# Programming Assignment 3: Rasterization
# Computer Graphics, Fall 2016

Submission deadline: Thursday, November 3rd, 11:59

You will have to submit your solution for this assignment before Thursday, November 3rd, 11:59 to ILIAS. Additionally, you will have to present your solution to one of the assistants. Please register for a time slot on ILIAS. Please create separate Eclipse projects or packages for the individual tasks in order to be able to demonstrate them separately. You may for example copy and extend the project "simple."

In this assignment, you will implement your own software rasterizer. The provided code is designed such that the formerly used OpenGL renderer can be exchanged easily by a software renderer implemented in Java. In order to do this, you should derive your implementation of `RenderPanel` (e.g. `SimpleRenderPanel`) from `jrtr.SWRenderPanel` instead of `jrtr.GLRenderPanel`. If you launch the application with this setting, you should see a black window. At this point, you are ready to start with your implementation.

## 1 Render triangle vertices (2 points)

In a first step, the vertices of the triangles should be projected and drawn correctly to the 2D image. Implement the method `SWRenderContext.draw` such that every vertex is transformed to image coordinates. Set the color of the corresponding pixel to white.

The output image is stored in an object of type `java.awt.image.BufferedImage` and is accessible via the instance variable `colorBuffer`. You can set the color value of a particular pixel using the method `BufferedImage.setRGB`. For the viewport transformation, note that the pixel $(0, 0)$ in the `BufferedImage` is at the top left instead of the bottom left corner.

Render some test images and compare them to the OpenGL version by replacing `SWRenderPanel` by `GLRenderPanel`.

## 2 Rasterization and Z-buffering (4 points)

Implement the procedure discussed in the lecture to rasterize triangles using homogeneous 2D coordinates. Use a Z-buffer to solve the visibility problem. To draw single triangles, you first have to collect the data of 3 vertices from the `VertexData` structure. The following code snippet gives you an idea of how to do this.

```
// Variable declarations
VertexData vertexData;
LinkedList<VertexData.VertexElement>;
int indices[];
float[][] colors;
Matrix4f t;

// Skeleton code to assemble triangle data
int k = 0; // index of triangle vertex, k is 0,1, or 2

// Loop over all vertex indices
for(int j=0; j<indices.length; j++)
{
  int i = indices[j];

  // Loop over all attributes of current vertex
  ListIterator<VertexData.VertexElement> itr =
    vertexElements.listIterator(0);
  while(itr.hasNext())
  {
    VertexData.VertexElement e = itr.next();
    if(e.getSemantic() == VertexData.Semantic.POSITION)
    {
      Vector4f p = new Vector4f
        (e.getData()[i*3],e.getData()[i*3+1],e.getData()[i*3+2],1);
      t.transform(p);
      positions[k][0] = p.x;
      positions[k][1] = p.y;
      positions[k][2] = p.z;
      positions[k][3] = p.w;
      k++;
    }
    else if(e.getSemantic() == ...
    // you need to collect other vertex attributes (colors, normals) too

    // Draw triangle as soon as we collected the data for 3 vertices
    if(k == 3)
    {
      // Draw the triangle with the collected three vertex positions, etc.
      rasterizeTriangle(positions, colors, normals, ...);
      k = 0;
    }
  }
}
```

As discussed in the lecture, first find the bounding box of the triangle (restricted to the 2D image area). Within the bounding box, compute the edge functions for each pixel and use them to find out if the pixel lies within the triangle. Using the Z-buffer, decide whether the pixel has to be drawn or not. For the Z-buffer, use $\frac{1}{w}$. For each drawn pixel, find the perspectively correct interpolated color from the colors of the 3 vertices of the triangle.

## 3 Texture mapping (4 points)

The aim of this task is to extend your software rasterizer to include texture mapping. First, you have to make some preparations. In the class `jrtr.SWTexture`, implement the method `load` that loads an image as a texture. This can be done using the following code:

```
BufferedImage texture;
File f = new File(fileName);
texture = ImageIO.read(f);
```

You can access the pixels of the texture through the methods in `BufferedImage`, similarly to what you do in the sofware rasterizer. In order to assign textures to single objects in the scene, store a reference to a `Texture` object in a `Material` object. 3D objects that are implemented using the class `Shape` can store a reference to a desired object of type `Material`. Whenever you rasterize a `Shape`, you can access the `Texture` via the `Material` of the `Shape`.

Extend your rasterizer such that at each pixel the texture coordinates are interpolated perspectively correct. The texture coordinates are then used to read the corresponding color value from the texture and assign it to the pixel. First, implement nearest-neighbor interpolation, then bilinear interpolation.

To render textures, your objects need to have suitably defined texture coordinates. For example, a square (made out of 2 triangles) with texture coordinates can be generated as follows:

```
float v[] = {-1,-1,1, 1,-1,1, 1,1,1, -1,1,1};
float t[] = {0,0, 1,0, 1,1, 0,1};
VertexData vertexData = renderContext.makeVertexData(4);
vertexData.addElement(v, VertexData.Semantic.POSITION, 3);
vertexData.addElement(t, VertexData.Semantic.TEXCOORD, 2);
int indices[] = {0,2,3, 0,1,2};
vertexData.addIndices(indices);
```

Keep in mind that texture coordinates $(0,0)$ and $(1,1)$ should by convention correspond to the bottom left and the top right corner of the texture. This means you have to accordingly convert the interpolated texture coordinates when accessing the texture.

Demonstrate your texture mapping functionality with a suitable scene. For this, equip your torus, cylinder, the house scene from the last assignment, or your fractal landscape with texture coordinates.