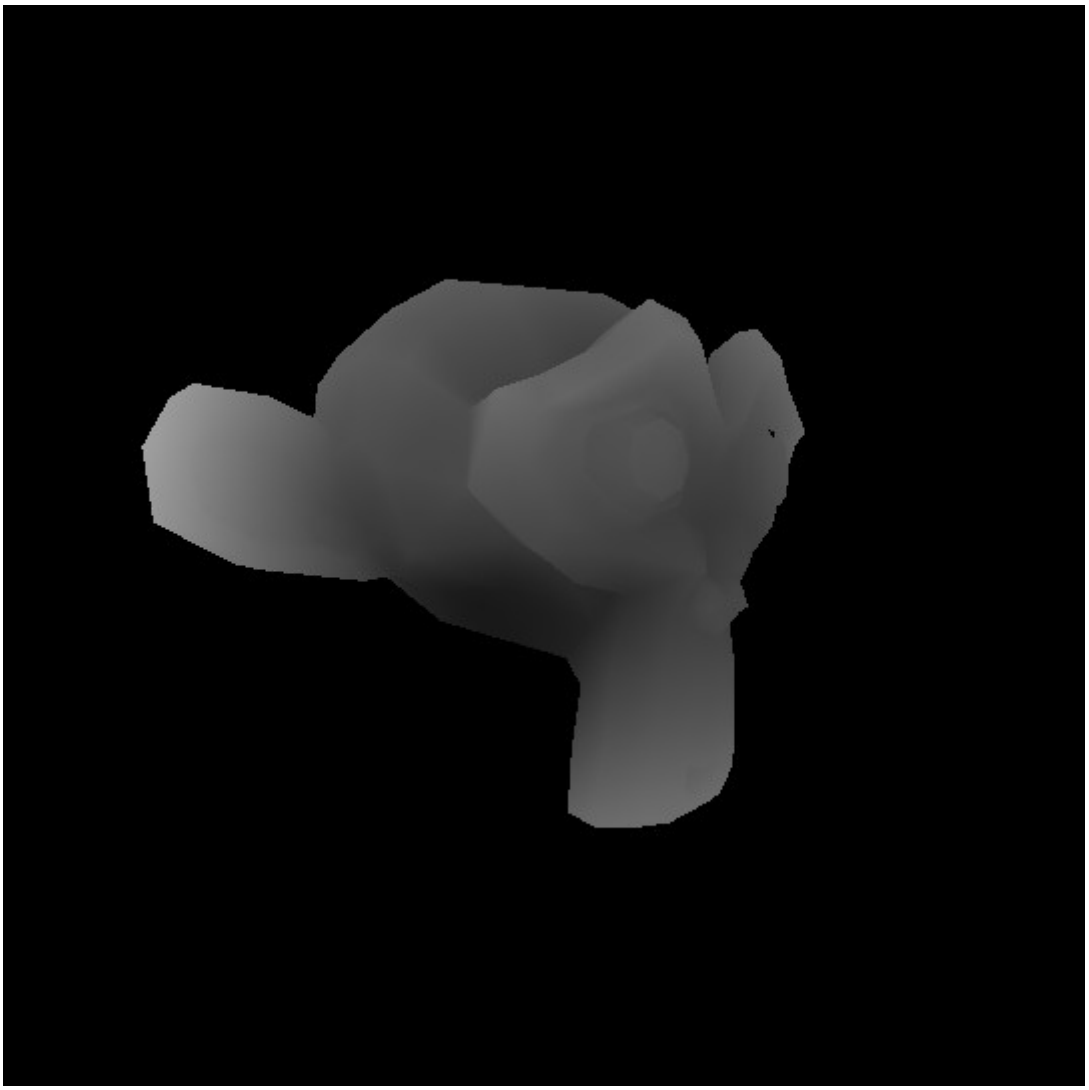
Introdução:

Neste trabalho foi consolidade o pipeline gráfico responsável pela renderização de vertices tridimencionais a partir de arquivos .obj. Foi utilizada como biblioteca matemática GLM e a versão do OpenGL escolhida foi a 3.3.

Originalmente o trabalho tinha como objetivo renderizar o objeto como wireframe (apenas arestas) mas foi encontrada dificuldade - explicada posteriormente – e por conta dela foi escolhido realizar o render do objeto com triangulos preenchidos, mas com shader de fragmentos que retorna uma cor diferente em função da distancia do vertice ao centro do modelo, garantindo que a cor ao longo de todo o render não seja homogenea, resultando numa ilusão de sombreamento. Foi obtido então o seguinte resultado:



Desenvolvimento:

OpenGL moderno é notoriamente verboso e complexo, tomando uma quantidade inesperada de linhas apenas para desenhar um triangulo na tela. Por esse motivo e tendo em mente o desenvolvimento de um codigo extensivel, foram desenvolvidas varias camadas de abstração. Em suma as abstrações desenvolvidas se relacionam da seguinte forma:

1- Um programa é iniciado pela criação de um Contexto, responsável por encapsular um vetor de Objetos e uma Camera; os primeiros possuem uma Mesh(vertices e futuramente vetores normais e coordenadas de textura) , dois Shaders(vertex e fragment) e uma posição, já a Camera encapsula sua posição, vetor direção e vetor Up.

2- O Objeto encapsula primitivas da API do OpenGL, como Vertex Array Objects e Vertex Buffer Objects, o primeiro é responsavel por manter metadados sobre a estrutura do buffer ativo, e o segundo permite a alocação e manipulaçao de memoria na GPU.

2- Shaders encapsulam carregamento, compilação, link e reporte de erros de arquivos de shader escritos em glsl.

A pequena biblioteca desenvolvida funciona da seguinte forma: o Contexto, após inicializado, inicia um laço no qual itera sobre todos os objetos nele contidos, ativa seu VAO e gera uma matriz de transformação a partir da posição do objeto e os parametros da camera ativa. Essa matriz de transformação é gerada através de uma cadeia de chamadas à funções como glm::rotate, glm::translate, glm::lookAt e glm::perspective e é passada como atributo uniforme para o shader program ativo. É suportada a renderização simultanea de multiplos objetos.

O desenvolvimento de tantos niveis de abstração permitiram uma função principal extremamente simples e clara comparada à utilização de OpenGL 3.0 de forma canonica:

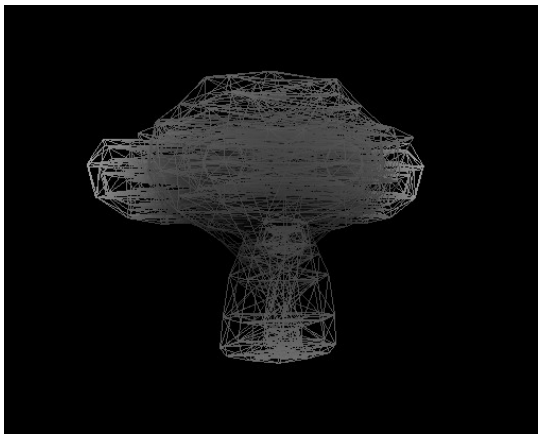
```
int main(){
    Context context("Opengl", 600,600);
    context.init();

    Mesh mesh = Mesh("./monkey_head2.obj");
    Shader vertex = Shader("./shaders/basicVertex.glsl", GL_VERTEX_SHADER);
    Shader fragment = Shader("./shaders/basicFragment.glsl", GL_FRAGMENT_SHADER);

    Object obj(mesh);
    obj.setShaders(vertex, fragment);
    obj.setUpAttributes();

    context.addObject(obj);

    context.setCamera({0,0, 1.3}, {0,0,-1}, {0,1,0});

    context.loop();
}
```

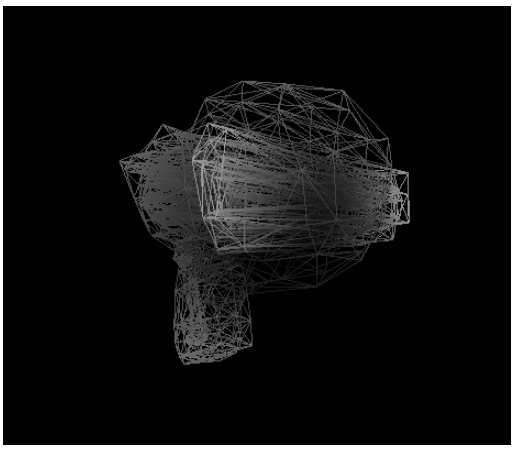
Dificuldades:

Uma dificuldade inesperada - e de provavél solução facil mas não descoberta pelo autor desse trabalho antes da data de entrega– foi a de renderização do objeto com um wireframe: o carregamento dos vertices a partir do arquivo .obj retorna uma sequencia de vertices que descrevem triangulos, o problema se resume aos seguintes pontos:

No momento da renderização é chamada a função glDrawArrays, que recebe como argumento o tipo da primitiva a ser renderizada. O resultado da chamada dessa função com o argumento GL_TRIANGLES é o esperado, são renderizados triangulos preenchidos perfeitamente.

No entanto, a chamada de glDrawArrays com qualquer outro tipo de primitiva resulta em um render distorcido. O planejado era utilizar como primitiva GL_LINE_STRIP, mas o resultado é a seguir:





Como ficou evidente na seguinda imagem, alguns triangulos estão sendo conectados de forma inesperada, resultado num caos imprevisto.

A primeira tentativa de solução foi reconstruir o vetor de vertices repetindo cada primeiro vertice após o fim de cada terceiro, formando um loop o qual poderia ser renderizado como GL_LINE_LOOP. O resultado foi semelhante ao visto acima.

Uma segunda solução imaginada foi a utilização de texturas com alpha zero, mas tal solução parecia ineficiente demais para ser a padrão.

Foi então decidido utilizar triangulos complestos e simular sombras através de uma simples modificação do fragment shader basico:

```
6   void main(){
7       float color = sqrt(sin(pow(length(thePosition),3)));
8       outColor = vec4(vec3(color), 1.0);
9   }
```