Bachelor Project

May 19, 2018

Barycentric Data Visualization for Triangle Meshes

Something

Costanza Volpini

Abstract

The usual way to visualize data for a triangle mesh is to associate the data with the vertices and then use linear interpolation over the mesh triangles. While this is the obvious way to go for data given at the mesh vertices, it is less natural for data given at the edges or triangles, since it requires to first aggregate the data neighbouring each vertex, thus introducing an additional averaging step. In this project we want to explore alternative data visualization techniques, using the power of barycentric coordinates and GPU programming.

Advisor Prof.Dr. Kai Hormann
Assistant Ph.D. Jan Svoboda

Advisor's approval (Prof.Dr. Kai Hormann): Date:

Contents

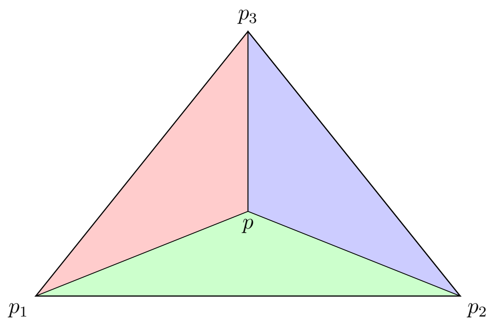

Figure 1: Let w_1 be the red area, w_2 the green one and w_3 the blue one. Normalizing each of them by the area of the triangle, we will get three values $(\lambda_1, \lambda_2, \lambda_3)$ that are the barycentric coordinates of p with respect to the triangle $[p_1, p_2, p_3]$.

1 Introduction

1.1 Barycentric coordinates

Barycentric coordinates, discovered by Mbius in 1827, consist of one of the most progessive area of research in computer graphics and mathematics thanks to the numerous applications in image and geometry processing. [7] The position of any point in a triangle can be expressed using a linear combination with three scalars using barycentric coordinates:

$$p = \lambda_1 p_1 + \lambda_2 p_2 + \lambda_3 p_3$$

where p_1, p_2 and p_3 are the vertices of a triangle and λ_1, λ_2 and λ_3 (the barycentric coordinates) are three scalars such that respect the following barycentric coordinates properties. [3]

- partition of unity: $\sum_{i=1}^{3} \lambda_i(p) = 1$
- reproduction: $\sum_{i=1}^{3} \lambda_i(p) p_i = p$
- Lagrange-property: $\lambda_i(p_j) = \delta_{i,j}$
- linearity: $\lambda_i \in \prod_1$
- non-negativity: $\lambda_i(p) \geq 0$ for $p \in [p_1, p_2, p_3]$

[5] The point is inside the triangle if $0 \leq \lambda_1, \lambda_2, \lambda_3 \leq 1$. If a barycentric coordinates is less than zero or greater than one, the point is outside the triangle. Barycentric coordinates allow to interpolate values from a set of control points over the interior of a domain, using weighted combinations of values associated with the control points (Fig. 1). [7]

1.2 Triangle meshes

A collection of triangles without any particular mathematical structure are called *triangle meshes* in many geometry processing algorithms. To derive a global parameterization for an entire triangle mesh we can define a 2D position for each vertex. Let be $\mathcal{M}$ a triangle mesh that consists of a geometric and topological component represented by a graph structure with a set of vertices $\mathcal{V} = \{v_1, ..., v_V\}$ and a set of triangular faces connecting them $\mathcal{F} = \{f_1, ..., f_F\}$ with $f_i \in \mathcal{V} \times \mathcal{V} \times \mathcal{V}$. The connectivity of a triangle mesh can be expressed in termes of the edges of the respective graph $\mathcal{E} = \{e_1, ..., e_E\}$ where $e_i \in \mathcal{V} \times \mathcal{V}$.

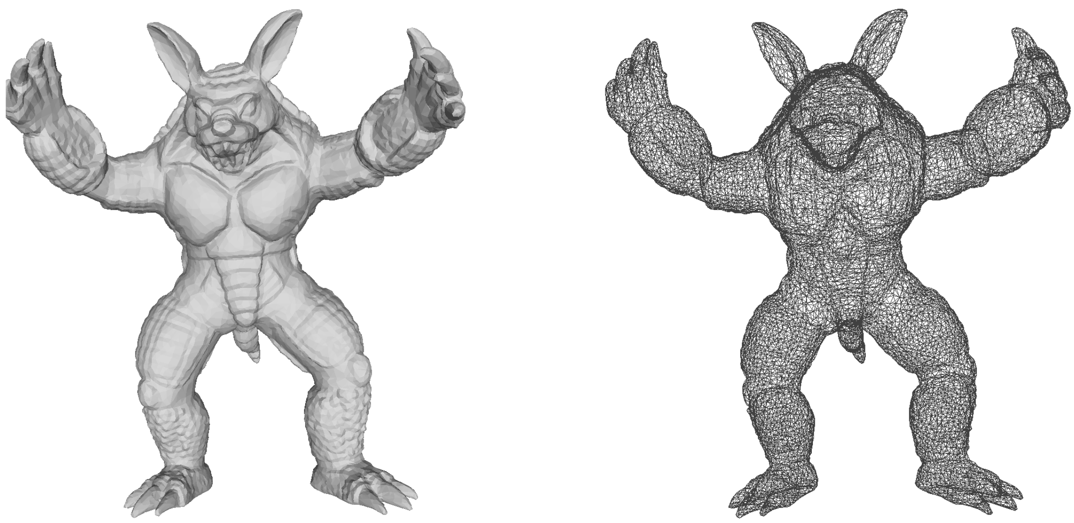

Figure 2: 3D triangle meshes.

1.3 Lighting - Phong lighting model

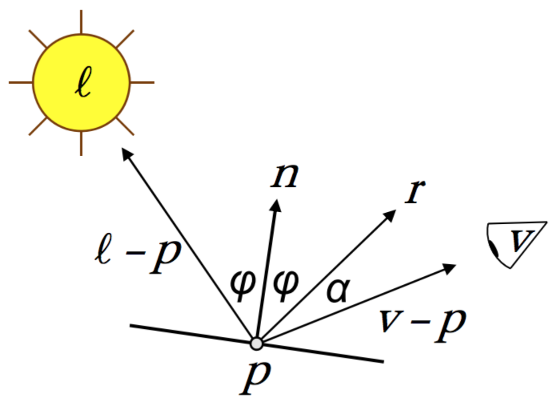

Figure 3: Lighting

Given a light source at position l with intensity I_l and a surface point at position p with normal n, we can define the angle between the incident light $(l - p)$ and the normal n as

φ. Let's bee our reflected light vector r defined as $r = 2n \cdot < n, l - p > -(l - p)$ and α the angle between that vector and the view direction $(v - p)$.

The *Phong lighting model* is defined as the sum of the self-emitting intensity, ambient term, diffuse reflection and specular reflection.

$$I = I_e + \rho_a \cdot I_A + \sum_{j=1}^{n} (\rho_d \cdot cos\varphi_j + \rho_s \cdot cos^k_{\alpha_j}) \cdot I_j$$

where I_e is the self-emitting intensity, ρ_a, ρ_d, ρ_s are the reflection constants (surface properties), n is the number of lights sources with intensities I_j and k is the shininess. [5]

1.4 Linear interpolation

Linear interpolation is an interpolation method where a linear interpolation is applied to get an equal spacing between the interpolated values. Given two numbers n_1 and n_2 (the start and final values of the interpolant), a linear interpolation can be made using a parameter t ($t \in [0, 1]$). [2]

$$n = n_1 + t(n_2 - n_1)$$

The standard linear interpolated visualisation is made passing three attributes (colors) for each vertex of a triangle. OpenGL will interpolate linearly the colors. That is possible thanks to the barycentric coordinates that will tell how much of each color is being mixed at any position.

1.5 Flat Shading

Flat shading is a way to compute the colour at each pixel (at a corner or the barycentre) using the triangle normal. Given a triangle $[p_1, p_2, p_3]$, the lighting is computed using the normal n[1] at $p = (p_1 + p_2 + p_3)/3$. This colour is used then for all pixels. Flat shading gives objects with flat facets. [5]

1.6 Gouraud Shading

Gouraud Shading is a way to compute the colour at each pixel assigning a normal to each corner and after having computed the color for each corner it linearly interpolates these colour values. Given a triangle $[p_1, p_2, p_3]$ and the normal at each corner n_1, n_2, n_3. The lighting is computed at p_i using normal n_i this applied to each corner return the colour values c_1, c_2, c_3. These colors are then linearly interpolate $c = \mu_1 c_1 + \mu_2 c_2 + \mu_3 c_3$. Gouraud shading gives objects that appear more curved. [5]

[1] $\widehat{n} = (p_2 - p_1) \times (p_3 - p_1)$ and then normalize it

1.7 Gaussian Curvature

The *Gaussian curvature* K is defined as the product of the principal curvatures (square of the geometric mean):

$$K = k_1 k_2$$

where k_1 and k_2 are the principal directions. A basic interpretation should be to imagine the *Gaussian curvature* like a logical `AND` since it will check if there is a curvature along both directions. The curvature of a surface is characterized by the principal curvatures, the *Gaussian curvature* and the *mean curvature* are simply averages of them. [1] Surfaces

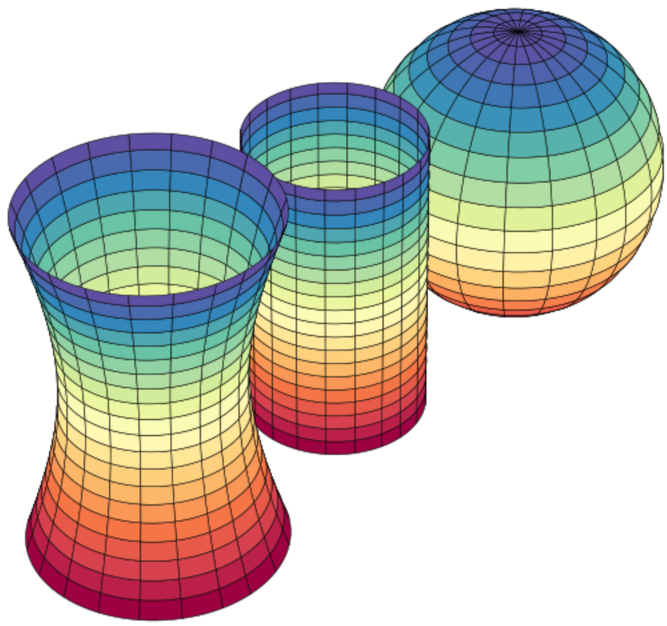

Figure 4: Negative curvature, zero curvature and positive curvature.

that have a zero gaussian curvature are called *developable surfaces* because they can be flattened out into the plane without any stretching. *Gaussian curvature* should be zero inside each mesh triangle and the same along edges since it can be flattened symmetrically into the plane by simply rotating one triangle about the common edge into the plane defined by the other. Consequently the *Gaussian curvature* is concentrated at the vertices of a triangle and defined as the *angle defect*

$$K(V) = 2\pi - \sum_{i=1}^{n} \theta_i$$

where θ_i are the angles of the triangle T_i adjacent to the vertex V at V. This should be seen as the integral of the Gaussian curvature over a certain region $S(V)$ around V, where these $S(V)$ form a partition of the surface of the entire mesh.

$$K(V) = \int_{S(V)} K dA$$

Negative curvature can be recognized by the fact that external directions curve in opposite directions, *zero curvature* has one external direction that has zero curvature, *positive curvature* has external directions that curve in the same direction (Fig. 4). The *Theorema Egregium*, discovered by C.F. Gauss in 1827, states that the *Gaussian curvature* is an intrinsic property of the surface that does not depend on the space, despite by the fact that is define as the product of the principal curvatures (whose value depends on how the surface is immersed in the space). Technically, the *Gaussian curvature* is invariant under isometries. We can then notice that triangle angles add up to less than 180° in negative curature, exactly 180° in zero curature, and more than 180° in positive curature. [4]

1.8 Mean Curvature

The *mean curvature* H is defined as the arithmetic mean of principal curvatures :

$$H = \frac{k_1 + k_2}{2}$$

where k_1 and k_2 are the principal directions. A basic interpretation should be to imagine the *mean curvature* like a logical `OR` since it will check if there is a curvature along at least one direction. [1] The *mean curvature* inside each mesh triangle is zero, but it does not vanish at edges. The *mean curvature* associated with an edge is defined as $H(E) = \| E \| \theta_E/2$ where $\theta_E/2$ is the signed angle between the normals of adjacent triangles (see Fig. 5).

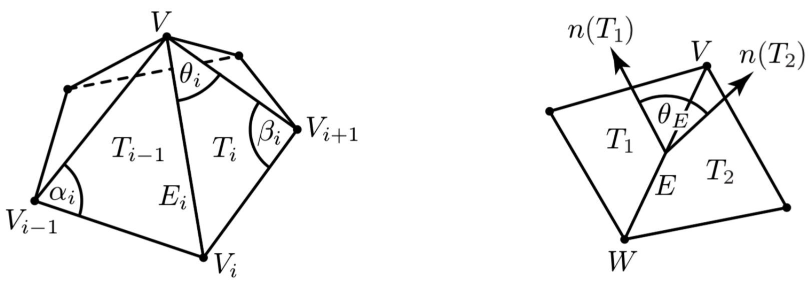

Figure 5: A vertex V with its neighbouring vertices V_i and adjacent triangles T_i. Angles opposite the edge E_i are denoted by α_i and β_i. The angle between the normals of adjacent triangles T_1 and T_2 with positive or negative sign is denoted as θ_E.

The *mean curvature* at a vertex V is defined as:

$$H(V) = \frac{1}{2} \sum_{i=1}^{n} H(E_i)$$

Averaging the mean curvatures of its adjacent edges guarantee that *mean curvature* of an edge is divide uniformly to both end points. [4]

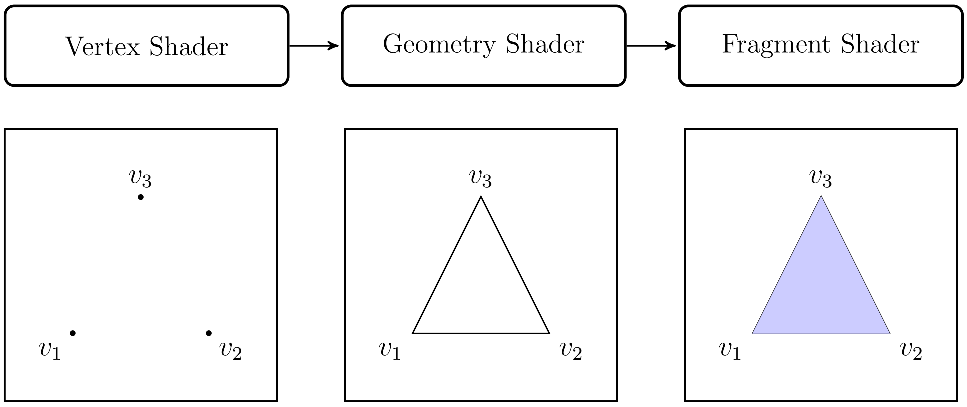

Figure 6: GPU pipeline

2 GPU program

A program that runs on GPU is called *shader*. Shaders are principally used to modify the representation and the behaviour of 3D objects. They are also used to create lighting effects. Shaders can perform tasks efficiently thanks to the GPU. That guarantee faster results than CPU since GPU is designed to work in parallel. These shader programs are written in *GLSL*.

2.1 GPU pipeline

A program will allow us to control the rendering pipeline since by default there is no pipeline set in OpenGL. It takes a set of vertices as input, then the *vertex shader* transforms them (translation, rotation, projection...) and passes the transformed vertices to the *geometry shader*. This shader takes vertices to create primitive shapes and then it rasterizes them. These rasterized flat images are then passed as input to the *fragment shader* that adds the lighting, apply textures and color these images (Fig. 6).

2.2 Vertex Shader

The program that perform vertex operations is called *vertex shader*. It receives one vertex at a time and then it passes the output to a *fragment shader* or to a *geometry shader*, if any.

2.3 Fragment Shader

Fragment shader performs color computation for every visible pixel of the rasterized object. It works on a fragment at a time, but thanks to the power of GPU it can work in parallel for all vertices (*vertex shader*) and fragments (*fragment shader*).

2.4 Geometry Shader

Geometry shader is used for layered rendering. It takes as input a set of vertices (single primitive, example: triangle or a point) and it transforms them before sending to the next shader stage. In this way, we can obtain different primitives. Each time we call the function `EmitVertex()` the vector currently set to `gl_Position` is added to the primitive. All emitted vertices are combined for the primitive and output when we call the function `EndPrimitive()`.

3 Vertex area based

The first goal is to extend the idea of flat shading from triangles to vertices and edges. The idea of flat shading is to draw all the pixels of a triangle with the same colour, and as such it is a natural way to visualize data given at the triangles (for example, the colour of an object, resulting from the lighting calculation using the triangle normal). The extension of this approach is to split the surface of the triangle mesh likewise into regions around vertices and edges and draw all pixels in these regions with the same colour, thus visualizing data given at the vertices or edges of the mesh in a piecewise constant, not necessarily continuous way, resembling the classical triangle flat shading. The aforementioned regions can easily be defined using barycentric coordinates and a simple GPU fragment program can be used to find out for each pixel to which region is belongs and which colour it should be painted with. Examples of vertex and edge data include discrete Gaussian and discrete mean value curvature. The results of this new technique should then be compared to the standard approach mentioned in the beginning.

3.1 Region around a vertex

We can split the surface of triangle meshes into regions around vertex (Fig. 8) and color them. These regions can be determined using barycentric coordinates and GPU fragment program. Visualizing data given at the vertices or edges of the mesh in a piecewise constant simulates the classical triangle flat shading. An example of this vertex data is the discrete Gaussian curvature.

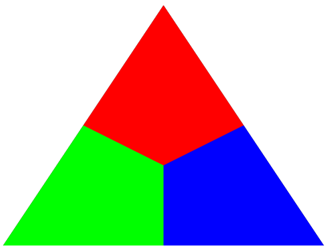

Figure 7: Vertex based area

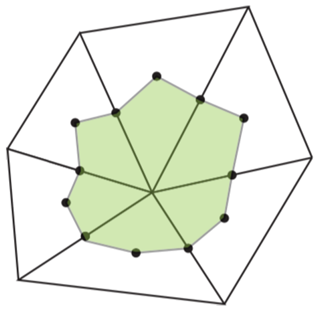

Figure 8: Region around a vertex

3.1.1 Max diagram - Vertex based area

Passing barycentric coordinates to the *fragment shader* will clearly demonstrate that we can get results different from the classic color interpolation. [6] There are different approaches to color interpolation focusing on the distance from vertices. For each point in a triangle, we can easily determine its closest vertex, which we use as a cue for coloring. A different approach from interpolating, can be found coloring vertex areas based on the minimum barycentric coordinate. The color is given by the region farthest from a vertex (Fig. 7, Pseudocode 1).

3.2 Flat shading extension

An extension of *flat shading* would be to have each vertex area to be in one constant color. This color can be taken from shading interpolation using the normal at the vertex and the vertex position. The color will then be computed as in *Gouraud shading*. The idea is to compute the color per vertex but instead of linearly interpolated it in each triangle (as *Gouraud shading* does) we color regions around a vertex with that constant color. To implement it, the barycentric coordinates, the vertex color, the normal at the vertex and the lighting calculations must be passed to the *fragment shader*. We want to avoid an automatic interpolation of colors, in order to return the resulting color using the *max diagram* we have used a *Geometry shader* that have access to all three vertex colors in *fragment shader*. (Pseudocodes: 2, 3, 4)

3.2.1 Comparison

Gouraud shading vs extension flat shading. TODO

3.3 Discrete Gaussian Curvature

Another interesting alternative data visualization technique is to compute the *Gaussian curvature* per vertex. That can be done summing up, for each vertex, angles at this vertex with adjacent triangles and then subtracting this value to 2π. After having obtain this value, called *angle defect* (Fig. 9), we map linearly this value to a color range. The resulting color will be the vertex flat shading visualisation of *Gaussian curvature*.

$$K(V) = 2\pi - \sum_j \theta_j$$

Gaussian curvature returns a constant color around each vertex (Fig. 10, Pseudocode 5). TODO: update image 3.4

3.4 Interpolated Gaussian Curvature

aka Gouraud Shading Gaussian Curvature TODO

3.4.1 Comparison

We want now to compare the *Gaussian curvature* (Fig. 11) with the *interpolated Gaussian curvature*. In Fig. 12 each vertex area is colored applying the method *max diagram*

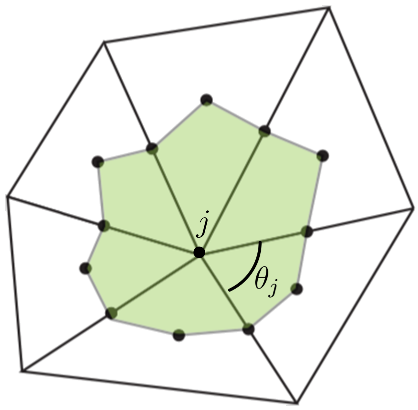

Figure 9: Angle defect

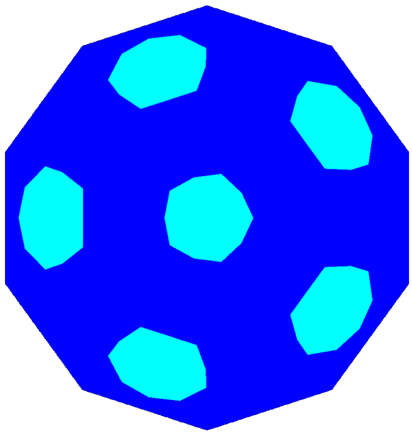

Figure 10: Gaussian curvature on icosahedron

described in the above subsection 3.1.1. Instead, in Fig. 11 the color is obtained with a linear interpolation.

Figure 11: Interpolated Gaussian curvature

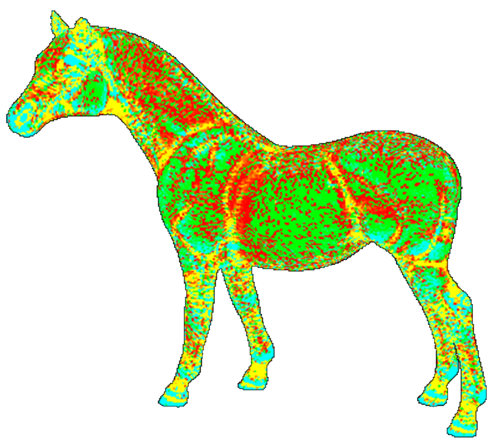

Figure 12: Gaussian curvature

Visualization of the principal curvatures of the model as colors from blue (highest values of curvature) to red (lower values of curvature), in both Fig. 12 and Fig. 11, better highlighs the geometry of the horse. These changes of curvature, positive (blue), flat (green) and negative regions (red), better emphasises the 3-dimensionality of the model. TODO

4 Results

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

5 Conclusions

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

6 References

[1] Discrete differential geometry. `http://brickisland.net/cs177/?p=144`.

[2] Maths for computer graphics. `https://nccastaff.bmth.ac.uk/hncharif/MathsCGs/Interpolation.pdf`.

[3] Scratchapixel 2.0. Ray tracing: Rendering a triangle. Scratchapixel 2.0 `https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/barycentric-coordinates`.

[4] K. Hormann. *Encyclopedia of Applied and Computational Mathematics*, chapter Geometry Processing, pages 593–606. Springer, Berlin, Heidelberg, 2015.

[5] K. Hormann. Slide of course: Computer graphics. iCorsi3.

[6] Amit Patel. Red blob games. `https://www.redblobgames.com/x/1730-terrain-shader-experiments`, Jul 2017. Accessed on 2018-03-03.

[7] J. Zhang, B. Deng, Z. Liu, G. Patanè, S. Bouaziz, K. Hormann, and L. Liu. Local barycentric coordinates, 2014.

A Pseudocodes

A.1 Chapter 3

TODO: CHECK AND UPDATE CODES

Listing 1: Max diagram - Vertex based area (Section: 3.1.1)

```
if (Coords.x > Coords.y && Coords.x > Coords.z) {
    vec3 blue = vec3(0.0f, 0.0f, 1.0f);
    FragColor = vec4(blue, 1.0f);
} else if(Coords.y > Coords.x && Coords.y > Coords.z) {
    vec3 green = vec3(0.0f, 1.0f, 0.0f);
    FragColor = vec4(green, 1.0f);
} else {
    vec3 red = vec3(1.0f, 0.0f, 0.0f);
    FragColor = vec4(red, 1.0f);
}
```

Listing 2: Vertex Shader for flat shading extension using lighting (Section: 3.2)

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec3 aColor;

struct Light {
    // ...
};

out vec4 vertex_color;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
    vec3 world_position = vec3(model * vec4(aPos, 1.0));
    vec3 world_normal = mat3(transpose(inverse(model))) *
        aNormal;

    // color obtained with lighting calculations
    vertex_color = get_result_color_lighting(...);

    gl_Position = projection * view * model * vec4(aPos, 1.0)
        ;
```

```
    }
```

Listing 3: Geometry Shader for flat shading extension (Section: 3.2)

```
#version 330 core
layout (triangles) in;
layout (triangle_strip, max_vertices = 3) out;

in vec4 vertex_color[3];
out vec3 coords;
out vec4 wedge_color[3];

void main() {
    wedge_color[0] = vertex_color[0];
    wedge_color[1] = vertex_color[1];
    wedge_color[2] = vertex_color[2];

    coords = vec3(1.0, 0.0, 0.0);
    gl_Position = gl_in[0].gl_Position;
    EmitVertex();

    coords = vec3(0.0, 1.0, 0.0);
    gl_Position = gl_in[1].gl_Position;
    EmitVertex();

    coords = vec3(0.0, 0.0, 1.0);
    gl_Position = gl_in[2].gl_Position;
    EmitVertex();

    EndPrimitive();
}
```

Listing 4: Fragment Shader for flat shading extension (Section: 3.2)

```
#version 330 core
in vec3 coords;
in vec4 wedge_color[3];
out vec4 fragColor;

void main() {
    // max diagram
    if (coords[0] > coords[1]) {
        if (coords[0] > coords[2]) {
            fragColor = wedge_color[0];
        } else {
```

```
                fragColor = wedge_color[2];
            }
        } else {
            if (coords[1] > coords[2]) {
                fragColor = wedge_color[1];
            } else {
                fragColor = wedge_color[2];
            }
        }
}
```

Listing 5: Vertex Shader for flat shading extension using gaussian curvature (Section: 3.3)

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 2) in vec3 gaussian_curvature;

out vec4 vertex_color;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

uniform float min_gc;
uniform float max_gc;
uniform float mean_negative_gc;
uniform float mean_positive_gc;


vec3 interpolation(vec3 v0, vec3 v1, float t) {
    return (1 - t) * v0 + t * v1;
}

vec4 get_result_color_gc() {
    float val = gaussian_curvature[0];
    vec3 red = vec3(1.0, 0.0, 0.0);
    vec3 green = vec3(0.0, 1.0, 0.0);
    vec3 blue = vec3(0.0, 0.0, 1.0);

    //negative numbers until 0 -> map from red to green
    if (val < 0) {
        return vec4(interpolation(red, green, val/min_gc)/(
           mean_negative_gc/5), 1.0);
    } else {
```

```
        //map from green to blue, from 0 to positive
        return vec4(interpolation(green, blue, val/max_gc)/(mean_positive_gc/5), 1.0);
    }
}

void main() {
    vec3 pos = vec3(model * vec4(aPos, 1.0));

    vertex_color = get_result_color_gc();

    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```