CPSC 478/578: Computer Graphics	Feb. 12, 2020
Spring 2020, Assignment 3	

Due: Wednesday, February 26, 2020 at 11:59 PM

Homework 3: Make Your Own Scanline Renderer

In this assignment, we're going to code up our own scanline renderer, like the one in WebGL, from (almost) scratch. You will not need to code up your own matrix and vector classes. Those are provided for you in the form of the Eigen library (`http://eigen.tuxfamily.org/`), which puts the classes `VEC3`, `VEC4`, `MATRIX3`, `MATRIX4`, and `VEC3I`[1] at your disposal. A brief introduction to this library is at the end of this PDF.

Suggested reading for this assignment is Chapter 4.6 in Ganovelli et al., Chapter 7 in Marschner and Shirley, and Chapter 3 of the OpenGL Red Book. The Red Book chapter in particular provides a superset of information that you require. You should not need to learn the OpenGL commands or the matrix stack in order to do this assignment.

1 The Viewport Matrix

Rather than coding up the entire scanline pipeline, switching the whole thing on at once, and praying that it all works on the first try, we are going to build the pipeline incrementally. Working backwards from the final image, let's get the viewport matrix $\mathbf{M}_{\text{vp}}$ working first.

A few geometries have been provided for you to aid in testing. Once you get triangle rasterization working (which you haven't yet), you should see Fig. 2 if you pass it the vertices from `void buildSquare()`. But before that, how are you supposed to know that the vertices transformed correctly? One way is to just stamp down the vertex locations in an image, such as in Fig. 1.

It is up to you to allocate a `float*` array that represents your final image and write the results out as a PPM.

2 Triangle Rasterization

Implement the barycentric coordinate-based rasterization method discussed in class, and that is also covered in Chapter 8.1.2 of Marschner and Shirley. Once rasterization is working, `void buildSquare()` should look like Fig. 2.

You are **not** implementing clipping in this assignment (i.e. nothing from Chapters 8.1.3 - 8.1.6 in Marschner and Shirley). All the triangles should end up inside your viewing volume at all times. If your code contains have a bug that places a vertex outside the volume, it

[1] Three integers instead of floats

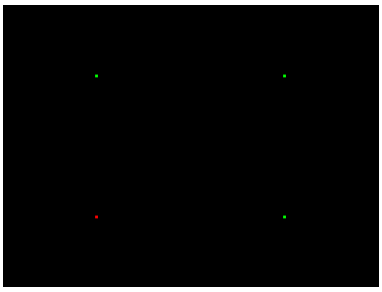


Figure 1: The results of `void buildSquare()` on an 800×600 image, once $\mathbf{M}_{\text{vp}}$ is working, but before rasterization is done. The dot sizes have been exaggerated so they are more visible. If you stamp the vertex to one pixel, the dots will only take up one pixel.

is easy to produce a segmentation fault, because the code can try to color a pixel that is outside the image bounds. If you are hitting lots of segfaults, this may be the problem.

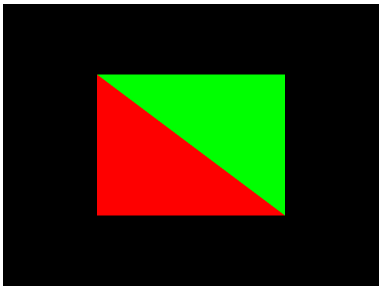


Figure 2: The results of `void buildSquare()` on an 800×600 image, once rasterization is working.

3 Orthographic Projection Matrix

Next, let's get $\mathbf{M}_{\text{ortho}}$ working. For this, the geometry in `void buildBigSquare()` is available to assist you. Whereas the geometry in `void buildSquare()` existed entirely inside the canonical view volume (-1.0 to 1.0 on all sides), `void buildBigSquare()` makes a big square that extends from 1.0 to 11.0.

Fig. 3 shows what you should see once $\mathbf{M}_{\text{ortho}}$ is working and you set `left = 0.0; right = 12.0; bottom = 0.0; top = 12.0; near = 12.0; far = 0.0;`.

4 Camera Matrix

Let's get the camera transform working. You should be able to look at the square at a slightly off-kilter angle now, such as in Fig. 4

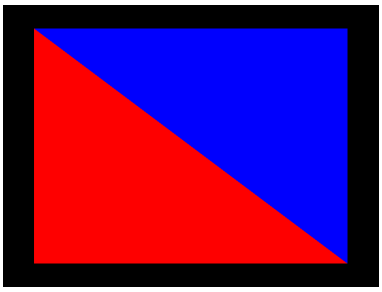


Figure 3: The results of `void buildBigSquare()` on an 800×600 image, once $\mathbf{M}_{\text{ortho}}$ is working.

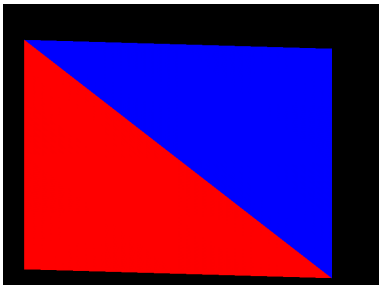


Figure 4: The results of `void buildBigSquare()` on an 800×600 image, with the camera matrix working. In this image, `eye = (0.2, 0.2, 1)`, `lookAt = (0, 0, 0)`, and `up = (0, 1, 0)`. Because we are not looking at it straight-on, the square is slightly off-kilter.

5 Perspective Projection Matrix

Following the section **Projection Transforms** in Chapter 3 of the OpenGL Red Book, implement a projection matrix that is parameterized by `fovy, aspect, near` and `far`. In order to test this stage, first scale the vertices of `void buildBigSquare()` by 0.5. Then you should get Fig. 5

6 Perspective Divide

Next, let's do the perspective divide. We can load up `void buildCube()` for this one, and it should draw a cube centered around the origin. If you get the perspective divide working, you should see Fig. 6.

It looks odd, like the cyan floor is somehow barging in front of everybody else. This is because we have not yet implemented Z-buffering, so whatever face was rasterized last will be the one that shows up. The cyan face is the last one pushed into the vector `indices`, so it's strange but fitting that it is drawn on top of everybody else.

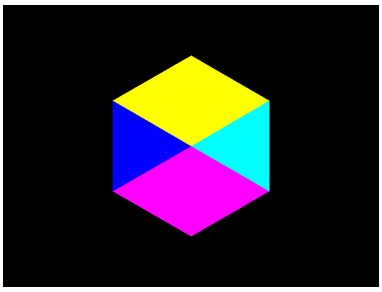


Figure 5: The results of `void buildCube()`, with all the vertices scaled by 0.5, on an 800 × 600 image, once the perspective projection matrix is working. The camera settings are `fovy = 65.0, aspect = 4.0 / 3.0, near = 1.0, far = 100.0` with `eye = (1,1,1), lookAt = (0,0,0), up = (0,1,0)`.

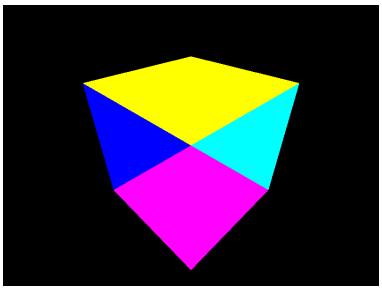


Figure 6: The results of `void buildCube()` on an 800 × 600 image, once the perspective divide is working. The camera settings are `fovy = 65.0, aspect = 4.0 / 3.0, near = 1.0, far = 100.0` with `eye = (1,1,1), lookAt = (0,0,0), up = (0,1,0)`.

7 Z-Buffering

Now let's implement Z-Buffering. When this is working, you should see something like Fig. 7. The cube faces that are in front actually show up in front, as expected.

8 Color Interpolation

Finally, let's do interpolation over the surface of each triangle.

Load up `void buildCubePerVertexColor()`, where the number of entries in `vector<VEC3>& color` will now match the number of *vertices* instead of *faces*. Using the barycentric coordinates you computed during triangle rasterization, interpolate the per-vertex color over the surface of each triangle. When this is working, you should see Fig. 8.

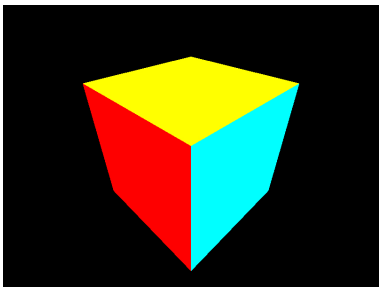


Figure 7: The results of `void buildCube()` on an 800×600 image, once Z-Buffering is working. The camera settings are `fovy = 65.0, aspect = 4.0 / 3.0, near = 1.0, far = 100.0` with `eye = (1,1,1), lookAt = (0,0,0), up = (0,1,0)`.

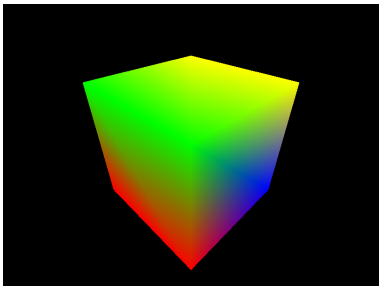


Figure 8: The results of `void buildCubePerVertexColor()` on an 800×600 image, once color interpolation is working. The camera settings are `fovy = 65.0, aspect = 4.0 / 3.0, near = 1.0, far = 100.0` with `eye = (1,1,1), lookAt = (0,0,0), up = (0,1,0)`.

9 Texture Mapping (578 Only)

Implement texturing by assigning (u, v) coordinates to each triangle, and applying the provided image file `sheep.ppm` to each face (Fig. 9). Make sure the texture coordinates are perspective-corrected.

10 Formatting Instructions

- Turn in a ZIP file named LastName_FirstName_Assignment_3.zip.
- Unzipping should produces a `main.cpp` and other files required for `make`.
- Write separate functions for each problem, e.g. `rasterization()`, `perspective_divide()`, and `z_buffering()`.

Figure 9: Apply this image to each cube face. You can orient them however you want.

- Running your binary with no command line arguments should produce images for each item above using the default parameters described in the Figures. Each image should be numbered according to the question, `1.ppm`, `2.ppm`, `3.ppm` ..., `9.ppm`.
- In your `main()` function, you may clear all necessary buffers and call new functions once rendering is complete for each separate problem.
- Your program should accept nine parameters from the command line. They should be: `eye_x eye_y eye_z lookat_x lookat_y lookat_z up_x up_y up_z`. We will be testing your code by entering different settings for these parameters. The output should be an image generated by your most complete version of the scanline algorithm, using the specified camera parameters.
- Include a `README.txt` if necessary to explain why some pieces of your answer work while others do not.

11 Points Breakdown

The assignment will be graded *on a Zoo lab machine*. Make sure your code builds there.

- **(10 points)** Viewport matrix.
- **(10 points)** Triangle rasterization.
- **(10 points)** Orthographic Projection Matrix.
- **(10 points)** Camera Matrix.
- **(10 points)** Perspective Projection Matrix.
- **(10 points)** Perspective Divide.
- **(10 points)** Z-Buffering.
- **(10 points)** Color Interpolation.
- **(10 points)** Texture Mapping (578 only).
- **(8 points)** Follow the formatting instructions.
- **(2 points)** At the top level directory, include a `HOURS.txt` that lists approximately how many hours you spent on this assignment. There is no right answer; we will only be analyzing this data anonymously.

There are a total of 90 points for 478 and 100 for 578. We will be running the Measure Of Software Similarity (MOSS) on the submissions for this assignment. Don't borrow code from your classmates.

12 Eigen Overview

The Eigen library (`http://eigen.tuxfamily.org/`) is very useful, but also quite large. Here are a few pieces that are useful for this assignment. This list is not exhaustive.

- You can add and multiply vectors and matrices them using the `*` and `+` operators. If you try to mix matrices of the wrong size, i.e. multiply a `MATRIX3` and a `VEC4`, it will give you a compile time error.
- Vector and matrix entries are not automatically initialized to zero. Call `setZero()` to accomplish this.
- You can access scalar elements using bracket notation for vectors:

```
VEC3 v;
v.setZero();
v[0] = 0.1;
v[1] = 0.2;
```

Use parenthetical notation for matrices:

```
MATRIX3 A;
A.setZero();
A(0,0) = 1.0;
A(0,2) = 2.0;
```

- To normalize a vector `v0`, call `v0.normalize();`. The vector `v0` is now normalized. However, to *assign* a normalized vector, use `normalized()`:

  ```
  VEC3 v0(1,2,3);
  VEC3 v1(4,5,6);
  v1 = v0.normalized();
  ```

  In this case, the vector `v1` is now a normalized copy of `v0`, but `v0` remains unchanged.

- You can send Eigen matrices and vectors to output streams, so

  ```
  cout << v0 << endl;
  ```

  will indeed work.

- The following will set a matrix to identity:

  ```
  MATRIX4 m;
  m = m.Identity();
  ```

- To transpose a matrix, to this:

  ```
  m = m.transpose();
  ```

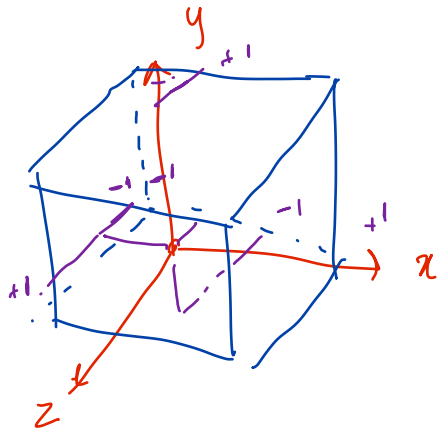
- To store an inverse matrix use this:

  ```
  m = m.inverse().eval();
  ```

  Note: it is possible to complete the assignment without using an inverse.

Viewport transformations

We want camera looking at canonical view volume in -z direction



$$\begin{bmatrix} \frac{xRes}{2} & 0 & 0 & \frac{xRes}{2} \\ 0 & \frac{yRes}{2} & 0 & \frac{yRes}{2} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} x \\ \end{bmatrix}$$

$4 \times 4 \quad 4 \times n$

↓ however many

e.g. $n = 8$.