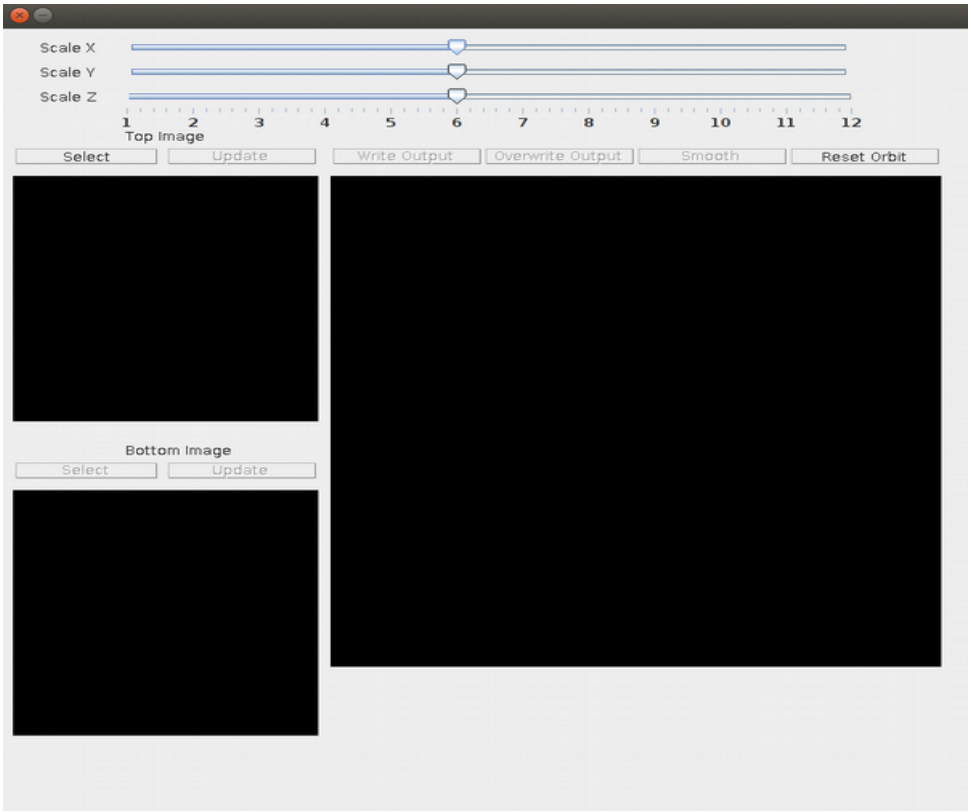
Criterion C: Development

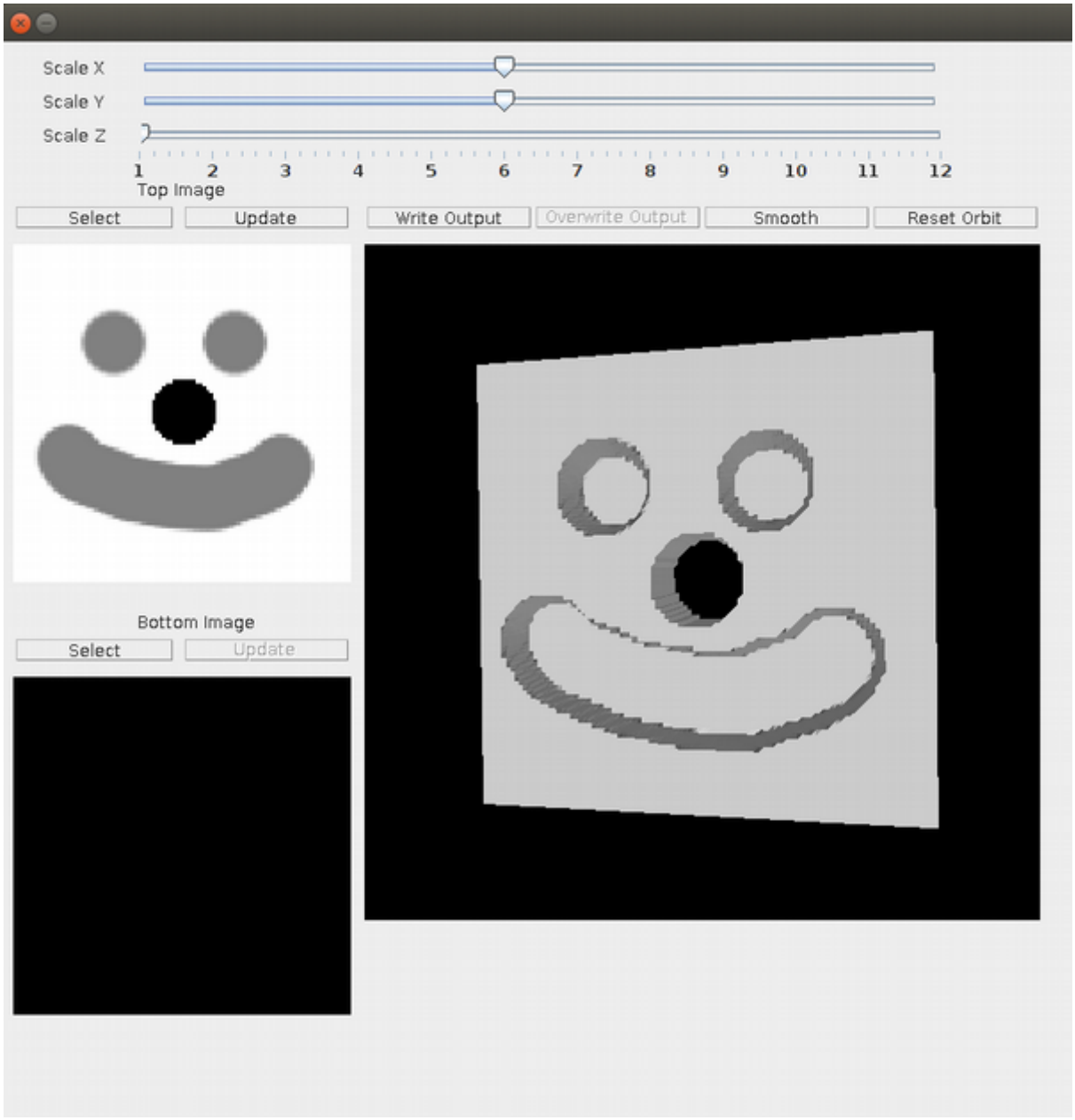
User Interface:

The user interface is implemented through the use of Java.awt framework with the use of Buttons and JSliders. Implementation of the interface was simplified by choosing not to allow resizing of the application. No layout manager was used. The constructor of the Runner class created and arranged all visual components. It also connected action methods of the buttons and sliders to those visual elements. The final GUI implementation was very close to the idea from the design document, with minor changes such that the model preview is easily viewable, and the buttons have a better layout.

Screenshot of program:



Screenshot with Image loaded:



Algorithmic Thinking:

Recompute

The recompute() method of the Model class is arguably the most important part of this whole project, as it is responsible for converting the image into the final 3D model. The algorithm was already written up in psydocode for the planning document, and only had to be modified minimally, to account for

overlap between top and bottom surfaces. An ArrayList is chosen to hold all the triangles here, such that adding them to the end is efficient, and memory allocation is dynamic. At the end of the method, the ArrayList is full of all triangles needed to display the object, and redundant triangles are removed.

```
1) Initialize a temporary 3D array of vertexes “varr” from arr:
        such that varr[x][y][i] == (x,y,arr[x][y][i])
        if z value of bottom is >= top
                Set both to new vertex that is average of the two //allows for permanently sealed solid
2) Create all triangles for the top & bottom.
  for col = 0 to colCnt-2
    for row = 0 to rowCnt-2
        Create triangle varr[col][row][0] , varr[col][row+1][0], varr[col+1][row][0]
        Create triangle varr[col][row+1][0] , varr[col+1][row+1][0], varr[col+1][row][0]
        Create triangle varr[col+1][row][1], varr[col][row][1] , varr[col][row+1][1]
        Create triangle varr[col+1][row][1], varr[col][row+1][1] , varr[col+1][row+1][1]
        Remove triangles if top and bottom are the same          // When no enclosed volume, there
should be a hole
3) Create all triangles for the left & right.
    for row = 0 to rowCnt-2
        Create triangle varr[0][row][1] , varr[0][row+1][1], varr[0][row][0]
        Create triangle varr[0][row+1][1] , varr[0][row+1][0], varr[0][row][0]
        Create triangle varr[colCnt-1][row][0] , varr[colCnt-1][row+1][0], varr[colCnt-1][row][1]
        Create triangle varr[colCnt-1][row+1][0] , varr[colCnt-1][row+1][1], varr[colCnt-1][row][1]
4) Create all triangles for the front & back.
    for col = 0 to colCnt-2
        Create triangle varr[col][rowCnt-1][0] , varr[col][rowCnt-1][1], varr[col+1][rowCnt-1][0]
        Create triangle varr[col][rowCnt-1][1] , varr[col+1][rowCnt-1][1], varr[col+1][rowCnt-1 [0]
        Create triangle varr[col][0][1] , varr[col][0][0], varr[col+1][0][1]
        Create triangle varr[col][0][0] , varr[col+1][0][0], varr[col+1][0][1]
5) Remove extras
    loop through all triangles and make sure they all have area, else remove them //artifact of
bottom>top
6) Call other methods
    call normalize to scale model to range of -.5 to .5
    call render to display model
```

Normalize

The normalize method in Model is used to scale and shift the model such that all vertexes are in the range of 0-1, This is done so that the model can easily be scaled up later to whatever size is set by the sliders. In this we decided to use a Hashmap to keep a directory of all Vertexes in the model, which makes sure that we do not scale and shift Vertexes multiple times, and it lets us quickly look

up and modify them. In the Vertex class there is also a method normalize, that will take in a shift factor and a scale factor and modify the method according to that. It was implemented in this way, such that we do not have to access the Vertex multiple times from the Model class, but instead only have one method call.

```java
/**
 * Finds the amount by which the vertexes need to be offset and scaled to be
 * between 0-1 Normalizes each vertex
 */
public void normalize() {
    Vertex max = new Vertex(Double.MIN_VALUE, Double.MIN_VALUE,
            Double.MIN_VALUE);
    Vertex min = new Vertex(Double.MAX_VALUE, Double.MAX_VALUE,
            Double.MAX_VALUE);
    HashMap<Vertex, Vertex> usedVerts = new HashMap<Vertex, Vertex>();
    for (Triangle t : myTriangles) {
        usedVerts.put(t.v1, t.v1);
        usedVerts.put(t.v2, t.v2);
        usedVerts.put(t.v3, t.v3);
    }
    Set<Vertex> theVerts = usedVerts.keySet();
    for (Vertex v : theVerts) {
        if (v.x > max.x)
            max.x = v.x;
        if (v.x < min.x)
            min.x = v.x;
        if (v.y > max.y)
            max.y = v.y;
        if (v.y < min.y)
            min.y = v.y;
        if (v.z > max.z)
            max.z = v.z;
        if (v.z < min.z)
            min.z = v.z;
        // find max and min
    }
    Vertex scale = new Vertex(1 / (max.x - min.x), 1 / (max.y - min.y),
            1 / (max.z - min.z));

    // calculate shift and scale
    for (Vertex v : theVerts) {
        v.normalize(min, scale);
    }
}
```

Code From: Model.java Method: normalize()

```java
/**
 * @param shift sets the offset amount
 * @param scale sets the amount to scale by
 */
public void normalize(Vertex shift, Vertex scale){
    this.x -= shift.x;
    this.y -= shift.y;
    this.z -= shift.z;
    this.x *= scale.x;
    this.y *= scale.y;
    this.z *= scale.z;

}
```

Code From: Vertex.java Method: normalize()

Smoothing

It was decided that smoothing would be implemented, even though we had not thought about it in the original planning, because due to the limitations of images, one can only have a max depth of 255, while the height and length could be any size determined by the picture. This allows us to remove some of the steps that might be seen when there are only 255 levels, and in general gives a cleaner output. IMPORTANT NOTE: The algorithm results in a model composed of triangles that completely enclose a volume with no self intersections and no openings. And any two adjacent triangles share vertex objects. This feature of sharing vertex objects is critical because it allows us to move vertexes easily without the risk of creating openings in the model surface. This was kept in mind specifically and designed and implemented such that we would be able to always be sure of this.

```
private void smooth() {
    HashMap<Vertex, ArrayList<Vertex>> adjacencies = new HashMap<Vertex, ArrayList<Vertex>>();
    for (Triangle t : myTriangles) {
        if (!adjacencies.containsKey(t.v1)) {
            ArrayList<Vertex> temp = new ArrayList<Vertex>();
            temp.add(t.v2);
            temp.add(t.v3);
            adjacencies.put(t.v1, temp);
        } else {
            ArrayList<Vertex> temp = adjacencies.get(t.v1);
            if (!temp.contains(t.v2))
                temp.add(t.v2);
            if (!temp.contains(t.v3))
                temp.add(t.v3);
        }

        if (!adjacencies.containsKey(t.v2)) {
            ArrayList<Vertex> temp = new ArrayList<Vertex>();
            temp.add(t.v1);
            temp.add(t.v3);
            adjacencies.put(t.v2, temp);
        } else {
            ArrayList<Vertex> temp = adjacencies.get(t.v2);
            if (!temp.contains(t.v1))
                temp.add(t.v1);
            if (!temp.contains(t.v3))
                temp.add(t.v3);
        }

        if (!adjacencies.containsKey(t.v3)) {
            ArrayList<Vertex> temp = new ArrayList<Vertex>();
            temp.add(t.v2);
            temp.add(t.v1);
            adjacencies.put(t.v3, temp);
        } else {
            ArrayList<Vertex> temp = adjacencies.get(t.v3);
            if (!temp.contains(t.v2))
                temp.add(t.v2);
            if (!temp.contains(t.v1))
                temp.add(t.v1);
        }
    }
    HashMap<Vertex, Vertex> replacements = new HashMap<Vertex, Vertex>();
    for (Vertex v : adjacencies.keySet()) {
        replacements.put(v, Vertex.getAverage(adjacencies.get(v)));
    }
    for (Vertex v : replacements.keySet()) {
        v.deepCopy(replacements.get(v));
    }
    render(myTriangles);
}
```

Code From: Model.java Method: smooth()

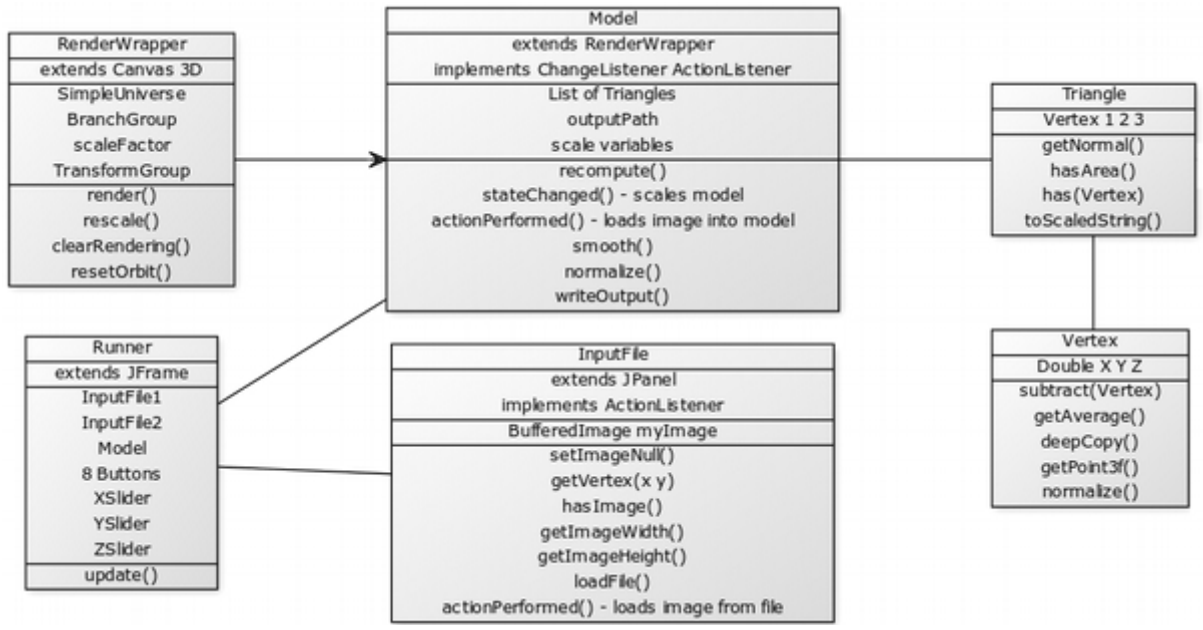
Data Structures:

RenderWrapper and Model

While the 3D preview could have also been implemented in Model, it was decided that we would separate this out into its own class, such that the 3D could be used in future projects, as well as simplifying and splitting off the program. RenderWrapper is a Canvas3D and is set up so that it creates and formats a Universe, which is a part of the Java3D library, and then displays it in the canvas. This Class takes care of all of the rotation, scaling, lighting, and displaying of the 3D model. RenderWrapper also contains all the code using Java3D, such that nowhere else in the program this is required. Model is a RenderWrapper, and to display the model that is created by recompute(), it only has to call render() and pass it a list of triangles. This greatly simplifies code and makes it more easy to understand, but also more maintainable and extensible.

```
public void render(ArrayList<Triangle> t) {
    group.detach();
    group.removeAllChildren();

    TriangleArray triArray = new TriangleArray(t.size() * 3,
            TriangleArray.COORDINATES);
    Vector3f[] normals= new Vector3f[t.size()*3];

    for (int i = 0; i < t.size(); i++) {
        triArray.setCoordinate(3 * i, t.get(i).v1.getPoint3f());
        triArray.setCoordinate(3 * i + 1, t.get(i).v2.getPoint3f());
        triArray.setCoordinate(3 * i + 2, t.get(i).v3.getPoint3f());
        normals[3*i]= new Vector3f(t.get(i).getNormal().getPoint3f());
        normals[3*i+1]= new Vector3f(t.get(i).getNormal().getPoint3f());
        normals[3*i+2]= new Vector3f(t.get(i).getNormal().getPoint3f());
    }

    GeometryInfo geometryInfo = new GeometryInfo(triArray);
    geometryInfo.setNormals(normals);
    NormalGenerator ng = new NormalGenerator();
    ng.generateNormals(geometryInfo);

    GeometryArray result = geometryInfo.getGeometryArray();
    Shape3D shape = new Shape3D(result, appearance);

    tg.removeAllChildren();
    tg.setTransform(scaleXYZ);
    tg.addChild(shape);
    group.addChild(tg);
    universe.addBranchGraph(group);
}
```

Code From: RenderWrapper.java Method: render()



Final Class Diagram that shows relationships

Libraries used:

Java3D

Website: http://www.java3d.org/

Java3D was used to be able to display the triangles we had generated to to our model, without having to write a program for this ourselves. All code using this library is encapsulated by our RenderWrapper class. We used code samples from many different sources to achieve our goal of displaying and rotating the 3D object. The following list contains all sources and what we used from each. In this section I required extensive help from my teacher to reach a deeper understanding of the materials we were using.

Sources:

- http://www.java3d.org/samples.html - Pyramid
  - Appearances
  - Lighting
  - Adding triangles
  - Methods containing this code: Constructor, render()

- http://www.java3d.org/position.html
  - Create transforms
  - Apply transforms to groups
  - Methods containing this code: Constructor, rescale(), render(),
- http://stackoverflow.com/questions/17464571/rotating-a-3d-view-with-mouse-movements-with-a-fixed-camera
  - How to set up orbital camera
  - Methods containing this code: Constructor
- https://docs.oracle.com/cd/E17802_01/j2se/javase/technologies/desktop/java3d/forDevelopers/J3D_1_3_API/j3dapi
  - Capabilities
  - Reset orbit
  - Resetting rendering
  - Methods containing this code: all in RenderWrapper

All else is available under standard Java API.

Problems Encountered and Solutions:

Reversed normal vectors:

When computing the normals of each of the triangles, each triangle has two normal vectors (one on each side), so it is important to consider in which order one uses the points to calculate the normal, because the right-hand rule will determine which side the normal is on. Our program must create triangles that form an object (enclosing a volume). All calculated normals should point to the outside of the object. In the design phase, this was reversed for all sides but the bottom and had to be corrected in the final product. For each triangle created with incorrect normal the code was changed from:

Create triangle VertexA , VertexB, VertexC

to

Create triangle VertexC , VertexB, VertexA

Bottom surface above top surface:

Because we are able to use two images to create the models, one for the top and the other for the bottom, we must account for intersections between the top and bottom. This was originally not accounted for and the intersections would lead to unprintable files. To solve this we clipped be bottom portion to be always less than the top

```
// fill array with vertex info from images
for (int x = 0; x < varr.length; x++) {
    for (int y = 0; y < varr[x].length; y++) {

        // load from images, 0 indicates top surface, 1 indicates bottom
        varr[x][y][0] = Runner.inputFile1.getVertex(x, y);
        varr[x][y][1] = Runner.inputFile2.getVertex(x, y);

        // If z value for bottom is >= z value of top
        // Let both share one vertex that is average of the two
        if (varr[x][y][0].z <= varr[x][y][1].z) {

            Vertex temp = new Vertex(x, y,
                    (varr[x][y][0].z + varr[x][y][1].z) / 2);
            varr[x][y][1] = temp;
            varr[x][y][0] = temp;
        }
    }
}
```

Code From: Model.java Method: recompute()

Remove touching triangles:

The code above results in some top and bottom triangles that share vertices, thus creating sections of the model with zero thickness. To fix this problem; if triangles share the same vertexes, regardless of order we consider them equal. The triangles are only added if they are not the equal (See code snibit below).

```
// make triangles for top and bottom surfaces
for (int col = 0; col < varr.length - 1; col++) {
    for (int row = 0; row < varr[col].length - 1; row++) {

        // create temporary triangles for top and bottom to allow for comparison
        Triangle temp1, temp2, temp3, temp4;
        // create triangles for top
        temp1 = new Triangle(varr[col][row][0], varr[col][row + 1][0],
                varr[col + 1][row][0]);
        temp2 = new Triangle(varr[col][row + 1][0],
                varr[col + 1][row + 1][0], varr[col + 1][row][0]);
        // create triangles for bottom
        temp3 = new Triangle(varr[col + 1][row][1],
                varr[col][row + 1][1], varr[col][row][1]);
        temp4 = new Triangle(varr[col + 1][row][1],
                varr[col + 1][row + 1][1], varr[col][row + 1][1]);

        // only add triangles if top and bottom do not share same vertices
        if (!temp1.equals(temp3)) {
            myTriangles.add(temp1);
            myTriangles.add(temp3);
        }
        if (!temp2.equals(temp4)) {
            myTriangles.add(temp2);
            myTriangles.add(temp4);
        }
    }
}
```

Code From: Model.java Method: recompute()

Triangles with no area:

The above code is still incomplete. When calculating the edge triangles that meet portions of the model with zero thickness some triangles will be created without any area. To fix this we remove such triangles in a post processing phase.

```
/**
 * @return true if none of the vertices are the same
 */
public boolean hasArea(){
    if (v1.equals(v2))return false;
    if (v1.equals(v3))return false;
    if (v3.equals(v2))return false;
    return true;
}
```

Code From: Vertex.java

```
// Some of the right, front, left, and back triangles have zero area.
// Remove them.
for (int i = myTriangles.size() - 1; i >= 0; i--) {
    Triangle t = myTriangles.get(i);
    if (!t.hasArea())
        myTriangles.remove(t);
}
```

Code From: Model.java Method: recompute()

Word Count: 1101