Computação Gráfica (3º ano de LCC)

Trabalho Prático (Fase 4) — Grupo 1

Relatório de Desenvolvimento

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

27 de maio de 2024

Conteúdo

Lista de Figuras

1 Introdução

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

Nesta fase modificamos o generator de forma que este tenha a capacidade de gerar figuras com as suas normais (serão usadas para a iluminação) e as coordenadas de textura.

Também tivemos de alterar o engine também de forma a ser capaz de suportar as novas informações fornecidas pelo generator.

Expandimos no ficheiro *XML* que representa o Sistema Solar (ficheiro Demo), implementando então as texturas e iluminação.

1.1 Estrutura do Relatório

- §2 - Detalhar as alterações feitas no generator, nomeadamente os cálculos usados para as coordenadas de textura e para as normais de cada primitiva.
- §3 - Explicação das alterações feitas no engine, para suportar a nova leitura dos ficheiros 3D, implementar as luzes e texturas.
- §4 - Explicação da implementação do *View Frustum Culling*.
- §5 - Conclusão.
- §6 - Imagens tiradas dos modelos gerados a partir dos ficheiros *XML* dos testes e da demo do Sistema Solar.

2 *Generator*

Passaremos a explicar o cálculo das normais e das coordenadas de textura de cada uma das primitivas.

Primeiramente é de notar que tivemos de desenvolver uma função para a normalização, a qual chamamos de normalize.

```
1 void normalize(float* a1, float* a2, float* a3) {
2   float l = sqrt((*a1) * (*a1) + (*a2) * (*a2) + (*a3) * (*a3));
3   if (l != 0) {
4     *a1 = *a1 / l;
5     *a2 = *a2 / l;
6     *a3 = *a3 / l;
7   }
8 }
```

Além disso tivemos de alterar a função write_file de forma que esta tenha em conta as normais e texturas nos ibos e na escrita no ficheiro.

- **Plano** Para as normais do plano colocamos (0,1,0) para todos os seus pontos.

  Relativamente às coordenadas de textura deste, calculamos de acordo com as variações do x e do z dos ciclos e serão então (i/divisions,j/divisions), em que i varia entre 1 e divisions e j 0 e divisions-1.

- **Cubo** As normais do cubo colocamos as seguintes em cada face:
  - (0,1,0) para o Topo.
  - (0,-1,0) para a Base.
  - (0,0,1) para a Frente.
  - (0,0,-1) para Trás.

– (1,0,0) para a Direita.
– (-1,0,0) para a Esquerda.

Relativamente às coordenadas da textura , usamos um método similar ao do plano em cada uma das suas faces. mas neste caso i varia como j , ou seja de 0 a divisions-1.

- **Cone** Para a base do cone as normais são (0,-1,0) e as para os outros casos é (cos(atan(raio / altura)) * sin(angulo),sin(atan(raio / altura)), cos(atan(raio / altura)) * cos(angulo))

  Para o cálculo das texturas temos (j/slices,i/stacks) em que i varia entre 0 e stacks - 1 e j entre 0 e slices - 1. A base é um caso especial , tendo (0.5f * sin(angle),0.5f * cos(angle)) como coordenadas de textura.

- **Esfera** Para as normais da esfera temos o cálculo usado nos pontos, excluindo a multiplicação com o raio.

  Para as texturas seguimos um raciocínio similar ao do cone, ou seja, (i/slices,j/stacks) em que i varia entre 0 a slices-1 e j entre 0 e stacks - 1.

- **Torus** O torus segue uma lógica similar à da esfera .Para as normais usamos o mesmo cálculo que o dos seus pontos mas retiramos a soma do raio interior (inner radius), mantendo o resto. Relativamente ás texturas do torus, nós colocamos algo similar às da esfera, mas acabamos por não conseguir.

- **Patchs Bezier** Para esta já foi necessário um esforço extra, tivemos de modificar a bezier_aux acrescentando o uso de três novas funções.

  bu que trata de calcular o **u**, da seguinte forma:

$$\mathbf{u} = \begin{bmatrix} 3 \times u \times u & 2 \times u & 1 & 0 \end{bmatrix} M \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} M^T \begin{bmatrix} v \times v \times v \\ v \times v \\ v \\ 1 \end{bmatrix}$$

  bv que trata de calcular o **v**, da seguinte forma:

$$\mathbf{v} = \begin{bmatrix} u \times u \times u & u \times u & u & 1 \end{bmatrix} M \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} M^T \begin{bmatrix} 3 \times v \times v \\ 2 \times v \\ 1 \\ 0 \end{bmatrix}$$

  Por fim para obter a normal dá-mos usa á função **multcr** que fará o seguinte cálculo:

$$\frac{\mathbf{u} \times \mathbf{v}}{\|\mathbf{u} \times \mathbf{v}\|}$$

  Por fim para as coordenadas de textura temos então:

  (j,i) ou (j,i+t) ou (j+t,i+t) ou (j+t,i)

  Dependendo do ponto da patch em questão. (t =1.0f / tesselation).

Estas foram todas as mudanças no generator, não foi mencionado previamente mas tivemos de ter muito cuidado com a escolha dos pontos das coordenadas de textura, se, por exemplo, o x da textura era por exemplo i/slices ou (i+1)/slices no caso da esfera.

3 Engine

Expliquemos as mudanças efetuadas no nosso motor de gráficos.

3.1 Alterações nas estruturas de dados

Para conseguirmos guardar a informação das normais e das coordenadas de texturas tivemos que alterar algumas estruturas de dados, nomeadamente *struct ident_prims*, *struct world* e tivemos que criar uma *struct lights*.

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

Adicionamos um vbo para as normais e um para as texturas, não é necessário um para os índices de cada pois o número de normais e de coordenadas de textura será o mesmo que de vértices.

```
struct light {
  char type[64];
  float posX, posY, posZ,
    dirX, dirY, dirZ,
    cutoff;
};
```

Criamos a *struct light* para guardar toda a informação associada a uma dada luz.

```
struct {
  struct {
    int h, w, sx, sy;
    char title[64];
  } win;
  camera cam;
  std::vector<struct trans> transformations;
  std::vector<struct prims> primitives;
  std::vector<struct light> lights;
} world;
```

Adicionamos um vetor para guardar as nossas luzes.

3.2 Leitura xml, loading texturas e leitura ficheiros 3D

Na leitura do ficheiro xml temos agora que ler também as luzes e as texturas. Uma adaptação simples no nosso código, pois vai em conta com o resto da leitura do ficheiro xml.

O *loading* das texturas é feito através da função fornecida nas aulas. A mesma é chamada sempre que um ficheiro 3D é lido no XML e é associado e guardado o *texID* à primitiva. Temos também uma flag a indicar se deve ser utilizado *MipMapping* ou não.

Ao lermos o nosso novo ficheiro 3D só temos que ter em conta que temos que ler mais 5 valores, 3 das normais e 2 das coordenadas de textura.

3.3 VBO's para normais e texturas

Com a adição das normais e texturas temos que guardar a informação dos mesmos em VBO's. Para tal fazemos como nas aulas.

```
// Generate the VBO
```

```
glGenBuffers(1, &aux.vbo);
glGenBuffers(1, &aux.ibo);
glGenBuffers(1, &aux.normals);
glGenBuffers(1, &aux.texCoord);

// 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);

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

// Bind the textures
glBindBuffer(GL_ARRAY_BUFFER, aux.texCoord);
glBufferData(GL_ARRAY_BUFFER, sizeof(doubles) * texCoord.size() , texCoord.data(),
    GL_STATIC_DRAW);

prims.push_back(aux);
triple center = {0.F,0.F,0.F};
triple extents = getBoxInfo(coords);
prims[i].box = AABox(center, extents.x, extents.y, extents.z);
```

3.4 Desenho com luzes, normais e texturas

Na nossa função *renderScene*, temos que, para cada luz, aplicar as devidas funções com os devidos valores.

```
for (int i = 0; i < world.lights.size(); i++) {
  if (!strcmp(world.lights[i].type, "point")) {
      float pos[4] = {world.lights[i].posX, world.lights[i].posY, world.lights[i].posZ,
    1.0};
     glLightfv(GL_LIGHT0 + i, GL_POSITION, pos);
     } else if (!strcmp(world.lights[i].type, "directional")) {
      float dir[4] = {world.lights[i].dirX, world.lights[i].dirY, world.lights[i].dirZ,
    0.0};       glLightfv(GL_LIGHT0 + i, GL_POSITION, dir);
  } else if (!strcmp(world.lights[i].type, "spot")) {
      float pos[4] = {world.lights[i].posX, world.lights[i].posY, world.lights[i].posZ,
    1.0};
     float dir[3] = {world.lights[i].dirX, world.lights[i].dirY, world.lights[i].dirZ};
      glLightfv(GL_LIGHT0 + i, GL_POSITION, pos);
    glLightfv(GL_LIGHT0 + i, GL_SPOT_DIRECTION, dir);
    glLightf(GL_LIGHT0 + i, GL_SPOT_CUTOFF, world.lights[i].cutoff);
      }
    }
```

No desenho, temos agora também que aplicar a cor do material com a função *glMaterialfv* e *glMaterialf* e caso o mesmo não tenha cor definida é aplicada a cor pré-definida fornecida pelo professor.

Temos também que utilizar os VBO's das normais e das texturas no desenho.

```
glBindTexture(GL_TEXTURE_2D, world.primitives[k].texID[g]);
```

```
2
3  glBindBuffer(GL_ARRAY_BUFFER, prims[i].vbo);
4  glVertexPointer(3,GL_FLOAT,0,0);
5
6  glBindBuffer(GL_ARRAY_BUFFER, prims[i].normals);
7  glNormalPointer(GL_FLOAT, 0, 0);
8
9  glBindBuffer(GL_ARRAY_BUFFER, prims[i].texCoord);
10 glTexCoordPointer(2, GL_FLOAT, 0, 0);
11
12 glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, prims[i].ibo);
13 total_proc++;
14 if (vfc)
15     prims[i].box.applyMVP(matrix);
16 if (!vfc || frustum.BoxInFrustum(prims[i].box) != Frustum::OUTSIDE) {
17     glDrawElements(GL_TRIANGLES,
18                    prims[i].index_count, // n mero de  ndices  a desenhar
19                     GL_UNSIGNED_INT, // tipo de dados dos indices
20                        0);// par metro n o utilizado
21     drawn++;
22  }
23  glBindTexture(GL_TEXTURE_2D, 0);
```

4 View Frustum Culling

De forma a melhorar o performance do Motor, decidimos implementar a técnica de View Frustum Culling, onde, dependendo da posição de um objeto em relação ao View Frustum, este é, ou não, enviado para o GPU para ser renderizado.

4.1 AABBs

O teste da inclusão/interseção do Frustum é feito com auxilio de "Caixas" alinhadas aos eixos - AABBs. Estas são definidas através de dois triplos, `center` e `extents`.

4.1.1 Atualização de AABBs

De forma a eficientemente realizar o teste, temos de verificar que, com as eventuais transformações geométricas, estas caixas continuam alinhadas aos eixos, para tal temos de aplicar as matrizes de transformação aos seus atributos.

4.2 Descoberta do Frustum

Através da câmera, conseguimos gerar todos os planos do Frustum:

```
1  Frustum(camera cam) {
2      const float Hfar = 2.F * tanf(TO_RADIANS(cam.proj.x*.5F)) * cam.proj.z,
3              Wfar = Hfar * cam.ratio,
4              Hnear = 2.F * tanf(TO_RADIANS(cam.proj.x*.5F)) * cam.proj.y,
5              Wnear = Hnear * cam.ratio;
6      triple HNear_up, WNear_r;
7      triple d, up = cam.up, r;
8      triple HFar_up, WFar_r;
9      triple fvc, nvc;
10     triple fc, nc;
11     triple v1,v2,v3;
```

```
    triple fbr, ftr, fbl,ftl,
        nbr, ntr, nbl, ntl;


    /* d, r and real up*/
    sub(cam.lookAt, cam.pos, d);
    normalize(d);
    cross(up, d, r);
    normalize(r);
    cross(d,r, up);

    scalar(up, Hfar*0.5F, HFar_up), scalar(r,Wfar*0.5F, WFar_r);
    scalar(d,cam.proj.z, fvc);
    /* Far Frustum Points*/
    add(cam.pos, fvc, fc);
    add(fc,HFar_up,ftl),sub(ftl,WFar_r,ftl);
    add(fc,HFar_up,ftr),add(ftr,WFar_r,ftr);
    sub(fc, HFar_up, fbl),sub(fbl,WFar_r,fbl);
    sub(fc,HFar_up, fbr), add(fbr,WFar_r, fbr);

    scalar(up, Hnear*0.5F, HNear_up), scalar(r,Wnear*0.5F, WNear_r);
    scalar(d,cam.proj.y, nvc);
    /* Near Frustum Points*/
    add(cam.pos, nvc, nc);
    add(nc,HNear_up,ntl),sub(ntl,WNear_r,ntl);
    add(nc,HNear_up,ntr),add(ntr,WNear_r,ntr);
    sub(nc, HNear_up, nbl),sub(nbl,WNear_r,nbl);
    sub(nc,HNear_up, nbr), add(nbr,WNear_r, nbr);

    pl[far] = Plane(ftr,ftl,fbl);
    pl[near] = Plane(ntl,ntr,nbr);
    pl[top] = Plane(ntr,ntl,ftl);
    pl[bot] = Plane(nbl,nbr,fbr);
    pl[left] = Plane(ntl,nbl,fbl);
    pl[right] = Plane(nbr,ntr,fbr);
}
```

Note-se que o plano aqui é determinado por uma normal e uma distância à origem, ambos quais conseguimos computar de forma simples através do "cross product" e do "dot product", respetivamente.

```
Plane( triple& v1,  triple& v2,  triple& v3)
{
    triple aux1, aux2;

    sub(v1, v2, aux1);
    sub(v3,v2, aux2);

    cross(aux2,aux1,normal);

    normalize(normal);
    point.copy(v2);
    d = -dot(normal,point);
}
```

A computação do frustum ocorrerá sempre que a câmera seja alterada.

4.3 Teste da Caixa contra o Frustum

Tendo a segurança que todos os objetos (sendo irrelevante posição, coordenadas, tamanho, ou tipo de primitiva) estão "cobertos" por uma **AABB**, podemos aplicar um teste extremamente simples que implica determinar os chamados vértices positivos e negativos de uma AABB, dependendo da normal do plano e verificar se, de facto, se encontra dentro, ou fora, do frustum através da distância "signed":

```
float distance(triple& p)
{
    return (d + dot(normal, p));
}
triple getVertexP(triple&normal)
{
    triple res;
    res.x = (normal.x < 0) ? (center.x - extents.x) : (center.x + extents.x);
    res.y = (normal.y < 0) ? (center.y - extents.y) : (center.y + extents.y);
    res.z = (normal.z < 0) ? (center.z - extents.z) : (center.z + extents.z);

    return res;
}
triple getVertexN(triple&normal)
{
    triple res = center;
    res.x = (normal.x > 0) ? (center.x - extents.x) : (center.x + extents.x);
    res.y = (normal.y > 0) ? (center.y - extents.y) : (center.y + extents.y);
    res.z = (normal.z > 0) ? (center.z - extents.z) : (center.z + extents.z);

    return res;
}
bool BoxInFrustum(AABox& b)
{

    triple tmp1, tmp2;
    bool res= INSIDE;
    for(int i=0; i < 6; i++) {
        tmp1 = b.getVertexP(pl[i].normal);
        tmp2 = b.getVertexN(pl[i].normal);
        if (pl[i].distance(tmp1) > 0)
            return  OUTSIDE;
        else if (pl[i].distance(tmp2) < 0)
            return INTERSECT;
    }
    return res;

}
```

Finalmente, estamos em condições para aplicar estes métodos no nosso motor:

```
total_proc++;
if (vfc)
    prims[i].box.applyMVP(matrix);
if (!vfc || frustum.BoxInFrustum(prims[i].box) != Frustum::OUTSIDE) {
    glDrawElements(GL_TRIANGLES,
                   prims[i].index_count, // n mero de  ndices  a desenhar
                   GL_UNSIGNED_INT, // tipo de dados dos  ndices
                   0);// par metro n o utilizado
    drawn++;
}
```

5 Conclusão

Efetuamos as alterações necessárias para sermos capazes de gerar normais, coordenadas de textura e ser também, capaz de interpretar as mesmas. Evoluímos o nosso sistema solar para incluir iluminação e texturas. Para além disto, também implementamos uma tática de otimização na forma do View Frustum Culling.

6 Resultados

Os resultados obtidos para cada teste fornecido.

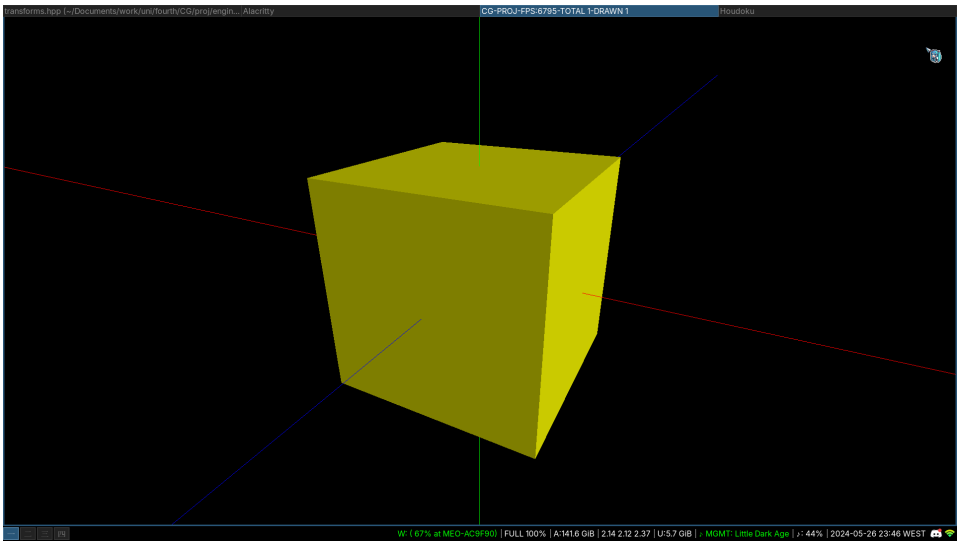

Figura 1: Teste 1

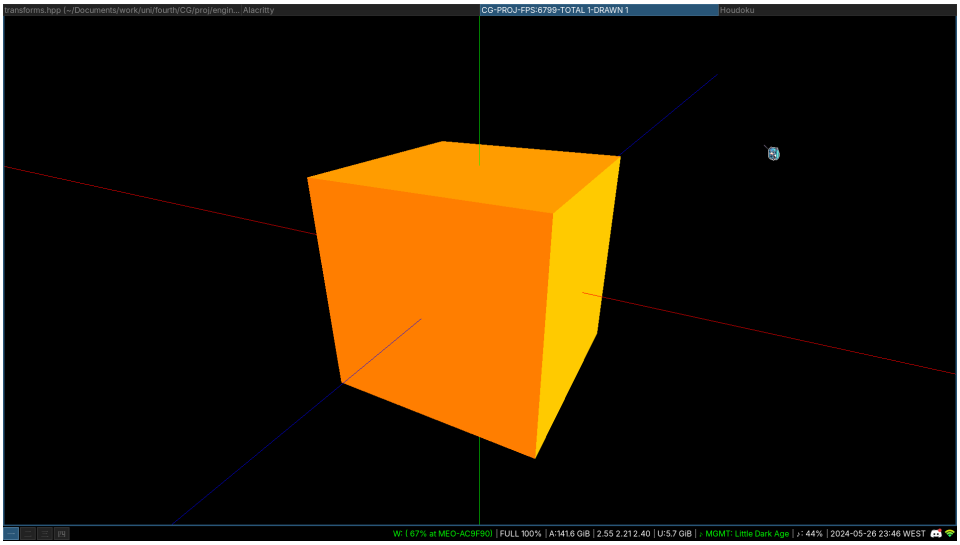

Figura 2: Teste 2

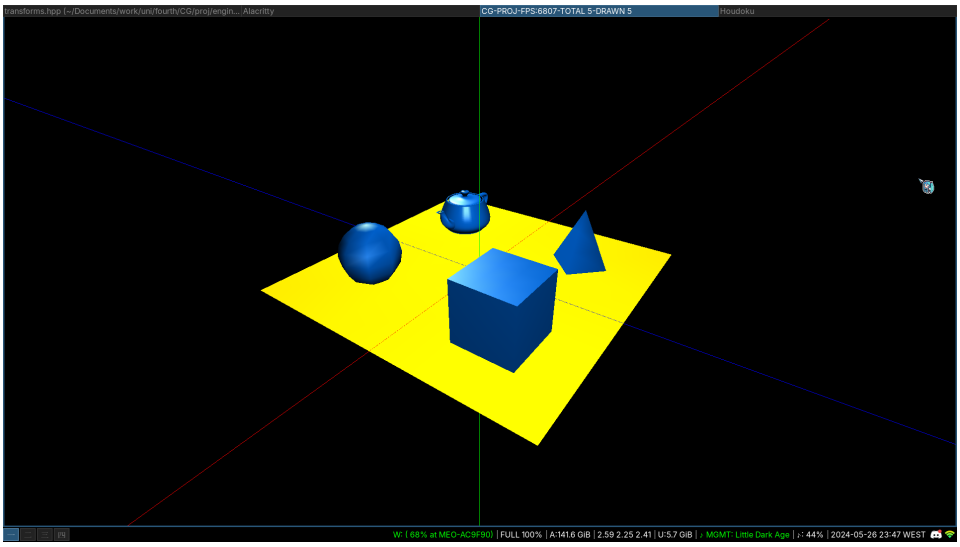

Figura 3: Teste 3

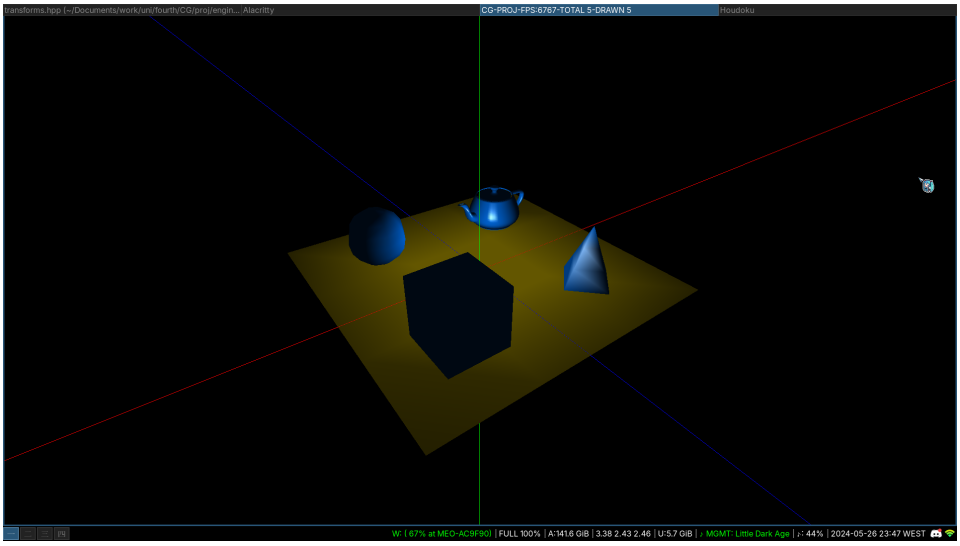

Figura 4: Teste 4

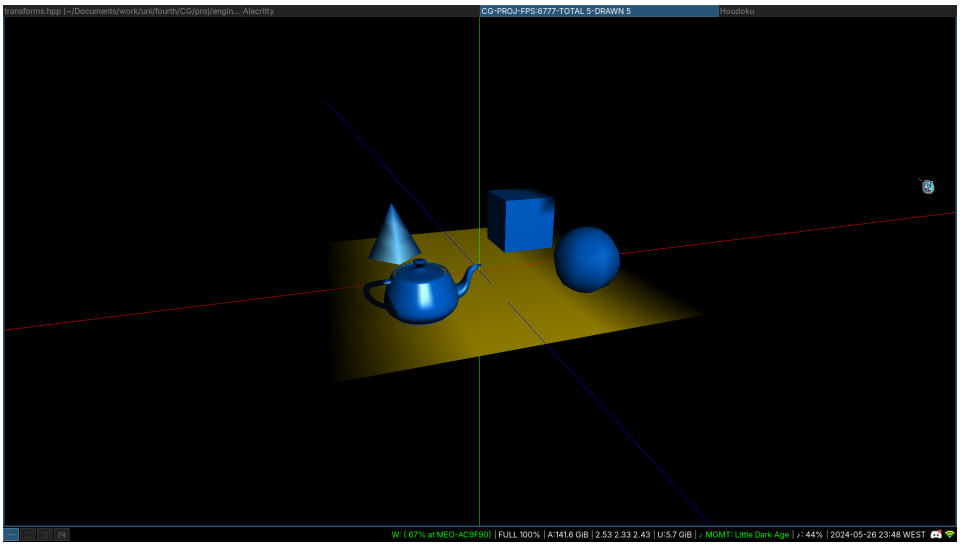

Figura 5: Teste 5

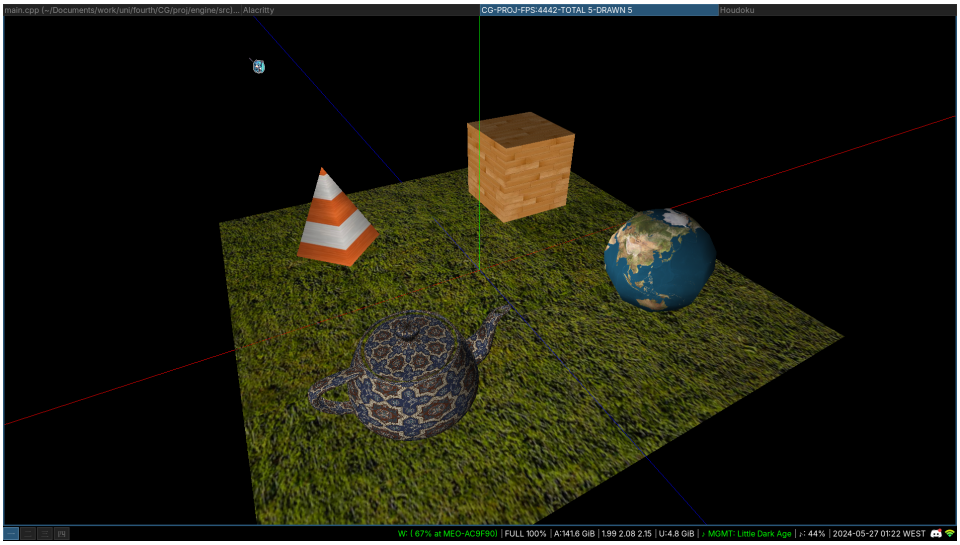

Figura 6: Teste 6

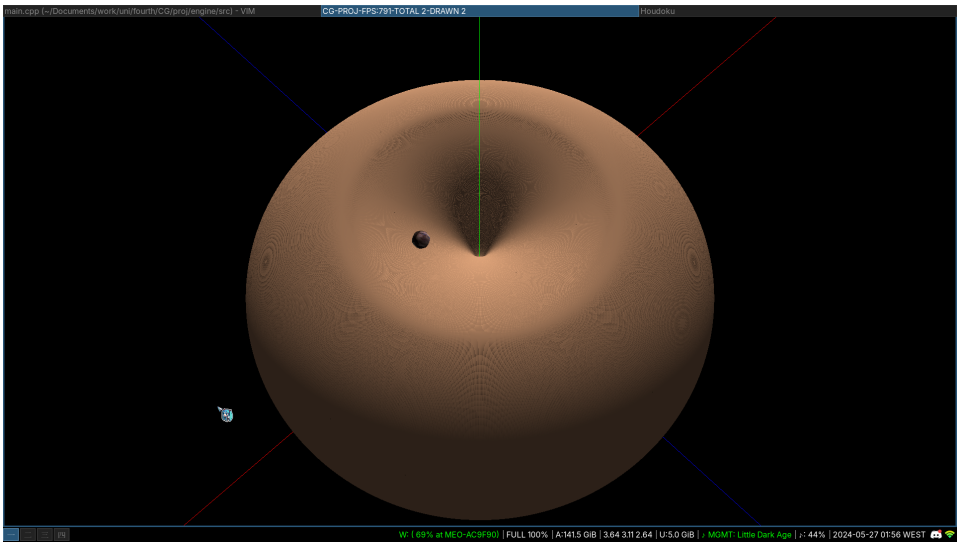

Figura 7: Teste 7

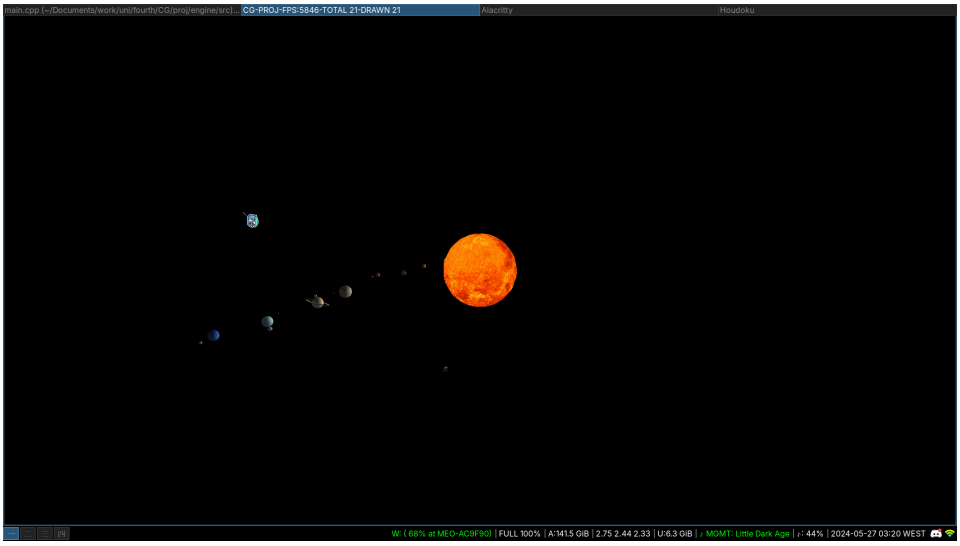

Figura 8: Sistema Solar