Computação Gráfica (3º ano de LCC)

Trabalho Prático (Fase 3) — Grupo 1

Relatório de Desenvolvimento

Bruno Dias da Gião (A96544) Bruno Miguel Fernandes Araújo (A97509)
João Luís da Cruz Pereira (A95375)

26 de abril de 2024

Conteúdo

Lista de Figuras

1 Introdução

Este relatório serve como um apoio ao trabalho, tem explicações sobre os raciocínios usados na terceira fase do projeto da unidade curricular Computação Gráfica.

Nesta fase tanto o generator como o engine foram modificados, sendo que foi necessário implementar superfícies de *Bézier* (generator), extensão das transformações *translate* e *rotate* para suportarem curvar cúbicas de *Catmull-Rom* (engine) e também a implementação de *VBO's*, no nosso caso, *VBO's* com indexação (generator e engine). Para além disso, implementamos a câmara exploradora.

Tivemos de expandir e melhorar o nosso ficheiro *XML* que através da leitura pelo engine, representará o Sistema Solar (ficheiro Demo).

1.1 Estrutura do Relatório

- §2 - Explicação do motivo e da implementação das alterações na geração dos ficheiros *3D* para uso de *VBO's*(2.1).
- §2 - Implementação das superfícies de *Bézier*(2.2).
- §3 - Implementação de *VBO's* com índices para melhoria de performance e diminuição da utilização de memória(3.1).
- §3 - Extensão das transformação *translate* e *rotate* para implementação de curvas *Catmull-Rom*(3.2).
- §4 - Explicação do novo desenvolvimento no ficheiro XML Demo.
- §5 - Implementação de câmara exploradora.
- §6 - Conclusão e trabalho futuro.
- §7 - Imagens e vídeo tirados dos modelos gerados a partir dos ficheiros *XML* dos testes e da demo do Sistema Solar.

2 *Generator*

Iremos explicar as alterações realizadas no nosso *generator*.

2.1 Alterações na geração dos ficheiros *3D*

Previamente os nossos ficheiros 3D estavam organizados da seguinte forma:

```
0.000 0.000 0.000
1.000 2.300 2.000
3.000 0.500 0.750
1.000 2.300 2.000
4.000 4.000 4.000
```

ou seja, tínhamos todos os vértices com as suas coordenadas listadas no ficheiro. Durante a implementação dos *VBO's* percebemos que estarmos a calcular quais seriam os índices de cada vértice iria atrasar o arranque do nosso programa *engine* e não iríamos ter qualquer benefício em termos de memória com a utilização dos índices dos *VBO's*. Então, decidimos passar esse pré-processamento para o *generator* e alterar o formato dos ficheiros *3D* para o seguinte formato:

```
0 0.000 0.000 0.000
1 1.000 2.300 2.000
2 3.000 0.500 0.750
1
3 4.000 4.000 4.000
```

ou seja, no início de cada linha, indicamos o índice de cada vértice e de seguida indicamos as coordenadas do mesmo. Caso o vértice já exista, adicionamos apenas o índice respetivo. Em termos de memória, num caso extremo como um *torus* com 1000 *slices* e 1000 *stacks*, passamos de ter um ficheiro *3D* com 250MB para um ficheiro de 75MB, uma diferença significativa.

Para obtermos estes resultados tivemos que fazer algumas alterações no nosso ficheiro *generator*. Criamos um *map* que como *key* irá conter as coordenadas do vértice em formato de string (Ex: "0.0001.0002.000") e terá como *value* o seu índice. Temos uma variável para guardar o valor atual do índice.

```
std::map<std::string, unsigned int> vi;
unsigned int i = 0;
```

Nas funções de geração das primitivas, invés de escrevermos diretamente os vértices para o ficheiro, calculamos o que irá ser a nossa *key* e invocamos uma função que irá determinar se o vértice já foi escrito no ficheiro e que determina o seu índice.

```
coord = std::to_string(px) + std::to_string(py) + std::to_string(pz);
write_file(coord, px, py, pz, output);
```

A função *write_file* recebe como argumentos a *string* que representa as coordenadas do vértice, as próprias coordenadas e o apontador para o ficheiro que queremos escrever. A mesma, verifica se a string *coord* já está no *map* (que guarda os vértices e o seu respetivo índice), caso esteja escreve para o ficheiro o *value* correspondente. Caso contrário, escreve o índice, as coordenadas e adiciona ao *map* o vértice com o seu índice. O valor atual do índice é incrementado.

```
void write_file(std::string coord,
    float x,
    float y,
    float z,
    FILE* output)
{
  char buff[512];
  size_t b_read;
  if (vi.find(coord) != vi.end()) {
    b_read = snprintf(buff, 512, "%u\n", vi[coord]);
    fwrite(buff, sizeof(int8_t), b_read, output);
  } else {
    b_read = snprintf(buff, 512, "%u %.3f, %.3f, %.3f\n", i, x, y, z);
    fwrite(buff, sizeof(int8_t), b_read, output);
    vi[coord] = i++;
  }
}
```

2.2 Superfícies de *Bézier*

Para a implementação da capacidade de gerar modelos conforme patches bezier, tivemos em conta os cálculos dados nas aulas, seguimos duas fórmulas que se encontram nos slides do ficheiro pdf 'curvas e superficies' fornecido pelo educando.

Primeiro tivemos em conta o slide 20 desse mesmo ficheiro que indicava como era feita a triangulação da patch.

Por fim, o cáculo necessário usando a fórmula de **p(u,v)** que se encontra no slide 19 desse ficheiro.

Antes destes cálculos, começamos por introduzir uma chamada para a nova função *gen_bezier* no caso de, na leitura do comando, apareça em ***argv[1]*** o argumento *patch*.

De seguida, em gen_bezier iremos inicialmente ler o ficheiro patch, guardando os indices num vetor de vetores de inteiros chamado **indexes**, e os pontos num vetor de triplos de floats chamado **points**.

Para os índices, lemos primeiramente o primeiro número do ficheiro e guardámo-lo numa variável inteira **npatches**, que representará o número de patches presente nesse ficheiro. Iteramos sobre esta, iteramos até 16 (número de índices

presentes numa patch) e guardamos em cada iteração deste, o índice num vetor, que no fim deste ciclo será guardado no vetor **indexes**. No fim **indexes** é um vetor de tamanho **npatches** com subvetores de tamanho 16.

Para os pontos, lemos primeiramente o primeiro número a seguir aos indices e guardámos-lo numa variável inteira **npoints**, que representará o número de pontos presente nesse ficheiro. Iteramos sobre esta e em cada iteração guardamos o x , y ,z num triplo (novo tipo definido numa struct).

Tendo então já todos os pontos de controlo guardados, iteramos sobre o vetor *indexes*, e cada iteração guardamos em 3 matrizes (px,py,pz) o x, y, z dos pontos de controlo. Agora podemos passar aos cálculos, multiplicamos cada destas novas matrizes com a matriz de *Bézier* à esquerda e à direita, e encaminhamos o resultado para uma função auxiliar que tratará dos cálculos restantes (**bezieraux**).

```
for (std::vector <int> quadrado:indexes) {
    for (int i = 0; i < 4; i++) {
      for (int j = 0; j < 4; j++) {
        triple point = points[quadrado[i * 4 + j]];
        px[j][i] = point.x;
        py[j][i] = point.y;
        pz[j][i] = point.z;

      }

    }

    multMM(bezierM, px, mataux);
    multMM(mataux, bezierM, px);

    multMM(bezierM, py, mataux);
    multMM(mataux, bezierM, py);

    multMM(bezierM, pz, mataux);
    multMM(mataux, bezierM, pz);

    bezieraux(px, py, pz, tesselation, output);
  }
```

Seguindo então o raciocínio da triangulação das *patches de Bézier*, iteramos até 1 saltando *1/tesselation* duas vezes (dois ciclos, um sobre **u** outro sobre **v**, de **p(u,v)**, de forma a podermos obter cada ponto do quadrado da *patch*, assim como tem no slide 20 do ficheiro já mencionado), em cada iteração completamos cálculo de **p(u,v)** sendo que já temos

$$M \times R \times Mt \tag{1}$$

(em que R é a matriz dos pontos de controlos e M a de matriz *Bézier*).

```
int32_t bezieraux(float px[4][4], float py[4][4], float pz[4][4], int tesselation, FILE*
    output) {

  float x1, x2, x3, x4, y1, y2, y3, y4, z1, z2, z3, z4;
  float t = 1.0f / tesselation;
    std::string coord;

  for (float i = 0; i < 1; i += t) {
    for (float j = 0; j < 1; j += t) {


      x1 = puv(i, j, px);
      x2 = puv(i + t, j, px);
      x3 = puv(i + t, j + t, px);
      x4 = puv(i, j + t, px);

```

```
16
17          y1 = puv(i, j, py);
18          y2 = puv(i + t, j, py);
19          y3 = puv(i + t, j + t, py);
20          y4 = puv(i, j + t, py);
21
22          z1 = puv(i, j, pz);
23          z2 = puv(i + t, j, pz);
24          z3 = puv(i + t, j + t, pz);
25          z4 = puv(i, j + t, pz);
```

Então os cálculos restantes que faltam são feitos à parte na função **puv**.

Nesta criamos os vetores U e V que têm $[u^3\, u^2\, u\, 1]$ e $[v^3\, v^2\, v\, 1]$, e supondo que

$$m = M \times R \times Mt \tag{2}$$

fazemos:

$$p(u,v) = U \times m \times V \tag{3}$$

Primeiro multiplicamos **m** com o vetor **V** e por fim **U** com **m**.

```
1     // calculo de p(u,v) para cada ponto da patch
2  float puv(float U, float V, float m[4][4]) {
3
4    // matriz onde tera o resultado de m * v
5    float res[4];
6
7    // matriz v
8    float v[4];
9
10   // Resultado final
11   float r;
12
13   //Criar a matriz v da
14   v[0] = powf(V, 3);
15   v[1] = powf(V, 2);
16   v[2] = V;
17   v[3] = 1;
18
19   //m * v
20   multMV(m, v, res);
21
22   // U * m * V
23   r = powf(U, 3) * res[0] + powf(U, 2) * res[1] + U * res[2] + res[3];
24
25   return r;
26 }
```

Por fim volta-se a ***bezieraux*** e escreve-se nos ficheiros os pontos que representam a triangulação da *patch Bézier* e que no engine serão desenhados.

3 *Engine*

Passaremos a explicar o trabalho relativo ao **Motor de Gráficos** concebido nas fases anterior:

3.1 Implementação de *VBO's* com índices

A implementação de *VBO's* com índices levou a que tivessemos que alterar a *struct ident_ prims*e também três funções no nosso *engine*: *read_ words*, *read_ 3d_ files* e *draw_ figs*.

3.1.1 *Struct ident_ prims*

A *ident_ prims* que previamente era composta pelo nome da primitiva e pelo *vector* das suas coordenadas é agora composta pelo nome, por *vbo*, *ibo* (que irão ser o *vector* de vértices e de índices, respetivamente), e também pelo número de vértices e pelo número de índices.

```
struct ident_prim {
  char name[64];
  GLuint vbo, ibo, vertex_count;
  unsigned int index_count;
};
```

3.1.2 Função *read_ words*

A função *read_ words* tem agora a responsabilidade de enquanto lê o conteúdo de um ficheiro, tem que também guardar num *vector* as coordenadas dos vértices (uma vez por vértice) e também guardar noutro *vector* os índices que lê do ficheiro.

```
void read_words(FILE *f, std::vector<struct triple>* coords, std::vector<unsigned int>* ind)
{
  char line[1024];
  char* num;
  unsigned int i;

  while (fgets(line, sizeof(line), f) != NULL) {
    num = strtok(line, " ");
    i = atoi(num);
    num = strtok(NULL, " ");
    if (num != NULL) {
      triple v;
      v.x = atof(num);
      num = strtok(NULL, " ");
      v.y = atof(num);
      num = strtok(NULL, " ");
      v.z = atof(num);
      num = strtok(NULL, " \n");
      coords->push_back(v);
      ind->push_back(i);
    } else {
      ind->push_back(i);
    }
  }
}
```

Tal como na fase anterior, percorremos o ficheiro linha a linha, porém, desta vez, podemos guardar o primeiro token na variável *i* que será o índice do vértice da linha atual. De seguida, avançamos para o próximo *token*, no caso do mesmo não ser *NULL*, quer dizer o vértice é novo e a linha contém também as suas coordenadas. Nesse caso, lemos as coordenadas, e adicionamos ao *vector coords* (cada ficheiro *3D* irá ter um associado) o vértice e ao *vector ind* (cada ficheiro *3D* irá ter um associado) o índice. Caso o *token* seja *NULL*, a linha não tem mais conteúdo, sendo assim, só temos que adicionar o *vector ind* o índice lido.

3.1.3 Função *read_3d_files*

A função *read_3d_files* tem agora também a responsabilidade de criar os VBO's para cada ficheiro *3D* e guardar os *vectors* dos vértices e dos índices associados a esse ficheiro nos VBO's respetivos. Para tal, por cada ficheiro *3D* são criados dois *vectors*, um para guardar as coordenadas e outro pra guardar os índices. É também usada uma instância da *struct ident_prim* que irá conter o conteúdo referido previamente.(3.1.1)

```
std::vector<struct triple> coords;
std::vector<unsigned int> ind;
```

De seguida, é invocada a função *read_words* que irá preencher esses mesmos *vectors*. Após isso, colocamos o número de vértices e o número de índices em *aux* que irá guardar os nossos *VBO's*. Tal como nas aulas práticas, geramos os nossos *VBO's*, indicamos quais são os *VBO's* ativos e copiamos os *vectors* para a placa gráfica. Por fim, adicionamos a nossa *struct aux* ao nosso *vector* de primitivas. Obtendo então a atual versão da função *read_3d_files*.

```
int read_3d_files(void)
{
  FILE* fd;
  int i, j;
  int flag = 0;
  struct ident_prim aux;
  for (i = 0; i < world.primitives.size(); i++) {
    flag = 0;
    for (j = 0; j < prims.size() && !flag; j++) {
      if (!strcmp(world.primitives[i].name, prims[j].name))
        flag = 1;
    }
    if (!flag) {
      fd = fopen(world.primitives[i].name, "r");
      if (!fd) {
        return -1;
      }
      std::vector<struct triple> coords;
      std::vector<unsigned int> ind;
      read_words(fd, &coords, &ind);
      strcpy(aux.name, world.primitives[i].name);

      aux.vertex_count = coords.size();
      aux.index_count = ind.size();

      // Generate the VBO
      glGenBuffers(1, &aux.vbo);
      glGenBuffers(1, &aux.ibo);

      // Bind the VBO
      glBindBuffer(GL_ARRAY_BUFFER, aux.vbo);
      glBufferData(GL_ARRAY_BUFFER, coords.size() * sizeof(triple), coords.data(),
    GL_STATIC_DRAW);

      // Bind the IBO
      glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, aux.ibo);
      glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(unsigned int) * ind.size(), ind.data(),
    GL_STATIC_DRAW);

      // Store the VBO ID in the vertices vector
      prims.push_back(aux);

      fclose(fd);
    }
```

```
  }
  return 0;
}
```

3.1.4 Função *draw_figs*

Na função *draw_figs* ao invés de desenharmos ponto a ponto, simplesmente acedemos a cada primitiva (conforme o grupo, como na fase anterior) e desenhamos os *VBO's* com índices tal como fizemos nas aulas práticas.

```
void drawfigs(void)
{
  int i, k, l, g;
  for (g = 0; g < global; g++) {
    glPushMatrix();
    for (l = 0; l < world.transformations.size(); l++) /* trans*/
      if (world.transformations[l].group == g) {
        world.transformations[l].t->do_transformation();
      }

    for (k = 0; k < world.primitives.size(); k++) {
      if (world.primitives[k].group == g) {
        for (i = 0; i < prims.size(); i++) {
          if (!strcmp(prims[i].name, world.primitives[k].name)) {
            glBindBuffer(GL_ARRAY_BUFFER, prims[i].vbo);
            glVertexPointer(3,GL_FLOAT,0,0);
            glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, prims[i].ibo);
            glDrawElements(GL_TRIANGLES,
                           prims[i].index_count, // numero de indices a desenhar
                           GL_UNSIGNED_INT, // tipo de dados dos indices
                           0);// parametro nao utilizado
          }
        }
      }
    }
    glPopMatrix();
  }
}
```

3.2 Extensão das transformações *transform* e *rotate*

Nesta fase, entendemos as transformações geométricas permissível para realizar animações, isto é, transformações ao longo do tempo.

3.3 Rotações

Fixando um parâmetro obtido no ficheiro de configuração, $t_{\text{expected}} \in \mathbb{Z}$, um valor a ser computado em cada *frame*, $t_{\text{elapsed}} \in \mathcal{F}$, e um valor que é atualizado com $\frac{\text{glutGet(GLUT\_ELAPSED\_TIME)}}{1000}$ sempre que $t_{\text{elapsed}} = t_{\text{expected}}$, $t_{\text{init}} \in \mathcal{F}$. Todos estes valores esperam-se estar em segundos.

$$R = \frac{360^\circ \times t_{\text{elapsed}}}{t_{\text{expected}}} \wedge t_{\text{elapsed}} = \frac{\text{glutGet(GLUT\_ELAPSED\_TIME)}}{1000} - t_{\text{initial}}$$

Seguindo estas fórmulas, rapidamente obtemos uma implementação em *OpenGL* que realize a rotação desejada.

3.4 Translações

As Translações são feitas de acordo com o padrão de curvas Catmull-Rom onde o detalhe encontra-se na obtenção de gt que precisa de assegurar que alcançando t_{expected} encontra-se no ponto inicial da curva.
Assumindo as mesmas variáveis das rotações:

$$gt = \frac{t_{\text{elapsed}}}{t_{\text{expected}}}$$

que, para estar fixo no intervalo, $[0, 1[$, introduz-se a condição no código:

```
if (elapsed < t_seconds) {
    ...
}
```

3.4.1 Alinhamento

O alinhamento é padrão em curvas Catmull-Rom, sendo obtido pela multiplicação da matriz de rotação dada por:

$$\vec{X}_i = P'\vec{(t)}_i \wedge \vec{Z}_i = \vec{X}_i \times \vec{Y}_{i-1} \wedge \vec{Y}_i = \vec{Z}_i \times \vec{X}_i$$

e $M = \begin{pmatrix} \vec{X} & \vec{Y} & \vec{Z} & \vec{0} \end{pmatrix}$, onde todos os vetores estão normalizados.

3.4.2 Algoritmo Rápido da Raiz Inversa

De forma a otimizar o *performance* do nosso motor, que sofria nas normalizações quando se adicionava algum detalhe em máquinas GPUs mais antiquados, optamos por introduzir o chamado "*Fast Inverse Square Root Algorithm*" introduzido no *Quake III Arena* [1], permitindo assim que o cálculo das componentes do vetor normalizado seja obtido exclusivamente por multiplicações.

4 Desenvolvimento da Demo

Nesta fase, com o intuito de criar um sistema solar dinâmico, adicionamos ao nosso ficheiro *XML* colocamos *rotate time* (versão extendida de *rotate*), para simular a rotação do planeta em volta de si mesmo e *translate time* (versão extendida de *translate*) para as órbitas dos planetas.

Como exemplo temos uma Lua de Marte, a Métis:

```
1         <group>
2         <transform>
3                 <translate time = "30" align="True">
4                     <point x="13" y="0" z="13"/>
5                     <point x="0" y="0" z="18"/>
6                     <point x="-13" y="0" z="13"/>
7                     <point x="-18" y="0" z="0"/>
8                     <point x="-13" y="0" z="-13"/>
9                     <point x="0" y="0" z="-18"/>
10                    <point x="13" y="0" z="-13"/>
11                    <point x="18" y="0" z="0"/>
12                </translate>
13                <rotate time="30" x="0" y="1" z="0"/>
14          <scale x="0.1" y="0.1" z="0.1" />
15        </transform>
16        <models>
17          <model file="sphere_10_20_20.3d" /> <!--  Metis -->
18        </models>
```

[1] Código Fonte do *Quake III Arena*

```
19        </group>
```

Como podemos ver, este subgrupo do grupo de Marte, tem no grupo *translate*, o *translate time* que vai simular a sua órbita e o *rotate time* de 30º à volta do eixo y , que simula a rotação à volta de si mesmo.

Por fim, acrescentar um cometa cujo modelo base é gerado usando o *patch Bézier* fornecido, *teapot*.

Temos então o seguinte grupo que representa o cometa no ficheiro *XML*:

```
1         <group>
2             <transform>
3                 <translate time = "40" align="true">
4                     <point x = "0" y = "-100" z = "100" />
5                     <point x = "100" y = "-100" z = "0" />
6                     <point x = "0" y = "200" z = "-20" />
7                     <point x = "-20" y = "200" z= "0" />
8                 </translate>
9                 <rotate angle="-60" x="1" y="0" z="0"/>
10                <scale x="2" y="2" z="2" />
11            </transform>
12            <models>
13                <model file="bezier_10.3d" /> <!-- Cometa -->
14            </models>
15        </group>
```

5 Câmara Exploradora

Para a adição da câmara exploradora tivemos que fazer algumas mudanças no nosso *engine*. Nomeadamente, na *struct world* adicionamos, no campo *cam*, *dist* que representa a distância da câmara ao ponto para onde está a olhar e os ângulos *alfa* e *beta* que nos indicam as coordenadas da câmara na esfera de raio *dist*.

```
1  struct {
2    struct {
3        int h, w, sx, sy;
4      char title[64];
5    } win;
6    struct {
7          float dist;
8          float alfa;
9          float beta;
10     struct triple pos;
11     struct triple lookAt;
12     struct triple up; /* 0 1 0 */
13     struct triple proj; /* 60 1 1000*/
14   } cam;
15   std::vector<struct trans> transformations;
16   std::vector<struct prims> primitives;
17 } world;
```

Na função *xml_ init*, após obtermos os valores da posição da câmara, usamos os mesmos para calcular os campos adicionados anteriormente. Os cálculos são efetuados como na aula prática e preenchemos dessa forma os valores de *dist*, *alfa* e *beta*.

```
1 world.cam.pos.x = posi->FloatAttribute("x");
2 world.cam.pos.y = posi->FloatAttribute("y");
3 world.cam.pos.z = posi->FloatAttribute("z");
4 world.cam.dist = sqrt(world.cam.pos.x*world.cam.pos.x +
5           world.cam.pos.y*world.cam.pos.y+
```

```
            world.cam.pos.z*world.cam.pos.z);
            world.cam.alfa = asin(world.cam.pos.x / (world.cam.dist * cos(world.cam.beta)));
            world.cam.beta = asin(world.cam.pos.y / world.cam.dist);
```

Alteramos também o nosso *renderScene*. Temos que, tal como na aula prática, converter as coordenadas esféricas para coordernadas cartesianas.

```
// calculate camera pos
world.cam.pos.x = world.cam.dist * cos(world.cam.beta) * sin(world.cam.alfa);
world.cam.pos.y = world.cam.dist * sin(world.cam.beta);
world.cam.pos.z = world.cam.dist * cos(world.cam.beta) * cos(world.cam.alfa);
```

Por fim, implementamos a função *processKeys*, tal como na aula prática, de forma a podermos movimentar a posição da câmara (mantendo o valor absoluto de *beta* inferior a 1.5 *rad*).

```
void processKeys(unsigned char c, int xx, int yy)
{
  switch (c) {
  case 'a':
    world.cam.alfa -= 0.1;
    break;
  case 'd':
    world.cam.alfa += 0.1;
    break;
  case 's':
    world.cam.beta -= 0.1;
    break;
  case 'w':
    world.cam.beta += 0.1;
    break;
  case 'l':
    draw -= 1;
    break;
  case '+':
    tesselation += 10;
    break;
  case '-':
    tesselation -= 10;
    break;
    case 'z':
        world.cam.dist -= 10;
        break;
    case 'x':
        world.cam.dist += 10;
    }


  if (tesselation <= 0.F) {
    tesselation = 100.F;
    draw = false;
  }
  else if (tesselation >= 1000000.F)
    tesselation = 1000000.F;
  if (world.cam.beta < -1.5f) {
    world.cam.beta = -1.5f;
  }
  else if (world.cam.beta > 1.5f) {
    world.cam.beta = 1.5f;
  }
```

```
45 }
```

6 Conclusão

Efetuamos as alterações necessárias no nosse **gerador** de ficheiros *3D* e no **motor** de gráficos para os mesmos serem capazes de, implementar superfícies de *Bézier*, curvas de *Catmull-Rom* e também a utilização de *VBO's* ao invés do modo imediato.

6.1 Trabalho Futuro

Para além do conteúdo requisito das fases ainda por realizar, temos certos objetivos pessoais que gostavamos de realizar ao longo da elaboração do projeto, nomeadamente:

- A implementação de 3 tipos de câmaras que podem ser predefinidas no input XML e alteradas ao longo da execução do engine, particularmente, uma câmara primeira pessoa e uma câmara terceira pessoa[2].
- Melhor tratamento de erros.

[2]ver Eve Online - `https://www.eveonline.com/`

7 Resultados

Os resultados obtidos para cada teste fornecido e para a demo do Sistema Solar.

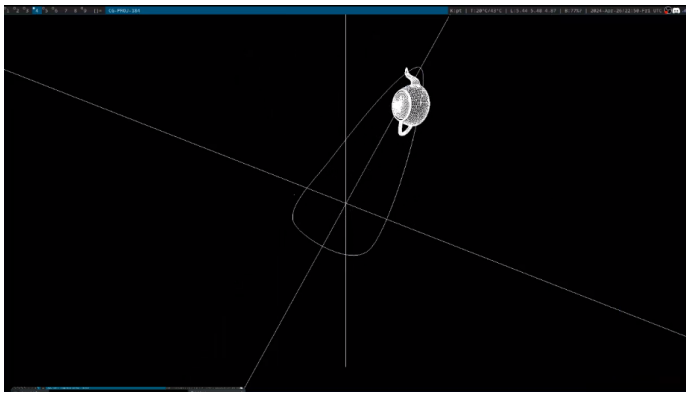

Figura 1: Teste 1

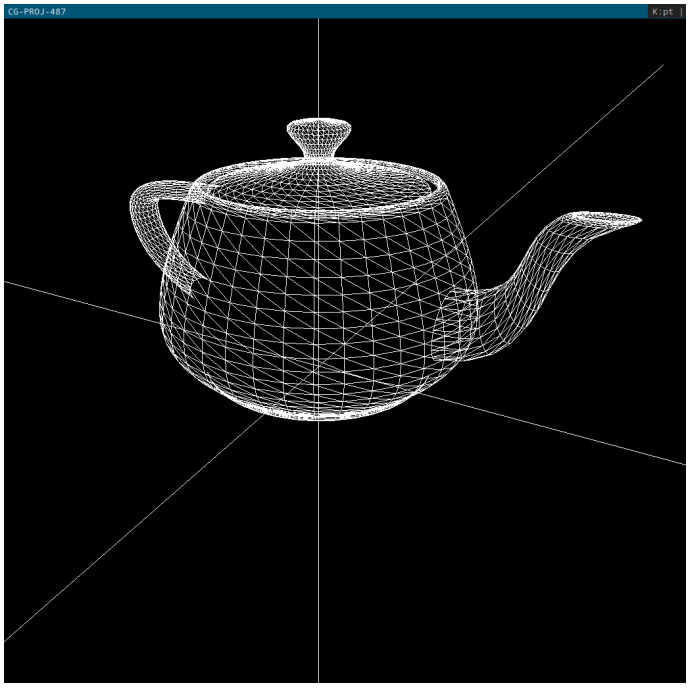

Figura 2: Teste 2

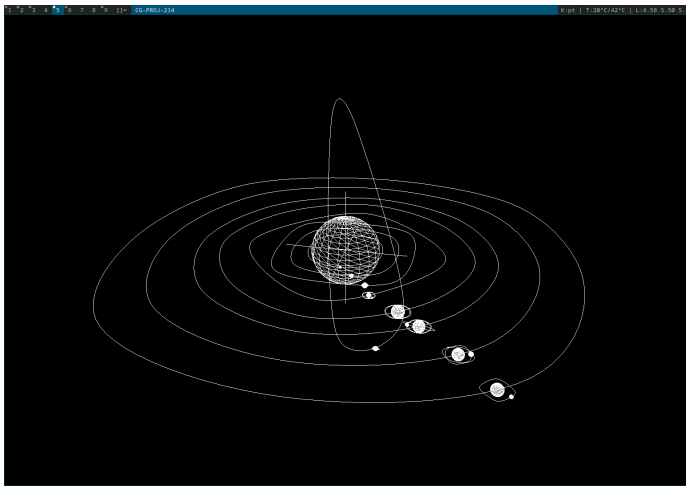

Figura 3: Sistema Solar