background Color
```

Figure 11: Geode node selected



```
▼ Editor:
 ▼ Adjust Scene Graph
 Current Scene Graph:
  ▼ Group
    ▼ Group
      ▼ Geode
        • Geometry
        • Geometry
        • Geometry
      ▼ Billboard
        • Geometry
        • Geometry
        • Geometry
    ▼ Group
      ▼ Geode
        • ShapeDrawable
 Selected Node:   Geode
 Options:
  ▼ Add Shape Drawable
    Sphere                         ▼ Shape to Add
    2.000      2.000      2.000      Position
    1.000000                         Radius
    R:102   G:179   B:  0   A:255  🟩 Sphere Color
    Add Sphere
  ▸ Add Geometry
```

Figure 12: Sphere added

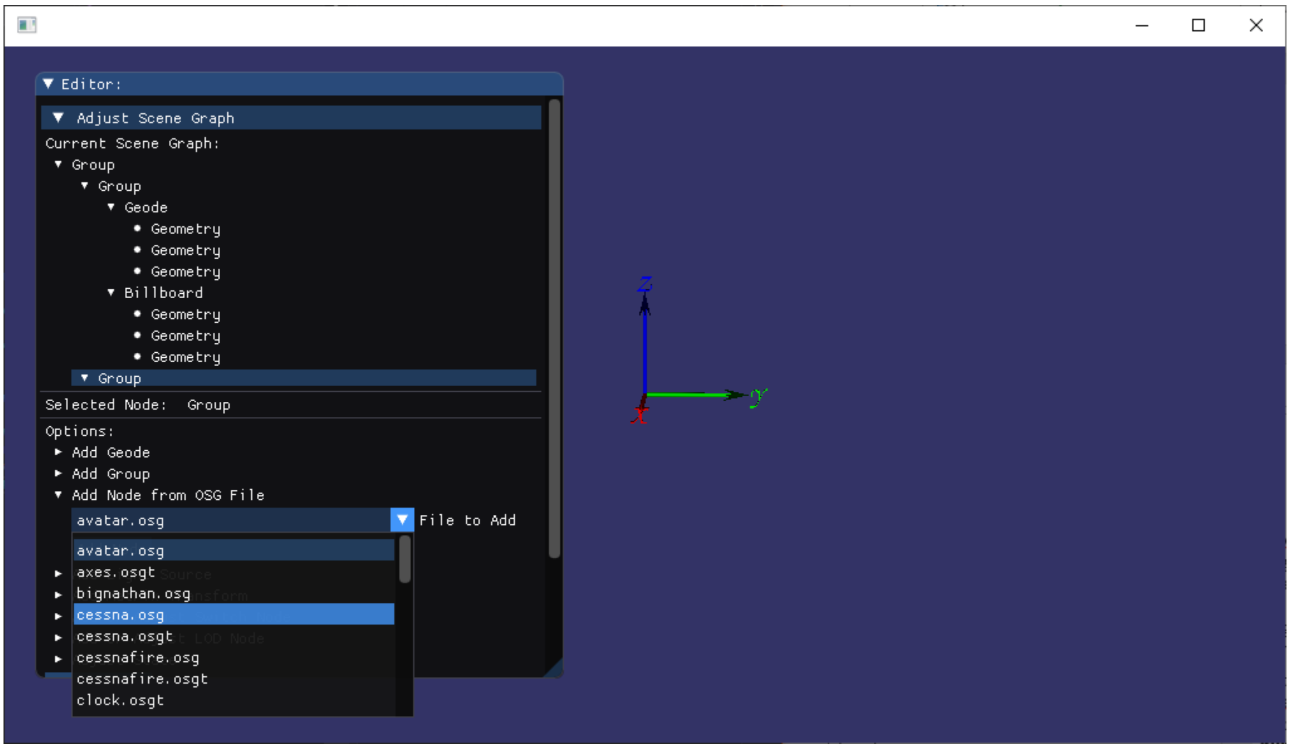


Figure 13: "Add Node from File" option opened

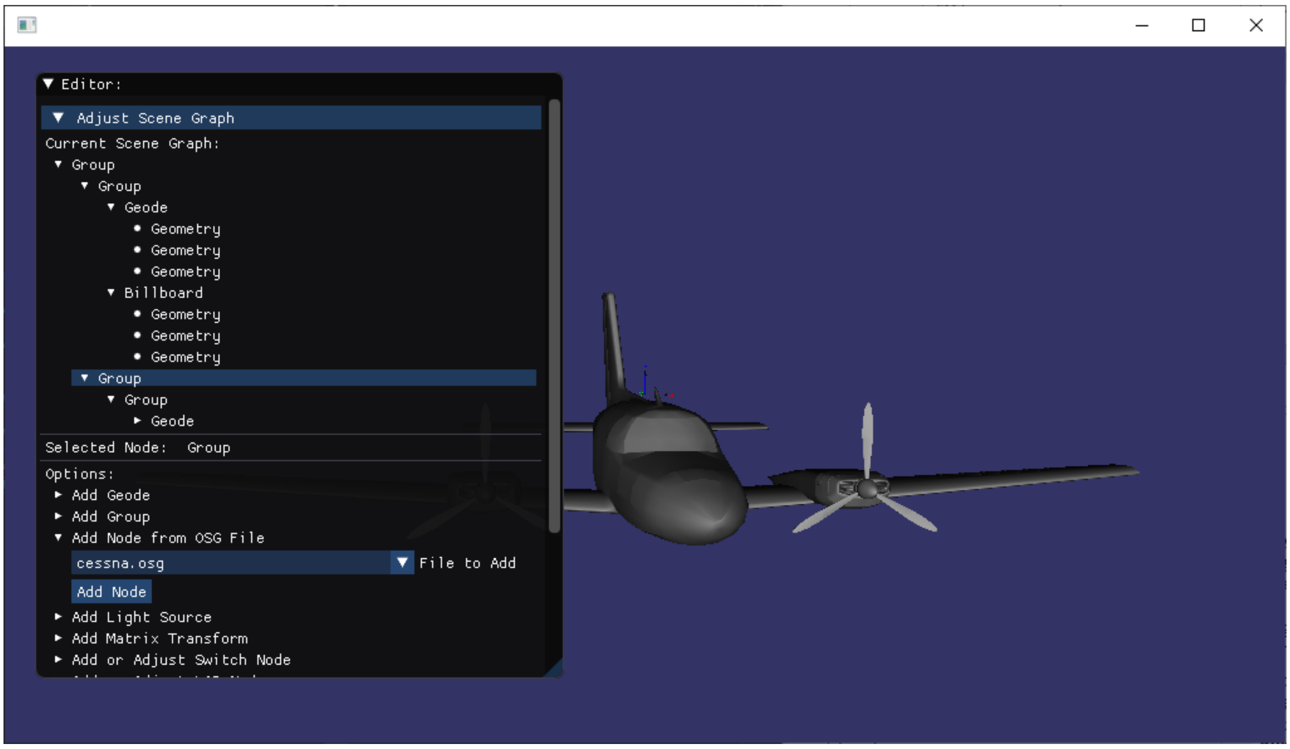


Figure 14: Nodes from cessna.osg file added to the scene graph

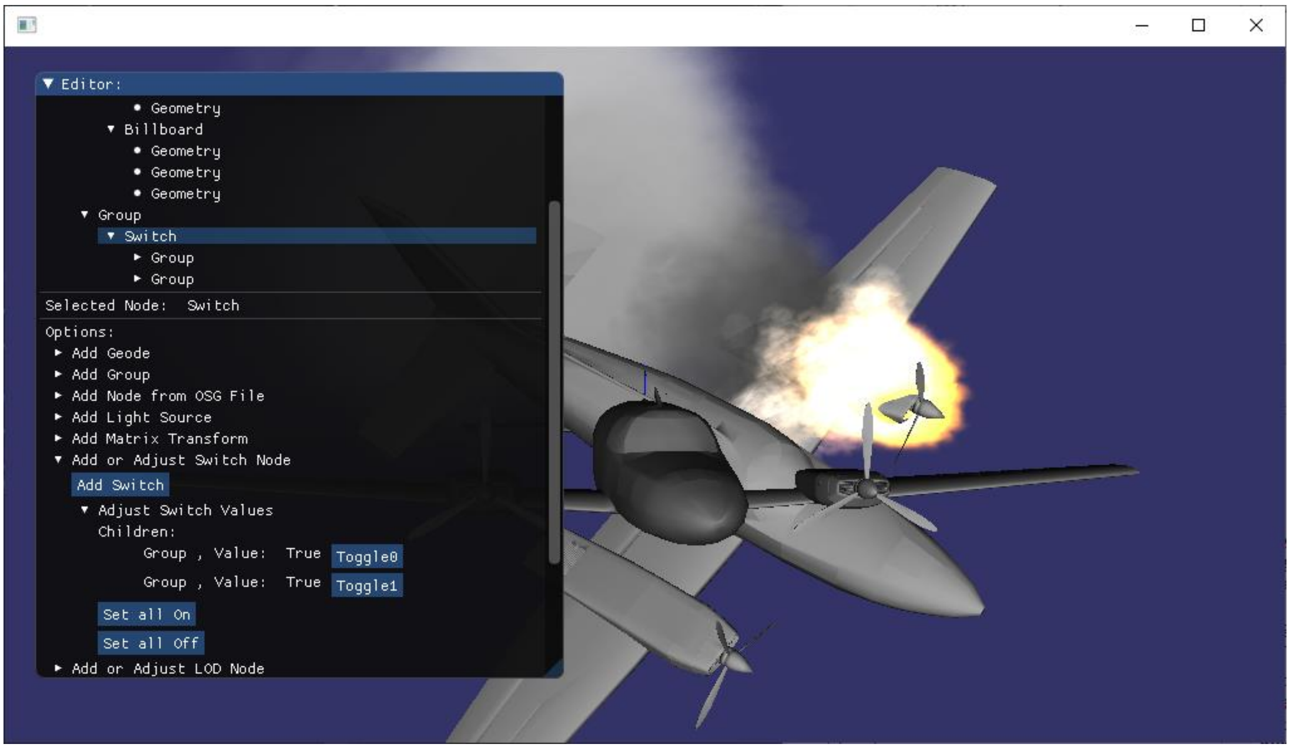


Figure 15: Switch node with nodes from cessna.osg and censsafire.osg as child nodes

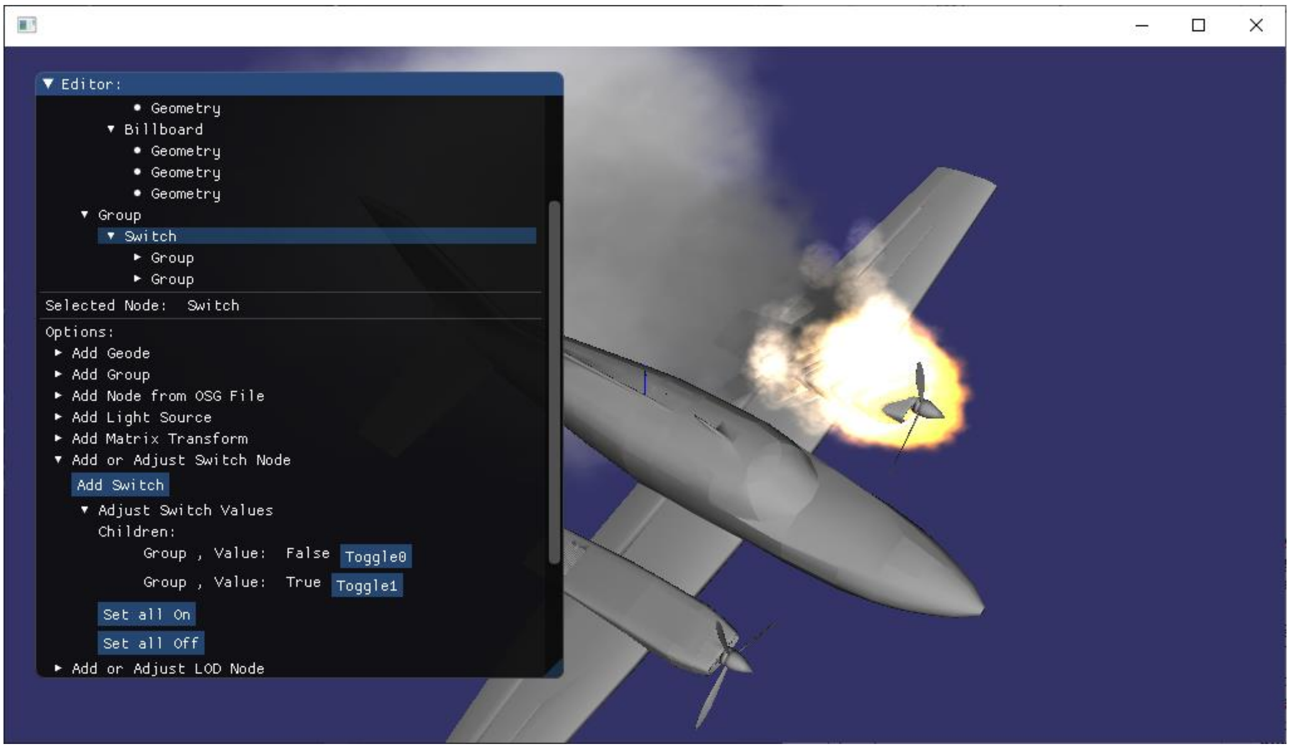
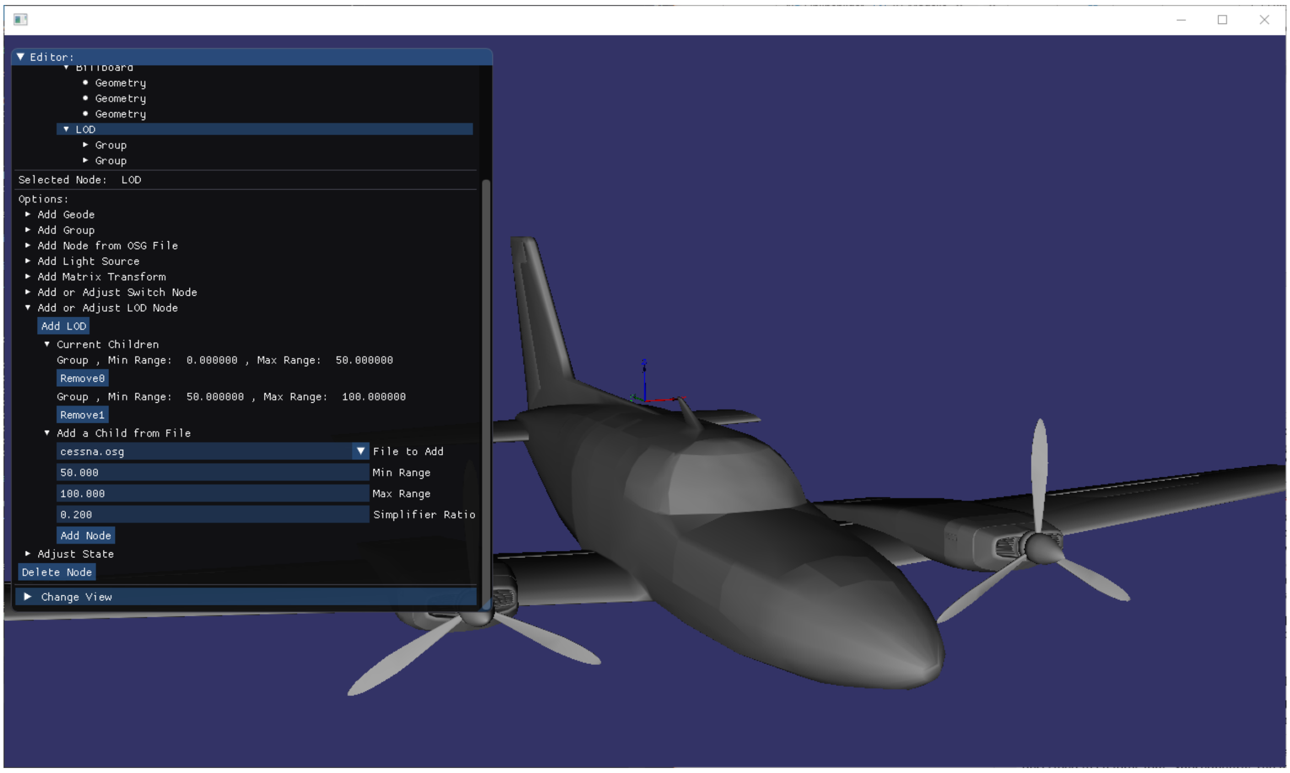


Figure 16: Switch node with nodes from cessna.osg and censsafire.osg as child nodes, while cessna.osg group is toggled to off



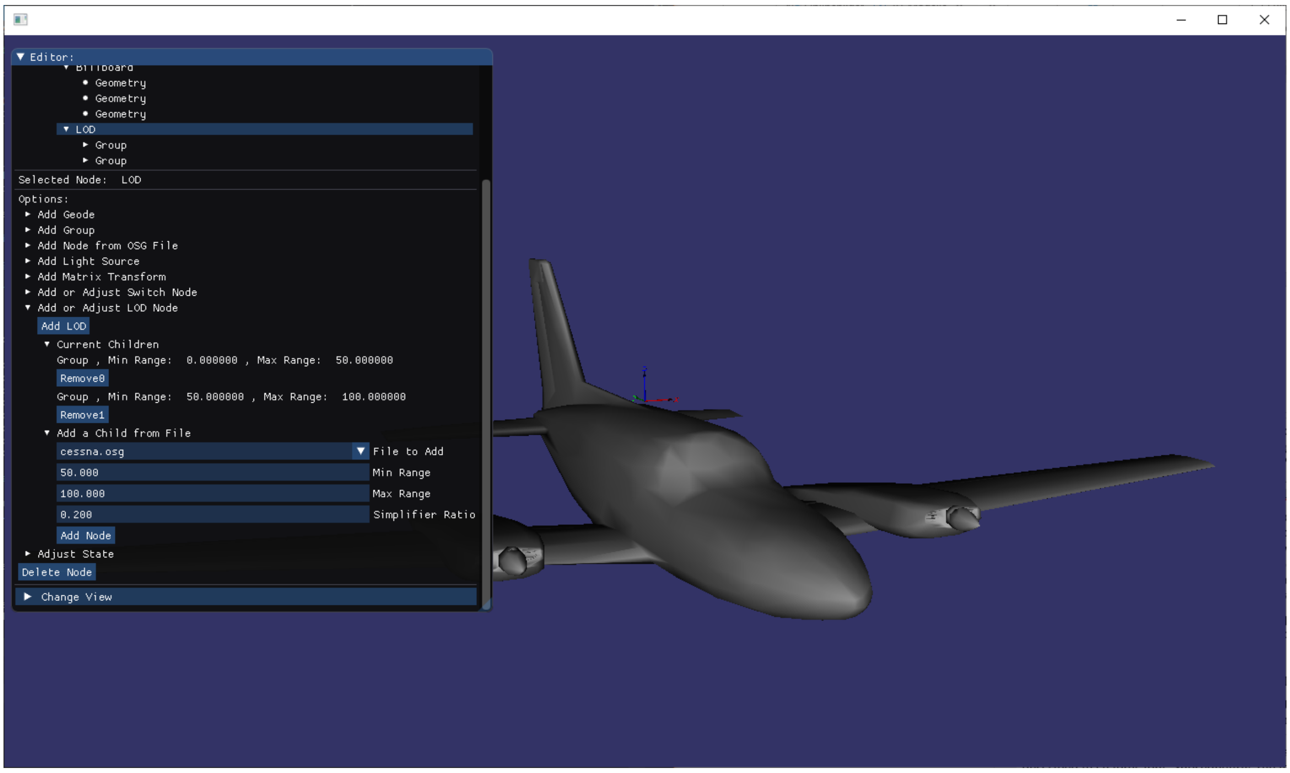
Selected Node: LOD

Group , Min Range: 0.000000 , Max Range: 50.000000

Group , Min Range: 50.000000 , Max Range: 100.000000

cessna.osg	File to Add
50.000	Min Range
100.000	Max Range
0.200	Simplifier Ratio

Figure 17: LOD node with nodes of two cessna.osg files as child nodes (first for range 0 to 50 and simplifier set to 1.0; second for range 50 to 100 with simplifier set to 0.2) - current distance to node between 0 and 50



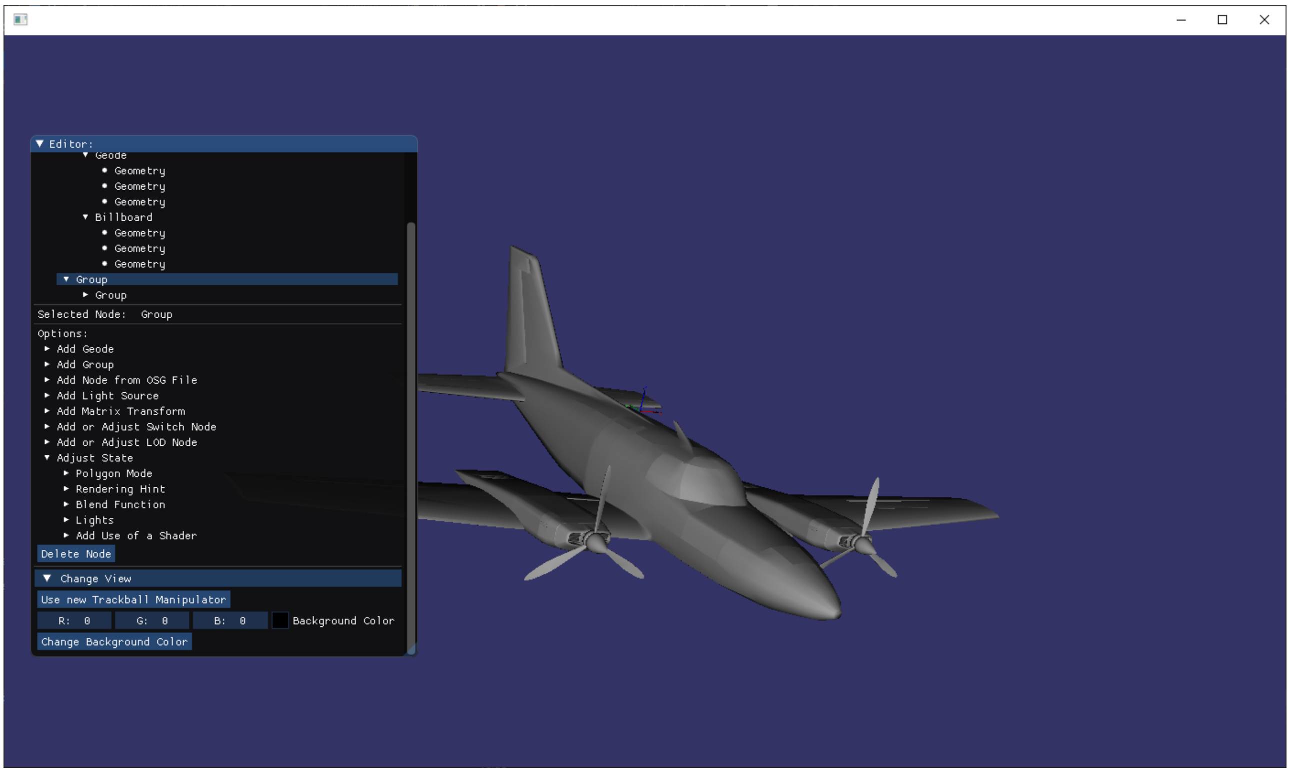
Selected Node: LOD

Group , Min Range: 0.000000 , Max Range: 50.000000

Group , Min Range: 50.000000 , Max Range: 100.000000

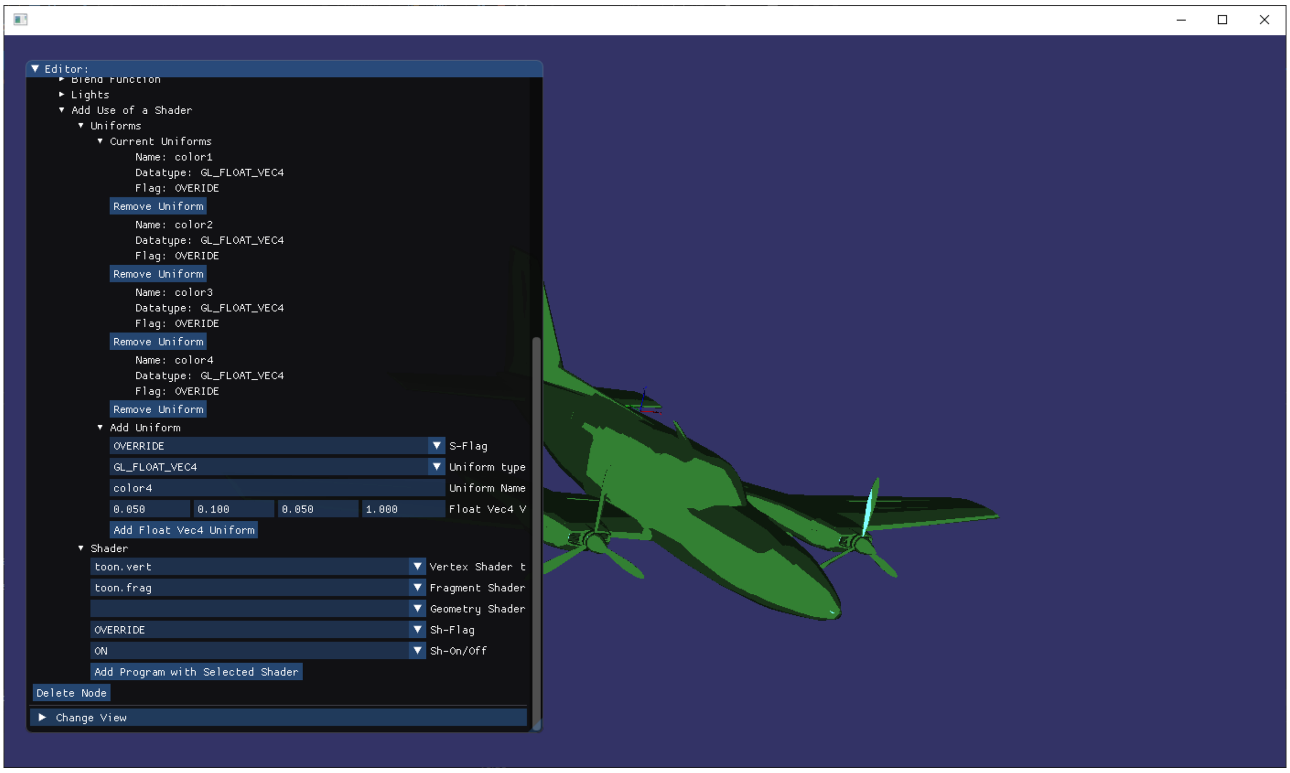
cessna.osg	File to Add
50.000	Min Range
100.000	Max Range
0.200	Simplifier Ratio

Figure 18: LOD node with nodes of two cessna.osg files as child nodes (first for range 0 to 50 and simplifier set to 1.0; second for range 50 to 100 with simplifier set to 0.2) - current distance to node between 0 and 50



```
▼ Editor:
      ▼ Geode
          • Geometry
          • Geometry
          • Geometry
      ▼ Billboard
          • Geometry
          • Geometry
          • Geometry
    ▼ Group
      ▸ Group
Selected Node:  Group
Options:
 ▸ Add Geode
 ▸ Add Group
 ▸ Add Node from OSG File
 ▸ Add Light Source
 ▸ Add Matrix Transform
 ▸ Add or Adjust Switch Node
 ▸ Add or Adjust LOD Node
 ▼ Adjust State
   ▸ Polygon Mode
   ▸ Rendering Hint
   ▸ Blend Function
   ▸ Lights
   ▸ Add Use of a Shader
Delete Node
▼ Change View
Use new Trackball Manipulator
R: 0    G: 0    B: 0    Background Color
Change Background Color
```

Figure 19: Group node from added cessna.osg file selected and "Adjust State" option opened



```
▼ Editor:
   ▸ Blend Function
   ▸ Lights
   ▼ Add Use of a Shader
     ▼ Uniforms
       ▼ Current Uniforms
            Name: color1
            Datatype: GL_FLOAT_VEC4
            Flag: OVERIDE
         Remove Uniform
            Name: color2
            Datatype: GL_FLOAT_VEC4
            Flag: OVERIDE
         Remove Uniform
            Name: color3
            Datatype: GL_FLOAT_VEC4
            Flag: OVERIDE
         Remove Uniform
            Name: color4
            Datatype: GL_FLOAT_VEC4
            Flag: OVERIDE
         Remove Uniform
       ▼ Add Uniform
         OVERRIDE                              S-Flag
         GL_FLOAT_VEC4                         Uniform type
         color4                                Uniform Name
         0.050   0.100   0.050   1.000         Float Vec4 V
         Add Float Vec4 Uniform
     ▼ Shader
       toon.vert                               Vertex Shader t
       toon.frag                               Fragment Shader
                                               Geometry Shader
       OVERRIDE                                Sh-Flag
       ON                                      Sh-On/Off
       Add Program with Selected Shader
Delete Node
▸ Change View
```

Figure 20: Added toon shader from [33] with uniforms displayed in the editor