Practical 2: 3D Rendering

Overview

This practical was concerned with the creation and display of face meshes using triangles, as well as face interpolation, Lambertian lighting and interactive input. I have completed this project only on the basic specification. My program includes:

- Interactive input that allows user to convey the interpolation weights via clicking.
- Rendering of face meshes using the base face shape/colour, offsets, weights, and interpolation weights.
- Painters Algorithm
- Lambertian reflectance

Implementation

My solution is implemented in Python 3. I choose python as it is high level enough to reduce the amount of boiler plate code for reading in data and manipulation lists/arrays (of which there is a lot in this practical). My design is centred around a self-implemented triangle class. This class is very useful as it stores the colours of its vertices as well as the vertices themselves, as well as a multitude of functions that implement Lambertian reflectance, centroids, depth, flat shading etc. All processing techniques are self-implemented, from mesh calculations to determinants.

Basic Face Rendering

The focal point and essence of this practical is centred around the rendering of a face using a triangle mesh. There are several important files that define and facilitate the rendering of faces. *sh_000.csv* defines the shape of the average face, each row pertains to the three-dimensional co-ordinates (x, y, z) of a position in this face shape. *sh_001.csv, sh_002* & *sh_003* all contain different faces based on the average face shape; therefore, these can be considered offsets from the original average face, required to produce other faces. *sh_ev.csv* defines the weightings of these offsets, with each row containing the weight that should be applied to the relevant *sh_00x* file. There are colour equivalents for each of these shape files that have the same structure but determine the RGB values of each triangle in the mesh. *mesh.csv* contains a series of rows, each row with three values. The rows contain three indices, these indices define a triangle between the points in *sh_00x* files and *tx_00x* files.

My program begins by reading in and storing all the shape and colour files (average, offsets, and weightings). These are read and stored using a *polygon_builder* instance. Once these have been read in, I then build the mesh, which is stored as a list of triangles (polygons). For each row in the mesh, I create an instance of my *triangle* class, which stores the vertices and colours defined by the mesh. A single vertex on the triangle must be calculated using the average face, offsets, and weightings. So, for each point on each triangle the following equation:

$$\begin{aligned} vertex = sh000.position &+ (sh001.position \times sh001.weight \times input\_weight\_1) \\ &+ (sh002.position \times sh002.weight \times input\_weight\_2) \\ &+ (sh003.position \times sh003.weight \times input\_weight\_3) \end{aligned}$$

This equation is simplified as the average is taken for each of x, y and z, and then they are recombined into a point in 3D space. A similar set of equations are used to determine the colour of each vertex. Once all the triangles in the mesh have been computed, the mesh can be rendered.

To render faces I use the library MatPlotLib. The rendering process is completed in *render.py*. Before plotting I find the maximum absolute value for each dimension (x, y, z) and normalise the co-ordinates between -1 and 1 (I was unable to get the matplotlib axes to behave as I desired). I also use flat shading (give each triangle a single uniform colour) and this colour is the average RGB value across the vertices of the triangle. Each triangle is then plotted on a 3D figure, which is then rendered with the view facing the face directly as shown in figure 1.

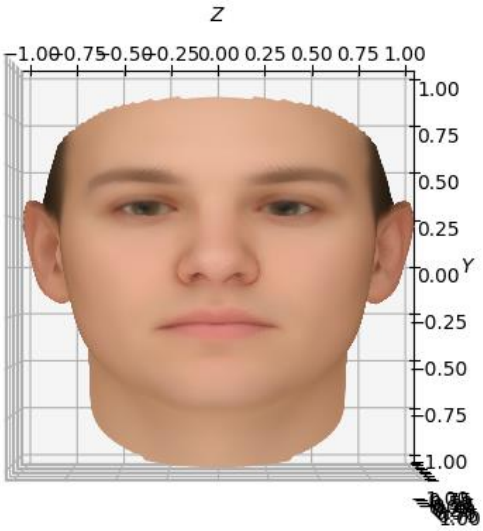

Figure 1 - A face mesh render with no lighting

Interactive Input

The interactive input aspect of the project is completed using the TKinter library. Figure 2 shows the input screen. The user may click anywhere inside the black triangle to convey the weight of each face, the closer to a vertex, the higher weight that face receives. The top vertex represents *sh_001*, the left vertex represents *sh_002* and the right vertex represents *sh_003*. After clicking the user should wait, the program is quite slow (roughly 2 minutes per render on my PC), this is because matplotlib produces publishable figures at the cost of time. The program will however display the progress of the process in the terminal window.

Figure 2 - The input screen for my program

Input from the user must be sanitised and as such the program checks that input is within the triangle. This is done with a similar approach to my convex hull in practical 1. By using the Z component from the cross product of vectors drawn between vertices and each edge I determine whether a click is within a triangle. The weight calculation is the inverse value of how far the click is from each vertex (small distance results in a large weight). The triangle is (almost) perfectly equilateral which helps with calculating weights and they are adjusted so that they sum to 1. Once the face has been rendered, a second TKinter window will appear with the face displayed inside it.

Painters Algorithm

The painter's algorithm requirement is a polygon-by-polygon approach to rendering. The algorithm requires that the polygons be sorted so that the furthest away polygons are rendered first. To do this I find the centroid of each triangle in the mesh and then sort

the triangles by the z component of their centroid using a quicksort. The polygons are then rendered in order from low z to high z. Later, I realised that MatPlotLib does this automatically. I left my implementation in, to keep the spirit of this aspect of the practical although this means the process is repeated.

Lighting

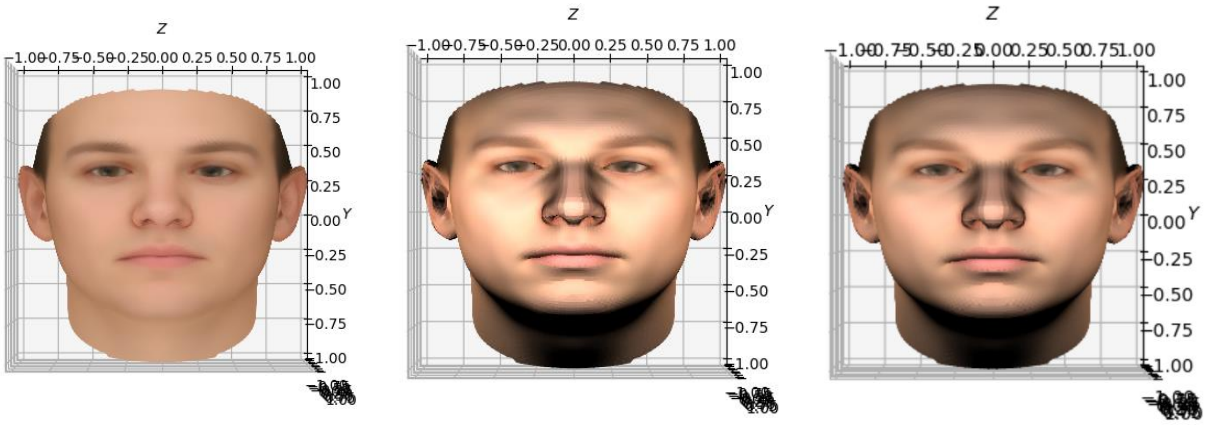

Figure 3 - A series of faces with different lightings. Left face has no lighting. Middle face has strong lighting. Right face has current lighting strength.

Finally, the basic specification also required that we introduce a light source. The light source is placed at position (10, 0.5, 0.5) which is essentially just in front of the face and aligned with the viewing direction. The practical specified flat shading, matte surfaces, and a unity diffuse coefficient. I have assumed unity diffuse coefficient to mean the diffusion co-efficient stays the same regardless of the wavelength. Therefore, to find the light-affected colour of each triangle I use Lamberts Reflectance. Calculations of the normal and the vector from the surface to the light source are performed in the triangle class. I then take the dot product of these two and multiply it by 2 (diffusion co-efficient) and then by 0.6 (intensity). The resulting value is then multiplied with the RGB values to get the final colours (if the value is beneath 0, I set it to 0). The result of this lighting can be viewed in figure 3 on the right side. These values were found through trial and error. The middle face shows an intermediate lighting which had much higher intensity. Unfortunately, due to flat shading, the polygons are much more visible in the nose after lighting the face. This is because the rate of change in the normal of each polygon is high across the nose, resulting in sharp changes in colour. Other than this, the lighting is very effective.

Usage

Required packages:

tkinter, PIL, math, csv, numpy, matplotlib, mpl_toolkits, pylab

These can be installed with a package manager such as PIP, e.g.:

```
pip install tkinter
```

Execution:

```
python main.py
```

The mesh files must be in the **same directory** as the python scripts.