Karlsruhe Institute of Technology
Computer Graphics Group

Graphics Programming Lab
Assignment 2

Modeling Tool

Abstract

We go on with the lab by extending your "Hello Cube!" application into a small modeling tool. You are asked to structure your code according to the model-view-controller concept; this allows you to add multiple views, cameras, and various representations of the scene to your tool easily. Eventually, the user will be able to create and design small scenes using your tool, which are internally stored within a scene graph you designed. A few additions of your own choice complete your modeler.

1 Introduction

First, we give a functional overview of the three core components you are asked to implement. We provide a concise outline of what we expect, however, you are given great freedom and flexibility during this assignment. The grading scheme at the end tells you how points will be distributed, and we are curious about the manifold solutions everybody will hand in. As always, feel free to contact us any time if you need assistance!

1.1 Model-View-Controller Architecture

Squeezing the whole application logic, input handling, and rendering into a single class might work out for very small applications like "Hello Cube!". However, this approach obviously does not scale. One encounters difficulties when trying to extend something, and much effort is needed to ensure that various states are always consistent.

Separating distinct parts of your application logic, particularly models and different representations of them, requires only a marginal amount of work when designing your code layout initially.

In this assignment, you are asked to stick to the *Model-View-Controller* (MVC) architecture. As the name suggests, this pattern separates *models*, which contain data and content; *views*, which can be arbitrary graphical or textual representation of these models; as well as *controllers*, which are responsible for handling and processing user input.

From Figure 2, you can gather how we interpret the MVC pattern in respect of a modelling tool. In the next subsection, we will detail the scene graph, which is going to be the most significant part of our model. Your modelling tool is supposed to feature two very distinct types of views: first, ordinary OpenGL renderings of the scene; second, a textual, hierarchical visualization in the form of an outliner. You already implemented mappings from user input to translations and rotations of a cube, clearly the task of the controller.

We attach great importance to the separation of models and views in your application. In contrast, we set no great store by a strict isolation of the controller, as it just comes in handy to process user input directly as events in OpenGL widgets or in the outliner. If you consider a solution more elegant and practical, stick to it. In fact, many other professional architectures distinguish mainly between models and views, for example, Microsoft's document/view architecture of their Foundation Class library.

1.2 Scene Graph

Image synthesis, that is, rendering, is the process of transforming given input data into a synthesized output image. This input data, the *scene*, consists of mathematically described objects. In the "Hello Cube!" application, your scene contained nothing more than a single cube. In this assignment, you will provide the possibility to add more objects to the scene. Therefore, you are asked to implement a very simple scene graph which is able to store multiple objects and transformations.

Figure 1 shows a basic structure. The entire scene is represented by the root node. Primitives, such as spheres, boxes, cylinder, cones, and tori are stored in leaf nodes. Transformations applied to these primitives are defined in interjacent nodes. In this assignment, you are asked to support *Rigid Body Transformations* (RBTs). These transformations hold both a translation vector and a rotation quaternion.

This basic scene graph features just a very flat hierarchy. You might want to implement grouping as one of the extensions we suggest in Sec. 6; that requires you to design your scene graph in a more flexible way.

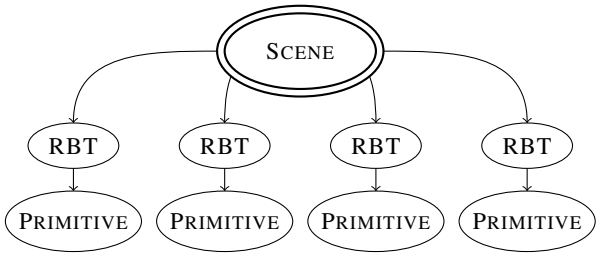


Figure 1: *A very simple scene graph.*

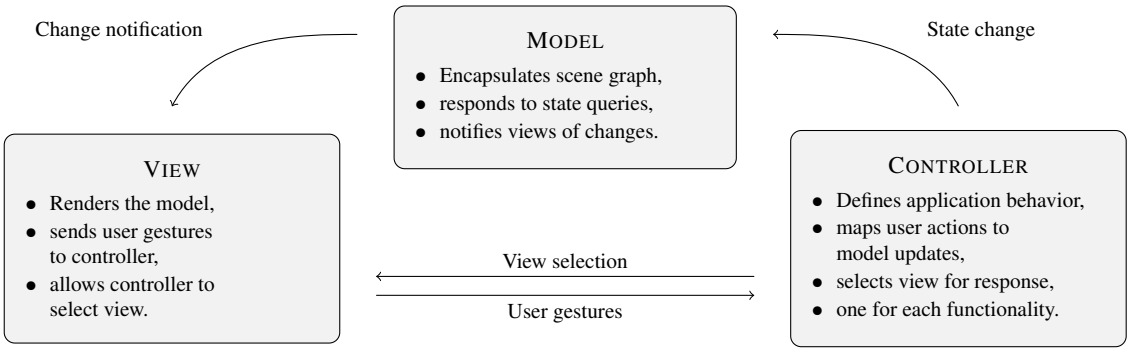


Figure 2: *Model-View-Controller pattern applied to your modelling tool. Associations can be established using method invocations or, more elegantly, Qt's signals and slots.*

1.3 Color Picking

The difficulty of *picking* is an inherent problem of raster graphics. For this, you need to answer the question, "which object was responsible that this particular pixel got this very particular color?". If your scene contains nothing more than a red sphere and a blue box in front of a black background, the answer is self-evident. But how to proceed if there are hundreds of red objects?

OpenGL provides a special render mode for this purpose, but we are not going to use it here. Consult [2] if you are interested in how it works, and be deeply grateful that we do not force you to use it.

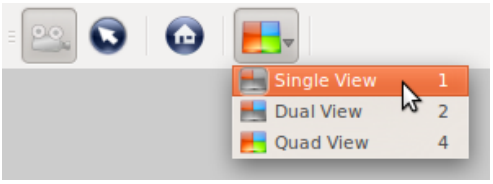
Performing ray casting for picking is another solution. It sounds like overkill at first, but is often done and works just fine. However, we stick to a simpler way.

You are asked to implement picking using an additional picking buffer. It features the same resolution as the regular color buffer that contains the rendering of the scene, and contains unique picking IDs of the corresponding objects rendered.

2 Multiple Views and Cameras

Start extending your "Hello Cube!" implementation by adding multiple *views* and *cameras*. Your program is to support single, double, and quad views as shown in Figure 3. An individual camera is needed for each viewport. Four cameras—perspective, front, left, and top—are assigned to the upper-left, upper-right, bottom-left, and bottom-right viewports, respectively. The double view shows only the upper two viewports. The single view narrows down to the perspective view.

Your application is supposed to have two different interaction modes: the *camera mode* and the *object manipulation mode*. To this end, add two actions to your tool bar to allow switching between these two modes; moreover, add a drop down menu to the tool bar to switch between different view modes:

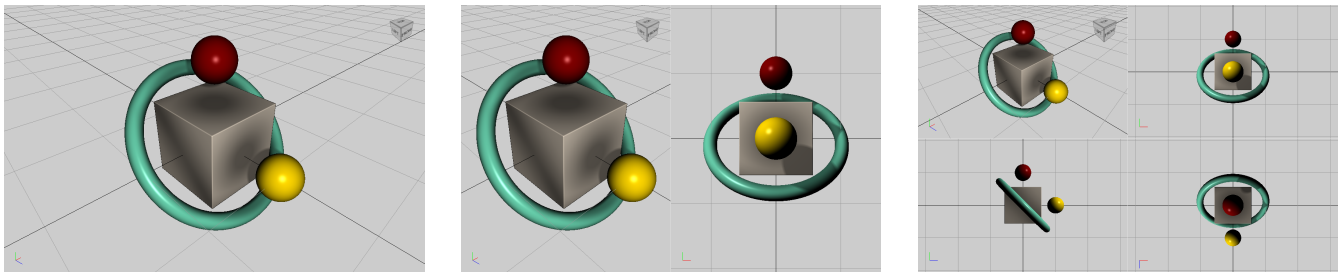


The camera and object manipulation modes are supposed to be mutually exclusive—remember `QActionGroup`? Add the menu for view mode selection to the main menu as well; it will appear as a sub menu there. In addition, set the shortcuts for these actions to `1`, `2`, and `4`, respectively.

In the camera mode, the user rotates the camera around its *point of interest* using the left mouse button. A camera always focuses its point of interest, which is initially at the origin. Using the mouse wheel, the camera zooms in and out, but keeps its point of interest. Using the right mouse button, the user translates both the camera's position and point of interest parallel to the image plane.

Unlike the perspective camera, all others should be orthographic, that is, set up an orthographic projection for them. Ask yourself why this makes sense for modelling purposes. Allow rotating the camera only for the perspective one, that is, lock rotations for the orthographic cameras. Store the perspective camera's rotation in a quaternion.

In the object manipulation mode, the user selects objects with a



(a) Single view, (b) dual view, (c) quad view.

Figure 3: *Several arrangements of viewports.*

single click on them, and deselects an object by clicking into the background. As in "Hello Cube!", an object is rotated by using the left mouse button, and translated by using the right mouse button. By using the mouse wheel, the camera still zooms in and out.

If the user holds `Ctrl` while clicking and moving the mouse, the application modes should be swapped. That is, right clicking on an object with the control key pressed translates the camera, not the object, while in object manipulation mode, and vice versa.

Start implementing your camera and view classes. Basically, you have two possibilities how to structure multiple views, and the choice is up to you:

- Your application features a single OpenGL widget that manages multiple views itself. During rendering, loop over all viewports and draw them into distinct regions using `glViewport()`.
- You instance an independent OpenGL widget dedicated for each of the individual viewports.

There is always one and only one *active* view. Your application is to initialize with a single viewport, which is active. If a user clicks into another viewport, set this viewport to active. Highlight the active viewport in a meaningful way, for example, draw a yellow outline around the viewport. If, for instance, the fourth viewport is active, and the user changes to a single view, change the active viewport back to the first one. Furthermore, if the user clicks on the "reset camera" button, the camera of the active viewport should be reset.

Provide a way to resize the viewports. If you chose to stick to a single OpenGL widget, you have to implement your own resize management. For instance, draw lines to separate different views, and allow the user to drag them around with the mouse. If you use multiple instances of the OpenGL widget class, lay them out using Qt's layout managers and use, for example, a `QSplitter` between every viewport for convenient resizing.

Now, you should be able to change into the quad view and see your cube from all sides. Check if everything above is implemented.

3 Adding Objects

... and deleting them

Implement a basic scene graph. For flat hierarchies, using a `QList` should be sufficient. Provide actions in the menu and tool bar to add spheres, boxes, cylinders, cones, and tori to the scene. Hint: check out GLU and GLUT primitives, `GLUquadric`, especially `gluSphere()`, `gluCylinder()` for cylinders and cones, and `glutSolidTorus()`. Obviously, you already have code for boxes. We don't mind what measurements the objects have, neither are we fixed on specific colors.

The above GLU and GLUT functions provide parameters to define the tessellation of new primitives. Set the tessellation according to the position of the tessellation slider at the moment of object creation. Come up with a meaningful mapping.

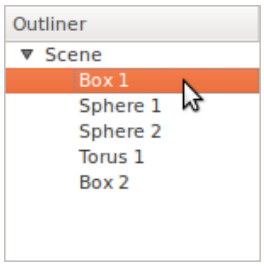
Add corresponding nodes to scene graph, as well as a dedicated RBT node for each primitive. Set the latest object added as *selected*. That is, the user can manipulate this object if the application is in the object manipulation mode.

Assign each primitive a unique ID for picking later on. Additionally, name each primitive in a straightforward way, for example, "Sphere 1", "Sphere 2", etc. for spheres. Display the name of the selected object in a field in the status bar!

Furthermore, provide a safe and consistent way to delete primitives and their corresponding RBT nodes. Provide the `delete` key as an additional shortcut.

4 Outliner

We further demonstrate the usefulness of the MVC concept by adding an *outliner* to the modelling tool. It should be a textual, hierarchical representation of the scene and could look like this:



First of all, consult the Qt reference documentation on model/view programming [1]. Next, derive from the `QAbstractItemModel` class and implement an item model for your scene graph. Finally, add a `QTreeView` to your application, which will be your outliner. Set its model to the item model you implemented. You can place your outliner anywhere you want inside your application. For instance, you could create a `QDockWidget` for this purpose. Have a look at Figure 1(a) from the "Hello Cube!" assignment sheet where these widgets can dock inside the `QMainWindow` (if they are not floating as top-level windows).

If the user clicks on an item in the outliner, the corresponding primitive should be selected. Don't forget to update the selected object's name in the status bar.

Moreover, provide the possibility to *rename* objects in the outliner! For this, you could design a context menu, allow the user to press `F2`, or start the renaming process after time-delayed double-clicks.

By now, you should have understood the power of the MVC concept. You feature totally different views next to each other, renderings in the main modelling widget and textual, hierarchical representations in tree views. Do not use the convenience class `QTreeWidget`! It is nasty to keep its default model consistent with your scene, and you miss the point of the MVC architecture!

5 Picking Objects

Before you proceed with this section, remove the wireframe, flat, and Gouraud shading modes from your application; use Phong shading exclusively and initialize everything related at start-up. Doubtlessly, you can remove the corresponding actions from the menu and tool bar. It is sufficient to implement color picking using shaders, and there is no need for the hassle of implementing it a second time in standard OpenGL with multiple rendering passes.

5.1 Framebuffer Object and Multiple Render Targets

Create a *FrameBuffer Object* (FBO) and attach *Multiple Render Targets* (MRTs) to it. Please consult the OpenGL Wiki [3] on this topic.

Have a look at Figure 4 to get an overview of how you should organize your MRTs within your FBO. Render the ordinary scene into a regular color buffer. Furthermore, render the unique picking IDs of the corresponding visible objects into another color buffer! Either pack the unique IDs into RGBA values or use a floating point texture. Do not forget to attach a depth buffer as well, because you definitely want to use depth testing in your modelling application!

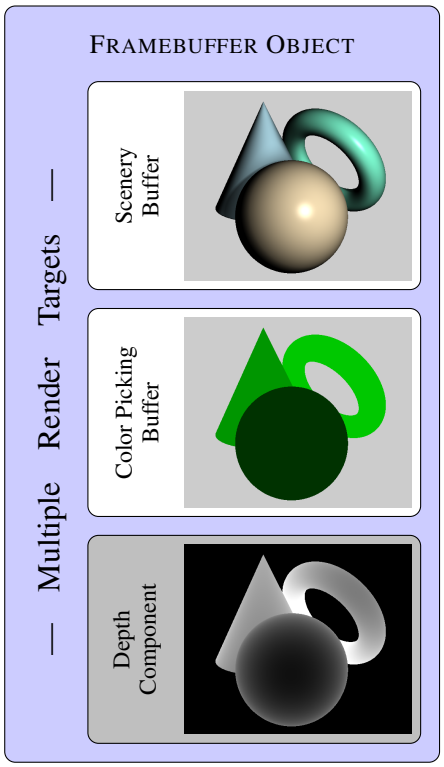
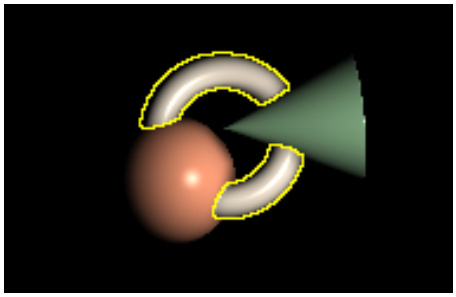


Figure 4: *An additional render target which stores unique picking IDs is used to implement color picking. A third component, the depth buffer, is required for depth testing.*

5.2 Selection Shader

Now, you need to display the scene inside the viewports. For this, create a new fragment shader and provide both color buffers of the FBO as input textures to this shader. Then, for each viewport, set up an orthographic projection and render a quad that spans the entire viewport. Set texture coordinates for this quad ranging from $(0, 0)$ to $(1, 1)$. Start by simply accessing the texture that contains the rendering of the scene at the position of the interpolated texture coordinates and color your fragments accordingly. Now, you should have an image of the rendered scene displayed on every viewport.

Another nice thing about having the picking buffer is that you can easily run edge detection on it. Upload the ID of the selected object to your newly created shader. Then, check if you are on a boundary of the corresponding object. If this is the case, color the current fragment yellow; else, pass on the color that you read out of the texture that contains the rendering of the scene:



This way, you can easily draw a nice looking outline around an object, and the user easily perceives which object is selected.

5.3 Picking in the Modelling Widget

With all previous tasks finished, it should be fairly easy to implement the actual color picking. If the user clicks on the main modelling widget, simply read out the value of the picking buffer at the mouse cursor position. Then, unpack it into the unique picking ID and select the corresponding object. Don't forget to render another frame so that the freshly selected object gets highlighted; update the object's name in the status bar as well!

If the user clicks on the background, deselect all objects and set the label in the status bar to `<None>`.

6 Sugar

Finally, you may implement *two different extensions* to your modelling tool of your own choice. Here are some suggestions; pick two from the list. They will increase in complexity as the list goes on. If you have another good idea what you would love to implement, feel free to contact us!

Configurable Grid. Your tool displays a *grid* as in Figure 3. You provide widgets to configure the grid size and step size (1 p) and a checkbox to show or hide the grid (1 p).

Saving & Loading. You can save (1 p) and load (1 p) scenes. Either dump them out in a binary stream (check out `QDataStream`) or in a user-readable fashion: either use XML; check out YAML [4], which is really cool by the way; or use your own format.

Scaling. You implemented transformation nodes for uniform and non-uniform scaling (1 p), which are applied after RBT nodes. Your tool scales objects conveniently by interpreting mouse events (1 p).

Grouping & Ungrouping. You implemented a group node (1 p). The user can group and ungroup objects by dragging and dropping objects into and out of groups using the outliner (1 p). Use *matrix stacks* for efficient transformations!

Lighting Suite. The user can add more lights to the scene, and they are stored in the scene graph as well (1 p). The positions and colors of the lights can be manipulated (1 p).

Materials & Texturing. You provide an interface to assign materials to objects (1 p), and they can also be textured (1 p).

Sophisticated Transformations. You implemented advanced transformations like bending, twisting, or waving objects (1 p). (Note that objects should be highly tessellated for good results.) The user can apply these transformations in a convenient way (1 p).

References

[1] Digia. Model/View Programming. `http://doc.qt.io/qt-5/model-view-programming.html`.

[2] OpenGL FAQ. Picking and Using Selection. `http://www.opengl.org/resources/faq/technical/selection.htm`.

[3] OpenGL wiki. Framebuffer Object. `http://www.opengl.org/wiki/Framebuffer_Object`.

[4] YAML. 1.2 Specification. `http://www.yaml.org/spec/1.2/spec.html`.

7 Grading

Note that your solution will *not be accepted* if its model—most notably the scene—is not clearly separated from views according to the MVC pattern!

You can achieve a total of 20 points in this assignment, based on the following criteria:

1 p	Your application features both a camera and an object manipulation mode. They are mutually exclusive. With `Ctrl` pressed, the modes are swapped during mouse event processing.
1 p	You implemented a camera class. In the camera mode, your application processes user input as expected. The perspective camera's rotation is stored in a quaternion.
1 p	Three orthographic cameras viewing the front, left, and top are implemented and work as expected. They cannot be rotated.
1 p	Your application features multiple views and its tool bar contains a drop down menu to choose between singe, double, and quad views. The state of the drop down menu is strongly consistent with the amount of viewports currently displayed.
1 p	The active view is highlighted. All viewports are resizeable. Clicking the "reset camera" button resets the camera of the active viewport.
1 p	You implemented a basic scene graph. It works absolutely reliably and does not produce huge memory leaks.
1 p	Spheres, Boxes, Cylinders, Cones, and Tori can be added to the scene using the menu and tool bar. Moreover, they can be deleted in a safe way.
1 p	During their creation, primitives are tessellated according to the state of the tessellation slider at that time.
1 p	You implemented rigid body transformation nodes for the scene graph and they work as expected.
1 p	You developed an abstract item model for the outliner.
1 p	You present the scene in a tree view using your abstract item model.
1 p	You implemented a framebuffer object with multiple render targets. The picking buffer stores unique picking IDs.
1 p	Selection of objects works both in the OpenGL rendering views and in the outliner. There is a strong consistency spanning all views which object is highlighted. You display the selected object's name in the status bar.
1 p	Objects can be renamed using the outliner. Changes are immediately reflected in the status bar.
2 p	The selected object is highlighted in the OpenGL rendering views using an edge detection shader.
4 p	You implemented two different proposals from the "Sugar" section. Each realization gives you 2 points max. You cannot score more than 4 points here, that is, you cannot compensate for points lost in the previous main tasks!

You get a *malus* if your code is not nicely written using meaningful variable names in proper English, or significantly lacks proper comments! Moreover, you get a malus if your solution produces errors or way too many warnings during compilation! Lastly, you lose points if your program crashes or freezes for weird reasons.

If we got the impression that you handed in copy-pasted code, and you are unable to explain your solution, we will expel you from the course!