OpenGL Assignment 1: Transformations and Meshes

January 31, 2022

Computer Graphics

1 Theoretical question (1 point)

This theoretical question can be submitted with a PDF file. We recommend using LaTeX, but we allow other submissions (including handwritten ones) as well.

This question is parametrised by the digits of one of your group member's student number. If your student number is $s1234567$, then $a = 1$, $b = 2$, $c = 3$, $d = 4$, $e = 5$, $f = 6$, $g = 7$.

In this exercise we will apply the Model and View transformations to a vertex of a mesh, essentially doing the work of a *vertex shader*. We assume that we are working in two dimensions and we are given a vertex with coordinates $v = (f, g + 2)$ in object space. The Model transformation takes the mesh from object space to world space. Then the View transformation places the mesh into the coordinate system of the camera.

1. First we rotate the mesh 90 degrees counterclockwise around the origin. Give the rotation matrix R in homogeneous coordinates.
2. Next we translate the mesh by $(d, e + 2)$. Give the translation matrix T in homogeneous coordinates. This concludes the Model transformation.
3. As seen from the origin of world space, the origin in camera space is positioned at coordinates (b, c). Give the transformation matrix V in homogeneous coordinates that transforms a point expressed in world space to a point in camera space.
4. Compute the final output of the vertex shader, i.e. map v from object to camera coordinates.
5. Explain why in this case it is not necessary to apply a projection matrix.

2 OpenGL implementation

In this assignment you will implement Model, View (optional) 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. One difference is that in this framework, *face culling* is enabled. What this entails is explained in Section 3.
Another difference is that this framework has Qt Widgets which you will use to control

the behavior of the program. The used widgets are explained in Section 6. Additionally, a `Model` 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.

3 Face culling

In OpenGL, a triangle has two sides or *faces*: front and back. The triangle may be rendered differently depending on which side is facing the camera. One possible application of this is (back-)face culling. Here, only triangles of which the front is facing the camera are rendered, and triangles of which the back is facing the camera are ignored.
Which side of the triangle is considered to be the front depends on its orientation or its *winding order*. The winding order is determined by the order of its vertices as seen from the camera, see Figure 3.1. If the vertices, going from A to B to C, spell out a counter-clockwise path, then the triangle's front faces the camera and it is rendered. If the vertices are in a clockwise orientation, then its back faces the camera and the triangle is ignored or *culled*. Note how the rotation of the triangle around the x-axis or y-axis can change the winding from counter-clockwise to clockwise and vice versa.

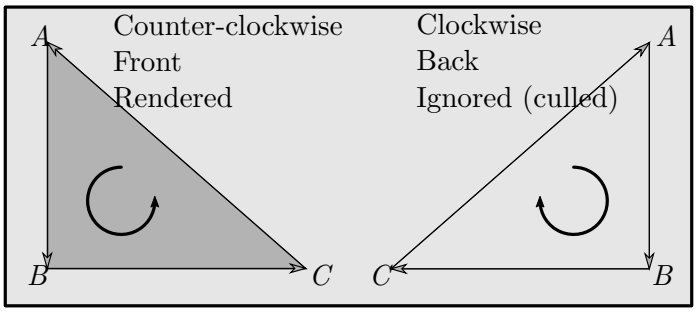


Figure 3.1: *Triangle winding order*

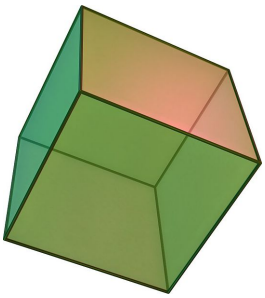
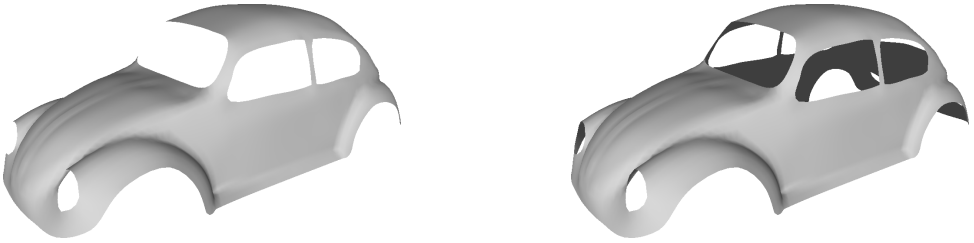


Figure 3.2: *A transparent cube* [†]

Here, we give an example where face culling is useful.
In this assignment, you will be defining, among other shapes, a cube with opaque sides. Every side consists of two triangles. In Section 6 you will implement rotation for this cube. Looking at the transparent cube in Figure 3.2, we can see that if it were opaque, only (a maximum of) three sides can be seen at a time. The other sides are occluded by the visible sides. Moreover, note that if we define — for every triangle — its front to be on the outside of the cube, then only the visible sides are rendered, and the occluded sides are ignored. By ignoring triangles that are not visible the rendering process is sped up.

Note how face culling is only useful if the geometry is closed and opaque, and if the camera is never inside of it. Figure 3.3 shows the effect of face culling on a model of a car. When defining the shapes in Section 4, take the winding order of each triangle into account. Face culling has been enabled in the framework in the `initializeGL` function by the use of the `glEnable(GL_CULL_FACE)` function call. For debugging, it can be useful to temporarily disable culling by commenting out this line. However, **make sure that back-face culling is enabled in your assignment submission**.

[†] https://en.wikipedia.org/wiki/Cube



(a) *Face culling enabled* ***(b)*** *Face culling disabled*

Figure 3.3: *Face culling example*

4 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 of last week to include a z-coordinate. The struct should contain six floats, three for the vertex coordinates and three for its color.
Use the `initializeGL` function to create the objects:

1. Create 8 instances of the Vertex object, defining a cube in 3D. The cube should be centered 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 an array of vertices as a sequence of triangles. Therefore, use the previously defined Vertex objects to create a Vertex array in which the cube is represented as a sequence of triangles. Every (unique) Vertex is part of multiple triangles and should thus appear multiple times in the array.

2. Create a VAO and VBO 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 is laid out 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 square pyramid, i.e. a pyramid with a square base, 4 triangles as sides and its apex at the coordinates $(0, 1, 0)$.

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 4.1 shows what the application should look like at this stage. (You may use different colors, of course.)

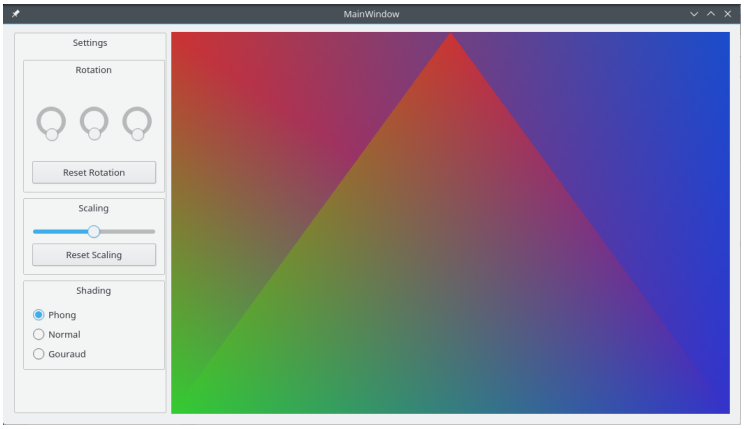


Figure 4.1: *The pyramid and cube without transformations.*

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 mesh 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.

5 Transformations (3 points)

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 Model and Projection 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 we will leave the camera at its default location: the origin looking along the negative z-axis. This means that we only need a model

transformation matrix and a projection transformation matrix. View transformation can be implemented as a feature for the competition (see "Suggestions for the competition" in Section 3).

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 by $(2, 0, -6)$ and the pyramid by $(-2, 0, -6)$.

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. Make sure you use suitable near and far plane values.

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 4.1.

Apply transformation

It is the task of the vertex shader to transform the location of each vertex from model space to screen space. 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 the execution of a shader program, and its value is specified after the shader program is bound, and before the render call where it is used. It is called a uniform because the value of the variable is constant for all shader invocations for one render call. In other words, the value of the uniform will be the same for all invocations of the vertex shader and all invocations of the fragment shader. Contrast this to the shader stage inputs and outputs, which may — and generally do — change between invocations. For example, the projection matrix uniform is the same for each vertex, but the individual vertex coordinates and color may change.

1. Change the vertex shader to accept two uniform 4×4 matrices: one for the model transformation and one for the projection transformation (see the comment in the vertex shader for an example).

2. Extract the locations of the uniforms (a GLint) in `createShaderProgram` using the `uniformLocation` member function of the `QOpenGLShaderProgram` after linking. You may also want to check the return value to see if the function call was successful.

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 5.1.

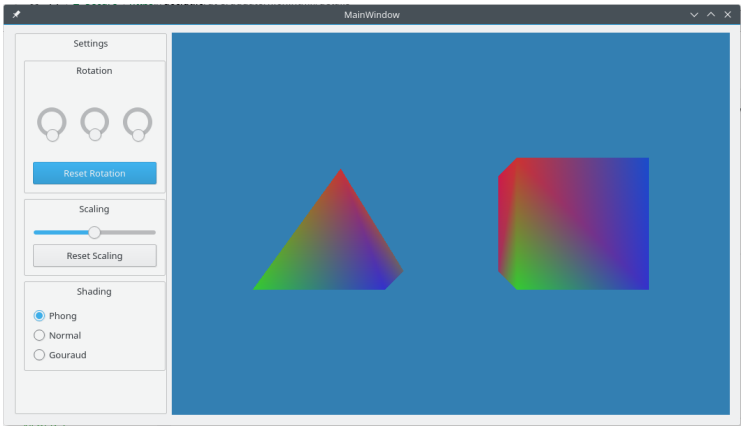


Figure 5.1: *The pyramid and cube with transformations.*

6 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 on 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, we 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` to be emitted 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 needing to understand the implementation details.
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 6.1.

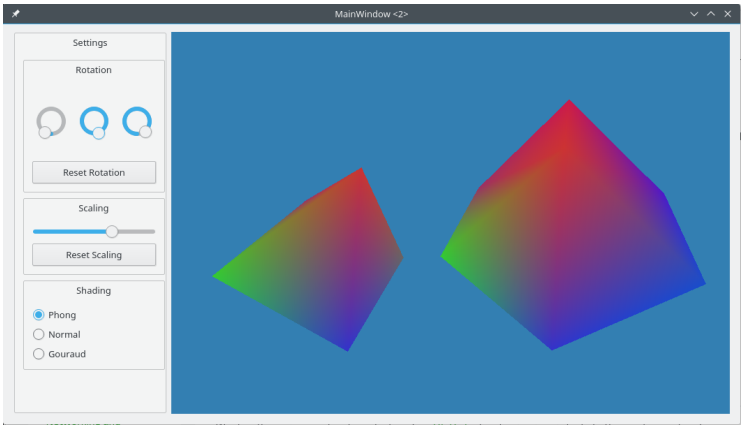


Figure 6.1: *The pyramid and cube with rotation and scaling.*

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.

7 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 file name of the object to load. Specify the path using the Qt resources file, just as with the shader source 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. Generate, fill, and destroy 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 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 7.1.

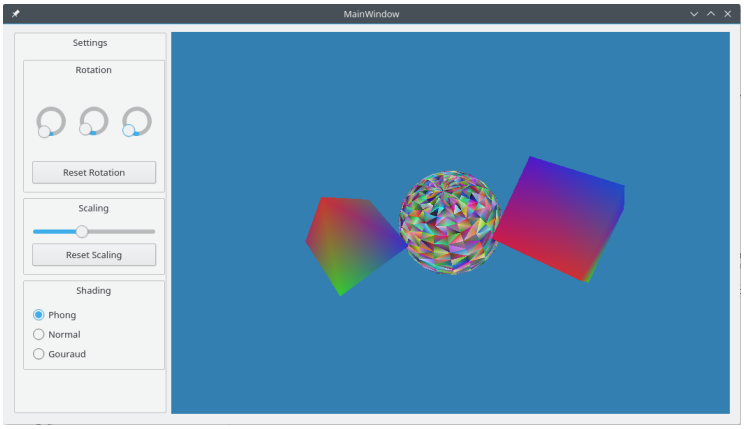


Figure 7.1: *The pyramid, cube and sphere with scaling and rotation.*

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

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

† To obtain a pointer to the data stored in a `QVector` for the third parameter of `glBufferData`, call the function `data` on your `QVector` object.

data is used. Unitizing a mesh linearly scales the vertex coordinates such that the model (exactly) fits in a cube with sides of length 2 and its origin at $(0, 0, 0)$. The `Model` class contains everything you need in order to implement unitizing. Only the implementation was not provided.