Assignment 5

Johan Moreira

November 30th, 2020

Problem 1: Load and Render a 3D model: Extend the provided code to load the same scenes used in Assignment 4, and render them using rasterization in a uniform color.

In order to render the bunny from the bunny.off file, we need to first read in the vertices and the faces into a V and F matrix respectively. The code to import the data points into V and F is shown below:

```
void load_off(const std::string &filename, MatrixXd &V, MatrixXi &F) {
        std::ifstream in(filename);
        std::string token;
        in >> token;
        int nv, nf, ne;
        in >> nv >> nf >> ne;
        V.resize(nv, 3);
        F.resize(nf, 3);
        for (int i = 0; i < nv; ++i) {
                in >> V(i, 0) >> V(i, 1) >> V(i, 2);
        }
        for (int i = 0; i < nf; ++i) {
                int s;
                in >> s >> F(i, 0) >> F(i, 1) >> F(i, 2);
                assert(s == 3);
        }
}
```

Now with the code above, we can import the data points from the *bunny.off* file. The second step is now to prepare the data and send the data through the rasterizer. The code to prepare the data is below.

```
vector<VertexAttributes> vertices;
for(unsigned i=0; i < F.rows(); ++i){
        for (unsigned j=0; j < F.cols(); ++j){
                 double x = V.row(F(i,j))[0];
                 double y = V.row(F(i,j))[1];
                 double z = V.row(F(i,j))[2];
                 vertices.push_back(VertexAttributes(x, y, z));
         }
}
rasterize_triangles(program,uniform,vertices,frameBuffer);
```

By just using the code above, we will get the following result:

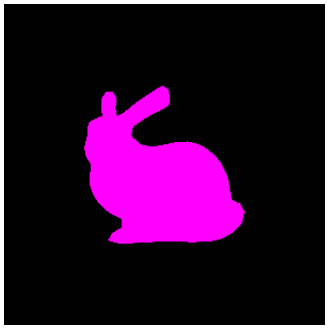

Image of 300x300 pixels

The problem with the code above is that it does not account for a change in the dimension of the image pixels. If we make the image $500x300$ pixels, we will see a distortion of the picture as shown below

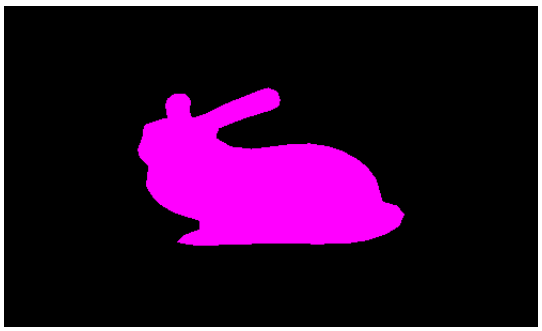

Image of 500x300 pixels

We can fix this problem by adding a transformation to compensate for the aspect ratio of the framebuffer. The transformation required is shown below

```
float aspect_ratio = float(frameBuffer.cols())/float(frameBuffer.rows());

uniform.view << 1, 0, 0, 0,
        0, 1, 0, 0,
        0, 0, 1, 0,
        0, 0, 0, 1;

if (aspect_ratio < 1)
        uniform.view(0,0) = aspect_ratio;
else
        uniform.view(1,1) = 1/aspect_ratio;
```

Now with this matrix, we can transform our vertices inside the vertexshader as follows:

```
program.VertexShader = [](const VertexAttributes& va, const UniformAttributes& uniform) {
        VertexAttributes out;
        out.position = uniform.view*va.position;
        return out;
};
```

Please note that we have passed the view matrix as a uniform attribute. The end result after applying this modification is as follows

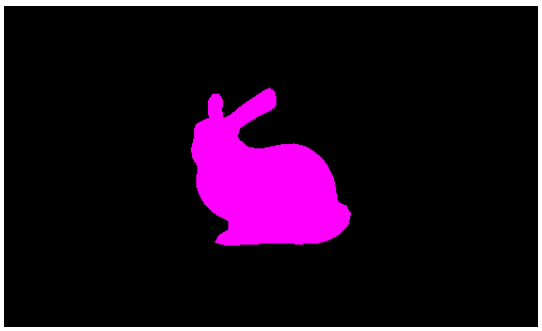

Image of 500x300 pixels

Finally, please note that for this problem, I used the world coordinates coming from the data points and no transformation was implemented.

Problem 2: Shading - In this exercise, implement different shading options for the triangles. The scene should always contain at least one light source, set as a shader uniform. Add an option to the code to render three different versions of the model: wireframe, flat shading, and per-vertex shading

1. Wireframe

In order to obtain the wireframe, I modified our code when preparing the data points. Instead of rasterizing triangles, I have to rasterize lines. Again for this problem, we will read the data points and then use these points to get our lines as follow

```
vector<VertexAttributes> vertices;
for(unsigned i=0; i < F.rows(); ++i){
        double x1 = (V.row(F.row(i)[0])[0]);
        double y1 = (V.row(F.row(i)[0])[1]);
        double z1 = (V.row(F.row(i)[0])[2]);
        double x2 = (V.row(F.row(i)[1])[0]);
        double y2 = (V.row(F.row(i)[1])[1]);
        double z2 = (V.row(F.row(i)[1])[2]);
        double x3 = (V.row(F.row(i)[2])[0]);
        double y3 = (V.row(F.row(i)[2])[1]);
        double z3 = (V.row(F.row(i)[2])[2]);

        vertices.push_back(VertexAttributes(x1, y1, z1));
        vertices.push_back(VertexAttributes(x2, y2, z2));
        vertices.push_back(VertexAttributes(x2, y2, z2));
        vertices.push_back(VertexAttributes(x3, y3, z3));
        vertices.push_back(VertexAttributes(x3, y3, z3));
        vertices.push_back(VertexAttributes(x1, y1, z1));
}

rasterize_lines(program,uniform,vertices,0.5,frameBuffer);
```

Additionally to the aspect ratio transformation implemented above, I added code to account for the depth of field. Note that in order to capture the depth of field, I added code to the fragment and blending shader, and added more uniform attributes as shown below,

```
// The vertex shader is the identity
program.VertexShader = [](const VertexAttributes& va, const UniformAttributes& uniform){
        VertexAttributes out;

        out.position = va.position;
        out.position = uniform.view*out.position;
        return out;
};

// The fragment shader uses a fixed color
program.FragmentShader = [](const VertexAttributes& va, const UniformAttributes& uniform){
        FragmentAttributes out(uniform.color[0],uniform.color[1],uniform.color[2]);
        out.position = va.position;
        out.distance = (out.position - uniform.camera_position).norm();
        return out;
};

// The blending shader converts colors between 0 and 1 to uint8
program.BlendingShader = [](const FragmentAttributes& fa, const FrameBufferAttributes& previous){
        if(fa.position[2] < previous.depth){
                FrameBufferAttributes out(fa.color[0]*255,fa.color[1]*255,fa.color[2]*255,fa.color[3]*255);
                out.depth = fa.position[2];
                return out;
        }
        else{
```

```
                return previous;
        }
};
```

The image we get after implementing the code above is

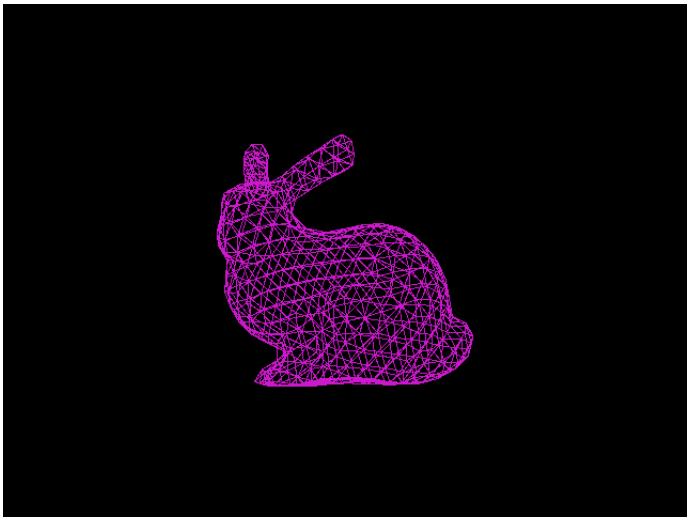

Image of 640x480 pixels

2. Flat Shading

For the flat shading, I reused the code above and added more code to it since the flat shading requires both triangles and line rendering. Since we need to calculate the color using the camera position and the light position(s), I changed the code for the vertex and fragment shader. Note that in order to accomplish the flat shading result, I computed a normal vector for each triangle and attached to every vertex. The code below shows the preparation of the data.

```
vector<VertexAttributes> vertices;

for(unsigned i=0; i < F.rows(); ++i){
        Vector3f vert1((V.row(F.row(i)[0])[0]), (V.row(F.row(i)[0])[1]), (V.row(F.row(i)[0])[2]));
        Vector3f vert2((V.row(F.row(i)[1])[0]), (V.row(F.row(i)[1])[1]), (V.row(F.row(i)[1])[2]));
        Vector3f vert3((V.row(F.row(i)[2])[0]), (V.row(F.row(i)[2])[1]), (V.row(F.row(i)[2])[2]));
        Vector3f v1 = vert2 - vert1;
        Vector3f v2 = vert3 - vert2;
        Vector3f n1((v1[1]*v2[2])-(v2[1]*v1[2]), -(v1[0]*v2[2])-(v1[2]*v2[0]), (v1[0]*v2[1])-(v1[2]*v2[0]));

        n1 = n1.normalized();

        vertices.push_back(VertexAttributes(vert1[0], vert1[1], vert1[2], n1[0], n1[1], n1[2]));
        vertices.push_back(VertexAttributes(vert2[0], vert2[1], vert2[2], n1[0], n1[1], n1[2]));
        vertices.push_back(VertexAttributes(vert3[0], vert3[1], vert3[2], n1[0], n1[1], n1[2]));
}
rasterize_triangles(program,uniform,vertices,frameBuffer);


vector<VertexAttributes> lines;
for(unsigned i=0; i < F.rows(); ++i){
        Vector3f vert1((V.row(F.row(i)[0])[0]), (V.row(F.row(i)[0])[1]), (V.row(F.row(i)[0])[2]));
        Vector3f vert2((V.row(F.row(i)[1])[0]), (V.row(F.row(i)[1])[1]), (V.row(F.row(i)[1])[2]));
        Vector3f vert3((V.row(F.row(i)[2])[0]), (V.row(F.row(i)[2])[1]), (V.row(F.row(i)[2])[2]));
        lines.push_back(VertexAttributes(vert1[0], vert1[1], vert1[2]));
        lines.push_back(VertexAttributes(vert2[0], vert2[1], vert2[2]));
        lines.push_back(VertexAttributes(vert2[0], vert2[1], vert2[2]));
        lines.push_back(VertexAttributes(vert3[0], vert3[1], vert3[2]));
        lines.push_back(VertexAttributes(vert3[0], vert3[1], vert3[2]));
```

```
        lines.push_back(VertexAttributes(vert1[0], vert1[1], vert1[2]));
}
rasterize_lines(program,uniform,lines,0.8,frameBuffer);
```

As mentioned before, I changed the code inside the vertex and fragment shader to compute the color of each triangle. The code to accomplish this task is shown below

```
program.VertexShader = [](const VertexAttributes& va, const UniformAttributes& uniform){
        VertexAttributes out;

        out.position = va.position;
        out.position = uniform.view*out.position;
        out.normal = va.normal;
        return out;
};

// The fragment shader uses a fixed color
program.FragmentShader = [](const VertexAttributes& va, const UniformAttributes& uniform){
        Vector4f ambient_color = uniform.ambient_color.cwiseProduct(uniform.color);
        Vector4f lights_color(0.0, 0.0, 0.0, 1);
        for(unsigned i=0; i<uniform.light_position.size(); ++i){
                Vector4f Li = (uniform.light_position[i] - va.position).normalized();
                Vector4f diffuse = uniform.mat_diffuse * max(Li.dot(va.normal), float(0.0));

                Vector4f h_unit = (Li - (va.position - uniform.camera_position)).normalized();
                Vector4f specular = uniform.mat_specular * pow(max(h_unit.dot(va.normal), float(0.0)), 256.0);

                Vector4f D = uniform.light_position[i] - va.position;
                lights_color += (diffuse + specular).cwiseProduct(uniform.light_intensity) /  D.squaredNorm();
        }

        Vector4f C = ambient_color + lights_color;
        C[3] = min(C[3],float(1));
        FragmentAttributes out(C[0],C[1],C[2]);
        out.position = va.position;
        out.distance = (out.position - uniform.camera_position).norm();
        return out;
};
```

The resulting picture obtained by using this code is shown below:

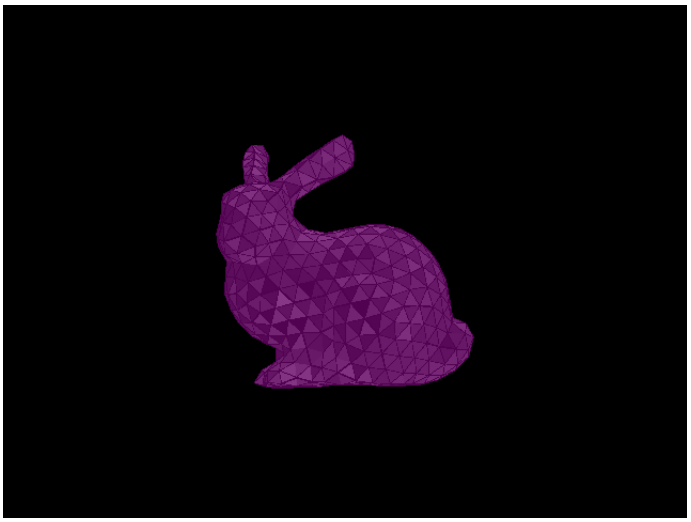

Image of 640x480 pixels

3. Per-Vertex Shading

The difference between this exercise and the previous one is that the normals are not the same for every vertex. I computed the per-face normals, and then I averaged them on the neighboring vertices. The other difference is that the color is calculated in the vertex shader rather than in the fragment shader. Below is the code to compute the per-vertex normals when preparing the vertices and normalsr.

```
MatrixXf normals = MatrixXf::Zero(V.rows(),3);
for(unsigned i=0; i < F.rows(); ++i){
        Vector3f vert1((V.row(F.row(i)[0])[0]), (V.row(F.row(i)[0])[1]), (V.row(F.row(i)[0])[2]));
        Vector3f vert2((V.row(F.row(i)[1])[0]), (V.row(F.row(i)[1])[1]), (V.row(F.row(i)[1])[2]));
        Vector3f vert3((V.row(F.row(i)[2])[0]), (V.row(F.row(i)[2])[1]), (V.row(F.row(i)[2])[2]));
        Vector3f v1 = vert2 - vert1;
        Vector3f v2 = vert3 - vert2;
        Vector3f n1((v1[1]*v2[2])-(v2[1]*v1[2]), -(v1[0]*v2[2])-(v1[2]*v2[0]), (v1[0]*v2[1])-(v1[2]*v2[0]));

        normals.row(F.row(i)[0])[0] += n1[0];
        normals.row(F.row(i)[0])[1] += n1[1];
        normals.row(F.row(i)[0])[2] += n1[2];
        normals.row(F.row(i)[1])[0] += n1[0];
        normals.row(F.row(i)[1])[1] += n1[1];
        normals.row(F.row(i)[1])[2] += n1[2];
        normals.row(F.row(i)[2])[0] += n1[0];
        normals.row(F.row(i)[2])[1] += n1[1];
        normals.row(F.row(i)[2])[2] += n1[2];
}

for(unsigned i=0; i<normals.rows(); ++i){
        normals.row(i) = normals.row(i).normalized();
}

for(unsigned i=0; i < F.rows(); ++i){
        Vector3f vert1((V.row(F.row(i)[0])[0]), (V.row(F.row(i)[0])[1]), (V.row(F.row(i)[0])[2]));
        Vector3f vert2((V.row(F.row(i)[1])[0]), (V.row(F.row(i)[1])[1]), (V.row(F.row(i)[1])[2]));
        Vector3f vert3((V.row(F.row(i)[2])[0]), (V.row(F.row(i)[2])[1]), (V.row(F.row(i)[2])[2]));

        Vector3f normal1 = normals.row(F.row(i)[0]);
        Vector3f normal2 = normals.row(F.row(i)[1]);
        Vector3f normal3 = normals.row(F.row(i)[2]);

        vertices.push_back(VertexAttributes(vert1[0], vert1[1], vert1[2], normal1[0], normal1[1], normal1[2]))
        vertices.push_back(VertexAttributes(vert2[0], vert2[1], vert2[2], normal2[0], normal2[1], normal2[2]))
        vertices.push_back(VertexAttributes(vert3[0], vert3[1], vert3[2], normal3[0], normal3[1], normal3[2]))
}

rasterize_triangles(program,uniform,vertices,frameBuffer);
```

Additionally please note that for this problem, the color is calculated in the vertex shader rather than in the fragment shader as shown below.

```
program.VertexShader = [](const VertexAttributes& va, const UniformAttributes& uniform){
        VertexAttributes out;
        out.position = va.position;

        Vector4f ambient_color = uniform.ambient_color.cwiseProduct(uniform.color);
        Vector4f lights_color(0.0, 0.0, 0.0, 1);
        for(unsigned i=0; i<uniform.light_position.size(); ++i){
                Vector4f Li = (uniform.light_position[i] - out.position).normalized();
                Vector4f diffuse = uniform.mat_diffuse * max(Li.dot(va.normal), float(0.0));

                Vector4f h_unit = (Li - (out.position - uniform.camera_position)).normalized();
```

```
                Vector4f specular = uniform.mat_specular * pow(max(h_unit.dot(out.normal), float(0.0)), 256.0)

                Vector4f D = uniform.light_position[i] - out.position;
                lights_color += (diffuse + specular).cwiseProduct(uniform.light_intensity) /  D.squaredNorm();
        }
Vector4f C = ambient_color + lights_color;
out.color = Vector4f(C[0],C[1],C[2],1);
out.position = uniform.view*out.position;
out.normal = va.normal;
return out;
};

// The fragment shader uses a fixed color
program.FragmentShader = [](const VertexAttributes& va, const UniformAttributes& uniform){
        FragmentAttributes out(va.color[0],va.color[1],va.color[2]);
        out.position = va.position;
        out.distance = (out.position - uniform.camera_position).norm();
        return out;
};
```

The resulting image after using the code above is the following:

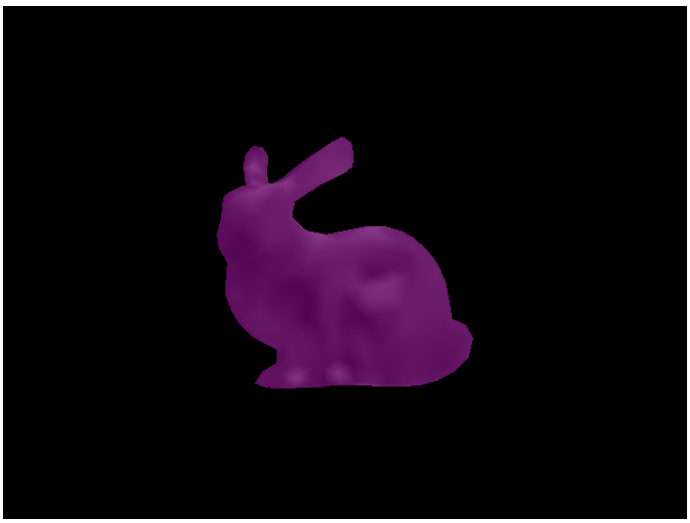

Image of 640x480 pixels

Problem 3: Object Transformation - Add an option to translate the object toward the camera, while rotating around its barycenter. Produce 3 gif videos, one for each rendering type used above.

In order to get the animated result, I modified the vertex shader and I added a piece of code that will generate the animation. Note that I also created the matrices for rotation and translation to camera matrices as shown below

```
const char * fileName = "triangle.gif";
vector<uint8_t> image;
int delay = 25;
GifWriter g;
GifBegin(&g, fileName, frameBuffer.rows(), frameBuffer.cols(), delay);

for (float i=0;i<=1.1;i+=0.05){
        Matrix<float,4,4> T;
        T << (1+i), 0, 0, 0,
                0, (1+i), 0, 0,
                0, 0, (1+i), 0,
                0, 0, 0, (1+1);

        Matrix<float,4,4> M_rot;
        M_rot << cos(i*2*M_PI), 0, sin(i*2*M_PI), 0,
                0, 1, 0, 0,
                -sin(i*2*M_PI), 0, cos(i*2*M_PI), 0,
                0, 0, 0, 1;

        uniform.M_s = T;
        uniform.M_rot = M_rot;

        frameBuffer.setConstant(FrameBufferAttributes());
        rasterize_lines(program,uniform,lines,0.8,frameBuffer);
        framebuffer_to_uint8(frameBuffer,image);
        GifWriteFrame(&g, image.data(), frameBuffer.rows(), frameBuffer.cols(), delay);
}
GifEnd(&g);
```

As mentioned before, I changed the code in the vertex shader as shown below. Additionally, please note that all the remaining code stays the same as in problem 2.

```
// Vertex Shader for the Wireframe animation
program.VertexShader = [](const VertexAttributes& va, const UniformAttributes& uniform) {
        VertexAttributes out;
        out.position = uniform.view*(uniform.M_rot * uniform.M_s) * va.position;
        return out;
};

// Vertex Shader for the Flat Shading animation
program.VertexShader = [](const VertexAttributes& va, const UniformAttributes& uniform){
        VertexAttributes out;
        out.position = uniform.view*(uniform.M_rot * uniform.M_s) * va.position;
        out.normal = va.normal;
        return out;
};

// Vertex Shader for the Per-Vertex Shading animation
program.VertexShader = [](const VertexAttributes& va, const UniformAttributes& uniform){
        VertexAttributes out;
        out.position = va.position;

        Vector4f ambient_color = uniform.ambient_color.cwiseProduct(uniform.color);
        Vector4f lights_color(0.0, 0.0, 0.0, 1);
```

```
        for(unsigned i=0; i<uniform.light_position.size(); ++i){
                Vector4f Li = (uniform.light_position[i] - out.position).normalized();
                Vector4f diffuse = uniform.mat_diffuse * max(Li.dot(va.normal), float(0.0));

                Vector4f h_unit = (Li - (out.position - uniform.camera_position)).normalized();
                Vector4f specular = uniform.mat_specular * pow(max(h_unit.dot(out.normal), float(0.0)), 256.0)

                Vector4f D = uniform.light_position[i] - out.position;
                lights_color += (diffuse + specular).cwiseProduct(uniform.light_intensity) /  D.squaredNorm();
        }

        Vector4f C = ambient_color + lights_color;
        out.color = Vector4f(C[0],C[1],C[2],1);
        out.position = uniform.view*(uniform.M_rot * uniform.M_s) * out.position;
        out.normal = va.normal;
        return out;
};
```

Please see all 3 gif files attached in the separate file

Problem 4: Perspective Camera - Implement a perspective camera and add support for multiple resolutions: the cameras should take into account the size of the framebuffer, properly adapting the aspect ratio to not distort the image whenever the framebuffer is resized.

To accomplish the perspective camera, I computed the matrix transformation for the camera, orthographic, and perspective as follow:

```
// Creating the camera transformation matrix
Vector4f g(0,0,-1,0);
Vector4f t(0,1,0,0);
Vector4f w = -g.normalized();
Vector4f u((t[1]*w[2])-(w[1]*t[2]), -(t[0]*w[2])-(t[2]*w[0]), (t[0]*w[1])-(t[2]*w[0]),0);
u = u.normalized();
Vector4f v((w[1]*u[2])-(u[1]*w[2]), -(w[0]*u[2])-(w[2]*u[0]), (w[0]*u[1])-(w[2]*u[0]),0);

Matrix4f M;
M << u[0], v[0], w[0], uniform.camera_position[0],
u[1], v[1], w[1], uniform.camera_position[1],
u[2], v[2], w[2], uniform.camera_position[2],
0, 0, 0, 1;

Matrix4f M_cam = M.inverse();
uniform.M_cam = M_cam;

// Creating the Orthographic transformation matrix
float n = -1.5;
float f = -6;
float top = tan(uniform.field_of_view/2) * n;
float r = aspect_ratio * abs(top);
float l = -r;
float b = -top;

Matrix4f M_ort;
M_ort << 2/(r-l), 0, 0, -(r + l)/(r - l),
0, 2/(top - b), 0, -(top + b)/(top - b),
0, 0, 2/(n-f), -(n+f)/(n-f),
0, 0, 0, 1;

Matrix4f M_per;
M_per << n, 0, 0, 0,
0, n, 0, 0,
0, 0, n+f, -f*n,
0, 0, 1, 0;

if(uniform.is_perspective){
        uniform.M_proj = M_ort * M_per;
}
else{
        uniform.M_proj = M_ort;
}
```

Additionally, I changed the code in the vertex shader as follows:

```
program.VertexShader = [](const VertexAttributes& va, const UniformAttributes& uniform){
        VertexAttributes out;
        out.position = uniform.M_cam*va.position;

        Vector4f ambient_color = uniform.ambient_color.cwiseProduct(uniform.color);
        Vector4f lights_color(0.0, 0.0, 0.0, 1);
        for(unsigned i=0; i<uniform.light_position.size(); ++i){
                Vector4f Li = (uniform.light_position[i] - out.position).normalized();
```

```
                Vector4f diffuse = uniform.mat_diffuse * max(Li.dot(va.normal), float(0.0));

                Vector4f h_unit = (Li - (out.position - uniform.camera_position)).normalized();
                Vector4f specular = uniform.mat_specular * pow(max(h_unit.dot(out.normal), float(0.0)), 256.0)

                Vector4f D = uniform.light_position[i] - out.position;
                lights_color += (diffuse + specular).cwiseProduct(uniform.light_intensity) /  D.squaredNorm();
        }

        Vector4f C = ambient_color + lights_color;
        out.color = Vector4f(C[0],C[1],C[2],1);
        out.position = uniform.M_proj*out.position;
        //out.position = out.position / out.position[3];
        out.position = uniform.view*out.position;

        out.normal = va.normal;
        return out;
};
```

The results is:

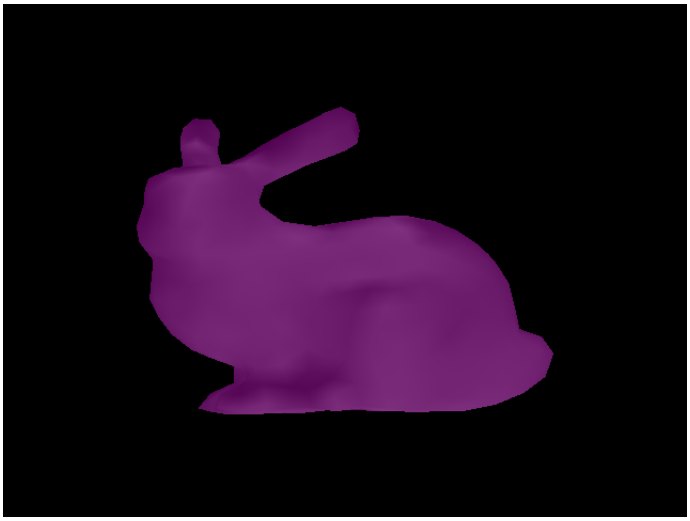

Image of 640x480 pixels