Università
della
Svizzera
italiana

Facoltà
di scienze
informatiche

Bachelor Project

Barycentric Data Visualization for Triangle Meshes

Costanza **Volpini**
costanza.volpini@usi.ch

Professor: Kai **Hormann**
Assistant: Jan **Svoboda**

Spring Semester 2018

Abstract

Abstract goes here

Dedication

to somebody

Declaration

I declare that..

Acknowledgements

I want to thank...

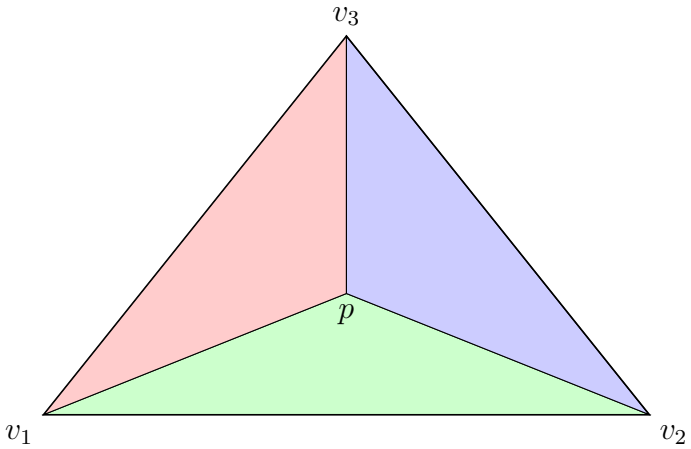
Contents

Chapter 1

Introduction

1.1 Barycentric coordinates

Let us consider a triangle with top, bottom left and bottom right vertices to which we have assigned the colors red, green and blue respectively. The triangle barycentre divides each median into two parts that have a 2:1 ratio.



Let call the red area w_1, the blue green one w_2 and the blue one w_3. Normalazing 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 $[v_1, v_2, v_3]$.

WRITE introduction of Barycentric coordinate (inventor and history) and all properties

1.2 Triangle mesh

See book

1.3 Lighting

formula + draw and explanation of angle (like at 90 degree some light..e.tc.)

1.4 Linear Interpolation

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

1.6 Gouraud Shading

Gouraud Shading can be calculated in the *vertex shader*. The main idea is to compute a normal at the vertex and an intensity for each vertex.

1.7 Curvature

1.7.1 Gaussian Curvature

Gaussian Curvature works like a logical `AND`, it will check if there is a curvature along both directions.

Surfaces 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.

1.7.2 Mean Curvature

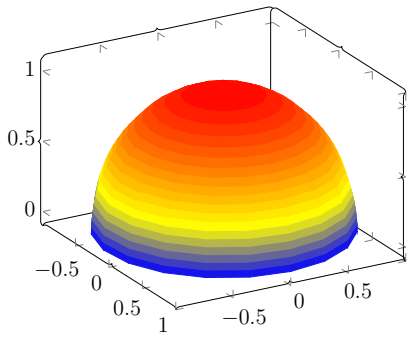


Figure 1.1: Positive gaussian curvature

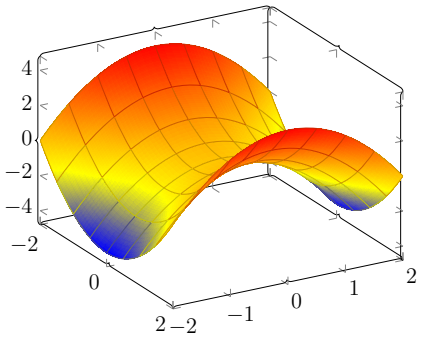


Figure 1.2: Negative gaussian curvature

Chapter 2

GPU program

shaders (geometry, vertex, fragment) + model view etc. arcball/trackball

Shader program is written in *GLSL*.

Vertex shader is done on every vertex (before rasterization). Fragment shader is done on every pixel (coloring per fragment).

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.

2.1 GPU pipeline

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

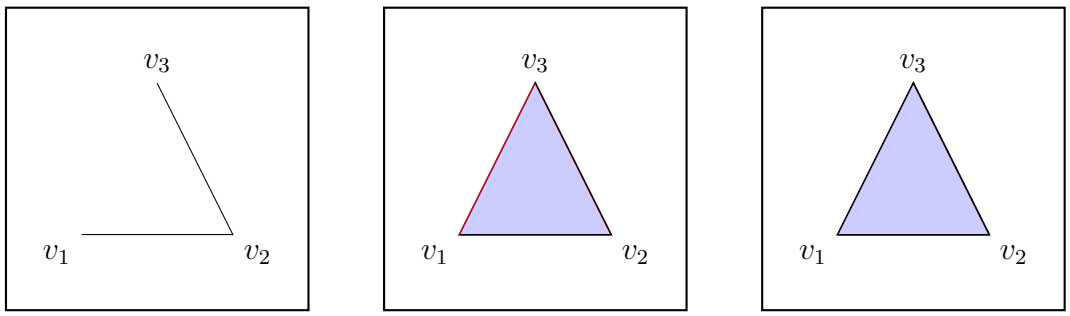
Fragment shader performs color computation for every visible pixel of the rasterized object. It works on a fragment at a time, but thanks to

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()`.



Chapter 3

Flat shading extension

Alternative data visualization techniques can be found using the power of barycentric coordinates and GPU programming.

The usual way to visualize data for a triangle mesh is to associate data to vertices and then interpolating over the mesh triangles, that does not work in case of edges and triangles.

3.1 Region around a vertex

We can split the surface of triangle meshes into regions around vertex (Fig. 3.2) 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.

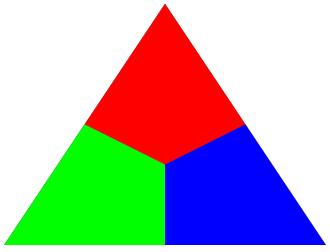


Figure 3.1: Vertex based area

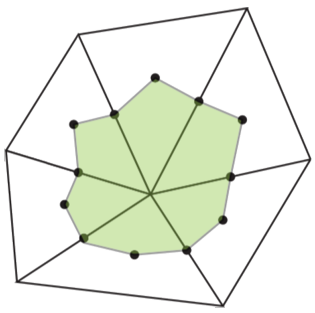


Figure 3.2: Region around a vertex

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.

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. 3.1).

```
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);
}
```

3.2 Color vertex area by lighting

Suppose now that you want 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. Then you can compute the color has it be done 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 that we need to pass the barycentric coordinates, the vertex color, the normal at the vertex and the lighting calculations to the *fragment shader*.

We want to avoid an automatic interpolation of colors, in order to return the resulting color using the *max diagram*, to do that we have used a *Geometry shader* in order to access to all three vertex colors in fragment shader.

Vertex Shader for extend flat shading:

```
#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);
}
```

Geometry Shader for extend flat shading:

```
#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();
    }
```

Fragment Shader for extend flat shading:

```
    #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];
            }
        }
    }
```

3.3 Color vertex area by Gaussian Curvature

We want to compute the *Gaussian curvature* per vertex. We want to sum up, for each vertex, angles at this vertex with adjacent triangles and then we subtract this value to 2π. After having obtain this value, called *angle defect* (Fig. 3.3), we map linearly this value to a color range.

Then we can pass this color to the *vertex shader* to see the color vertex flat shading visualisation of Gaussian curvature.

$$G = 2\pi - \sum_j \theta_j$$

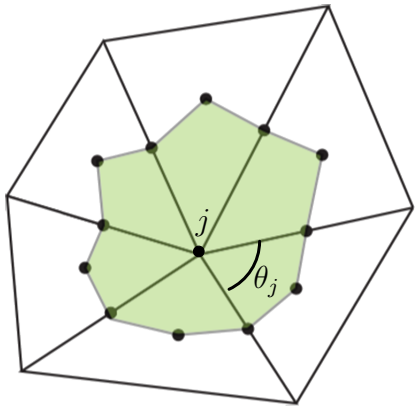


Figure 3.3: Angle defect

Gaussian curvature should return a constant color around each vertex (Fig. 3.4).

Vertex Shader for gaussian curvature:

```
#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;
```

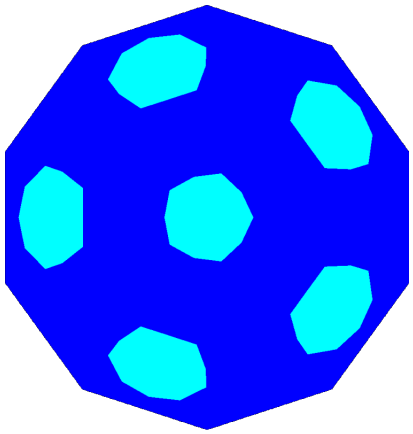


Figure 3.4: Gaussian curvature on icosahedron

```
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);
}
```

3.4 Evaluation and Comparison

We want now to compare the classic linear interpolation visualisation (Fig. 3.5) with two possible way to extend the flat shading. In Fig. 3.6 and Fig. 3.7 we have color each vertex area following the method *max diagram* described in the above subsection 3.1.1, this extension of the flat shading returns a more 3-dimensional model. In fact, as we can see in Fig. 3.6 and Fig. 3.7, the horse get more realistic thanks to the lighting calculation for Fig. 3.6 and to the *Gaussian curvature* for Fig. 3.7, in particular this last result in a more detailed model. Visualization of the principal curvatures as colors (Fig. 3.7), marking

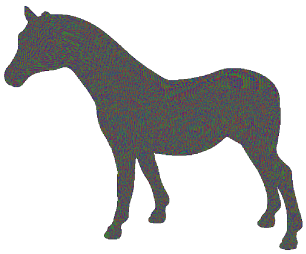


Figure 3.5: Interpolation

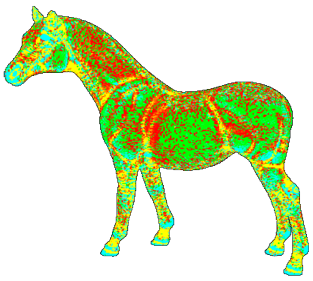


Figure 3.6: Lighting calculation

Figure 3.7: Gaussian Curvature

then every change of signs highlighting the 3-dimensionality of the model.

Appendix A

Appendix Title