The 3D Renderer

The goal of this assignment is to write a program that implements a simple but complete pipeline for rendering 3D shapes represented by polygons. This assignment is a lot larger than the previous two, you should provision your time accordingly.

Resources

The assignment webpage also contains:

- An archive of template code (including a GUI you may use) and a small example class.
- An archive of model files and images of what they look like.
- A marksheet.

To Submit

You should submit four things:

- All the source code for your program, including the template code. **Please make sure you do this**, without it we cannot give you any marks. **Again: submit all your `.java` files**.
- Any other files your program needs to run that *aren't* the data files provided.
- A report on your program to help the marker understand the code. The report should:
  - describe what your code does and doesn't do.
  - describe any bugs that you have not been able to resolve.
  - outline how you tested that your program worked.
  - describe any of the extensions you did and discuss the results.

  The report should be clear, but it does not have to be fancy – very plain formatting is all that is needed. It must be either a `txt` or a `pdf` file. It does not need to be long, but you need to help the marker see what you did.

A simple 3D rendering pipeline

3D graphics rendering ranges from fast approximate rendering, used for example in video games and other real time rendering, through to complex ray tracing and sophisticated modeling, for generating high quality images and animated films. There are a wide variety of data structures and algorithms used in 3D rendering.

For this assignment, you must write a program that performs a simple rendering of shapes represented by polygons, using a Z-buffer, using the algorithms described in the lectures.

The simplifications:

- All polygons are triangles.
- Only diffuse reflection will be required - no specular reflection.
- There will be a single directed light source, with a fixed intensity, placed at infinity in a specified direction (so that the incident light is from the same direction at every point on the shapes).
- The shapes will be automatically zoomed and translated to all fit in the image, so that no objects need to be clipped.
- The polygon surfaces are assumed to be flat and have a uniform reflectance so that the light reflected from every point on the polygon is the same (there is no need to interpolate the reflected light intensity or the surface normals).
- The coordinate system is assumed to have the origin at the top left x increases to the right, y increases down, z increases away from the viewer (this is a right handed system that is also convenient for converting to a 2D image).

Outline of your renderer

Your program should implement the pipeline described in the lectures:

- Read the lightsource direction and all the polygons from the file
- Rotate all the polygons so that the view direction is along the z-axis. Rotate the lightsource direction by the same amount.
- Compute the bounding box of the collection of polygons then translate and scale all the polygons so that they all fit within the image (ie, 0<=x<windowWidth and 0<=y<windowHeight).
- Mark all the polygons that are facing away from the viewer (ie, the Z component of their normal vector is positive) as hidden.

- Compute the normal and the reflected light intensity (as a `Color`) of every non-hidden polygon. The reflected light should be the sum of the reflected ambient light (independent of the normal of the polygon) and the diffuse reflected light, which depends on the angle between the polygon normal and the direction of the light source.
- Use a Z-buffer to render the image
  - initialise the Z-buffer (eg, gray background, and an infinite depth at each point)
  - For each non-hidden polygon
    - Compute the EdgeLists for the polygon, specifying the left and right values for x and z for each line of the image
    - Compute the z values of all the pixels on the polygon (by stepping through each row of the edgelists array, and interpolating from the left to the right values), for each x and y, putting the z value (and the color value of the light from the polygon) in the Z-buffer if the depth is less than the depth of the current value in the Z-buffer.
- Turn the array of colors in the Z-buffer into an image and save it to a file and/or display it in a window.

Some template code has been provided to get you started, along with a small test suite to help track down bugs. Using the template code is entirely optional, there are many ways to design a large program like this and you are welcome to take the approach you think is best. The test suite is just there to help you isolate which parts of your pipeline are broken, you **will not be marked** on whether tests pass or not.

Warning: Some tests might fail depending on how you've implemented the pipeline (especially how you handle the edge cases of edge lists). This doesn't necessarily mean your renderer is wrong.

Warning: The test suite is still new. If you think there are bugs in it please post on the forums.

Stages and marking

See the provided marking guide for a more detailed breakdown of the marks for different stages.

Stage 1 – Implement the rendering pipeline described above without steps 3 and 4, i.e. assume that viewer is already looking along the Z axis and all the objects are within the window.

- Read polygons and light source from file.

- Mark hidden polygons.
- Compute shading of each polygon, using both the ambient light (set by the user) and the positional light (from the file).
- Render with Z-buffer.

It can be quite hard to get all the glitches and off-by-one errors out of the edge lists and the z-buffer rendering. You can get most of these marks, even if there are still little problems with your rendering.

Stage 2 – Get the rendering correct!

Stage 3 – Extend your pipeline to include steps 3 and 4.

- Store a viewing direction and modify it when the users presses movement keys (arrows or WASD).
- Rotate the polygons (and the lightsource) so that the Z axis now corresponds to the direction of the viewer.

Stage 4 – Make some difficult improvements to the program.

- Extend your program to allow multiple light sources in different directions, with different colours and intensities. The computation of the reflected light will become a bit more complicated, and you will need to extend the `GUI.java` class to support this functionality, if you used it.
- Remove the assumption that the faces represent a shape with flat faces. The polygons should be viewed as an approximation to a curved surface, so that the reflected light ('shading') should not be uniform across each polygon, and should change smoothly at the edges. Accomplish this by implementing Gouraud shading:
  - At each vertex, compute the 'vertex normal' – the average of the normals of all the polygons meeting at that vertex. This is an approximation to the normal of the curved surface underneath the vertex.
  - Compute the reflected light intensity at that vertex using that normal.
  - Extend the EdgeLists computation to interpolate the reflected light intensity along the edges, as well as the z value. You will need to extend the EdgeLists to hold x, z, *and* shading values.
  - Extend the final rendering computation to interpolate the reflected light along the scanline from leftX to rightX, as well as interpolating the value of z.

  For a completely over-the-top assignment, use Phong shading instead of Gouraud (it interpolates the normals, along with the z value, then recomputes the shading at each point).

Data files

There are six files of polygons on which you can test your program:

- `tetras.txt` contains three tetrahedrons (4 triangles each) of about the same size and shape. The middle tetrahedron is all green (and upside down), so that the only difference between its faces should be due to the lighting. The other two have one green face each, but the other faces are different.
- `bigblocks.txt` contains two large blocks, which will need to be scaleed and translated to fit in the window.
- `shapes.txt` contains a collection of multicoloured blocks, some of which intersect with each other.
- `ball.txt` which is a large ball shape made out of 321 triangles. The faces are slightly different shades of blue/green
- `car.txt` is a old-timey car made out of 324 triangles, using 3 different colours.
- `monkey.txt` an ugly monkey head (courtesy of blender), in different shades of orange. (Almost 1000 polygons.)

Each polygon file has the direction of the light source on the first line specifed by a vector (three floating point numbers) pointing at the light source. The file has a polygon on every following line. Each polygon is specified by nine floating point numbers, specifying the x, y, and z of each of the three vertices (in order), followed by three integers specifying red, green, and blue reflectivity of the surface (0 = does not reflect at all, 255 = reflects completely). The numbers are separated by single spaces.

For each data file, there is also an image file showing what the object would look like when rendered (using the simple shading, not the challenge version).

Implementation hints

- Each polygon should be represented by three vertices, in order, such that the vertices are in anti-clockwise order when viewed from the outward side. That means that the cross product of the first edge (v2-v1) and the second edge (v3-v2) will be normal (right angles) to the polygon face, in the outwards direction.
- Each polygon will also have the reflectivity of the surface, represented as three numbers - the red, green, and blue reflectivity.
- You could use the provided java library files for representing and manipulating 3D vectors (`Vector3D.java`) and matrix transformations (`Transform.java`), if you find them helpful. *You do not need to use them.*

- We have provided a template GUI (`GUI.java`) that you can use to load files, display your render, and so on. It has three methods that need implementing: `onLoad`, `onKeyPress`, and `render`. You are free to use this class as-is, modify it, or ignore it.
- The `Scene`, `Polygon`, and `EdgeList` classes have been provided as data structures for the renderer. These have method stubs that you'll need to fill in.
- The `Pipeline` class provides method stubs for each step of the pipeline, which you should fill in and then combine together in the `Renderer` class.
- The provided test suite should help you isolate which parts of the pipeline are bugged. However, these tests were made with 'perfect rendering' in mind. It's possible to have a fully functioning renderer even with many tests failing.
- If using `GUI.java`, the values of the sliders used to control ambient light can be retrieved with the `getAmbientLight` method.
- If you are using the `Vector3D` class, then all floating point numbers should be `float`, not `double`. (Literal `float` constants need an 'f' on the end: `3.145f` is a `float`, `3.145` is a `double`.)
- Stick with a right hand coordinate system in which the x axis goes from left to right, the y axis goes from top to bottom, and the z axis goes from near to far. This will let you map from the model space to the image space trivially.
- You will need to compute a unit normal of each polygon in order to render it (but do this after rotating, translating, and scaling it).
- You should compute the bounding box of each polygon (max and min x and y, rounded to integers). You will need to do this after rotating the polygons in order to work out the translation and scaling. You will also need to do it again after the translation and scaling.
- I have provided sample code (in `ImageExample.java`) for turning a 2D array of Color values into a jpg image and then saving it, and displaying it. You don't need to use it, but you may find it helpful.
- Start with a minimal pipeline that does not rotate, translate, or scale the polygons (ie, leaves out steps 3 and 4 above).
- Include lots of debugging code (eg, that prints out the polygons and the edgelists) until you have the code working - ie, assume that you will get it wrong at first and write the debugging code as you go. It helps to write a toString method for each of your classes (there are examples in `Transform.java`, `Vector3D.java`).
- You are welcome to create your own shapes. If you know how to use Blender, you can create shapes, convert all the polygons to triangles, and then export to 'raw faces' to

get all the polygons. You would then need to add the colour numbers to the end of each line.