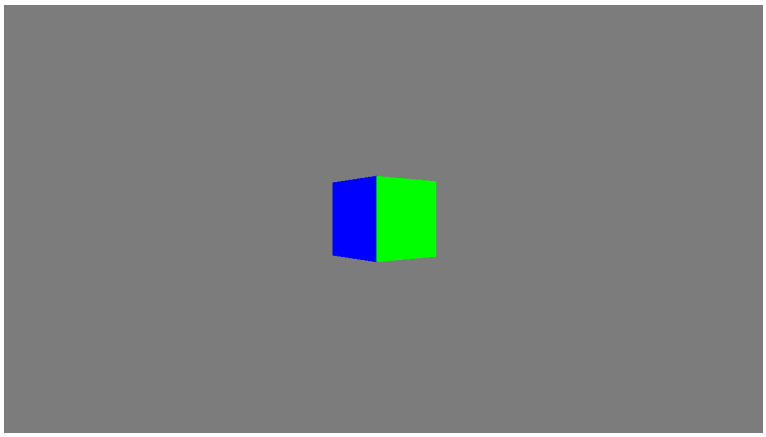
COMP3811 – Exercise G.3 OpenGL Cube

Contents



In this exercise, you will render a simple 3D object: a cube. The exercises walks you through the steps to extend the previous program (Exercise G.2) to move from 2D to 3D. You will also implement a few of the necessary matrix/vector operations that this requires.

It is recommended that you have completed Exercise G.2 before this one.

Note: You are allowed to reuse code from your solutions to exercises in the courseworks, *if and only if* you are the sole author of the code in question. Keep this in mind when working on the exercises. While you are encouraged to discuss problems with your colleagues, you should not directly share any code. If you find solutions to your problems online, study the solution and then implement it yourself. Never copy-paste code, and always make sure you understand the code that you write!

If you run into problems, you are expected to attempt resolving your issues yourself first, e.g., by debugging your program. Do not just immediately ask for help. Assistants may ask you to show/explain what you've attempted so far. Help will only be provided if you have made a fair attempt first.

Recommendation: It might be tempting to just copy-paste the code from the PDF document into your solution while working through this exercise. I would recommend against this. You will gain far more by spending the small amount of extra time it takes to write the code yourself.

1 Input data

Make sure that you have your Exercise G.2 solution at hand. Much of the setup will be very similar (e.g., setting up a VAO and drawing with it).

The first step is to define new input data. In this case, we will want to render a cube. A cube has six faces. Each face consists of two triangles, which have three vertices each. That's a total of $6 \times 2 \times 3 = 36$ 3D vertices. Each vertex has a 3D position and a RGB color (we want to give each side of the cube a distinct color such that we can distinguish the sides from each other).

We define the cube as an unit cube (e.g., spanning from $(-1, -1, -1)$ to $(+1, +1, +1)$. We will later use a camera transform to place our virtual camera such that we are looking at the cube from the outside.

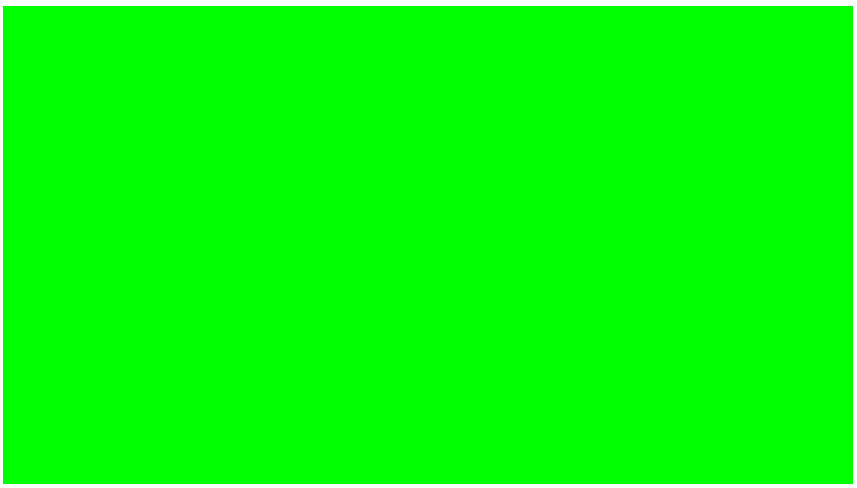


Figure 1: *Clip space unit cube. The entire image is filled with a solid color. We see exactly one face of the cube, stretched across the entire viewport.*

Now, it's (still) relatively easy to write these vertices down by hand. However, to save you some debugging, the included `exercise3/cube.hpp` defines two C arrays with the necessary data: `kCubePositions` and `kCubeColors`. These are similar to the `kTriPositions` and `kTriColors` arrays from the previous exercise (albeit the positions in `kCubePositions` are now 3D positions instead of 2D!).

Task: Modify the VBO and VAO set up (Exercise G.2) to use the data from the `cube.hpp` header. Do not forget to update the VAO such that positions are formed from three floats (instead of two as previously). Also make sure you update the call to `glDrawArrays` to draw the correct amount of vertices:

```
// Draw triangles: 6 faces, 2 tris per face, 3 vertices per tri
glDrawArrays( GL_TRIANGLES, 0, 6*2*3 );
```

Task: Implement the vertex and fragment shaders (`assets/default.vert` and `assets/default.frag`). Make sure that the inputs to the vertex shader match the new VAO format (e.g. 3D positions). If you base your shader code on Exercise G.2, simplify the fragment shader by removing the color modulation and simply pass-through the color data.

For now, pass-through the positions in the vertex shader as well. When you run the program, you should see something similar to Figure 1.

2 Matrices

In order to render the cube with a proper 3D projection, you will need a few different 4×4 matrices. First, you will implement functions to create and work with the matrices. Then you will use these helpers to compose a "model-view-projection" matrix that transforms from model space to clip space and pass it to OpenGL.

2.1 Matrix helpers

In the `vmlib/` folder, you will find a few new headers, including `vec3.hpp`, `vec4.hpp` and `mat44.hpp`. These define the `Vec3f` (3D vector), `Vec4f` (4D vector) and `Mat44f` (4×4 matrix) types, respectively. They are similar in spirit to the `Vec2f` and `Mat22f` that you are already familiar with.

The vector classes are fully implemented, albeit you might want to add your own functions to them as you see fit. Study the existing code briefly. Some of the functions and operators of `Mat44f` are left empty and it is up to you to implement them.

Task: Implement the 4×4 matrix-matrix multiplication. Verify that it works correctly before proceeding (this will save you a bunch of time debugging later). Recall that

$$Q_{i,j} = \sum_{k=0}^{3} L_{i,k} \, R_{k,j}$$

with $Q_{i,j}$ being the elements of the resulting 4×4 matrix, L being the left-hand side matrix in the multiplication, and R being the right-hand side one.

Task: Implement the 4×4 matrix-vector multiplication. Again, verify that it works correctly before proceeding.

Next, we want to define the helper functions to create specific matrices. The specific functions are:

- Rotation matrices: `make_rotation_{x,y,z}()`
- Translation matrix: `make_translation()`
- Perspective projection matrix: `make_perspective_projection()`

Task: Implement the three `make_rotation_{x,y,z}` functions. Verify that they work correctly. (E.g., minimally check the results for a few well-known cases.)

Recall that

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & +ca & -sa & 0 \\ 0 & +sa & +ca & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, R_y = \begin{pmatrix} +ca & 0 & +sa & 0 \\ 0 & 1 & 0 & 0 \\ -sa & 0 & +ca & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, R_z = \begin{pmatrix} +ca & -sa & 0 & 0 \\ +sa & +ca & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Here, $ca = \cos\alpha$ and $sa = \sin\alpha$ for a rotation with angle α. You may assume that the `aAngle` argument is given in radians.

Task: Implement the `make_translation` function. Verify that it works correctly.

Recall that

$$T = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

for a translation of $\mathbf{t} = (t_x, t_y, t_z)$.

Task: Implement the `make_perspective_projection` function. Verify that it works correctly.

Recall that

$$P = \begin{pmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{pmatrix},$$

where $sx = \frac{s}{\text{aspect}}$, $sy = s$ with $s = \frac{1}{\tan\frac{\text{fov}}{2}}$. Further, $a = -\frac{f+n}{f-n}$ and $b = -2\frac{f\,n}{f-n}$. Here, fov is the field of view (angle), aspect is the aspect ratio of the target surface (window/screen), and n and f are the near and far plane distance, respectively. You may assume that the `aFovInRadians` argument is indeed given in radians.

2.2 3D Transformations

Now that we have implemented the helper functions, it is time to put them to use. Our model, the unit cube defined in Section 1, is defined in model space. We want to transform it into clip space using the transformation pipeline discussed in the lectures. Refer to Figure 2 for a review of the spaces that occur in the vertex processing stage.

You will define a matrix for each of the transformations that need to take place:

- `Mat44f model2world`: transform from model to world space
- `Mat44f world2camera`: transform from world to camera space
- `Mat44f projection`: transform from camera to clip space ("projection")

The model to world transform will change each frame. We will use it to make the cube rotate around the Z axis. We will want to place the camera slightly back (i.e., on the positive Z axis), so that it is not inside of the cube. Hence, we will use a world to camera transformation with a translation along the Z axis. Finally, the projection uses the `make_perspective_projection` function to define a perspective projection matrix.

Task: Locate the line that contains the comment *//TODO: define and compute projCameraWorld matrix*. Start by defining the `model2world` transform as follows:

```
Mat44f model2world = make_rotation_y(angle);
```

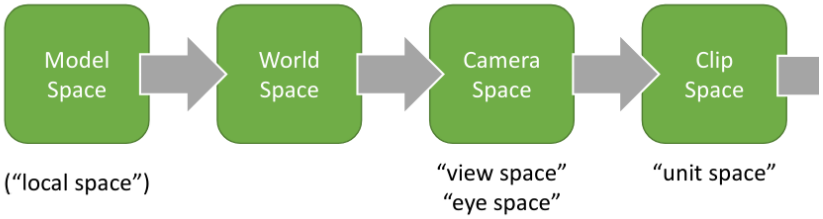


Figure 2: *Spaces in the vertex processing stage. Models are defined in* model space. *They are placed somewhere in the* world space *using a model-to-world transform.* Camera space *is a space relative to the camera. Here, the camera is located at the origin and is looking down the negative z-axis. The world-to-camera transform takes coordinates in world space and moves them into camera space. Finally, the output of the vertex processing stage is defined in* clip space. *The* `gl_Position` *built-in in GLSL is a clip space position. The projection matrix transforms camera space coordinates to clip space. The output from the vertex processing step is in clip space, so the coordinates should not be transformed further. In particular, the fixed-function processing will perform the homogenization / "perspective division" (division with* w*), and we* must not *do it ourselves.*

The `angle` variable is already defined (similar to the last exercise) and increases each frame.

Task: Next, define the `world2camera` matrix:

```
Mat44f world2camera = make_translation( { 0.f, 0.f, -10.f } );
```

We place the camera at $z = +10$. However, the transformation moves the world relative to the camera (rather than placing the camera relative to the world), which means that the translation is inverted (e.g. translation by -10 along the z-axis).

Task: Define the `projection` matrix:

```
Mat44f projection = make_perspective_projection(
    60.f * 3.1415926f / 180.f, // Yes, a proper π would be useful. (C++20: mathematical constants)
    fbwidth/float(fbheight),
    0.1f, 100.0f
);
```

We use a field of view (fov) of $60°$ (but convert it to radians for the `make_perspective_projection` function). The aspect ratio is the width of the window divided by its height by convention. Finally we set the near distance to 0.1 meaning that the near plane is at $z = -0.1$. The far distance is 100, placing the far plane at $z = -100$.

Task: Finally, concatenate the defined matrices to create the `projCameraWorld` matrix that combines all the transformations:

```
Mat44f projCameraWorld = projection * world2camera * model2world;
```

Task: The final step is to pass this matrix to the vertex shader and use it to transform the input vertices. Update the vertex shader (`assets/default.vert`) and declare a new uniform value:

```
layout( location = 0 ) uniform mat4 uProjCameraWorld;
```

Transform the input position with the uniform matrix before writing the results to the built-in `gl_Position`:

```
gl_Position = uProjCameraWorld * vec4( iPosition, 1.0 );
```

We extend the 3D input position to a homogeneous coordinate vector (4D). The input position is a point. Hence, we set the w component to 1.

Finally, we pass computed matrix to OpenGL with `glUniformMatrix4fv`. The call to `glUniformMatrix4fv` must occur after the `glUseProgram` call, but before the call to `glDrawArrays`:

```
glUniformMatrix4fv(
    0,
    1, GL_TRUE, projCameraWorld.v
);
```

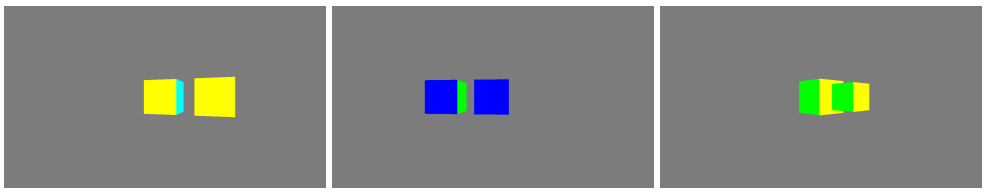


Figure 3: *Second cube "orbiting" the first cube. The cubes are tidally locked. Without depth testing, the second cube is drawn over the first one, even if it is technically positioned further away (rightmost image).*

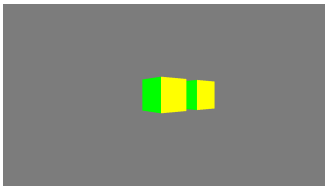


Figure 4: *Two cubes with depth testing.*

The uniform is at location 0 (first argument), as declared in the GLSL shader. There is one matrix. OpenGL should transpose the matrix to convert from the row-major storage that `Mat44f` uses to column-major storage that OpenGL expects internally. Finally, we pass a pointer to the 16 float values of the `Mat44f` matrix.

Test the program. You should see a spinning cube on screen (compare to teaser image).

2.3 More cubes

Task: Add one more cube to the scene. Create a separate model to world transform for it. Define it such that the second cube is initially to the right of the original cube (e.g., at $x = 3$). The second cube should move in a circle around the first cube, rotating at the same rate as the first cube's center. Refer to Figure 3 for screenshots at a few different time steps.

3 Depth testing

Refer to Figure 3. Clearly something is going wrong - whichever cube is drawn second is always drawn over the one drawn first, regardless of their positioning. We fix this by enabling depth testing, which will discard fragments if they are occluded by something that has been drawn earlier.

Task: Configure GLFW such that we always get a 24 bit depth buffer:

```
glfwWindowHint( GLFW_DEPTH_BITS, 24 );
```

The window hint must be set before creating the window, i.e., before the call to `glfwCreateWindow`.

Task: Enable depth testing. In the code, find the place where the global GL state is set up. From Exercise G.2, you should already be enabling back-face culling and sRGB color output. Also enable depth testing:

```
glEnable( GL_DEPTH_TEST );
```

The default depth testing mode is `GL_LESS`, which is good enough for now. See `glDepthFunc` for more information. Similarly, we can set a clear value for the depth buffer with `glClearDepthf`, however the default value of 1 is again the right one.

Task: Clear the depth buffer each frame with `glClear`:

```
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
```

Combining both the `GL_COLOR_BUFFER_BIT` and `GL_DEPTH_BUFFER_BIT` tells OpenGL to clear both the color and depth attachments in one go.

Now run the program. Both cubes should show correctly now (Figure 4).

4 "Arc ball" camera

In this final step, we will implement a simple "arc ball" camera that lets the user control the view position. In practice, this involves reading some user input and computing the `world2camera` transform accordingly.

Study the `CamCtrl_` structure (nested inside `State_` in `main.cpp`). Additionally, briefly look at the two GLFW callback methods that are defined towards the end of the `main.cpp` file. It currently implements a simple control scheme:

- Space actives and deactivates camera control.
- When controlling the camera, the mouse can be moved to rotate the camera around the origin of the world. The variables `phi` and `theta` (angles) describe the direction of the camera.
- The distance at which the camera is from the origin is described by the `radius` variable. When camera control is active, the `W` and `S` keys decrease and increase the radius, respectively. The radius changes at a constant rate w.r.t. wall time (i.e., it is independent of frame rate).

A predefined instance of `State_` exists as `state` in the `main()` function. So, for example, `phi` can be accessed as `state.camControl.phi`.

Task: Construct your `world2camera` matrix from the camera state described above. You will need to construct the following matrices:

```
Mat44f Rx = make_rotation_x( state.camControl.theta );
Mat44f Ry = make_rotation_y( state.camControl.phi );
Mat44f T = make_translation( { 0.f, 0.f, -state.camControl.radius } );
```

The `world2camera` matrix is a combination of the three matrices. Find the right order to multiply these matrices together. Moving the mouse left and right should rotate the camera left and right around the cubes at a constant distance. Moving the mouse up and down should rotate the camera up and down instead. (You may want to temporarily disable the constant rotation of the cubes for debugging.)