Prof. Dr. F. Sadlo, D. Aumiller *Heidelberg, November 28, 2016*

Exercise Sheet 6

Assignment 6.1 Bresenham Algorithm [3 Points]

1. Rasterize the two given lines with the help of Bresenham's algorithm. For this, use the line equation $F(x,y) = y(x_1 - x_0) + x(y_0 - y_1) + y_1 x_0 - y_0 x_1$, with $F(x+1, y+0.5)$. Provide your steps and mark the pixels, that get filled by the algorithm.
2. What would the first line look like if it would be drawn using antialiasing? Provide your steps and the intensity for each pixel.

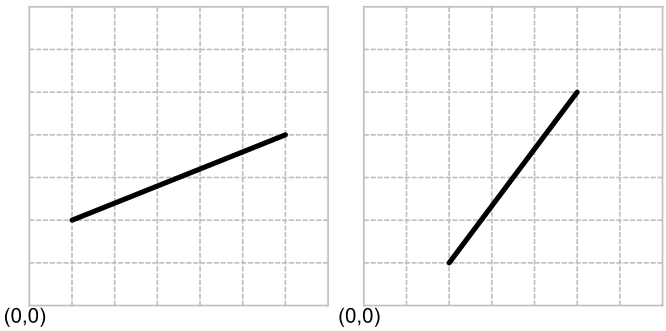

Figure 1: Line 1 (left) and Line 2 (right).

Assignment 6.2 Rasterization I: Transformations [9 Points]

Hint: You will not be able to use your own code from the last exercise, since this code brings a significant extension to the existing model!
Also make sure to upload the file `SimpleRasterizer.cpp` for each subtask.

On the next exercise sheets, we wil implement a complete 3D software rasterizer. As a basis, we will use the raytracer from the last exercises, with a few extensions. As a new base class for each scene object, the class `Raytracer::Scenes::SceneObject` is now used. This class provides (among others) the methods `SetGlobalTransformation()` and `GetGlobalTransformation()`, with which the global transformation from model to world coordinates can be set and requested. The method `GetGlobalToLocal()` provides the inverse transformation from world to model coordinates. In addition, one can transform an object relative to its parent object, with the help of:

- `GetTransformation()` and `SetTransformation()`
- `GetGlobalToLocal()`

- `GetPosition()` and `SetPosition()`
- `UpdateTransformations()`

In addition to that, the class `Mesh` is derived from the base class `SceneObject`, which can handle triangle meshes. Unlike the other exercises, we will not be using ray tracing, but instead convert the objects to the screen coordinates with the help of transformations. The main part of the rasterizer is handled by the class `SimpleRasterizer.cpp`. Therefore, hand in a different version of your file `SimpleRasterizer.cpp` for each of the three subtasks. Since there is currently no implementation for the drawing the lines (it will follow later), the function `DrawTriangle()` will only draw the corner points of the triangle. The result should look somewhat like the attached screenshot, which is provided with the skeleton.

1. **Model Transformation**
   Implement the function `SimpleRasterizer::RenderMesh`. Call the function `TransformAndLightTriangle()` for each triangle of the mesh. Provide it with the correct transformation matrix of the mesh and the normals. The used library *glm*[1] provides various matrix operations for this. Afterwards call `DrawTriangle()` for each triangle.
   Hint: The matrices are the same for all triangles in a mesh. They should therefore be generated only once.
2. **Camera Transformation, Projection Transformation**
   The camera and projection transformations are constant for one frame, thus they do not need to be recalculated for each mesh. Therefore, the calculation is performed in `SimpleRasterizer::Render`. They can be combined in one matrix (`SimpleRasterizer::viewProjectionTransform`). Create the camera transformation, the projection transformation, as well as the combination of both. Consider what kind of transformation you will need for the camera transformation (or how this is related to the global transformation of the camera). For the projection transformation, glm once again provides various functions to generate such transformation matrices[2], thus you do not need to fill the matrices by hand. Think about the kind of projection matrix you need, and use the respective values of the camera.
3. **Normalization Transformation, Viewport Transformation**
   Now implement `SimpleRasterizer::TransformAndLightTriangle()`. Apply the transformations you have calculated up to this point (`modelTransform`, `modelTransformNormals`, and `viewProjectionTransform`) to the vertices and normals of the triangle. Light the vertices in the appropriate coordinate system with `LightVertex()`. Finally, perform the normalization transformation and the viewport transformation.
   Hint: The normalized coordinates are in a coordinate system between $(-1, -1)$ and $(1, 1)$, meaning the origin is located in the center, and the coordinate values increase in an rightward or upward direction. The window coordinate system is between `(0,0)` and `(image.width, image.height)`, where the coordinates increase in rightward and downward direction.

Submission: Dezember 05, 2016, 6:00 pm via Moodle

[1] `https://glm.g-truc.net/0.9.4/api/a00133.html`
[2] `https://glm.g-truc.net/0.9.4/api/a00151.html`