OpenGL 1

February 7, 2018

Computer Graphics

Transformations and User Input

In this assignment you will implement Model, View and Projection transformations in a way that allows the user to control scaling and rotation. The assignment uses an updated framework, so download the new code from Nestor. The difference between the code of last week is that this framework has Qt Widgets which you will use to control the behaviour of the program. The used widgets are explained in section 3: Scaling and rotations. Additionally, a class which can read `.obj` files is included in order to work with meshes. The framework should compile out of the box and show a blue tinted window. Please take some time to familiarize yourself with the changes.

1 Shape drawing (1 point)

The first task of this assignment is drawing a pyramid and a cube. You will use the techniques of last week's assignment to define the shapes, create the buffers, and make the draw commands.

Define shapes

Last week you defined the vertices for a triangle in 2D. Each vertex contained an x and y coordinate and an rgb color. A third coordinate is required in order to draw in 3D, so copy and adapt the *vertex* struct (or class) of last week to include a z coordinate. The struct should contain six floats, three for the vertex coordinates and three for its colour.

Use the `initializeGL` function to create the objects:

1. Make an array of vertices specifying a unit cube in 3D. The cube should be centred on the origin of the 3D space and each vertex should be on 1 or -1 in each dimension. The draw call that you will use later expects the array of vertices to be a sequence of triangles, so you will have to repeat some vertices.

2. Create a VBO and VAO for the cube.

3. Fill the VBO with the vertex data. The vertex array created in step 1 does not have to remain in memory after this step.

4. Specify how the data layout with `glVertexAttribPointer` and `glEnableVertexAttribArray`.

5. Do not forget to clean up the VBO, VAO and any other allocations in the destructor.

6. Repeat these steps for a pyramid.

Drawing

Drawing the cube and pyramid is very similar to drawing last week's triangle. Use `glDrawArrays` with the `GL_TRIANGLES` mode. However, since you are using two shapes you have to tell OpenGL which one you want to use for the draw command. This is where the *vertex array object* (VAO) comes in, binding the cube's VAO tells OpenGL which buffers to use and how the data is specified in those buffers. So bind one of the VAOs and make a draw call, then bind the other VAO and make another draw call. Figure 1.1 shows what the application should look like at this stage. (You may use different colors of course.)

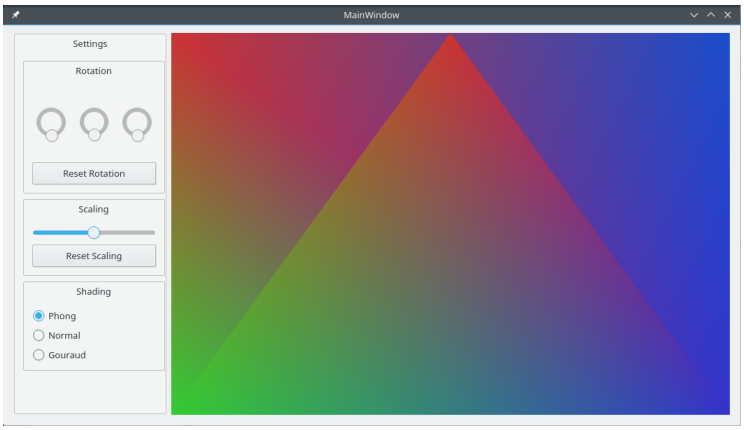

Figure 1.1: *The pyramid and cube without transformations.*

Suggestions for the competition

As you might have noticed the vertex data arrays were quite a bit larger than the number of vertices of the object that was specified. A cube consists of 8 vertices, but the array had to have 36 vertices. This redundancy becomes more difficult to deal with the larger the model. Indexed drawing is an alternative way to represent the to be rendered vertices which prevents this problem. Instead of specifying triangles directly, each vertex is only given once. The triangles are created by specifying the indices of the vertices that have to be combined. See the documentation of `glDrawElements` for more information.

Another cool technique is procedural shape generation. Instead of specifying the vertex coordinates by hand use the parametric equation of a shape to generate them. For instance using the parametric equation of a sphere.

2 Transformations (3 points)

The Model, View, and Projection transformations are used to specify how the vertices of a model have to be placed in the world, how the camera is positioned in the world (or how the world is positioned with respect to the camera), and how the world is projected through the camera. In this task you will implement these transformations such that the pyramid and cube are shown side by side, instead of filling the screen entirely.

Transformation matrices

Qt has many classes that you can use for the math related to 3D transformations. So, make sure you take a look at what these classes can already do for you when working on this task. For instance: QMatrix4x4 and QVector3D.

As explained in the lectures, affine transformations in 3D can be implemented using matrix multiplication in 4D. Therefore, we will use 4×4 matrices to represent the Model, View, and Projection matrices. In this assignment you will leave the camera at its default location: the origin looking along the negative z-axis.

1. Create `QMatrix4x4` data-members in `MainView` representing the Model transformation for the cube and the pyramid.
2. Set their value such that the cube is translated to $(2, 0, -6)$ and the pyramid to $(-2, 0, -6)$. Make sure you use suitable near and far plane values.
3. Create a `QMatrix4x4` data-member for the Projection transformation and set its value for a perspective projection with a field of view of 60 degrees.
4. The user of the application may change the size of the window, which might change the aspect ratio of the surface we are rendering to. Make sure you update the perspective projection each time this happens (see the `resizeGL` function). If you do not do this squares are not shown as squares, as in Figure 1.1.

Apply transformation

It is the task of the vertex shader to compute the location of each vertex. So the values of the transformation matrices have to be passed along to the GPU. A way to do this is through uniforms. A uniform is a (global) variable in a shader, while its value is specified on the CPU. It is called a uniform because the value of the variable does not change between shader invocations like the shader stage inputs and outputs. For example, the projection matrix is the same for each vertex, but the vertex's coordinates and colour may change.

1. Change the vertex shader to accept a uniform 4×4 matrix for the model transformation one for the projection transformation (See the comments in the vertex shader).
2. Extract the locations of the uniforms (a GLint) in `createShaderProgram` using the `uniformLocation` member function of the `QOpenGLShaderProgram` after linking.
3. Use the locations found in the previous step to set the value for each uniform in `paintGL`. For this you can use the `glUniformMatrix4fv` function after the shader program has been bound. The `4fv` suffix specifies that a 4×4 matrix of floats is used. Note that the Model transformation should be updated before each draw command, as the cube and pyramid use a different transformation.
4. Finally, change the `main` function of the vertex shader such that it applies the transformations.

After these steps the application should look something like Figure 2.1.

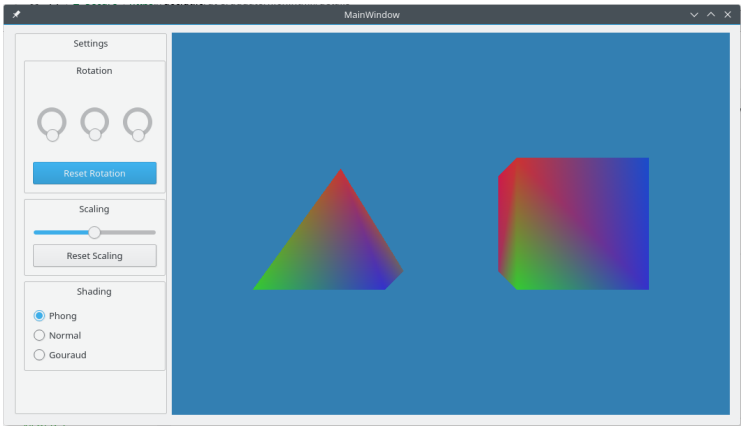

Figure 2.1: *The pyramid and cube with transformations.*

3 Scaling and Rotation (2 points)

In this task you will use the widgets of the application to control the scaling and rotation of the cube and pyramid.

User Input

The framework supports two kinds of user input. The first are keyboard and mouse events. Qt will generate an event every time the user does something with the mouse or keyboard in our application. The functions that will be called in these events are located in `user_input.cpp`. These functions are members of `MainView` so you can use everyting of that class in their implementation. So far, they only print a message indicating that they have been called. See if you can activate all of the callback functions by running the application and interacting with it!

The second kind of user input comes from the Qt Widgets. The application has widgets for scaling, rotation and shading. The widgets for shading will be used in the next OpenGL assignment, so you can ignore them for now. The GUI of the framework was designed in Qt Creator. Double-click on `mainwindow.ui` (under 'Forms') to enter the design view. On the left is a list of widgets that can be used. In the middle is a preview of what the application will look like. And on the right is an hierarchy of objects with their properties. The designer allows you to build a GUI by dragging and dropping the widgets that you want to use and changing their properties. (You can exit the design view by clicking on the 'edit' button in the left toolbar)

How the GUI-forms created in the designer are integrated into the `MainWidow` class is specific to Qt and a lot of this process happens behind the scenes. Therefore, I will explain the process using the widgets that are already in use. Start by selecting one of the rotation dials in the preview. Notice how one of the `QDials` is highlighted in the hierarchy view. The name of the `QDial` is important. Navigate back to `MainWindow.h` and look at the section `private slots:`. Every time a user interacts with a widget, the widget creates a `Signal` [†]. `Signals` can be connected to `Slots`, such that every time a widget generates a `Signal` the connected `Slot` will be called. Qt automatically connects `Slots` to a `Signal` when the name of the slot matches with `on_[Widget Name]_[Signal name]` and the arguments of the `Slot` match with those of the `Signal`. So in `MainWindow.h`

[†]See the documentation of a widget to see which interactions cause a `Signal` and what information it will provide.

the `on_RotationDialX_sliderMoved` function is called every time the user interacts with RotationDialX in a way that changes its value. The new value is given as an argument.

Currently, the framework takes care of all the interaction with widgets that is required. You will **not** have to touch the code in `MainWindow` in this assignment. Instead, change the implementation of `setRotation` and `setScale` in `MainView.cpp` to solve the tasks in this assignment. These setter functions are called by the `Slots` of the `MainWindow` class, so you can use the widgets without having to worry about how they work.

In summary, every time the user changes the rotation dials `setRotation` is called with the new values and every time the user changes the scale `setScale` is called with the new value.

Implement scaling and rotation

In this task you will use the widgets to control the rotation and scale of the objects. The cube and pyramid should rotate around their own origin, i.e. they should not move to another place. The uniforms for the transformation matrices are updated before each draw-call, so you only have to change the value of the matrices in `setRotation` and `setScale` and call `update` to render a new frame.

Try to find the minimum and maximum value of each widget before you change the implementation. Use this information so you can scale the input in a way that gives the user reasonable control over the rotation and scaling. The application should look something like Figure 3.1.

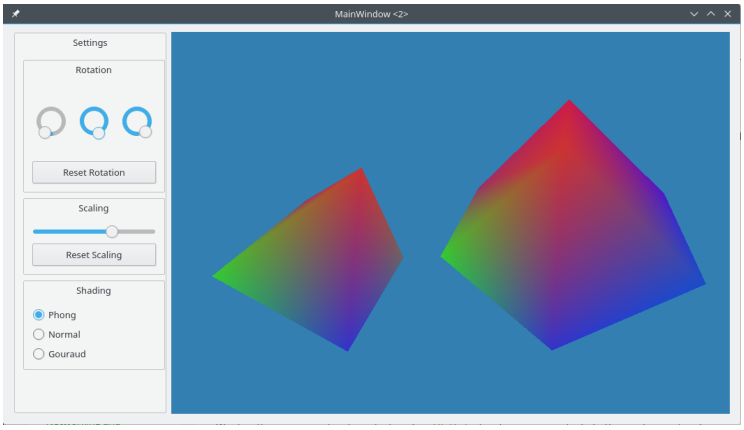

Figure 3.1: *The pyramid and cube with rotation and scaling.*

Suggestions for the competition

The framework contains all the event-listeners needed to create FPS-like interaction, where the user looks with the mouse and moves the camera with the 'wsda' keys. If you want to implement this make sure that the user can 'escape' the mouse-events, so s/he can close the application or use the widgets. You will also need to add a View transformation matrix, as moving the camera changes the View transformation.

4 Object loading (2 points)

In this task you will add a third object to the application using the provided `Model` class to load a mesh.

Load a Mesh

1. Load the sphere object file that is already in the project by constructing a `Model` object. The constructor takes as argument the filename of the object to load. Specify the path using the Qt resources file, just as with the shader-files.
2. Use `Model::getVertices` to obtain a `QVector<QVector3D>` containing all the vertex coordinates in the mesh in order. Construct an array of `Vertex` objects from these coordinates and specify a random color for each vertex. [†]
3. Create, fill, and properly clean a VAO and VBO for the mesh. Do not forget to specify the vertex attributes.
4. Add a draw call for the mesh in paintGL.
5. Add a Model transformation matrix for the mesh and translate it to $(0, 0, -10)$. The sphere should also rotate in its place, just like the pyramid and cube.
6. Unlike the cube and pyramid, the Sphere's coordinates exceed $[-1, 1]$, therefore you have to change the scaling in order to show it with the same size. Using a factor of 0.04 on the scaling value returned by the widget should resolve the problem.

The application should now look something like Figure 4.1.

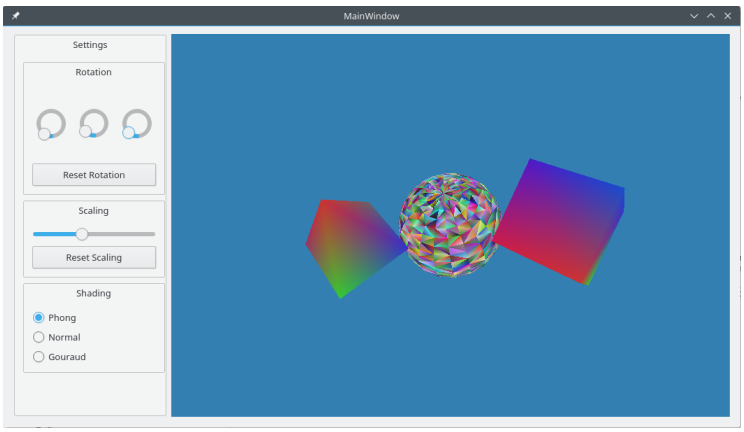

Figure 4.1: *The pyramid, cube and sphere with scaling and rotation.*

Suggestions for the competition

Loading models that use a different scaling makes it difficult to manage rendering, as you experienced with the sphere model. Therefore, meshes are typically unitized before their data is used. Unitizing a mesh linearly scales the vertex coordinates such that the model (exactly) fits in a unit cube. The `Model` class contains everything you need in order to implement unitizing. Only the implementation was not provided.

[†] http://en.cppreference.com/w/cpp/numeric/random/rand

Deadline

See Nestor (*Time Schedule*). Details on how to submit your work can also be found on Nestor (*Lab Assignments*).

Assignment submission

Please use the following format:

- Main directory name: `Lastname1_Lastname2_OpenGL_1`, with the last names in alphabetical order, containing the following:
- Sub-directory named Code, containing the modified Qt framework (please do **NOT** include executables)
- Sub-directory named Screenshots with your screenshots or videos.
- README, a plain text file with a short description of the modifications/changes to the framework along with user instructions. We should not have to read your code to figure out how your application works!

The main directory and its contents should be compressed (resulting in a **zip** or **tar.gz** archive) which is the file that should be submitted (using the *Assignment Dropbox*). An example of a file to be submitted associated with the first OpenGL assignment would be: `Kliffen_Talle_OpenGL_1.tar.gz`

Assessment

See Nestor (*Assessment & Rules* → Assignment assessment form).