3D Rendering

1. Introduction

The assignment asks to implement a program that can synthesise and render 3D faces. This implementation covers all the basic requirements and some of the advanced requirements. The extensions include: large assortment of faces, face rotation, multiple light sources, limited face scaling, an attempt at perspective projection. A jar file is provided for easy execution. See the appendix for instructions to execute and use the system.

2. Loading and Storing Data

The provided files are loaded as described in the practical specification. The points of Face X are calculated by multiplying the values in sh_X.csv by the X^{th} weight in sh_EV.csv and adding them to sh_000.csv. The same method is applied for calculating the colours of the points. The location and the colour of a point are stored in the same class.

Points are represented with Homogeneous coordinates. Therefore, all affine transformations are implemented via matrix multiplications. The library “jblas” provides optimized implementations to store matrices and multiply them with vectors. The location of a point is a column vector with 4 values (x, y, z, w). All transformation matrices are 4 by 4 squares. When multiple transformations are applied, the dot products are consecutively calculated.

The application defines “polygon” as an array of points. In theory, it should support polygons with many vertices. However, it has only been tested with polygons with 3 vertexes. Therefore, any uses of the word “polygon”, both in the code and in this report, hereby refer to triangles.

3. Drawing a face

The application includes two face renderers. One is used to make previews of selected faces. The other is used to fully render the synthesised face. They both sort the polygons and drawn them sequentially. When drawing a 3D polygon, first the colour is chosen, then the polygon is projected into 2D space. The "*Synthesised Renderer*" is more powerful and supports different options.

3.1. Shading Model

The shading model determines how colour is interpolated on a triangle. A single shading model is used throughout the system. Flat shading was chosen because it is simple and easily processed. Both renderers use this shading model.

3.2. Light Sources

The implementation supports two types of light sources – directional and point. Each light source provides three light intensities – of red, green, and blue. Each light source provides a method that determines the direction of incoming light to a point. For directional, this is simply the direction of the light, regardless of the destination point. For a point light source, it is the vector from the target point to the light source (calculated via vector subtraction). The direction of incoming light and the intensities are used by an illumination model. The "Synthesised Renderer" can switch between Point and Directional light. The lighting colour attempts to simulate the sun with (r=255, g=225, b=200). This colour, and the position of the point light source can be changed in the configuration file ("settings.props") without re-compiling the code.

3.3. Illumination Model

An illumination model determines the intensity of light at a given point. The system currently supports only Lambert's Illumination Model. However, it has been designed in such a way as to be easily extendible.

This implementation supports stacking of multiple light sources. The final light intensity is the sum of intensities provided by the light sources. This equation from the lecture slides has been used: $I = (\hat{\omega_{i1}} \cdot \hat{n}) I_1 K + (\hat{\omega_{i2}} \cdot \hat{n}) I_2 K + ..$. Following the advice from the practical specification, a unity diffuse coefficient has been used (K=1). The intensity is a column vector with three values between 0 and 1. They correspond to the intensities of red, green, and blue. They are defined by the light source. The surface normal is computed at the mean point of the triangle. Both normals are calculated but the one pointing towards the viewing direction is used in the formula. The dot product with the direction of incoming light gives a brightness level. Both vectors are normalized, so the brightness is between -1 and 1. A negative value means that the light source does not illuminate the triangle, in which case the computation ignores it. Non-negative values specify "how well" the triangle is illuminated.

The final intensity is a red-green-blue vector. Since flat shading is used, this intensity is multiplied by the colour of the average point to produce an adjusted colour. That colour is used to fill the

triangle. Only the “Synthesised Renderer” uses an illumination model. The “Preview Render” assumes full intensity at all points.

3.4. Projection

The projection determines how a 3D polygon is projected onto a 2D surface. The system supports orthographic projection. An unsuccessful attempt is made at perspective projection.

The coordinates of vertexes are in a matrix in homogeneous form. Therefore, projection is achieved with just a matrix multiplication. This operation takes a 3D homogeneous point and returns a 2D homogeneous point. Each vertex of the polygon is projected separately. The resulting points are dehomogenised and connected to form a 2D polygon.

Orthographic projection is achieved with a simple matrix that removes the Z coordinate altogether. Perspective projection was attempted in a similar manner with a matrix that scales the X and Y coordinates with respect to Z and a distance f. This caused points that are far away to be projected close to the origin. Mathematically, this makes sense, since X and Y are reduced. However, in practice, this causes the face to be stretched towards the top-left corner of the screen. I tried centring the face in the top corner, then projecting, then translating the projection to the middle of the screen. This didn’t immediately work. Unfortunately I don’t have time to debug it, so it is left as was.

The “Preview Renderer” uses orthographic projection. The “Synthesised Renderer” can switch between the orthographic and perspective projection.

3.5. Triangle Overlap

The previous subsections have described how an individual triangle is painted on the screen. This implementation does not take into account how triangles might affect each other. Painter’s Algorithm has been adopted. Triangles are first sorted by their depth. The depth of a triangle is defined as the Z coordinate of the middle point. Painting starts at the “furthest” triangle and ends with the “closest”. This way triangles in the front, cover the ones in the back. Both renderers use this algorithm.

I attempted to implement Z-Buffering, so that I could account for partially overlapping triangles. I first wrote a short algorithm that paints the points of a triangle individually. It used plane interpolation to generate enough points from within a triangle $p = p_1 + \alpha * (p_2 - p_1) + \beta * (p_3 - p_1)$. Where p is the generated point and p_1, p_2, and p_3 are the vertexes of the triangle. Several executions were made and they iterated values for α and β at different precisions (0.1, 0.01, 0.05). However, this was very slow even without the actual buffering of the Z coordinate. Since I could not come up with an efficient way to implement Z-Buffering, I decided to stick with Painter’s algorithm and focus on other aspects of the graphics.

4. Synthesised Face

4.1. Synthesis

Compared to the other parts of this practical, synthesising a face is actually quite simple. A synthesised face is just a linear combination of other faces. The implemented algorithm can combine any number of faces. Each face has an associated weight between 0 and 1. The total sum of the weights is 1. Each face is practically a vector of points. A point is practically a vector of 6 values (x, y, z, r, g, b). Therefore, the face synthesis is though of as consecutive matrix by scalar multiplications (face by weight) and then matrix with matrix additions (weighted face plus weighted face). Note that the implementation does not deal with them like that. It uses objects to simplify the code from a programmer's point of view. However, I imagine that if synthesis was implemented at a graphics card level, a matrix representation like that one would be used.

4.2. Input

Even though the algorithm supports synthesis of many faces, the interface supports the input of exactly three faces with three weights. As suggested by the practical specification, the user is presented with an upwards oriented triangle. At each vertex of the triangle is a face selector. It includes a face preview (via the "Preview Renderer"), a weight above it, and the face ID below it. The ID is the 'xxx' in the file name "sh_xxx.csv". The weight is the value that will be used for synthesis. Both of these can be entered through the keyboard. Little input validation is done and the system has not be tested against invalid values (non-existing files, weights not in range of 0 to 1). Given valid values, it works well.

The weights can also be set by mouse-clicking within the triangle. The plane interpolation equation $p = top + \alpha * (\text{left} - top) + \beta * (\text{right} - top)$ is used to determine the weights. Where *p* is clicked point, *top* is the top point of the triangle, *left* is the bottom left point, and *right* is the bottom right point. This equation is represented via a matrix. Matrix multiplication is performed to determine the values for α and β. The weights of the faces are then: $1 - \alpha - \beta$ for the top face, α for the left face, and β for the right face.

When the user manually enters weights, the position of point *p* is changed. Given the above equation, all values on the right side are given, so they are applied like that. I imagine that in a graphics card, it would be more optimal to form a matrix of [top, left-top, right-top] and multiply it by [1, α, β].

4.3. Interaction

The given face coordinates are displacements from the nose. As in, the nose is (0,0), the eyes are above the X axis, the mouth is below the X axis, the ears are around the Y axis. The face is also very large. In Java, the origin is the top left corner of the window. Thus, directly displaying the loaded face will cause only some triangles to appear. Scaling it down shows that only a quarter of it fits on screen. To display the full face, the system flips it, scales it down, and translates it to the centre of the screen.

Several possible interactions with the synthesised face are supported. The rotations and scaling are implemented as matrix multiplications. Rotation is a quick sequence of three matrix multiplications. The first translates the face to the origin (0,0), then the actual rotation is performed, then the face is translated back to the middle of the screen. The actual rotation is done around an axis, so in order to rotate the face around its own axis, that axis has to be aligned with the axis of the grid. All six rotations are supported – 3 dimension $_{x}$ 2 directions. An individual rotation tuns the face by 15 degrees. This can be configured from the “settings.props” file.
Two scaling matrices are supported. One increases all dimension, and one decreases all dimensions. Scaling does not depend on an axis, so the face is not translated.

In addition to face transformations, the “Synthesised Renderer” can also change light source and projection matrix. All operations are performed from the key board. See the Interaction Section of the Appendix for keys.

5. Conclusion

Overall, this submission completes the basic specification, the ad first advanced specification, and some the highly advanced specification. It can correctly synthesise and render 3D faces. It allows some interaction with the created faces. Doing this assignment has given me great insight into what happens behind the scenes to display a 3D image. There are many operations that need to be performed and there’s therefore no wonder why graphics cards are dedicated to one purpose.

Running all of these computations on the CPU is slow, when compared to the ability of modern day Computer Graphics. If I were to do this sort of task again, I would approach it as follows. Represent a face with two large 2D matrices. A 4 by N location matrix where each column is a 3D point in homogeneous coordinates. A 3 by N colour matrix where each column is the colour of the respective point. Store the triangle relations separately in a 3 by M matrix where each column is a triangle of indexes into the location matrix. Express the shading and illumination models as matrix multiplications. Projection is already in matrix form but might need adjustments. Find a way to pass all matrix operations to the GPU.

6. Appendix

6.1. Running

The system can be run with “java -jar FaceRendering.jar” from within the submission directory. The file “settings.props” is a plain-text configuration file for the elements of the GUI. It should be in the working directory when running the program. The application is also configured to look for a “data” folder within the working directory. This would be the folder storing the “sh_xxx.csv” files and so on. This submission includes several of them, so as to easily demonstrate that the program is working. To use other sets of files, open “settings.props” with any text editor, and change the paths to the appropriate locations in the file system.

```
# Base File paths
mesh file=data/mesh.csv
average face shape=data/sh_000.csv
shape weights=data/sh_ev.csv
average face color=data/tx_000.csv
color weights=data/tx_ev.csv

# Face Selector
shape prefix=data/sh_
texture prefix=data/tx_
filename suffix=.csv
```

6.2. Source Code Files

The source code written for this assignment is in the packages ‘main’, ‘model’, and ‘renderers’. The classes in the ‘util’ package have been written for previous university assignments and are being re-used for convenience.

6.3. Interacting

When the application starts, give it a few seconds to preview the faces.

When selecting a face for the vertex of a triangle, enter the string ID. This is usually a 0-padded 3-digit integer. The program attaches the ‘prefix’ and ‘suffix’ to it. Those are defined in the configuration file and can be seen above. Example: enter 051 for ID, ***press enter***, and the program will look for “data/sh_**051**.csv” and “data/tx_**051**.csv”. You can enter the weights manually or by clicking the triangle. Then click the “Synthesise” button in the upper right corner to synthesise a face from the three selected faces and their weights. That face will be displayed in a new window.

Here is a list of key presses that will trigger different functionalities in that window.

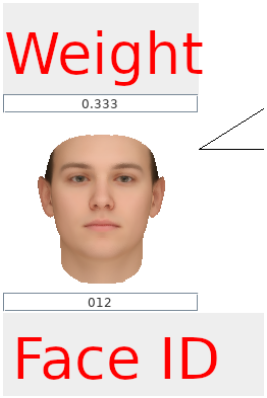
Key	*Action*
Up arrow	Rotate the face up, around the X axis
Down arrow	Rotate the face down, around the X axis
Left arrow	Rotate the face left, around the Y axis

Right arrow	Rotate the face right, around the Y axis
Page Up	Rotate the face counter-clockwise, around the Z axis
Page Down	Rotate the face clockwise, around the Z axis
W	Scale the face, making it larger
S	Scale the face, making it smaller
D	Set the light source to Directional lighting, if it wasn't already
F	Set the light source to Point Light Source, if it wasn't already
O	Use orthographic projection, if it wasn't already
P	Use perspective projection, if it wasn't already

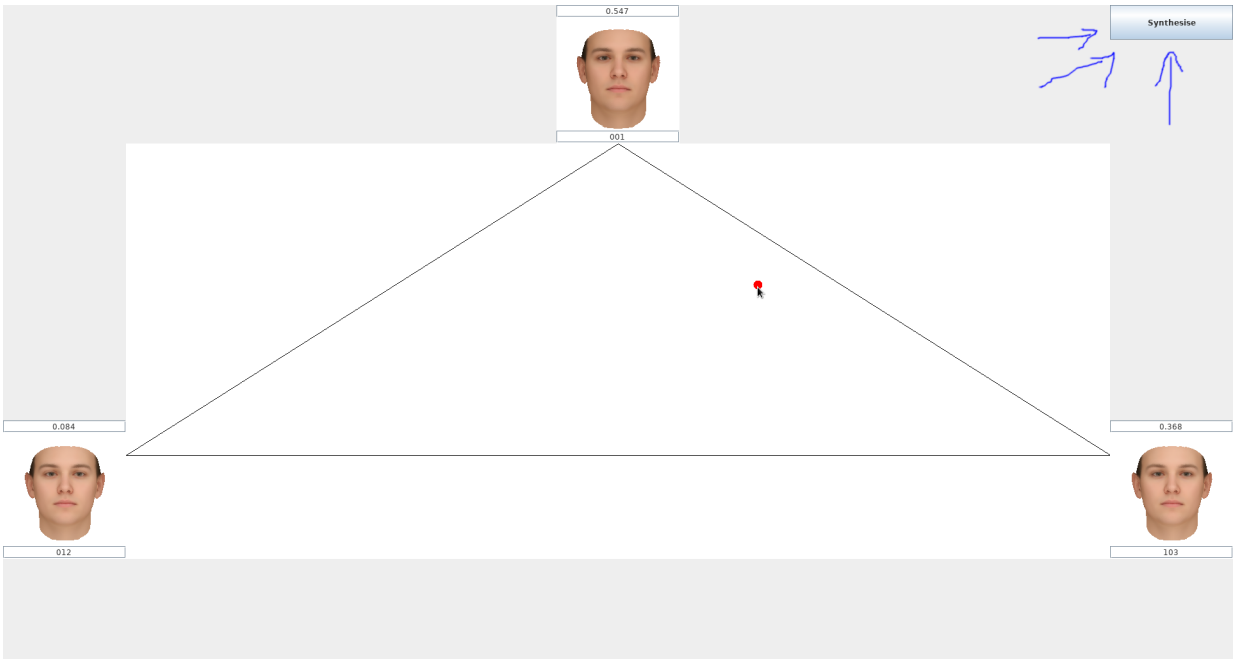
Note that the face selection window is still active and can be used to synthesise anther face completely independently. Use Alt-F4 to exit a window.

6.4. Samples

Selecting an input face:



Select a point, then click "Synthesise":



Directional Light aligned with the view angle. On a Rotated Face:

Point Light from the top-left, in front of the screen. On a scaled, more rotated Face: