Master 1 International Computer Science
Anastasia Kuznetsova, Luca Sciullo

Computer Graphics Project

Real-time hatching and line drawing

Part 1. Hatching rendering.

As a base for this part of the project we have chosen the tp3 lab – Textures.

Algorithm:

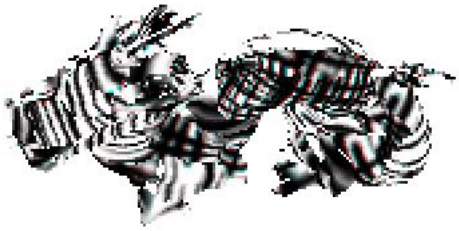
1. In tp2.cpp we have loaded and binded to the texture the provided hatches.bmp file in the function loadTexture();
2. We have send texture as myTextureSampler to the Fragment shader, and texture coordinates as textureuv to the Vertex Shader;
3. In the Vertex Shader we identified the distance to the camera as a Z coordinate of a view direction, and we send it to the Fragment Shader;
4. In the Fragment Shader we have calculated the diffuse value as a dot product between normal N and the light direction L
5. Since the diffuse value is between 0 and 1, we could directly define the intervals, depending on which to apply different parts of the provided texture.

```
if(diffuse < 0.33)
        x = 2; //third column of the texture
else if(diffuse < 0.66)
        x = 1; //second column of the texture
else
        x = 0; //first column of the texture
```

6. For the distance to the camera we had to extract 2 from the received value, to be able to map it to the [0;1] interval

```
if(distanceCamera-2 < 0.33 && distanceCamera-2 > 0)
        y = 0; //first row of the texture
else if(distanceCamera-2 < 0.66 && distanceCamera-2 >= 0.33 )
        y = 1; //second row of the texture
else
        y = 2; //third row of the texture
```

7. Finally we have added the bias, to make the hatches more defined and visible, and we combined the x coordinates for the diffuse term and y coordinates for the distance to the camera to obtain final texture and apply it to the color.



Picture 1. Dragon with hatching. Dark side, close to the camera – hatches are thin.

Picture 2. Dragon with hatching. Dark side, far from the camera – hatches are blurry.

Picture 3. Dragon with hatching. As seen from the bright side.

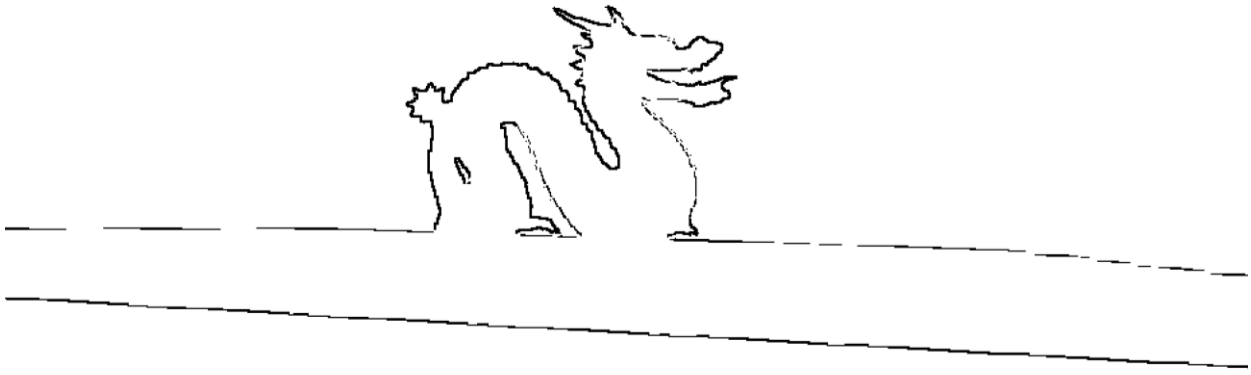
Part 2. Line drawing

As a base for the second part of the project we have chosen the tp4 – Shadows.

Algorithm:

1. We have rendered the depth map a seen from the camera and sent the rendered texture to the FragmentShader;
2. The Sobel edge detection algorithm required the width and height of the window, the radius of edge detection (the level of precision of edge detection), these values were sent as uniform from tp4.cpp directly in the FragmentShader;
3. To compute the colors for the plain according to the edges position, we have created the function simple_edge_detection(), which receive as an input the step of detection (x and y, width and height respectively) and the texture coordinates (fargmentUV) and returns the vec3 (color of the pixel);
4. Simple edge detection algorithm:
    a. For each pixel algorithm is applied on the pixels surrounding it in the given radius;
    b. Current location for the measurements of the intensity level are defined as a vec2 of central pixel and step in x and y;
    c. Current intensity value is computed as a sqrt(color.x+y+z^2);
    d. If current intensity level is smaller than center intensity -> increase the level of darkness, otherwise assign the current intensity value to the maximum intensity;
    e. Basing on the maximum intensity, center intensity, the level of darkness and the radius we assign black color to the pixels situated on the edges and white color otherwise.

The final result is provided by the VertexShader2DQuad and FragmentShader2DQuad, but we also tried to apply the edge detection algorithm on the 3Mesh (and we didn't succeed), the following code is provided in VertexShader3DMesh and FragmentShader3DMesh.



Picture 4. Dragon and bottom plane with edge-detection algorithm applied. View 1.

Picture 4. Dragon and bottom plane with edge-detection algorithm applied. View 2.

Picture 4. Dragon and bottom plane with edge-detection algorithm applied. View 3.