3D Rendering

140005087

April 2018

Introduction

This report describes a Java application which is capable of rendering a human face in three dimensions. The Java application implements all relevant graphics techniques from scratch. Pixel values are set directly in a buffer which then drawn to the screen without using any 2d or 3d graphics APIs. The application utilises Gradle for a seamless build process. From the submission folder at the command line simply invoke **gradle build** followed by **gradle run** to start the application.

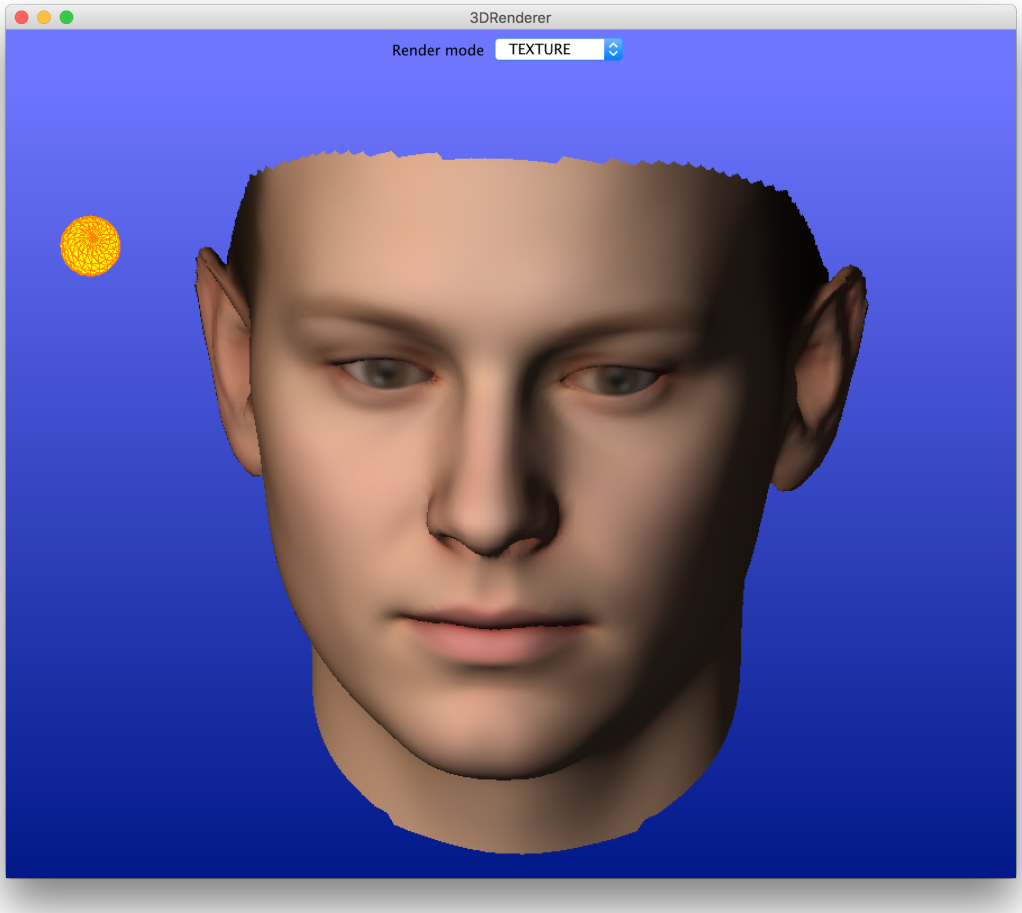

Figure 1: Rendered face

To rotate the face simply left mouse drag in the view screen, to zoom rotate the mouse wheel, and to pan middle mouse drag. The illumination source is indicated by the yellow ball. This may be relocated by holding the keyboard arrow keys in the desired direction. You may need to click in the view area to ensure the window has focus before pressing the arrow keys. The light source will always move relative to the screen so to achieve light source displacement in a new plane simply rotate the model. The combo-box will set the render mode.

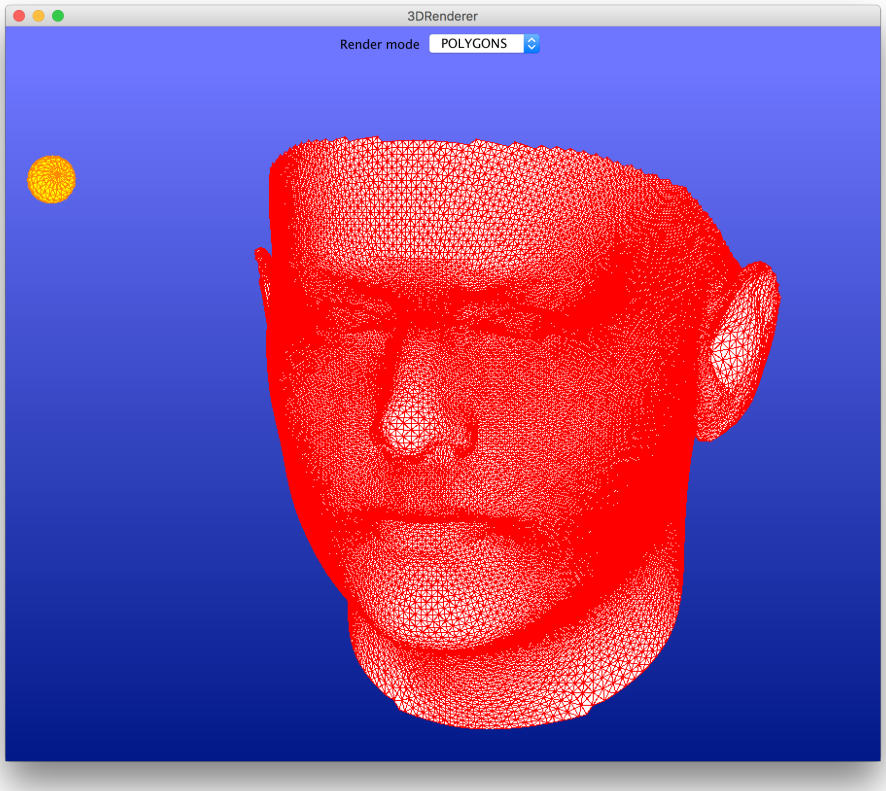

Figure 2: Polygons

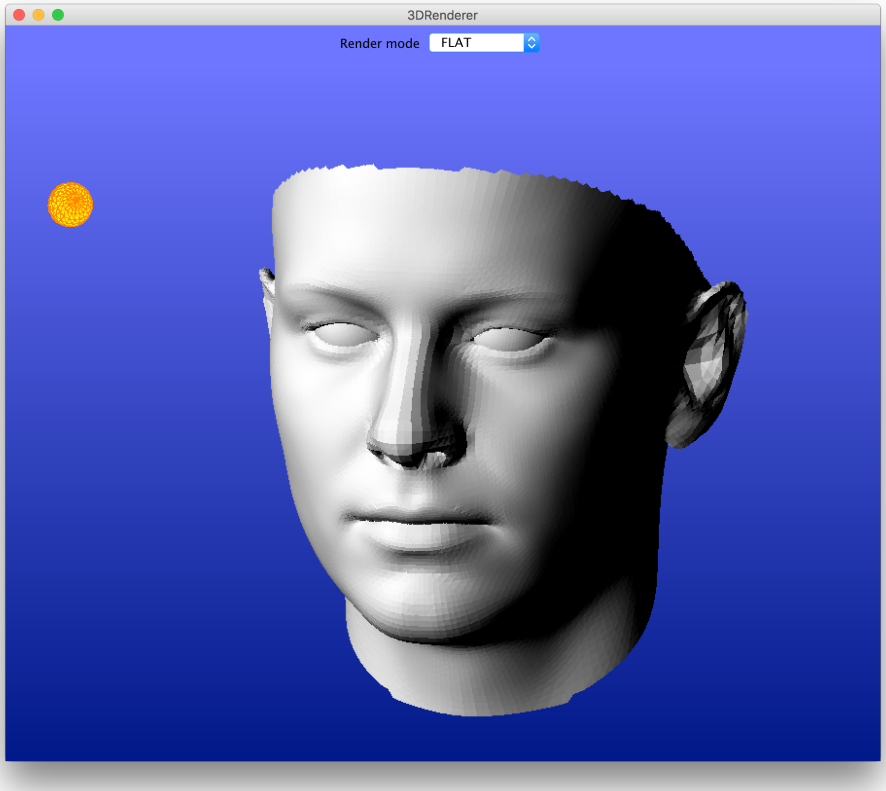

Figure 3: Flat rendering proportional to illumination and surface normal

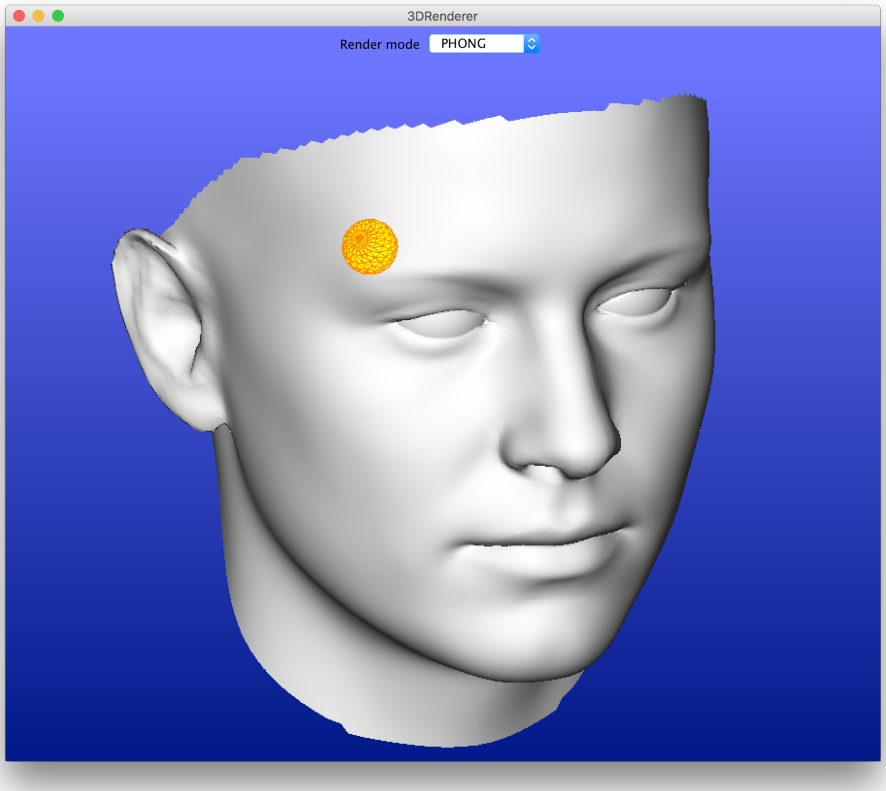

Figure 4: Phong rendering

Implementation

The implementation has required quite a lot of code across many Java classes, however we follow the usual Model, View, Controller pattern and the most important implementation details and algorithms reside in the RenderView and in Triangle2D.

```
public class RenderView extends JPanel {

    enum Mode { POLYGONS, FLAT, PHONG, Z_BUFFER, TEXTURE }

    private Mode mode = Mode.POLYGONS;
    private Scene scene = null;
    private Triangle2D[] triangles = null;
    private Rectangle modelBounds = null;
    private AffineTransform3D rotation = AffineTransform3D.identity()
            .rotatedBy(Math.PI, 0.0)
            .scaledBy(1.0,-1.0,1.0);
    private AffineTransform3D zoom = AffineTransform3D.identity();
    private AffineTransform3D scroll = AffineTransform3D.identity();

    ...
```

The scene contains all of the data parsed from the input files in model coordinates. It provides an iterator of all of the Triangle3D objects from the scene. The main task of the render view is to translate from these Triangle3D objects in model coordinates to an array of Triangle2D objects in screen coordinates. When the render view is called to paint itself it delegates to each Triangle2D's draw method.

```
    ...

    public void refreshTriangles() {
        Iterator<Triangle3D> it = scene.getTriangles();

        AffineTransform3D transform = AffineTransform3D.identity()
                .concatenateWith(scroll)
                .concatenateWith(zoom)
                .concatenateWith(rotation);

        Triangle2D[] triangles = new Triangle2D[scene.numTriangles];
        int i = 0;
        Triangle2D triangle2D;
        while (i < scene.numTriangles) {
            Triangle3D transformed = it.next().applying(transform);
            if (mode == Mode.FLAT) {
                triangle2D = flatMode(transformed, scene.getLightSource());
            } else if (mode == Mode.TEXTURE) {
                triangle2D = textureMode(transformed, scene.getLightSource());
            } else if (mode == Mode.PHONG) {
                triangle2D = phongMode(transformed, scene.getLightSource());
            } else {
                triangle2D = polygonMode(transformed);
            }
            triangles[i] = triangle2D;
            i++;
        }
        this.triangles = triangles;

        Rectangle bounds = null;
        for (int j = 0; j < triangles.length; j++) {
            Rectangle b = triangles[j].getBounds();
            if (bounds == null) {
                bounds = b;
            } else {
                Rectangle.union(bounds, triangles[j].getBounds(), bounds);
            }
        }
        modelBounds = bounds;
    }

    ...

    void renderModel(MyContext context) {
        Rectangle dirtyRect = new Rectangle(0, 0, getWidth(), getHeight());
        for (Triangle2D t : triangles) {
            t.drawInto(context);
        }
    }
}
```

Triangle2D objects may overlap each other and it is important to ensure that Triangle2D objects which should be occluded do not draw their pixels to the screen on-top of their occluder. Initially this was achieved by sorting the Triangle2D array by a z index for each triangle so that the front most triangle pixels would overwrite those that should be behind. However sorting is an $\mathcal{O}(nlogn)$ operation and this introduced significant lag when the face was rotated interactively. Therefore per pixel Z-buffering was implemented so that this sorting would be unnecessary. This works by storing the z value of the closest thing to the screen drawn at a certain pixel so that the next time that pixel is to be drawn the algorithm can check to see if there is already something closer draw at that pixel, in which case the pixel shouldn't be drawn. One of the render modes in the application allows the contents of the Z-buffer to be drawn to the screen (the z values are first normalised to colors where black is the furthest point and white is the closest).

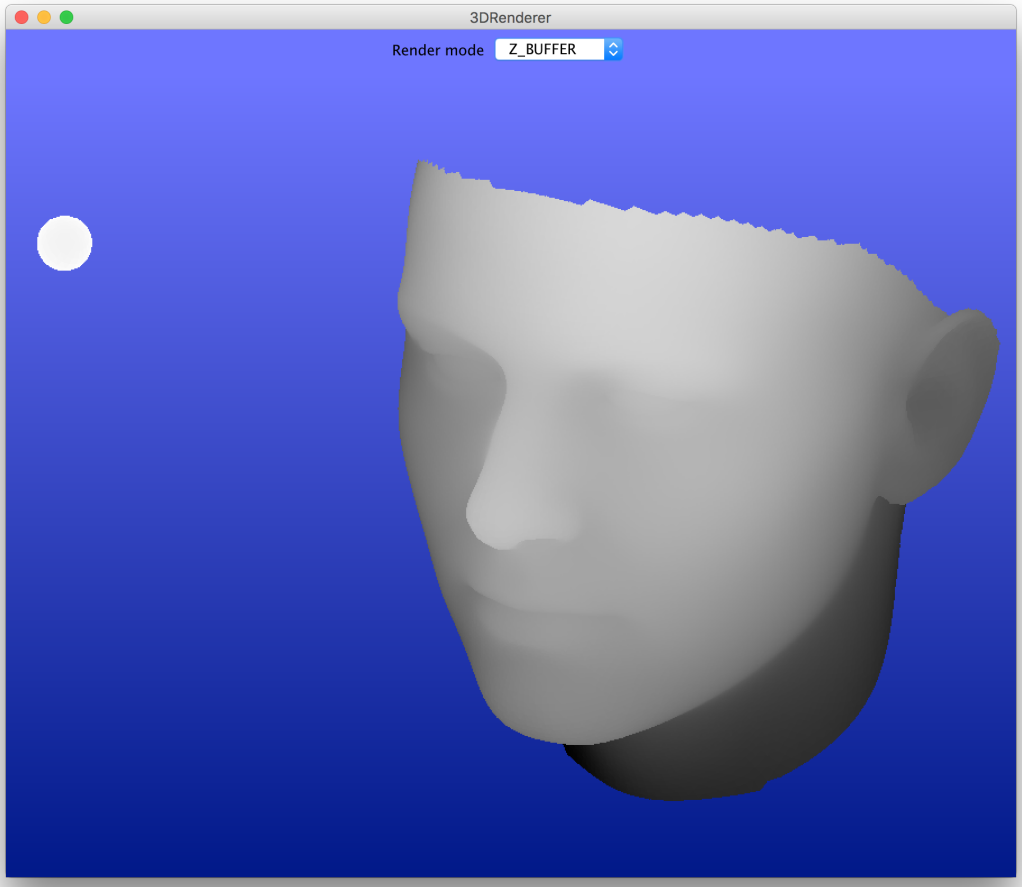

Figure 5: Z-buffer

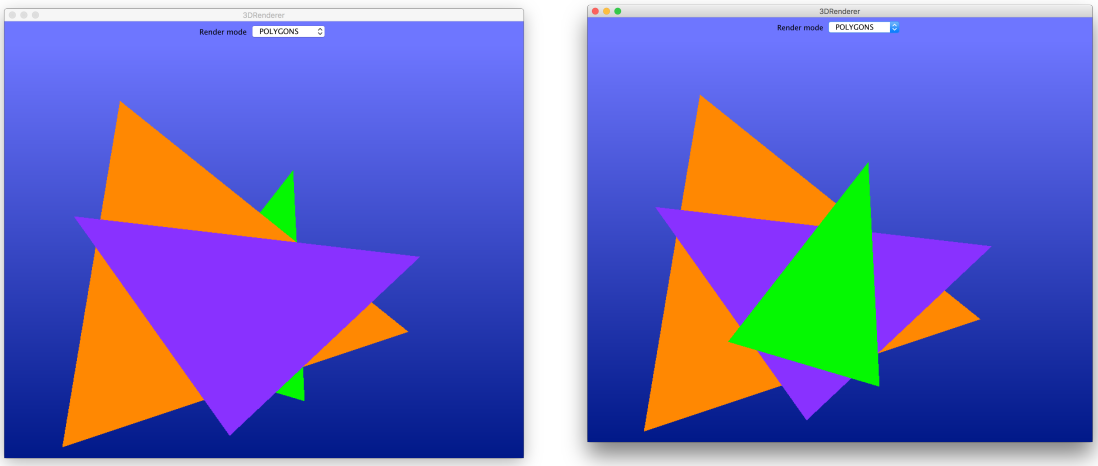

Figure 6: Z-ordering

By utilising z-buffering to avoid triangle sorting the performance is greatly improved and the face rotates smoothly with little to no lag.

Triangle rasterization

Each Triangle2D is responsible for drawing it pixels into the pixel buffer and so it needs an efficient algorithm to do this. This algorithm is truly a performance critical part of the implementation and so algorithm was designed very carefully to short circuit on simple calculations wherever possible and to pre-compute as much as possible when the Triangle2D is created. The crux of the algorithm relies on understanding the mathematical properties of projective coordinates.

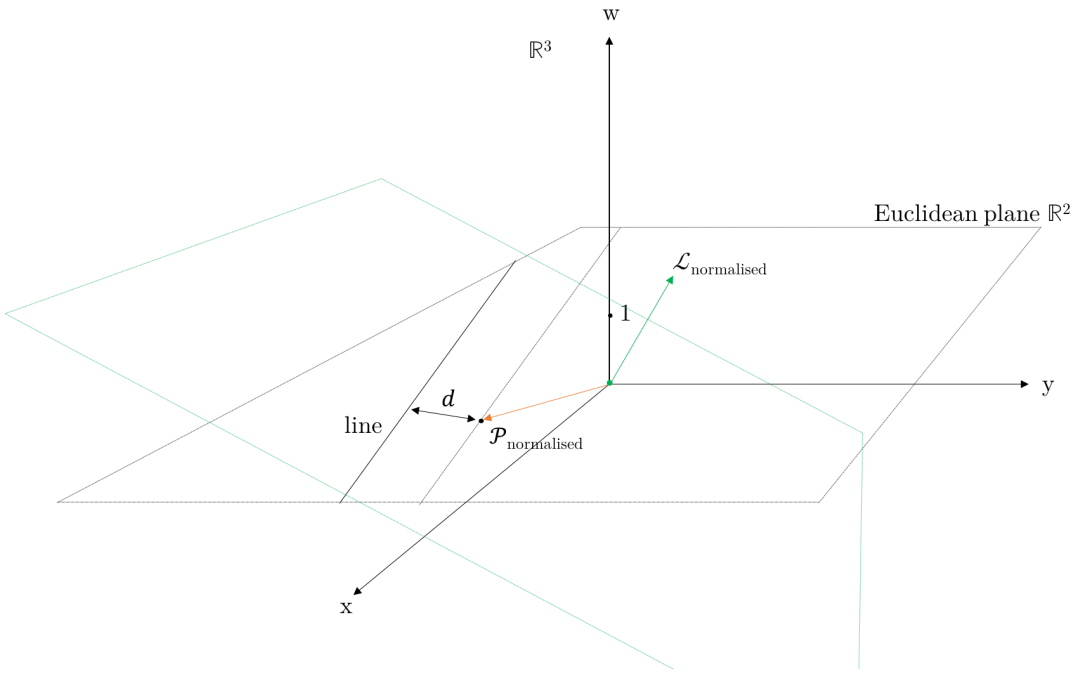

Figure 7: Projective space
[2]

In homogenous coordinate a 2d point is described by a vector P in a 3d projective space that pierces the 2d euclidean plane at that point. Since any vector in that direction corresponds to the same point we normalize by dividing each component by the w component, resulting in coordinates of the form $(x, y, 1)$ where x and y correspond to the euclidean position. A line is described by a vector L which defines a plane in the projective space that intersects the euclidean plane at the line. If this vector L is normalised by dividing through by $\sqrt{x^2 + y^2}$ then the line is in normal form and it has the following useful property for our algorithm, namely that the perpendicular distance of any point from the line is given by $d = \mathcal{L} \cdot \mathcal{P}$. Crucially the sign of this dot product indicates the side of the line that the point resides.

Our algorithm works as follows.

- When the Triangle2D is constructed pre-compute and store the lines representing the edges in normal from. These are calculated from $\mathcal{L} = \mathcal{P}_1 \times \mathcal{P}_2$ and then normalize.
- Pre-compute and store whether the vertexes are ordered in a clockwise or anti-clockwise sense.
- Pre-compute and store the rectangular bounds of the Triangle2D.
- When it comes time to draw the triangles first check and short circuit on whether the triangle vertexes are clockwise (precomputed) see fig 8.
- Then loop over the pixels in the intersection of the context bounds and the triangle bounds, this is a typically a relatively small number of pixels see 9.
- For each of the pixels inside the intersection box, calculate the three dot products with the three edge lines in normal form to get the signed distance from these lines. If all of these are negative then the pixel is inside the triangle. Moreover in the same calculation we can determine if the pixel is sufficiently close to the edge of the triangle that we may wish to color it red if we are in polygon render mode. Thus we are able to draw the edges of our triangles at no additional cost.

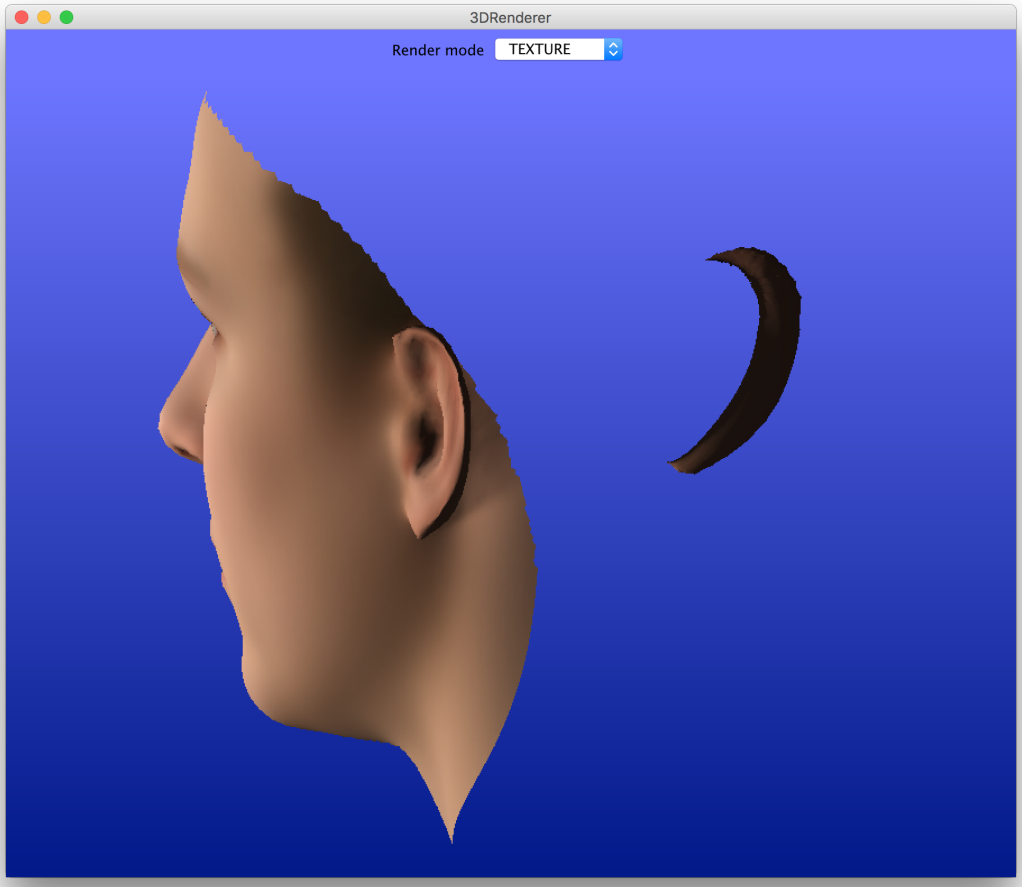

Figure 8: Backface culling by vertex anti-clockwise order

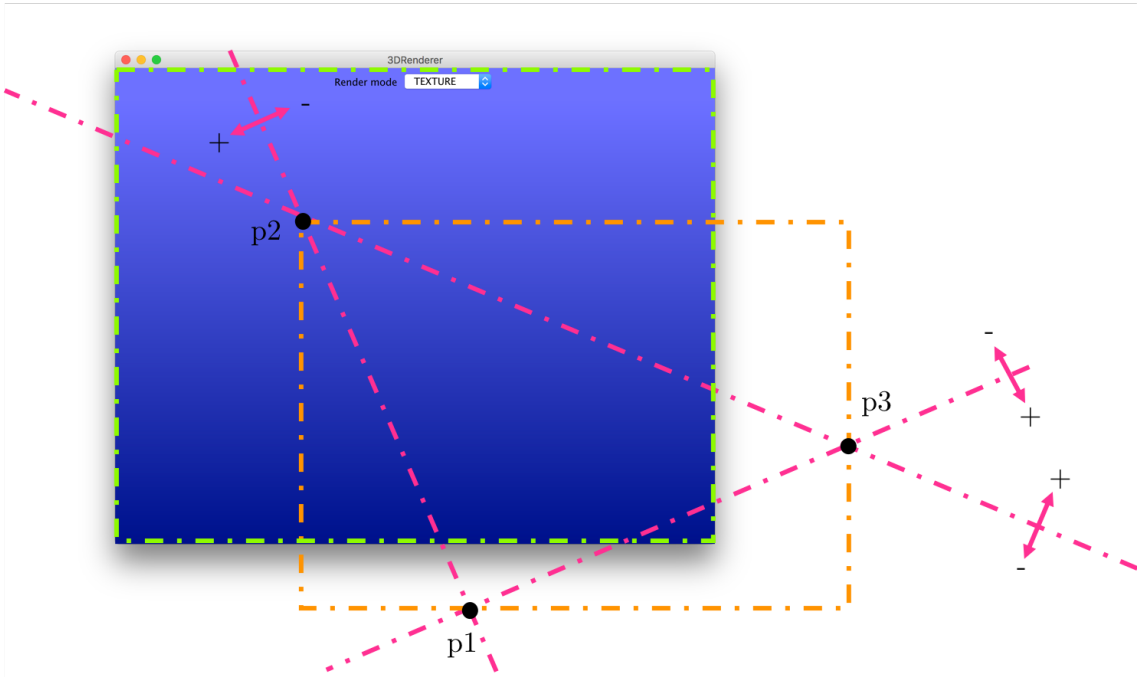

Figure 9: Triangle rasterization

Shading

We have just described the general form of our triangle rasterization algorithm. There are a number of different variants in our Triangle2D object corresponding to the different render modes and these introduce additional calculations. Notably the texture shading raterization algorithm combines barycentric interpolation of the vertex colors and also phong shading.

```
public class Triangle2D {
    ...
    private void textureDrawInto(MyContext context) {
        Rectangle intersect = context.bounds.intersection(bounds);
        int maxX = context.getWidth();
        int maxY = context.getHeight();

        for (int x = intersect.x; x <= (intersect.x + intersect.width) && x < maxX; x++) {
            for (int y = intersect.y; y <= (intersect.y + intersect.height) && y < maxY; y++) {
                PixelType pix = pixelType(x, y);
                if (pix == PixelType.INSIDE || pix == PixelType.EDGE) {
                    if (z < context.zBuffer.getBufferedZ(x, y)) {
                        Point2D p = new Point2D(x, y);
                        double brightness = phongBrightness(p);
                        double ambient = 0.15;
                        brightness = ambient + brightness * 0.85;
                        int rgb = interpolateColor(p);
                        int red = (int)(((rgb >> 16) & 0xFF) * brightness);
                        int green = (int)(((rgb >> 8) & 0xFF) * brightness);
                        int blue = (int)((rgb & 0xFF) * brightness);
                        rgb = red;
                        rgb = (rgb << 8) + green;
                        rgb = (rgb << 8) + blue;
                        context.pixels.setRGB(x, y, rgb);
                        context.zBuffer.setZ(x, y, z);
                    }
                }
            }
        }
    }

    double[] barycentricCoords(Point2D p) {
        Vector2D v2 = Vector2D.Subtract(p, points[0]);
        double d20 = Vector2D.Dot(v2, v0);
        double d21 = Vector2D.Dot(v2, v1);
        double[] bary = new double[3];
        bary[1] = (d11 * d20 - d01 * d21) * invDenom;
        bary[2] = (d00 * d21 - d01 * d20) * invDenom;
        bary[0] = 1.0d - bary[1] - bary[2];
        return bary;
    }

    double phongBrightness(Point2D p) {
        double[] bary = barycentricCoords(p);
        double dx = bary[0] * normals[0].dx + bary[1] * normals[1].dx + bary[2] * normals[2].dx;
        double dy = bary[0] * normals[0].dy + bary[1] * normals[1].dy + bary[2] * normals[2].dy;
        double dz = bary[0] * normals[0].dz + bary[1] * normals[1].dz + bary[2] * normals[2].dz;
        Vector3D normal = new Vector3D(dx, dy, dz).normalized();
        double brightness = Vector3D.dotProduct(directionToLightSource, normal);
        if (brightness < 0) {
            brightness = 0;
        }
        return brightness;
    }
}
```

In order to get Phong shading to work the shape data was processed to find the vertex normals for each triangle. This was achieved by constructing an adjacency dictionary with the key being the vertex and the value being the set of adjacent triangles, from which a vertex normal could be calculated as the average of the normals of the adjacent triangles.

Figure 10: Barycentric color interpolation

Above we see the barycentric algorithm interpolating the color across the triangles in the eye. The specific algorithm for calculating the barycentric coordinates is shown below, which is reported to be the most efficient [1].

```
// Compute barycentric coordinates (u, v, w) for
// point p with respect to triangle (a, b, c)
void Barycentric(Point p, Point a, Point b, Point c, float &u, float &v, float &w)
{
    Vector v0 = b - a, v1 = c - a, v2 = p - a;
    float d00 = Dot(v0, v0);
    float d01 = Dot(v0, v1);
    float d11 = Dot(v1, v1);
    float d20 = Dot(v2, v0);
    float d21 = Dot(v2, v1);
    float denom = d00 * d11 - d01 * d01;
    v = (d11 * d20 - d01 * d21) / denom;
    w = (d00 * d21 - d01 * d20) / denom;
    u = 1.0f - v - w;
}
```

Moving the light source

The light source is moved by holding down the arrow keys, and it always moves in a plane parallel to the screen. However the location of the light source needs to be determined in the model space for it to be compatible with the normals expressed in model space (it is not a good idea to try transform the normals as they do not transform under affine transformations the same way as points [3]). We therefore invert the model-to-scene transform so that we can express a translate a light position displacement vector from screen coordinates to model space.

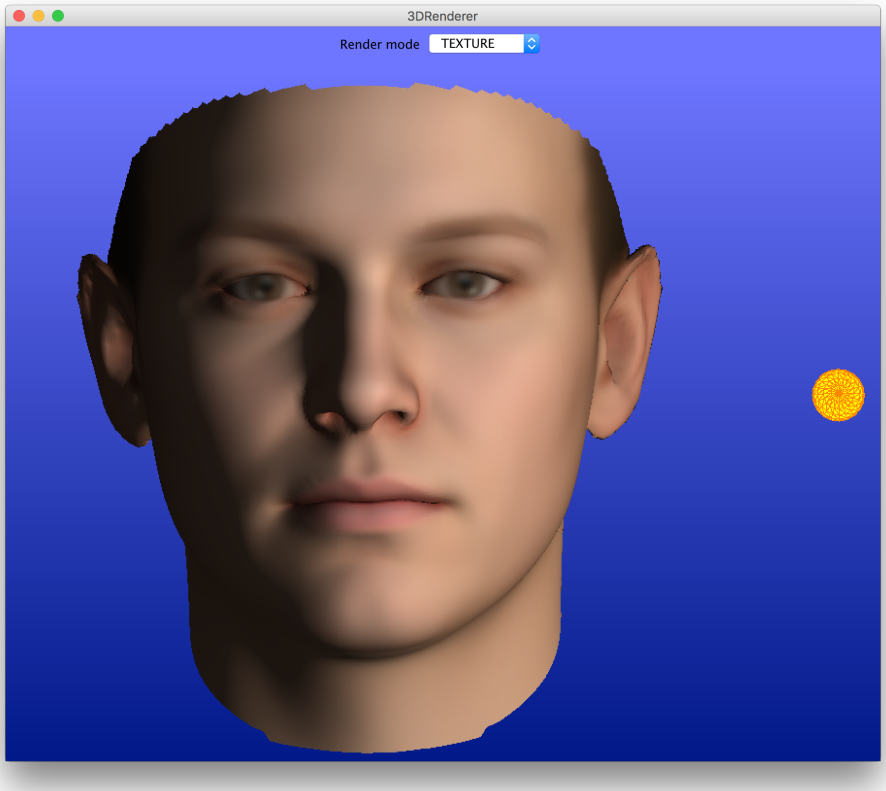

Figure 11: Moving the light source a

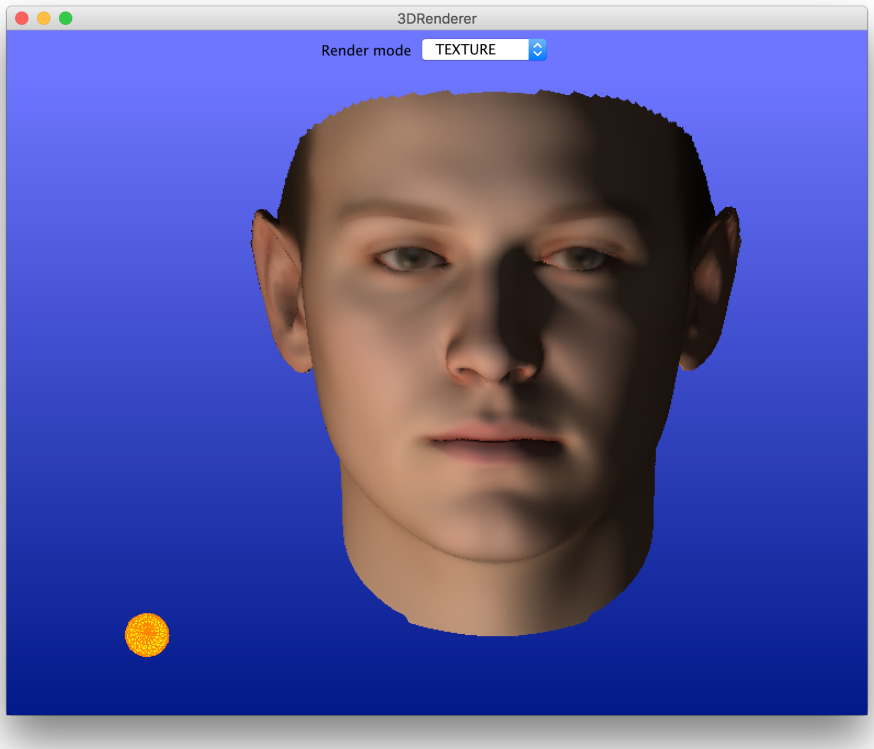

Figure 12: Moving the light source b

References

[1] Christer Ericson. *Real-Time Collision Detection*. 1st ed. CRC Press, 2004.

[2] Duncan Marsh. *Applied Geometry for Computer Graphics and CAD*. 2nd ed. Springer-Verlag, 2005.

[3] Peter Shirley. *Fundamentals of Computer Graphics*. 3rd ed. CRC Press, 2009.