University of St Andrews

Computer Graphics

CS4102

Interactive 3D Modelling

Author:
150008022

May 16, 2020

Goal

The aim of this practical is to understand the key principles behind various techniques frequently used for the rendering of 3D objects, and to get hands-on experience with their implementation and manipulation.

1 Running the Program

The project uses `maven` for lifecycle and dependency management. `LWJGL` was used for this project, which provides a way to use `OpenGL` from Java. OpenGL *may* be a cause of compatibility issues when running, but an early and forward compatible profile was used to minimise the risk of this.

Usage
`mvn clean install` will compile the project, run all tests, and produce an exectuable jar.
`cd target` to enter the directory where the jar has been compiled.
`java -jar FaceModelling-jar-with-dependencies.jar` to run the program. Pass the `-h` flag to see options.
Camera control:

- W to move into the screen
- S to move out of the screen
- A to move left
- D to move right
- Right click and drag to change orientation
- Left click on selection triangle to change weightings (ray tracing not perfect, works best when facing straight on).

2 Implementation

2.1 Loading Mesh Data

A `FaceLoader` instance initialises by parsing the indices (`mesh.csv`), and the average face coordinates (`sh_000.csv`), vertex weighting (`sh_ev.csv`), average face colour (`tx_000.csv`), and colour weightings (`tx_ev.csv`). The indices are converted from 1-indexed to 0-indexed when being loaded in. The `FaceLoader` instance then loads the deltas for both coordinates and colours for each face, calculating the actual coordinates for each vertex of each face.

The arrays of face data are then passed to the `MeshLoader` which creates a vertex array object (VAO) for each face, and binds a vertex buffer object (VBO) for the vertices, indices, and colours to them.

The use of indices instead of simply listing all coordinates saves on memory as the one vertex can be used in many triangles, as demonstrated in figure 1.

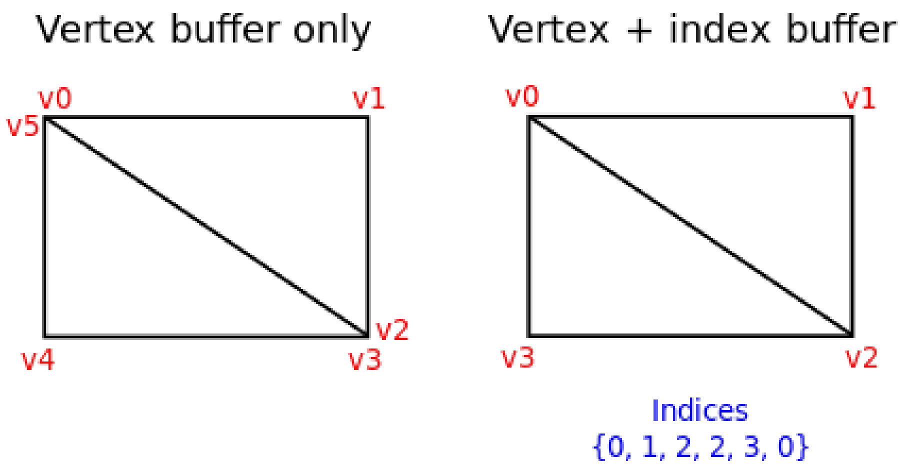

Figure 1: How indices are used when describing meshes.

2.2 Depth Testing and Painters Algorithm

OpenGL does not natively implement painters algorithm for depth testing, and does not expose an API for carrying it out . Instead, it uses Z-buffers, Z-Buffers involve recording the z-index last drawn at each pixel of the window view. A pixel with a greater z-index than the current value will be drawn and the z-buffer updated to reflect the new value. Otherwise the depth test fails and the object will not be drawn at that coordinate. This functionality is enabled using the `glEnable(GL11.GL_DEPTH_TEST)` call. An optimisation called early z-testing is often used which is applied earlier on in the rendering pipeline in order to reduce the number of fragments that need to be processed in total.

Painters algorithm and Z-buffers are both viable solutions to the same problem; the former is more computationally expensive while the latter requires more memory. Z-Buffers are preferred in most cases for modern hardware since memory has become cheaper. Painters algorithm also requires additional computation in the case where objects intersect. Methods to solve this involve splitting the intersecting shapes and rendering these instead.

A separate renderer was implemented that used painters algorithm. This renderer was significantly slower both due to the added computation necessary for the painters algorithm, and also due to the fact that the sorting was performed using the CPU rather than the GPU.

The first step involved applying each transformation to the set of coordinates. To try improve performance, this task was parallelised using a thread pool. Then, for each triplet of indices (i.e. each triangle), the maximum Z value was recorded. Next, maximum z values for each triangle were sorted lowest to highest using a custom `quicksort` implementation. The custom implementation was necessary so that every swap performed on the array of maximum z values would also be performed on the indices array, effectively sorting each triangle by maximum z value. The new indices would then be passed as a VBO for OpenGL, which renders triangles in order of appearance in the indices VBO.

The algorithm does not account for intersecting shapes, which is especially noticeable at the nose of the faces. Methods for dealing with such intersections include splitting the intersection shapes and painting (or not) the subsections individually.

2.3 Selection Triangle

The selection triangle is rendered with the three control faces at each vertex. A left click on the triangle updates the weighting to be used when producing the interpolated face. This required carrying out ray casting with respect to the mouse position, and was achieved using the steps described by [1]. Mapping the left click to a position on the selection triangle involved converting from the 2D screen space coordinates to OpenGLs 3D device coordinate system (Figure 2).

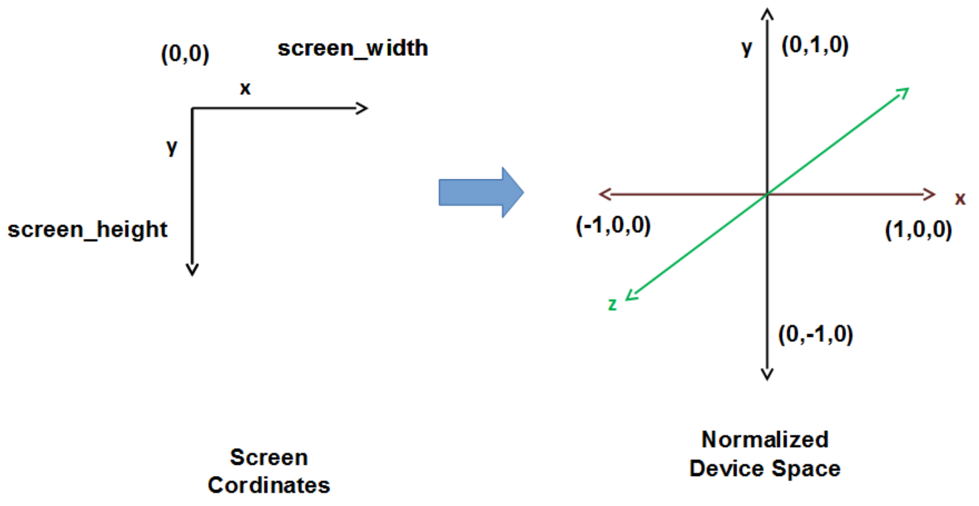

Figure 2: Converting from screen space to OpenGL [2].

We then apply the inverse of the transformation matrices we used to convert to screen space to get the coordinates in world space. A marker is shown on the triangle to represent the current weighting, and the colour of the marker also reflects the current weighting. The weightings are calculated using a rearrangement of the following system of equations [3]:

$$P_x = W_{p1}X_{p1} + W_{p2}X_{p2} + W_{p3}X_{p3}$$

$$P_y = W_{p1}Y_{p1} + W_{p2}Y_{p2} + W_{p3}Y_{p3}$$

$$1 = W_{p1} + W_{p2} + W_{p3}$$

If our user clicks outside the bounds of the selection triangle, a negative weighting would be produced, which we can check for before updating the output face.

2.4 Interpolated Face

For each vertex of the face, the coordinates can be calculated using the equations for Barycentric coordinates described above. The output face is rendered at a larger scale than the control faces, just to the right of the selection triangle.

2.5 Shading

OpenGL supports both flat and Gouraud shading methods. However, the flat shading method only uses the colour from the first vertex of each triangle. My interpretation of the practical specification was that flat shading should involve using the average colour from each vertex to calculate the colour of every fragment on the face of the triangle. When accounting for lighting, the face of each triangle would have the same value which could be calculated by taking the cross product of two edges of the triangle (which two affects the direction of the normal), as shown in figure 3. This would require that each vertex in the mesh that is part of multiple triangles would have a normal value for each triangle it was part of, due to how OpenGL processes VAOs with indices. This was realised later into development, and would require heavy refactoring to correct.

Gouraud shading involves calculating a normal per vertex, where this normal is weighted by each triangle that the vertex belongs to. This weighting can be calculated many ways. Often logic is included so that hard edges are not rounded. For this practical, the steps were taken (from [4]):

1. Set all vertex normals to zero
2. For each face, calculate the face normal and add it to the vertex normal for each vertex its touching
3. Normalize all vertex normals

The normal for each point on the surface of each triangle is calculated by again using Barycentric coordinates.

Again, the main issue with implementing the shading fully was the abstraction of OpenGL. Though the concepts were thoroughly understood, I was not able to implement it.

2.6 Projection

Both perspective and orthographic projection can be set.

For the perspective projection matrix, the focal length can be set from command line options.

The transformations from model coordinates to view coordinates involves the multiplication of a series of different transformations:

$$Transformation = [\text{Projection}][\text{View}][\text{World}]$$

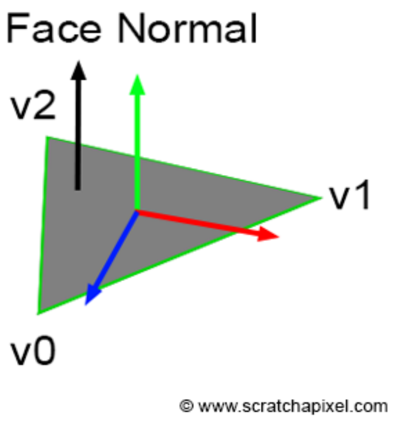

Figure 3: Calculating face normal.

The world matrix maps the model coordinates (i.e. our face vertices) to how we want it represented in our 'world'. This includes transformations such as translation, scaling, and rotation. A scene graph was implemented so that models could be rendered relative to each other. This is exemplified by the control faces which are positioned relative to the selection triangle. This involved recursively building a world matrix for each parent.

The view matrix is constructed from the current camera position. The camera movement was based on the guide from [2].

The perspective projection matrix is calculated using the field of view (hard coded to 60), the aspect ratio (width over height), and the focal length. Since the camera is used, the focal length described the distance just in front of the camera (hard coded to 0.01f) to the far distance set by the user (default 5f). The effect of the far distance can be seen if the camera is moved backwards far enough, as the model begin to disappear.

The orthographic projection matrix is calculated by specifying the left-most coordinate (-1 in OpenGL) rightmost (the aspect ratio was used so that the image was not too stretched by non-square windows), the lowest and highest coordinate, and the near and far distances again.

Since the orthographic view is effectively like having a camera infinitely far away, it behaves differently to the perspective camera. Zooming in or out does not affect the appearance of the models, therefore the relative sizes of objects are constant regardless of their distance from the viewer. The widths and heights of objects also remain constant as the user moves in any other direction.

On the other hand, the perspective projection stretches objects as they are approached. Zooming in effectively increases the scale of the model.

3 Conclusion

Though I was not able to implement everything, the time spent reading to understand the concepts was very beneficial. Development was drastically hindered by several issues. For example, when implementing Painters algorithm, a custom implementation of quicksort was used that would cause a stack overflow. Time was lost on the assumption that this was due to a logical error, but it was in fact simply due to the large number of elements being sorted, and was corrected by increasing the stack size for the JVM (as mentioned in the usage). In hindsight the use of OpenGL quickly became more of a curse than a blessing. It was difficult to determine how much of the functionality was to be written from scratch (for example, it was assumed acceptable to use the `JOML` matrix functions such as `setPerspective()` when constructing projection matrices). For example, when calculating pixel colours, OpenGL uses Gauraud shading by default, and interpolates the colours using Barycentric coordinates for each fragment. The time spent determining if it was possible to perform these operations myself efficiently would have been better spent including a lighting model which could have been applied to each fragment with knowledge of the vertex normals (these are calculated in the `Util` class, but not used).

Nevertheless, this project has greatly improved my understanding of computer graphics.

4 Summary of Implementation

Requirements

- Render faces using index, coordinate, and colour buffers
- Painters algorithm
- Click triangle to select weighting between different faces
- Interpolate to create face from control faces with weighting
- Orthographic projection
- Perspective projection
- Adjustable focal length
- Comparison of projection methods

Extensions

- Camera is able to move around 3D world
- Comparison between painters algorithm and z-buffers
- Control faces are also visible during selection
- Use of OpenGL for optimised rendering
- Scene graph to render children objects relative to parents

References

[1] Anton Gerdelan. *Mouse Picking with Ray Casting*. `https://antongerdelan.net//opengl/raycasting.html`.

[2] Antonio Hernández Bejarano. *3D Game Development with LWJGL*. `https://lwjglgamedev.gitbooks.io/3d-game-development-with-lwjgl`.

[3] CodePlea. *Interpolating in a Triangle*. `https://codeplea.com/triangular-interpolation`.

[4] *Gouraud Shading*. `http://www-users.mat.umk.pl/~wrona/3d_tutor/shading.html`.