Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Computação Gráfica

Ano Letivo de 2023/2024

3ª Fase - VBOs e Curvas

Diogo Matos
A100741

Júlio Pinto
A100742

Mário Rodrigues
A100109

Miguel Gramoso
A100835

April 27, 2024

CG

Índice

1. Introdução

Este trabalho foi desenvolvido no âmbito da unidade curricular de Computação Gráfica (CG) no ano letivo 2023/2024.

A terceira fase do projeto consistiu em adicionar novas funcionalidades tanto ao ***engine*** como ao ***generator***. No *engine* foram implementados **VBOs** (*Vertical Buffer Objects*), para substituir a abordagem anterior, tal como o suporte para **animações** utilizando **curvas de *Catmull-Rom***. Para além disso, no *generator*, foi adicionada a possibilidade de utilizar ***patches*** para gerar **Curvas de Bézier**.

2. *Generator*

Nesta fase foi implementada a capacidade de interpretar *patches* de Bézier e convertê-los num ficheiro `.3d` contendo os triângulos do modelo.

2.1. Leitura de ficheiros `.patch`

Para a criação de modelos a partir dos *patches* de Bézier, é crucial que a nossa aplicação seja capaz de efetuar a leitura e processamento dos ficheiros `.patch` que armazenam toda a informação acerca do modelo a construir. Cada ficheiro deste tipo é composto por um conjunto de *patches* e por um conjunto de pontos.

Começamos o processo pela leitura dos *patches* e o seu posterior armazenamento em *arrays* de comprimento 16.

```
size_t num_patches;
  file >> num_patches;

  std::vector<std::vector<size_t>> patches(num_patches,
                                           std::vector<size_t>(16));

  for (size_t i = 0; i < num_patches; ++i) {
    for (size_t j = 0; j < 16; ++j) {
      size_t idx;
      file >> idx;
      file.ignore();
      patches[i][j] = idx;
    }
  }
```

Listing 1: Leitura e armazenamento dos patches

Cada elemento de um *patch* representa um índice, que representa um ponto. Proseguimos então com a leitura e armazenamento desses pontos num vetor.

```
size_t numberOfControlPoints;
  file >> numberOfControlPoints;

  for (size_t i = 0; i < numberOfControlPoints; ++i) {
    float x, y, z;
    file >> x;
    file.ignore();
    file >> y;
    file.ignore();
    file >> z;
    file.ignore();

    controlPoints.push_back(Point(x, y, z));
  }
```

Listing 2: Leitura e armazenamento dos pontos

Com os *patches* e os pontos armazenados, partimos então para o cálculo das coordenadas do objeto `.3d`, tendo em conta um nível de tesselação fornecido pelo utilizador.

Para calcularmos os pontos da curva optamos por utilizar uma técnica chamada **Interpolação de Bézier**, que permite criar curvas suaves que se aproximam de pontos específicos de controlo. Esta técnica tem como base os polinómios de Bernstein.

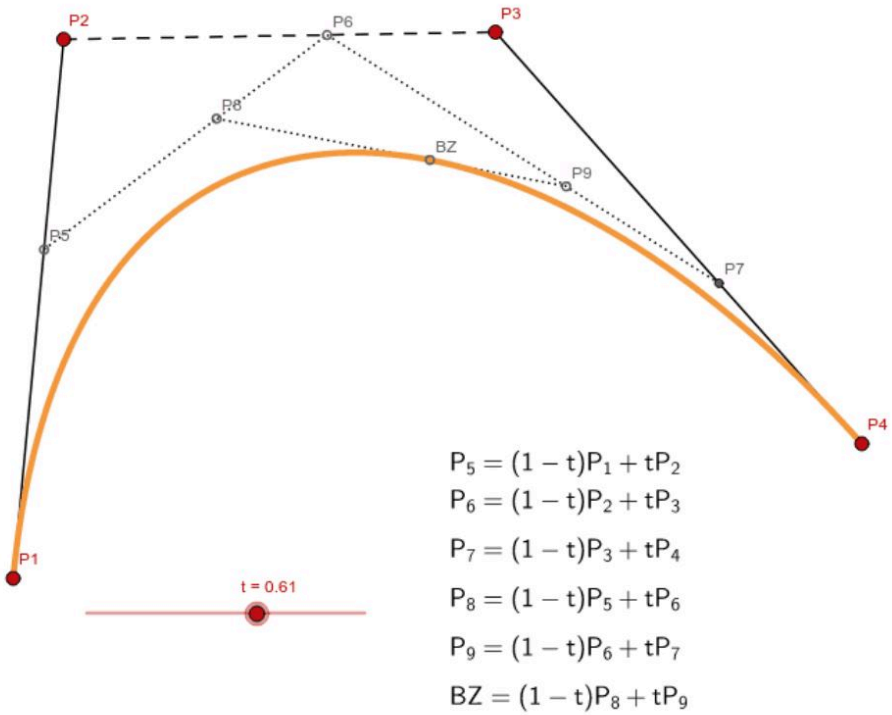

Figura 1: Exemplo de uma curva de Bezier, com tesselação de 0,61

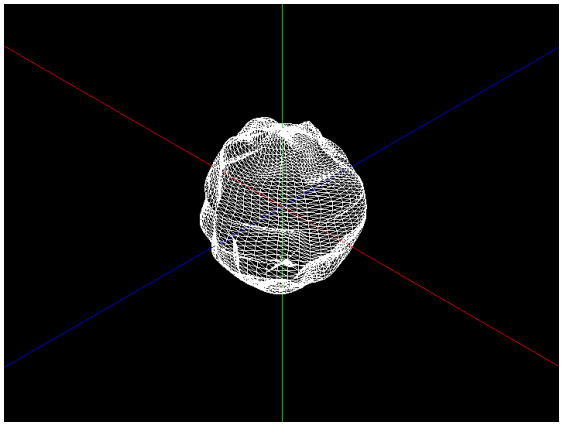

Figura 2: Cometa gerado por patch de bezier

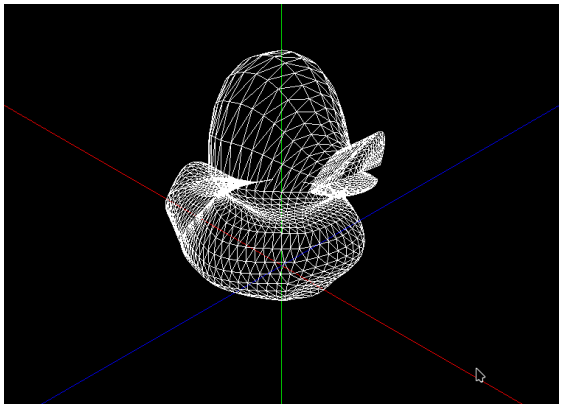

Figura 3: Pato de borracha gerado por patch de bezier

Utilizando o seguinte código, é nos possível calcular os pontos desejados:

```
for (size_t i = 0; i < num_patches; ++i) {
    std::vector<Point> patchControlPoints;
    for (size_t j = 0; j < 16; ++j) {
      patchControlPoints.push_back(controlPoints[patches[i][j]]);
    }

    for (int u = 0; u < tessellation; ++u) {
      for (int v = 0; v < tessellation; ++v) {
        float u1 = (float)u / tessellation;
        float v1 = (float)v / tessellation;
        float u2 = (float)(u + 1) / tessellation;
        float v2 = (float)(v + 1) / tessellation;

        Point p1 = bezierPatch(patchControlPoints, u1, v1);
        Point p2 = bezierPatch(patchControlPoints, u2, v1);
        Point p3 = bezierPatch(patchControlPoints, u1, v2);
        Point p4 = bezierPatch(patchControlPoints, u2, v2);

        points.push_back(p1);
        points.push_back(p3);
        points.push_back(p2);

        points.push_back(p2);
        points.push_back(p3);
        points.push_back(p4);
      }
    }
  }
  return points;
}

Point bezierPatch(const std::vector<Point>& controlPoints, float u, float v) {
  Point p(0, 0, 0);
  for (int i = 0; i < 4; ++i) {
    for (int j = 0; j < 4; ++j) {
      float b = bernstein(i, u) * bernstein(j, v);
      p.x += controlPoints[i * 4 + j].x * b;
      p.y += controlPoints[i * 4 + j].y * b;
      p.z += controlPoints[i * 4 + j].z * b;
    }
  }

  return p;
}

float bernstein(int i, float t) {
  switch (i) {
    case 0:
      return pow(1 - t, 3);
    case 1:
      return 3 * t * pow(1 - t, 2);
    case 2:
      return 3 * pow(t, 2) * (1 - t);
    case 3:
      return pow(t, 3);
    default:
      return 0;
  }
}
```

Listing 3: Cálculo dos pontos `.3d`

3. *Engine*

3.1. `Models`

De maneira a conseguirmos implementar os **VBOs** e **IBOs**, achamos necessário criar a classe **`Models`**:

```
class Model {
 public:
  std::string filename;
  std::vector<Point> vbo;
  std::vector<unsigned int> ibo;
  int id;
  bool initialized = false;

  Model(std::string filename, std::vector<Point> points);

  void setupModel();
  void drawModel();

  std::vector<Point> getPoints();

 private:
  GLuint _vbo, _ibo;
  std::vector<Point> _points;
  Model(std::string filename, std::vector<Point> vbo,
        std::vector<unsigned int> ibo, int id, std::vector<Point> points);
};
```

Listing 4: Atributos da Classe Model

Esta classe engloba o vetor de pontos a enviar para a GPU, o vetor de índices, um inteiro identificador do modelo, um booleano que sinaliza se o modelo já foi, ou não, inicializado e ainda um conjunto de variáveis privadas para o devido funcionamento da classe.

Para criar um `Model`, chamamos o construtor:

```
Model::Model(std::string filename, std::vector<Point> points) {
  this->filename = filename;
  this->id = counter;
  this->vbo = generateVBO(points);
  this->ibo = generateIBO(points, this->vbo);
  this->initialized = false;
  this->_points = points;
  counter++;
}
```

Listing 5: Construtor público da Classe Model

Este construtor executa as funções `generateVBO()` e `generateIBO()`, responsáveis por gerar o conjunto de pontos únicos do modelo e os indíces correspondentes a cada ponto a ser desenhado, respetivamente.

Para além disso, executa também as funções `setupModel()` e `drawModel()`. `setupModel()` é responsável por enviar o VBO e o IBO para a GPU e é chamada da primeira vez que o modelo é renderizado. Já a

`drawModel()` simplesmente utiliza as funções das bibliotecas GLUT e GLEW de modo a desenhar o modelo, assumindo que este está já carregado na GPU.

```
void Model::setupModel() {
  std::vector<float> floats = vPointstoFloats(this->vbo);

  // Generate and bind vertex buffer
  glGenBuffers(1, &this->_vbo);
  glBindBuffer(GL_ARRAY_BUFFER, this->_vbo);
  glBufferData(GL_ARRAY_BUFFER, sizeof(float) * floats.size(), floats.data(),
               GL_STATIC_DRAW);

  // Generate and bind index buffer
  glGenBuffers(1, &this->_ibo);
  glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, this->_ibo);
  glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(unsigned int) * this->ibo.size(),
               this->ibo.data(), GL_STATIC_DRAW);
}
```

Listing 6: Função `setupModel`

```
void Model::drawModel() {
  if (!this->initialized) {
    this->initialized = true;
    setupModel();
  }

  glColor3f(1.0f, 1.0f, 1.0f);
  glBindBuffer(GL_ARRAY_BUFFER, this->_vbo);
  glVertexPointer(3, GL_FLOAT, 0, 0);

  glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, this->_ibo);
  glDrawElements(GL_TRIANGLES, this->ibo.size(), GL_UNSIGNED_INT, 0);
}
```

Listing 7: Função `drawModel`

3.2. VBOs (*Vertical Buffer Objects*)

Como mencionado previamente, para gerar os VBOs utilizamos a função `generateVBO()`:

```
std::vector<Point> generateVBO(const std::vector<Point>& points) {
  std::vector<Point> vbo;
  std::unordered_map<Point, int, PointHash> index_map;
  for (const Point& point : points) {
    if (index_map.find(point) == index_map.end()) {
      index_map[point] = vbo.size();
      vbo.push_back(point);
    }
  }
  return vbo;
}
```

Listing 8: Definição de `generateVBO`

Esta função recebe um vetor de pontos e utiliza um `unordered_map` para associar os pontos a um índice. Ou seja:

```
-1 0 -1       --->  Indíce 0
-1 0 -0.5     --->  Indíce 1
-0.5 0 -1     --->  Indíce 2
-0.5 0 -1     --->  Pass
-1 0 -0.5     --->  Pass
-0.5 0 -0.5   --->  Indíce 3
```

Listing 9: Exemplo de como seria *mapeado* o ficheiro `.3d`

```
[ -1 0 -1  , -1 0 -0.5 , -0.5 0 -1 , -0.5 0 -0.5 ]
```

Listing 10: Exemplo de como seria o vector gerado

3.3. IBOs (*Index Buffer Objects*)

Para a geração do IBO seguimos uma ideia similar, recebendo o vetor do VBO e o vetor de pontos:

```
std::vector<unsigned int> generateIBO(const std::vector<Point>& points,
                                      const std::vector<Point>& vbo) {
  std::vector<unsigned int> ibo;
  std::unordered_map<Point, int, PointHash> index_map;
  for (size_t i = 0; i < vbo.size(); ++i) {
    index_map[vbo[i]] = i;
  }
  for (const Point& point : points) {
    ibo.push_back(index_map[point]);
  }
  return ibo;
}
```

Listing 11: Definição de `generateIBO`

Tal como para o VBO, esta função também utiliza um `unordered_map`, como forma de associar os pontos aos seus respetivos indíces. Desta forma, voltando a olhar para o ficheiro `.3d` anterior e o VBO gerado por esse ficheiro, teríamos a seguinte associação:

```
-1 0 -1       --->  Indíce 0
-1 0 -0.5     --->  Indíce 1
-0.5 0 -1     --->  Indíce 2
-0.5 0 -1     --->  Indíce 2
-1 0 -0.5     --->  Indíce 1
-0.5 0 -0.5   --->  Indíce 3
```

Listing 12: Exemplo da associação dos pontos ao seu respetivo indíce

3.4. Transformações com Tempo

Um dos principais objetivos desta fase foi a implementação de animações, sendo necessário introduzir transformações que ocorrem de forma cíclica em intervalos de tempo bem defenidos, nomeadamente **rotações** e **translações**.

Devido à natureza destas transformações, não é possível o seu armazenamento através da matriz de transformações, pelo que optamos por introduzir duas novas classes, `Rotations` e `Translations`, responsáveis por armazenar a informação relacionada com estas transformações.

```
class Rotations {
 public:
  float time;
  float x;
  float y;
  float z;
};
```

Listing 13: Classe `Rotations`

```
class Translations {
 public:
  float time;
  bool align;
  std::vector<Point> curvePoints;
  Point y_axis;
};
```

Listing 14: Classe `Translations`

3.4.1. Rotação

Também implementada como parte da classe `Rotations`, a função `applyRotation()` é responsável por calcular o ângulo de rotação correto e aplicar a respetiva rotação, dado o tempo atual. Esse cálculo consiste na divisão do tempo atual pelo tempo da rotação, multiplicado por 360 graus. De seguida apresenta-se a função `applyRotation()`:

```
void Rotations::applyRotation(float elapsed_time) {
  if (this->time == 0) {
    return;
  }
  float angle = 360 * (elapsed_time / this->time);
  glRotatef(angle, this->x, this->y, this->z);
}
```

Listing 15: Aplicação de uma rotação através da função `applyRotation()`

3.4.2. Translação

De forma semelhante ao que acontece na classe `Rotations`, a função `applyTranslation()` é responsável por calcular e aplicar a translação correta, dado o tempo atual. Para além do tempo atual, é também utilizado um valor booleano que indica o alinhamento ou não do objeto com a curva, assim como um conjunto de pontos que permitem definir uma curva de Catmull-Rom, variáveis estas que se encontram definidas na própria classe.

```
void Translations::applyTranslation(float elapsed_time) {
  if (this->time == 0) {
    return;
  }

  float time = elapsed_time / this->time;

  std::pair<Point, Point> position_dir =
      catmollRomPosition(this->curvePoints, time);
  Point pos = position_dir.first;
  Point dir = position_dir.second;

  glTranslatef(pos.x, pos.y, pos.z);

  if (this->align) {
    Point x = dir.normalize();
    Point z = Point(x).cross(this->y_axis).normalize();
    Point y = Point(z).cross(x).normalize();
    glMultMatrixf(rotationMatrix(x, y, z).data());
  }
}
```

Listing 16: Aplicação de uma translação

De modo a calcular a posição atual do ponto, foi criada a função auxiliar `catmullRomPosition()`, que recebe o vetor de pontos para o cálculo da curva, assim como o progresso relativo do objeto na animação, resultado da divisão entre o tempo atual e a duração da animação (`elapsed_time / this->time`).

Para o processo de cálculo da curva, é crucial obter a matriz de geração das curvas de Catmull-Rom, que se define da seguinte forma:

```
static const std::array<std::array<float, 4>, 4> cutmoll_rom_matrix{{
    {-0.5f, +1.5f, -1.5f, +0.5f},
    {+1.0f, -2.5f, +2.0f, -0.5f},
    {-0.5f, +0.0f, +0.5f, +0.0f},
    {+0.0f, +1.0f, +0.0f, +0.0f},
}};
```

Com todos os dados necessários estabelecidos, é possível então definir a função. De seguida exploramos por partes a implementação da mesma.

O primeiro passo é o cálculo do valor do tempo (`t`) para a matriz, assim como o segmento no qual o ponto se encontra (`segment`).

```
float t = global_time * curve.size();
int segment = (int)floor(t);
t -= segment;
```

Com estes valores definidos, podemos avançar para o cálculo dos quatro pontos que definem o segmento onde o ponto atual se encontra (`p1` a `p4`), podendo de seguida criar uma matriz com estes valores (`p`).

Para além disso definimos também as matrizes, neste caso de apenas uma linha, para os valores do tempo, tanto para calcular o ponto atual (`timeP`) como a sua orientação, *i. e.* a sua derivada (`timeDP`).

```
int first = segment + curve.size() - 1;
  Point p1 = curve[(first + 0) % curve.size()];
  Point p2 = curve[(first + 1) % curve.size()];
  Point p3 = curve[(first + 2) % curve.size()];
  Point p4 = curve[(first + 3) % curve.size()];

const std::array<std::array<float, 4>, 3> p{{
      {p1.x, p2.x, p3.x, p4.x},
      {p1.y, p2.y, p3.y, p4.y},
      {p1.z, p2.z, p3.z, p4.z},
  }};
```

```
const std::array<float, 4> timeP = {t * t * t, t * t, t,
                                    1};  // time matrix for point
const std::array<float, 4> timeDP = {3 * t * t, 2 * t, 1,
                                     0};  // time matrix for derivate
```

Por útlimo, realizamos a multiplicação destas matrizes obtendo um par de pontos (`Point`) que definem o ponto atual e o seu alinhamento, *i. e.* a sua derivada.

```
std::array<float, 3> pv{};  // Point
  std::array<float, 3> dv{};  // Derivative

  for (size_t i = 0; i < 3; ++i) {
    std::array<float, 4> a{};

    for (size_t j = 0; j < 4; ++j) {
      for (size_t k = 0; k < 4; ++k) {
        a[j] += p[i][k] * matrix[j][k];
      }
    }

    for (size_t j = 0; j < 4; j++) {
      pv[i] += timeP[j] * a[j];
      dv[i] += timdeDP[j] * a[j];
    }
  }

  return {
      Point(pv[0], pv[1], pv[2]),
      Point(dv[0], dv[1], dv[2]),
  };
```

3.5. Sistema Solar

Como forma de melhor testar as novas funcionalidades incluídas nesta fase, melhoramos a *scene* relativa ao Sistema Solar, escalando-a de maneira a ser possível observar a rotação em torno do Sol e a rotação do Sol em torno de si mesmo. Optamos também por adicionar o anel de Urano e o cometa Hayley, este último sob a forma de um *teapot*. Uma *demo* das animações do Sistema Solar pode ser encontrada na pasta `demos/` do projeto.

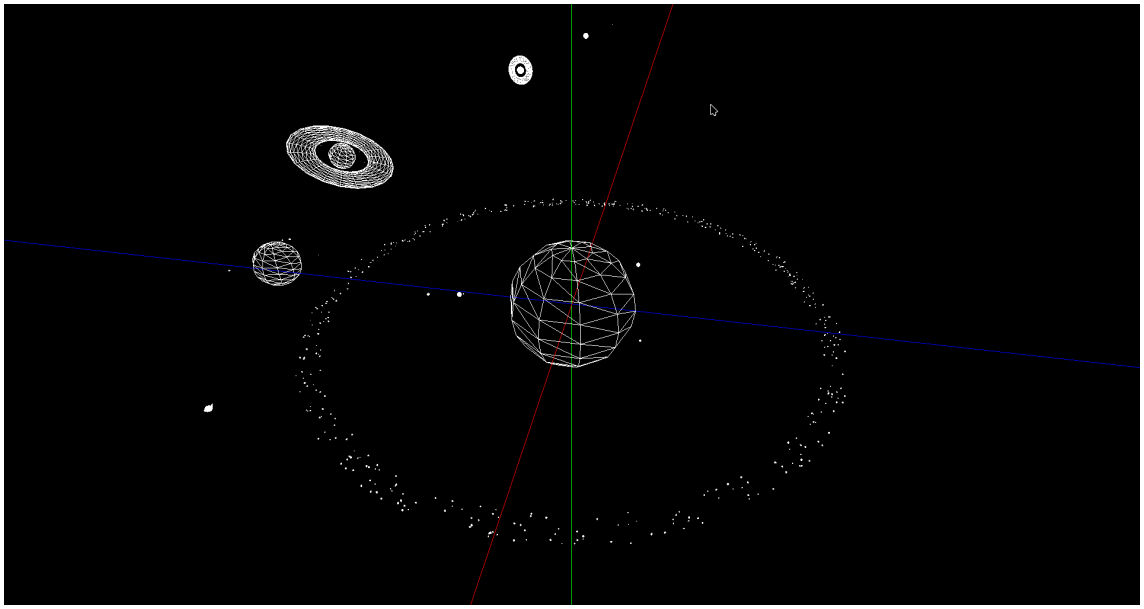

Figura 4: Sistema Solar

4. Testes

De modo a testar as diversas funcionalidades implementadas nesta fase, foram realizados um conjunto de testes que se apresentam de seguida.

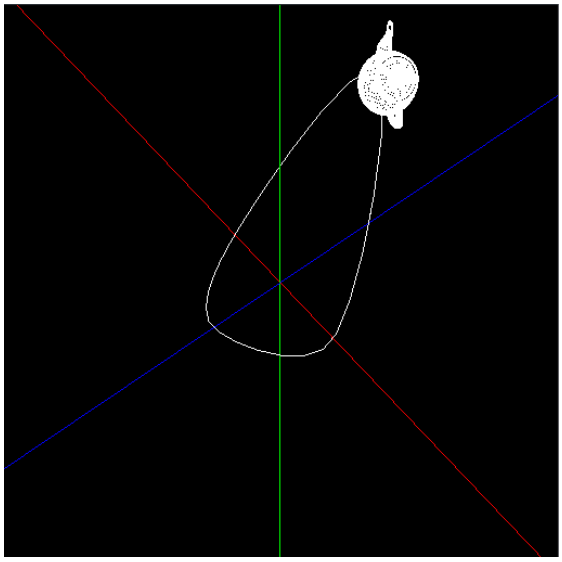

Figura 5: Teste nº1 - *Teapot* a mover-se segundo uma curva

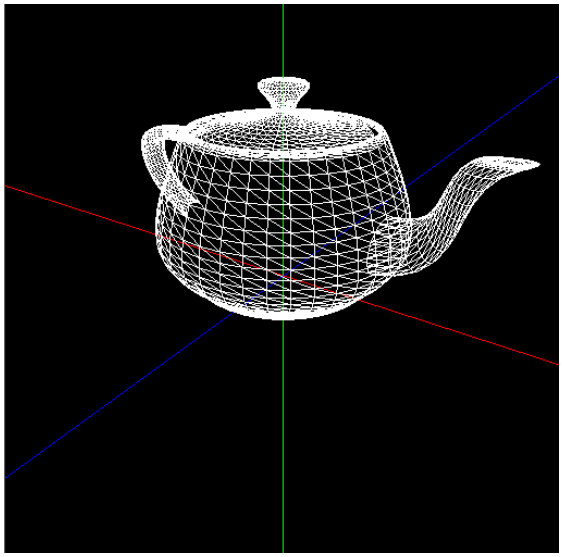

Figura 6: Teste nº2 - *Teapot*

Para os testes que implicam animações, é possível encontrar um ficheiro `.webm`, na pasta `demos/` do projeto, com o nome do teste.

5. Conclusões e Trabalho Futuro

Consideramos que os resultados que obtemos nesta fase foram os esperados. Planeamos realizar algumas otimizações de maneira a melhorar a gestão de memória do processo tal como a eficiência de alguns dos cálculos necessários.

Para além disso, tal como mencionado na fase anterior, planeamos implementar alguns extras como `ImGUI` e *frustum culling*. Acreditamos que estes extras irão melhorar o nosso projeto e nos permitirão desenvolver ainda mais conhecimentos sobre a área da Computação Gráfica.