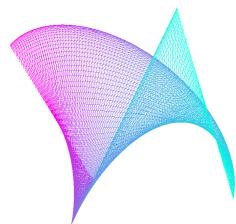


Universidade do Minho
Departamento de Informática

Solaris



Computação Gráfica
Grupo 24 - Fase 4

5 de junho, 2022



Beatriz
Rodrigues
(a93230)



Francisco Neves
(a93202)



Guilherme
Fernandes
(a93216)



João Carvalho
(a93166)

Índice

| | | |
|----------|--------------------------------|-----------|
| 1 | Introdução | 3 |
| 2 | <i>Generator</i> | 4 |
| 2.1 | Plano | 4 |
| 2.2 | Caixa | 5 |
| 2.3 | Cone | 7 |
| 2.4 | Esfera | 9 |
| 2.5 | Cilindro | 10 |
| 2.6 | Torus | 12 |
| 2.7 | Patches de Bezier | 13 |
| 3 | <i>Engine</i> | 15 |
| 3.1 | Model | 15 |
| 3.1.1 | ModelBuffer | 15 |
| 3.1.2 | Cores | 17 |
| 3.2 | <i>TextureBuffer</i> | 17 |
| 3.3 | Iluminação | 19 |
| 3.3.1 | <i>Parse State</i> | 20 |
| 4 | <i>Script</i> | 21 |
| 5 | <i>Scenes</i> | 22 |
| 5.1 | Sistema Solar | 22 |
| 5.2 | Boneco de Neve | 26 |
| 5.3 | Desportos | 26 |
| 5.4 | Ovo da Páscoa | 27 |
| 6 | Testes | 28 |
| 7 | Conclusões | 33 |

1. Introdução

De forma a permitir a evolução do projeto *solaris*, nesta fase, sendo a última do projeto, é requerida a definição de luzes e texturas. Para isto ser possível, foram efectuadas diversas alterações, tanto nos ficheiros das primitivas presentes no *Generator*, como na *Engine* do programa.

Assim, aliada à alteração na *script* geradora do sistema solar, permite-se a presença de texturas nos diversos planetas, bem como de luzes realistas no sistema em questão.

Por fim, foram ainda criadas diversas outras *scenes* que permitem verificar as mais diversas potencialidades do sistema em questão, apesar deste ter o foco claro no desenvolvimento do Sistema Solar.

2. Generator

De forma a efetuar as mudanças necessárias e pretendidas, foi requerida a alteração da construção modo dos ficheiros *.3d* das primitivas que o programa contém. Assim, foi alterado o modo de construção destes ficheiros fornecendo, além dos pontos dos sólidos, os seus vetores normais e as suas coordenadas de textura, visto que uma textura em OpenGL irá estar contida num referencial 2D com ambos os eixos entre 0 e 1.

Assim, foram alteradas as seguintes primitivas:

2.1 Plano

Tendo em conta que os planos gerados acabam por ser sempre paralelos ao plano xz, é possível determinar que as normais nesta primitiva irão corresponder sempre a (0,1,0).

Relativamente à textura, esta é aplicada repetidamente ao longo do plano, tal como demonstrado no código abaixo, em que *tn* corresponde às coordenadas de textura de um ponto *pn*.

```
for (size_t i = 0; i < n_divisions; ++i) {
    for (size_t j = 0; j < n_divisions; ++j) {
        auto t1 = Vec2(j, i);
        auto t2 = Vec2(j, i + 1);
        auto t3 = Vec2(j + 1, i);
        auto t4 = Vec2(j + 1, i + 1);

        auto i1 = t1 / nd;
        auto i2 = t2 / nd;
        auto i3 = t3 / nd;
        auto i4 = t4 / nd;

        auto p1 = starting_point + length * Vec3::cartesian(i1.x(), 0, i1.y());
        auto p2 = starting_point + length * Vec3::cartesian(i2.x(), 0, i2.y());
        auto p3 = starting_point + length * Vec3::cartesian(i3.x(), 0, i3.y());
        auto p4 = starting_point + length * Vec3::cartesian(i4.x(), 0, i4.y());

        points.push_back(Vertex(p1, n, t1));
        points.push_back(Vertex(p2, n, t2));
        points.push_back(Vertex(p3, n, t3));
```

```

        points.push_back(Vertex(p2, n, t2));
        points.push_back(Vertex(p4, n, t4));
        points.push_back(Vertex(p3, n, t3));
    }
}

```

Desta forma, aplicando a seguinte textura, obtém-se o resultado associado.



Figura 2.1: Textura

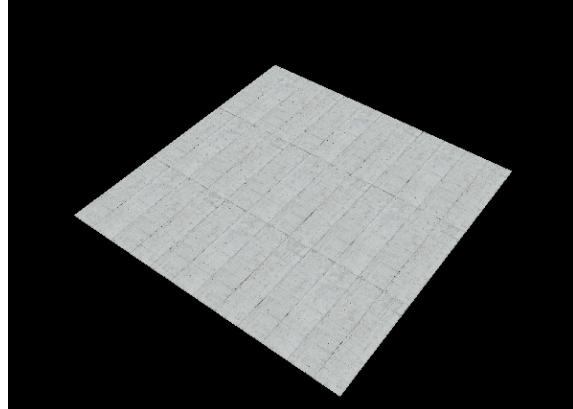


Figura 2.2: Plano com textura

2.2 Caixa

As normais dos pontos da caixa são obtidas tendo em conta o lado na qual este está contido. Por exemplo, se estiver no topo da caixa, a normal é $(0,1,0)$, enquanto que se estiver na base da caixa é $(0,-1,0)$.

Por outro lado, para uma dada textura como a indicada abaixo, esta é dividida horizontalmente em quartos e verticalmente em terços, e posteriormente aplicada. Segue-se abaixo código abreviado que ilustra a aplicação da textura à base e ao topo da caixa.

```

auto n_base = Vec3::cartesian(0, -1, 0);
auto t_base = Vec2(2/4.f, 0.f);

...

auto n_top = Vec3::cartesian(0, 1, 0);
auto t_top = Vec2(2/4.f, 1.f);

for (size_t j = 0; j < n_divisions; ++j) {
    for (size_t k = 0; k < n_divisions; ++k) {
        auto i1 = Vec2(k, j) / nd;
        auto i2 = Vec2(k, j + 1) / nd;
        auto i3 = Vec2(k + 1, j) / nd;
        auto i4 = Vec2(k + 1, j + 1) / nd;

        // Base triangles

```

```

auto p1 =
    starting_point + length * Vec3::cartesian(i1.x(), 0, i1.y());
auto t1 = Vec2(t_base.x() - i1.y() / 4.0f, t_base.y() + i1.x() / 3.0f);

auto p2 =
    starting_point + length * Vec3::cartesian(i2.x(), 0, i2.y());
auto t2 = Vec2(t_base.x() - i2.y() / 4.0f, t_base.y() + i2.x() / 3.0f);

auto p3 =
    starting_point + length * Vec3::cartesian(i3.x(), 0, i3.y());
auto t3 = Vec2(t_base.x() - i3.y() / 4.0f, t_base.y() + i3.x() / 3.0f);

auto p4 =
    starting_point + length * Vec3::cartesian(i4.x(), 0, i4.y());
auto t4 = Vec2(t_base.x() - i4.y() / 4.0f, t_base.y() + i4.x() / 3.0f);

points.push_back(Vertex(p3, n_base, t3));
points.push_back(Vertex(p2, n_base, t2));
points.push_back(Vertex(p1, n_base, t1));

points.push_back(Vertex(p3, n_base, t3));
points.push_back(Vertex(p4, n_base, t4));
points.push_back(Vertex(p2, n_base, t2));

// Top triangles
p1.set_y(length / 2);
t1 = Vec2(t_top.x() - i1.y() / 4.0f, t_top.y() - i1.x() / 3.0f);

p2.set_y(length / 2);
t2 = Vec2(t_top.x() - i2.y() / 4.0f, t_top.y() - i2.x() / 3.0f);

p3.set_y(length / 2);
t3 = Vec2(t_top.x() - i3.y() / 4.0f, t_top.y() - i3.x() / 3.0f);

p4.set_y(length / 2);
t4 = Vec2(t_top.x() - i4.y() / 4.0f, t_top.y() - i4.x() / 3.0f);

points.push_back(Vertex(p1, n_top, t1));
points.push_back(Vertex(p2, n_top, t2));
points.push_back(Vertex(p3, n_top, t3));

points.push_back(Vertex(p2, n_top, t2));
points.push_back(Vertex(p4, n_top, t4));
points.push_back(Vertex(p3, n_top, t3));

...
}

}

```

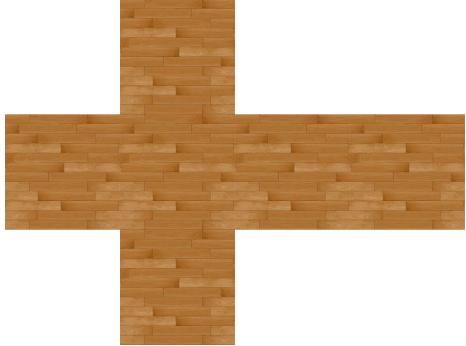


Figura 2.3: Textura

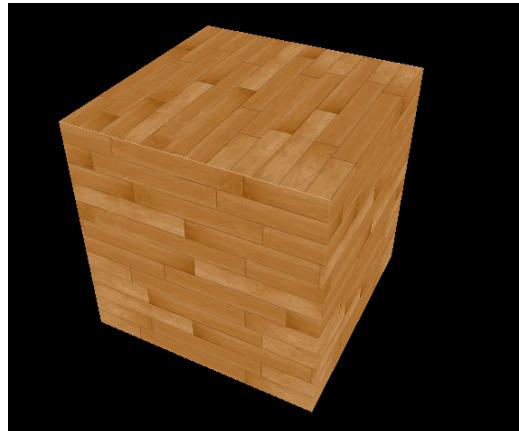


Figura 2.4: Caixa com textura

2.3 Cone

Em primeiro lugar, indentificou-se que a normal dos pontos contidos na base desta primitiva será $(0, -1, 0)$. No entanto, para os restantes pontos é necessário efetuar o seu cálculo, através do ângulo α (que corresponde ao ângulo da *slice* em que o ponto está contido) e do ângulo β , que é calculado por

$$\tan(\frac{\text{raio}}{\text{hipotenusa}})$$

Para as texturas, é necessária uma imagem em que o lado esquerdo representa a lateral do cone, enquanto o lado direito representa a base. Representamos em vetores de 2 dimensões as seguintes informações:

- Centro da parte da textura correspondente à lateral do cone $\rightarrow (1/4, 1/2)$
- Centro da parte da textura correspondente à base do cone $\rightarrow (3/4, 1/2)$
- Raio $\rightarrow (1/4, 1/2)$, em que $1/4$ é o raio na horizontal e $1/2$ o raio na vertical

Através das informações anteriores, sabemos aplicar as texturas, tendo em conta a *stack* e a *slice* em que se encontra o ponto de forma a adequar à posição da textura, como evidenciado na porção de código que se segue.

```

auto t_top_center = Vec2(1.f / 4.f, 1.f / 2.f);
auto t_bot_center = Vec2(3.f / 4.f, 1.f / 2.f);
auto t_radius = Vec2(1.f / 4.f, 1.f / 2.f);

for (size_t stack = 0; stack < n_stacks; ++stack) {
    auto next_radius = radius + (radius_step * (stack + 1));
    auto next_height = stack_height * (stack + 1);

    auto curr_alpha = 0.f;
    for (size_t slice = 0; slice < n_slices; ++slice) {

        auto next_alpha = alpha * (slice + 1);
    }
}

```

```

auto n_side = Vec3::spherical(1.f, curr_alpha, beta);
auto n_side_next = Vec3::spherical(1.f, next_alpha, beta);

...
auto t1 = t_top_center +
    next_radius * t_radius * Vec2(sin(curr_alpha), cos(curr_alpha));
auto t2 = t_top_center +
    next_radius * t_radius * Vec2(sin(next_alpha), cos(next_alpha));
auto t3 = t_top_center +
    curr_radius * t_radius * Vec2(sin(curr_alpha), cos(curr_alpha));
auto t4 = t_top_center +
    curr_radius * t_radius * Vec2(sin(next_alpha), cos(next_alpha));

...
if (stack == 0) {
    // draw base
    auto o = Vec3::cartesian(0, 0, 0);
    points.push_back(
        Vertex(
            p4, n_bot, t_bot_center + t_radius * Vec2(
                sin(next_alpha), cos(next_alpha)
            )
        )
    );
    points.push_back(
        Vertex(
            p3, n_bot, t_bot_center + t_radius * Vec2(
                sin(curr_alpha), cos(curr_alpha)
            )
        )
    );
    points.push_back(Vertex(o, n_bot, t_bot_center));
}
curr_alpha = next_alpha;
}

curr_radius = next_radius;
curr_height = next_height;
}

```

A aplicacão da textura pode ter um resultado como o ilustrado de seguida.



Figura 2.5: Textura

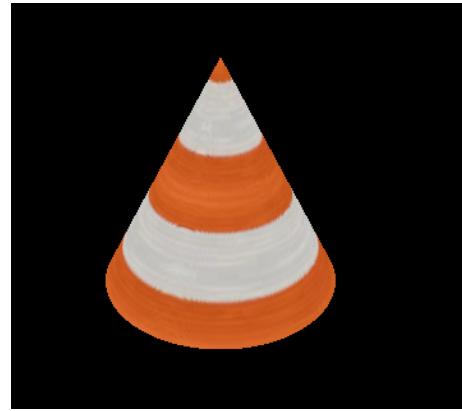


Figura 2.6: Cone com textura

2.4 Esfera

O cálculo da normal de um ponto na esfera é efetuado através da normalização das coordenadas do ponto, uma vez que podem representar o vetor que o une ao centro da esfera.

Para corresponder um dado ponto a um ponto na textura, basta dividir o número da *stack* atual pelo número total de *stacks* e dividir o número da *slice* atual pelo número total de *slices*.

```
auto ns1 = static_cast<float>(n_slices);
auto nst = static_cast<float>(n_stacks);
auto stacks_height = M_PI / nst;
auto alpha = 2 * M_PI / ns1;
auto current_stack_beta = M_PI / 2.f;

for (size_t stack = 0; stack < n_stacks; ++stack) {
    auto next_stack_beta = M_PI / 2.f - stacks_height * (stack + 1);
    auto current_slice_alpha = 0.0f;
    auto current_tex_y = (n_stacks - stack) / nst;
    auto next_tex_y = (n_stacks - stack - 1) / nst;

    for (size_t slice = 0; slice < n_slices; ++slice) {
        auto next_slice_alpha = alpha * (slice + 1);
        auto current_tex_x = slice / ns1;
        auto next_tex_x = (slice + 1) / ns1;
        ...

        Vertex vtx1 = Vertex(p1, Vec3(p1).normalize(),
                             Vec2(current_tex_x, current_tex_y));
        Vertex vtx2 = Vertex(p2, Vec3(p2).normalize(),
                             Vec2(next_tex_x, current_tex_y));
        Vertex vtx3 = Vertex(p3, Vec3(p3).normalize(),
                             Vec2(current_tex_x, next_tex_y));
        Vertex vtx4 = Vertex(p4, Vec3(p4).normalize(),
                             Vec2(next_tex_x, next_tex_y));
        ...

        current_slice_alpha = next_slice_alpha;
```

```

    }

    current_stack_beta = next_stack_beta;
}

```

O resultado pode ser algo do género das seguintes figuras.



Figura 2.7: Textura



Figura 2.8: Esfera com textura

2.5 Cilindro

As normais dos pontos da base e do topo do cilindro são simples de obter, visto que são, respetivamente, $(0, -1, 0)$ e $(0, 1, 0)$. No entanto, para os pontos nas laterais, determinamos que a normal de um ponto será correspondente ao raio que o une à projeção do centro da base do cilindro para a mesma *stack* em que se encontra, dependendo depois então de um ângulo α , : $(\sin(\alpha), 0, \cos(\alpha))$.

Quanto à textura, é necessário definirmos alguns valores essenciais:

- Centro do topo $\rightarrow (7/16, 3/16)$
- Centro da base $\rightarrow (13/16, 3/16)$
- Início da lateral $\rightarrow (0, 3/8)$
- Altura da lateral $\rightarrow 5/8$
- Raio $\rightarrow 3/16$

Através do enunciado acima, conseguimos deduzir as coordenadas de textura de todos os outros pontos. Os pontos presentes na lateral do cilindro são dependentes da *slice* da *stack* em que estão contidos que influencia a distância ao ponto de início da lateral, enquanto que a base e o topo dependem do ângulo α , que influencia a posição relativa à parte da textura que lhes é adequada.

```

for (size_t stack = 0; stack < n_stacks; ++stack) {
    auto next_height = stack_height * (stack + 1);

    auto curr_alpha = 0.0f;
    for (size_t slice = 0; slice < n_slices; ++slice) {
        auto next_alpha = alpha * (slice + 1);

        auto nc = Vec3::cartesian(sin(curr_alpha), 0, cos(curr_alpha));
        auto nn = Vec3::cartesian(sin(next_alpha), 0, cos(next_alpha));

        ...
        auto t1 = t_side +
            Vec2(slice / ns1, (stack + 1) * t_side_height / nst);
        ...
        auto t2 = t_side +
            Vec2((slice + 1) / ns1, (stack + 1) * t_side_height / nst);
        ...
        auto t3 = t_side +
            Vec2(slice / ns1, stack * t_side_height / nst);
        ...
        auto t4 = t_side +
            Vec2((slice + 1) / ns1, stack * t_side_height / nst);

        ...

        if (stack == 0) {
            // draw base
            Vec3 o = Vec3::cartesian(0, 0, 0);
            points.push_back(Vertex(
                p4, n_bot, t_bot_center + t_radius * Vec2(
                    sin(next_alpha), cos(next_alpha)
                )
            ));
            points.push_back(Vertex(
                p3, n_bot, t_bot_center + t_radius * Vec2(
                    sin(curr_alpha), cos(curr_alpha)
                )));
            points.push_back(Vertex(o, n_bot, t_bot_center));
        }

        if (stack == n_stacks - 1){
            // draw top
            Vec3 o = Vec3::cartesian(0, next_height, 0);
            points.push_back(Vertex(o, n_top, t_top_center));
            points.push_back(Vertex(
                p1, n_top, t_top_center + t_radius * Vec2(
                    sin(curr_alpha), cos(curr_alpha)
                )
            ));
        }
    }
}

```

```

        points.push_back(Vertex(
            p2, n_top, t_top_center + t_radius * Vec2(
                sin(next_alpha), cos(next_alpha)
            )
        ));
    }
    curr_alpha = next_alpha;
}
curr_height = next_height;

}

```

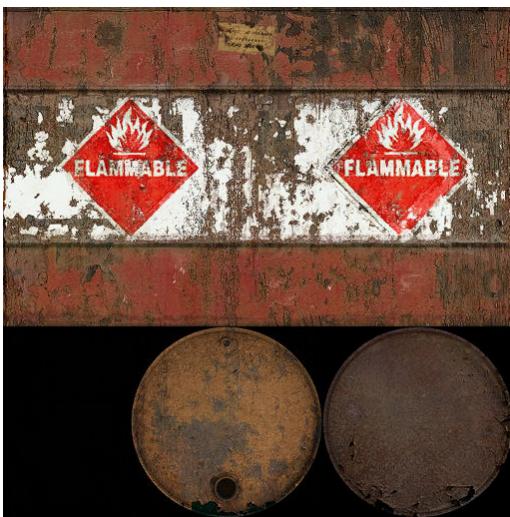


Figura 2.9: Textura

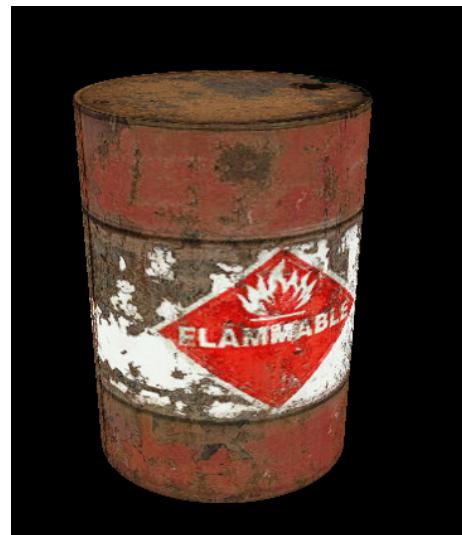


Figura 2.10: Cilindro com textura

2.6 Torus

As normais do torus calculam-se de acordo com o ângulo Φ e Θ . Seguem a mesma lógica que a esfera, mas orientam-se pelo centro do anel do torus. São dadas por

$$(\cos(\Phi) * \cos(\Theta), \sin(\Phi), \cos(\Phi) * \sin(\Theta))$$

Quanto às texturas, tal como na esfera, obtêm-se as coordenadas de textura dividindo a *slice* atual pelo número de *slices* e a *stack* atual pelo número de *stacks*.

```

for(size_t stack = 0; stack < n_stacks; stack++){
    auto next_theta = theta_step * (stack + 1);

    auto current_tex_y = stack / nst;
    auto next_tex_y = (stack + 1) / nst;

    auto curr_phi = 0.f;
    for(size_t slice = 0; slice < n_slices; slice++){
        auto next_phi = phi_step * (slice + 1);

```

```

auto current_tex_x = slice / ns1;
auto next_tex_x = (slice + 1) / ns1;

...
Vec2 t1 = Vec2(current_tex_x, current_tex_y);
Vec2 t2 = Vec2(next_tex_x, current_tex_y);
Vec2 t3 = Vec2(next_tex_x, next_tex_y);
Vec2 t4 = Vec2(current_tex_x, next_tex_y);
...

```

Com isto, é possível obter algo do género das seguintes figuras.



Figura 2.11: Textura

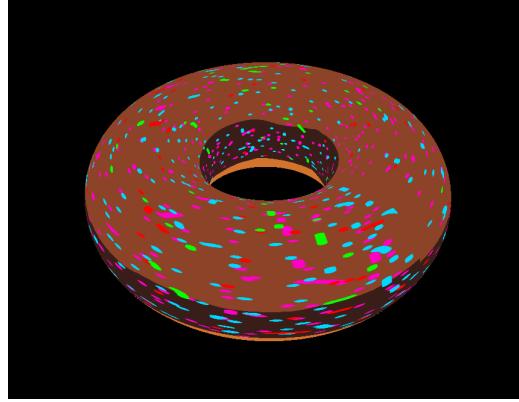


Figura 2.12: Torus com textura

2.7 Patches de Bezier

De modo a ser possível obter a normal de um ponto de um *patch* de *Bezier*, é necessário calcular dois vetores tangentes a este ponto, utilizando as seguintes fórmulas.

$$\frac{\delta p(u, v)}{\delta u} = [3u^2 \quad 2u \quad 1 \quad 0] \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} M^T V^T$$

$$\frac{\delta p(u, v)}{\delta v} = U M \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} M^T \begin{bmatrix} 3v^2 \\ 2v \\ 1 \\ 0 \end{bmatrix}$$

A normalização do *cross product* entre os dois vetores resultantes das operações acima permite obter a normal pretendida.

As coordenadas de textura são simples de obter. Basta dividir os vetores u e v pelo nível de tesselação.

```

for (size_t ui = 0; ui < tess; ++ui) {
    auto u = ui / f_tess;
    auto nu = (ui + 1) / f_tess;

    for (size_t vi = 0; vi < tess; ++vi) {
        auto v = vi / f_tess;
        auto nv = (vi + 1) / f_tess;

        ...

        auto vtx0 = Vertex(p0.first, p0.second, Vec2(u, v));
        auto vtx1 = Vertex(p1.first, p1.second, Vec2(u, nv));
        auto vtx2 = Vertex(p2.first, p2.second, Vec2(nu, v));
        auto vtx3 = Vertex(p3.first, p3.second, Vec2(nu, nv));

        ...
    }
}

```

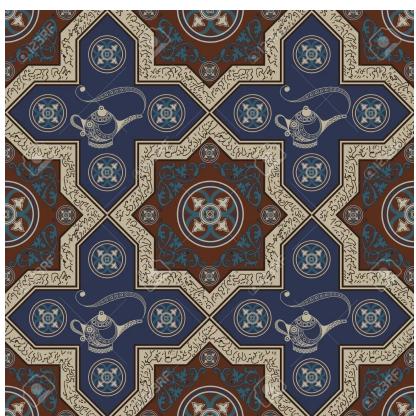


Figura 2.13: Textura



Figura 2.14: Bezier com textura

3. Engine

No lado da *Engine*, no que lhe concerne, foi necessária a ativação de funcionalidades de luzes e texturas, sendo ainda requerido que esta consiga ler e aplicar os novos dados dos ficheiros *.3d* gerados.

Assim, é importante referir que o ficheiro XML deverá então permitir indicar o tipo de luz (*diffuse*, *specular*, *emissive* ou *ambient*) e a *shininess* de um objeto, bem como as origens da luz (*point*, *directional* ou *spot*).

3.1 Model

3.1.1 ModelBuffer

Nesta fase, foi adicionado um GLuint para o índice do *VBO* das normais e das coordenadas de textura.

```
class ModelBuffer {
private:
    GLuint _positions, _normals, _texcoords;
    size_t _n_vertices;
    ...
}
```

Para além disso, a função `ModelBuffer::try_from_file(std::string_view file_path)` passa a ler os novos ficheiros *.3d*, guardando as normais e coordenadas de textura em *VBOs*.

```
auto ModelBuffer::try_from_file(std::string_view file_path) noexcept
    -> cpp::result<ModelBuffer, ParseError>
{
    std::ifstream file_stream(file_path.data());
    if (!file_stream.is_open()) {
        return cpp::fail(ParseError::PRIMITIVE_FILE_NOT_FOUND);
    }
    float x, y, z, nx, ny, nz, tx, ty;
    auto positions = std::vector<float>();
    auto normals = std::vector<float>();
    auto texcoords = std::vector<float>();
    while (file_stream >> x >> y >> z >> nx >> ny >> nz >> tx >> ty) {
        positions.push_back(x);
        positions.push_back(y);
```

```

    positions.push_back(z);

    normals.push_back(nx);
    normals.push_back(ny);
    normals.push_back(nz);

    texcoords.push_back(tx);
    texcoords.push_back(ty);
}

file_stream.close();

GLuint buffers[3];
glGenBuffers(3, buffers);

glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);
glBufferData(
    GL_ARRAY_BUFFER,
    sizeof(float) * positions.size(),
    positions.data(),
    GL_STATIC_DRAW
);

glBindBuffer(GL_ARRAY_BUFFER, buffers[1]);
glBufferData(
    GL_ARRAY_BUFFER,
    sizeof(float) * normals.size(),
    normals.data(),
    GL_STATIC_DRAW
);

glBindBuffer(GL_ARRAY_BUFFER, buffers[2]);
glBufferData(
    GL_ARRAY_BUFFER,
    sizeof(float) * texcoords.size(),
    texcoords.data(),
    GL_STATIC_DRAW
);

return ModelBuffer(
    buffers[0],
    buffers[1],
    buffers[2],
    positions.size()
);
}

```

Por fim, a função de desenho dos *ModelBuffers* passa a dar *bind* aos vetores de normais e coordenadas.

```

auto ModelBuffer::draw() const noexcept -> void {
    glBindBuffer(GL_ARRAY_BUFFER, _positions);

```

```

glVertexPointer(3, GL_FLOAT, 0, 0);
glBindBuffer(GL_ARRAY_BUFFER, _normals);
glNormalPointer(GL_FLOAT, 0, 0);
glBindBuffer(GL_ARRAY_BUFFER, _texcoords);
glTexCoordPointer(2, GL_FLOAT, 0, 0);

glDrawArrays(GL_TRIANGLES, 0, _n_vertices);
}

```

3.1.2 Cores

Quando um model é lido pode ter diferentes componentes de cor associado a ele: *diffuse*, *ambient*, *specular*, *emissive*, *shininess*.

```

class Model {
private:
    ModelBuffer _model_buffer;
    std::optional<TextureBuffer> _texture_buffer;
    Color _diffuse;
    Color _ambient;
    Color _specular;
    Color _emissive;
    float _shininess;
    ...
}

```

Ao dar parse do XML, se um objeto não tiver alguma componente da cor, é utilizada uma componente *default*.

3.2 *TextureBuffer*

Foi decidido que uma textura seria representada pelo *id* do seu *VBO*. Para além disso, é ainda importante referir que esta é opcional para o *Model*.

```

class TextureBuffer {
private:
    GLuint _id;
    ...
}

```

A leitura do ficheiro de textura é feita utilizando a biblioteca DevIL.

```

auto TextureBuffer::try_from_file(std::string_view file_path) noexcept
-> cpp::result<TextureBuffer, ParseError>
{
    GLuint t;
    ilGenImages(1, &t);
    ilBindImage(t);
    if (!ilLoadImage(file_path.data())) {
        return cpp::fail(ParseError::TEXTURE_FILE_NOT_FOUND);
    }
}

```

```

auto width = ilGetInteger(IL_IMAGE_WIDTH);
auto height = ilGetInteger(IL_IMAGE_HEIGHT);

ilConvertImage(IL_RGBA, IL_UNSIGNED_BYTE);

auto image_data = ilGetData();

GLuint id;
 glGenTextures(1, &id);

glBindTexture(GL_TEXTURE_2D, id);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA,
             GL_UNSIGNED_BYTE, image_data);
glGenerateMipmap(GL_TEXTURE_2D);

return TextureBuffer(id);
}

```

Para desenhar um modelo, é necessário dar *bind* à textura.

```

auto TextureBuffer::bind() const noexcept -> void {
    glBindTexture(GL_TEXTURE_2D, _id);
}

```

Quando um modelo é desenhado, são aplicadas as diferentes componentes da cor utilizando a função `glMaterialfv`, é dado *bind* à textura e desenhado o modelo. Por fim é dado *unbind* à textura.

```

auto Model::draw() const noexcept -> void {
    float d[] = {_diffuse.r, _diffuse.g, _diffuse.b, 1.0f};
    float a[] = {_ambient.r, _ambient.g, _ambient.b, 1.0f};
    float s[] = {_specular.r, _specular.g, _specular.b, 1.0f};
    float e[] = {_emissive.r, _emissive.g, _emissive.b, 1.0f};
    glMaterialfv(GL_FRONT, GL_DIFFUSE, d);
    glMaterialfv(GL_FRONT, GL_AMBIENT, a);
    glMaterialfv(GL_FRONT, GL_SPECULAR, s);
    glMaterialfv(GL_FRONT, GL_EMISSION, e);
    glMaterialf(GL_FRONT, GL_SHININESS, _shininess);
    if(_texture_buffer) {
        _texture_buffer->bind();
    }
    _model_buffer.draw();
    glBindTexture(GL_TEXTURE_2D, 0);
}

```

3.3 Iluminação

Uma luz pode ser pontual, direcional ou *spotlight*. Cada luz possui um *id* e está associada a um método para a posicionar.

```
class Light {
protected:
    GLenum _id;
    auto static try_get_new_id() noexcept -> cpp::result<GLenum, ParseError>;
    Light(GLenum id) : _id(id) {}

public:
    void turn_on() const noexcept;
    auto virtual place() const noexcept -> void = 0;
};

class PointLight: public Light {
private:
    Vec3 _position;
    ...
}

class DirectionalLight: public Light {
private:
    Vec3 _direction;
    ...
}

class SpotLight: public Light {
private:
    Vec3 _position;
    Vec3 _direction;
    float _cutoff;
    ...
}
```

Quando é adicionada uma nova luz, primeiro verifica-se se com a inserção desta se ultrapassa o número limite de luzes do OpenGL (8).

```
auto Light::try_get_new_id() noexcept -> cpp::result<GLenum, ParseError> {
    static GLenum curr_light_id = GL_LIGHT0;
    if (curr_light_id > GL_LIGHT7) {
        return cpp::fail(ParseError::LIGHT_LIMIT);
    }
    return curr_light_id++;
}
```

As luzes são ligadas no arranque do programa, antes de entrar no *loop* principal do glut. Uma vez que as luzes diferentes da GL_LIGHT0 são pretas é ainda necessário alterá-las para branco.

```

auto Light::turn_on() const noexcept -> void {
    static GLfloat dark[4] = {0.2, 0.2, 0.2, 1.0};
    static GLfloat white[4] = {1.0, 1.0, 1.0, 1.0};

    glEnable(_id);
    glLightfv(_id, GL_AMBIENT, dark);
    glLightfv(_id, GL_DIFFUSE, white);
    glLightfv(_id, GL_SPECULAR, white);
}

```

Em cada *frame*, as luzes são posicionadas com o método `place()`.

```

auto PointLight::place() const noexcept -> void {
    float pos[4] = {_position.x(), _position.y(), _position.z(), 1.0f};
    glLightfv(_id, GL_POSITION, pos);
}

auto DirectionalLight::place() const noexcept -> void {
    float dir[4] = {_direction.x(), _direction.y(), _direction.z(), 0.0f};
    glLightfv(_id, GL_POSITION, dir);
}

auto SpotLight::place() const noexcept -> void {
    float pos[4] = {_position.x(), _position.y(), _position.z(), 1.0f};
    float dir[4] = {_direction.x(), _direction.y(), _direction.z(), 0.0f};
    glLightfv(_id, GL_POSITION, pos);
    glLightfv(_id, GL_SPOT_DIRECTION, dir);
    glLightf(_id, GL_SPOT_CUTOFF, _cutoff);
}

```

3.3.1 Parse State

Nas fases anteriores, foi utilizado um `unordered_map` para armazenar os *ModelBuffers* já lidos, de forma a não se repetir a leitura do mesmo ficheiro *.3d*.

Já nesta fase expandiu-se esse sistema para também suportar o armazenamento de *TextureBuffers*. Para isto, foi criada uma nova classe denominada de *ParseState*.

```

class ParseState {
private:
    std::unordered_map<std::string, ModelBuffer> _model_buffers;
    std::unordered_map<std::string, TextureBuffer> _texture_buffers;

public:
    auto insert_model_buffer(std::string const&) noexcept
        -> cpp::result<ModelBuffer, ParseError>;
    auto insert_texture_buffer(std::string const&) noexcept
        -> cpp::result<TextureBuffer, ParseError>;
};

```

4. *Script*

Tendo em conta as mudanças necessárias para a construção do Sistema Solar de forma a que este pudesse englobar as novas funcionalidades do programa, a sua *script* de geração foi passível de alterações.

Desta forma, cada planeta foi alvo de atribuição de uma textura que o pudesse representar fidedignamente, pelo que os *paths* das texturas foram adicionados ao ficheiro CSV que contém as diversas informações dos planetas. Além destes, ainda o Sol, as diversas luas e os asteroides possuem, também eles, a si associados uma textura.

Por outro lado, foi também necessária a adição de luzes e da componente *color* aos diversos astros, sendo de especial realce a componente emissiva do Sol que permite que este emita luz.

5. *Scenes*

De forma a enriquecer e testar as diversas funcionalidades do programa foram criadas diversas *scenes* que visam demonstrar as potencialidades do *software* desenvolvido.

Assim, desenvolveram-se as seguintes *scenes*:

5.1 Sistema Solar

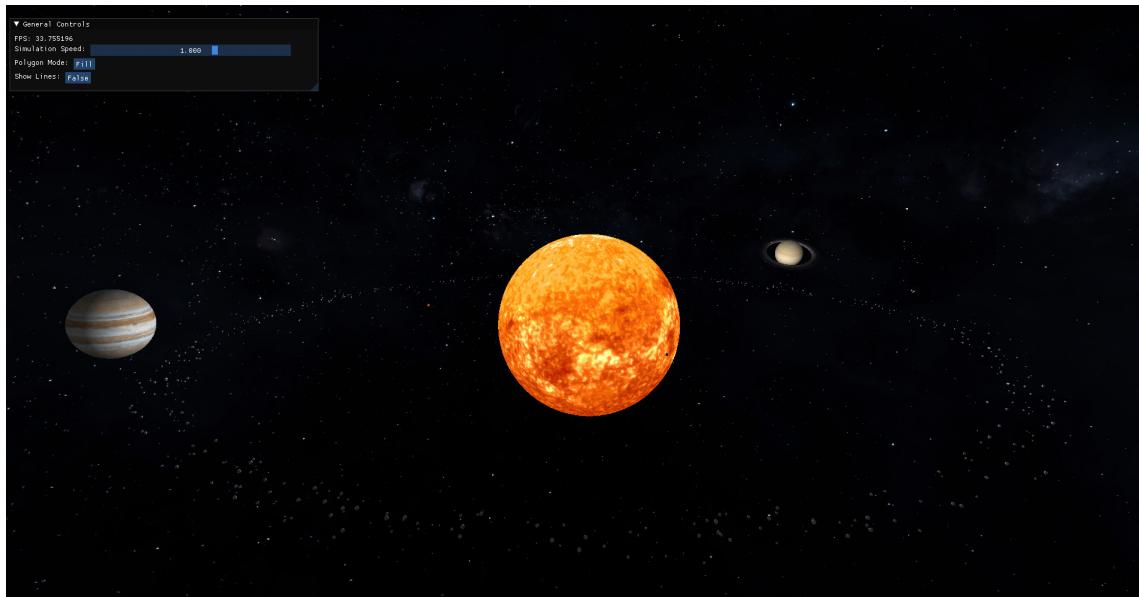


Figura 5.1: Sistema Solar

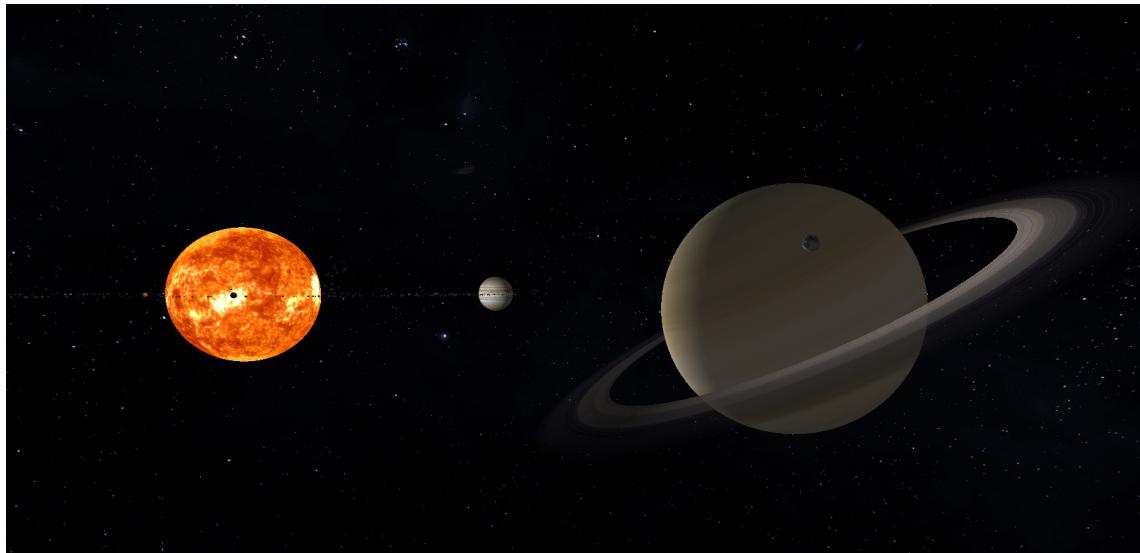


Figura 5.2: Saturno



Figura 5.3: Sol

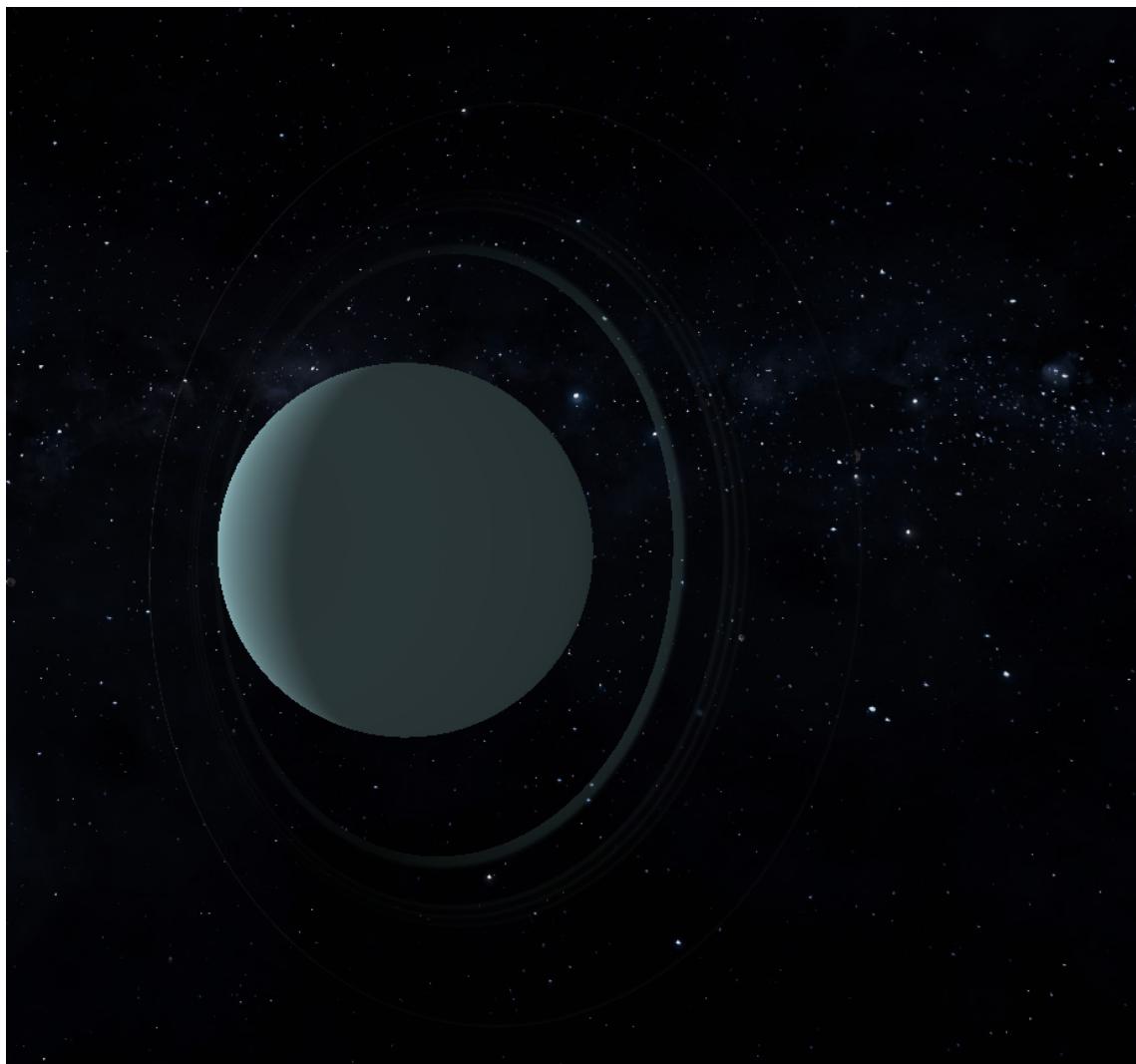


Figura 5.4: Urano

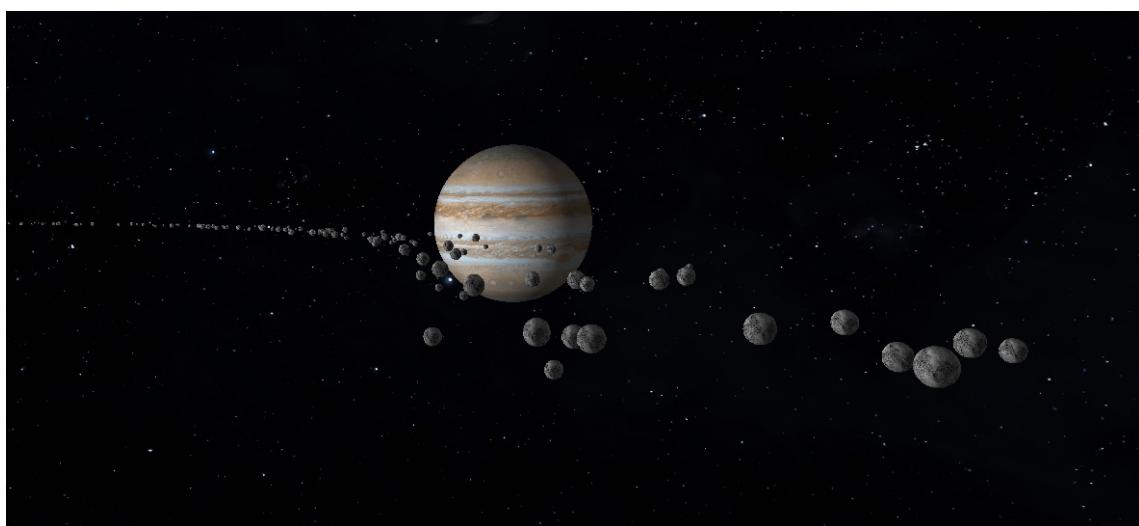


Figura 5.5: Cintura de Asteróides

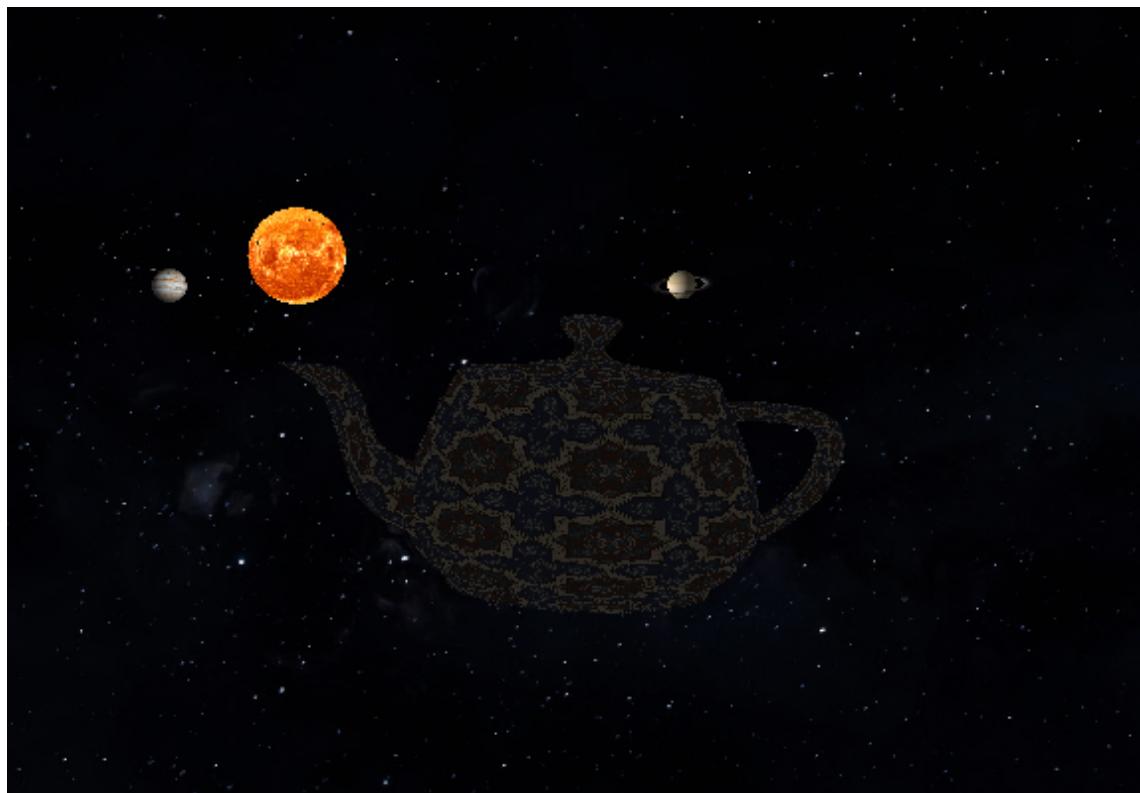


Figura 5.6: *Teapot*

5.2 Boneco de Neve



Figura 5.7: Boneco de Neve

5.3 Desportos

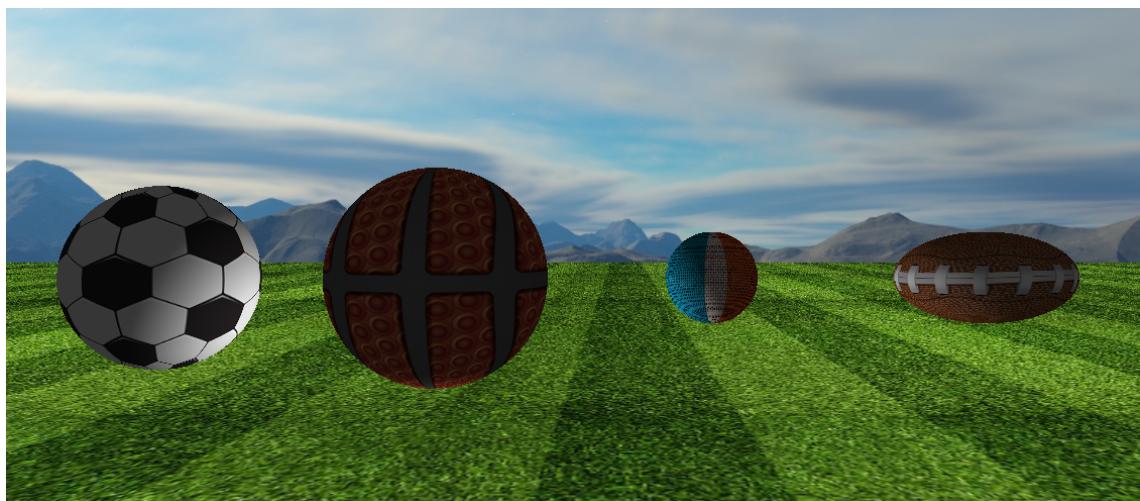


Figura 5.8: Desportos

5.4 Ovo da Páscoa

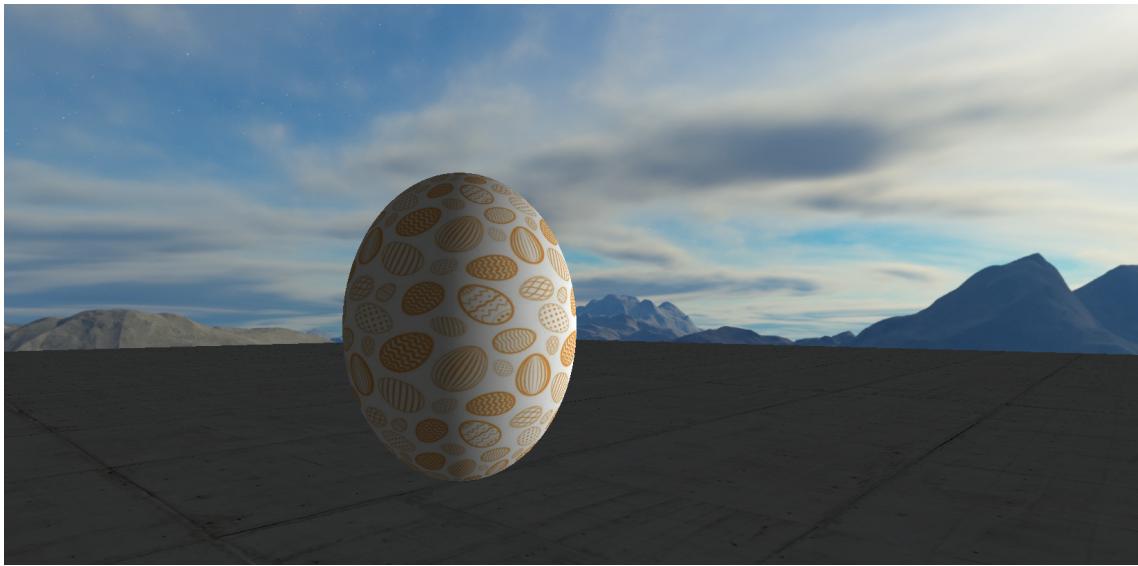


Figura 5.9: Ovo da Páscoa

6. Testes

De forma a realizar outros testes na aplicação, foram verificadas as *scenes* desenvolvidas pela equipa docente:

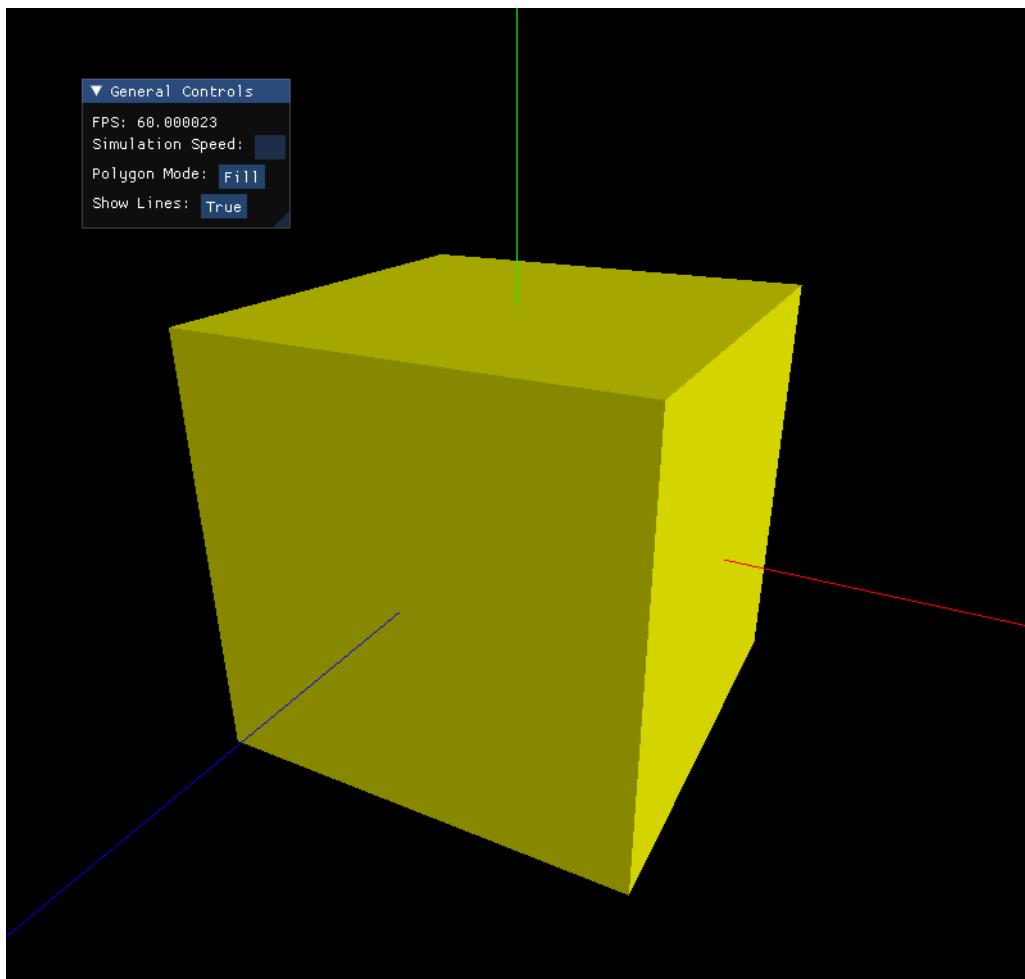


Figura 6.1: Teste 1

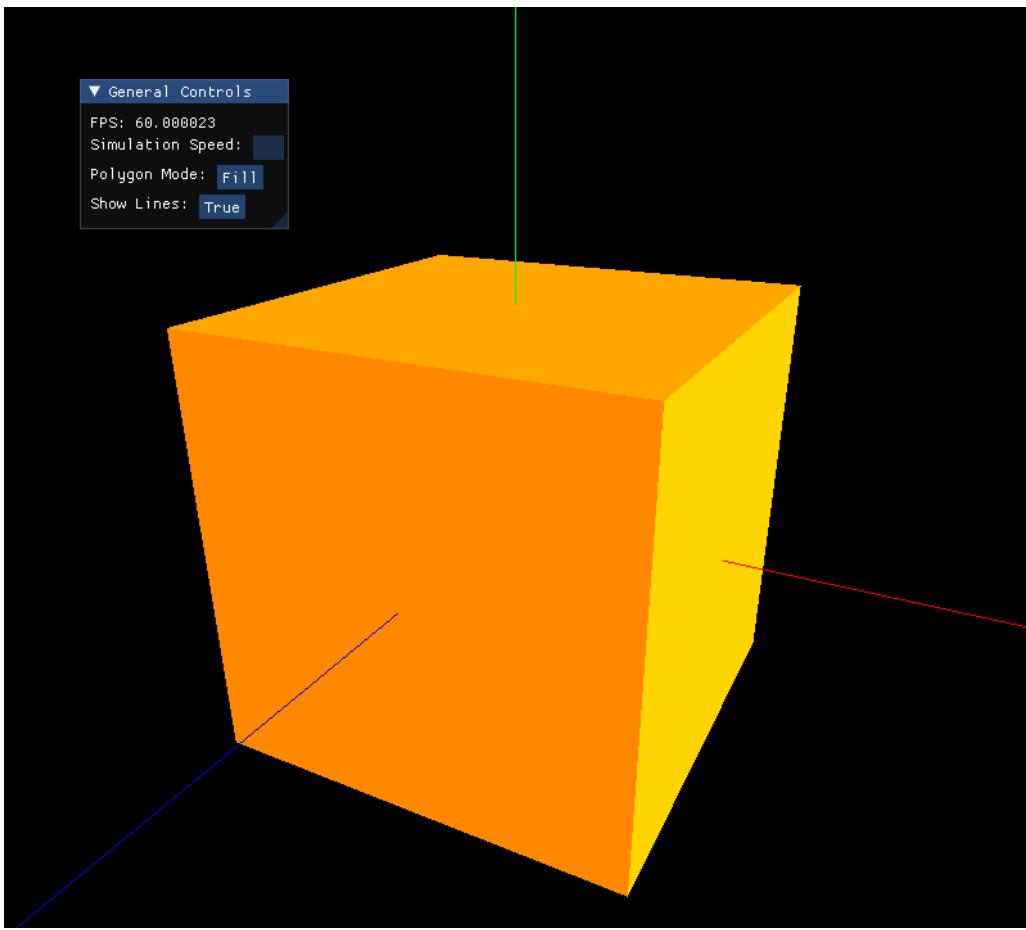


Figura 6.2: Teste 2

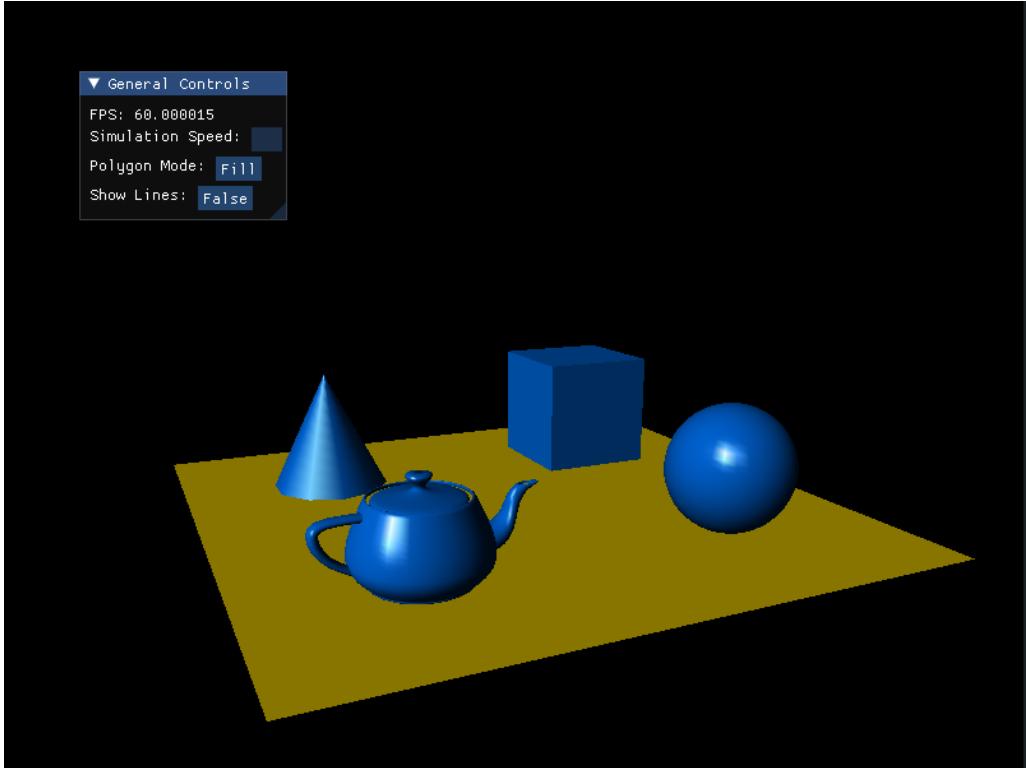


Figura 6.3: Teste 3

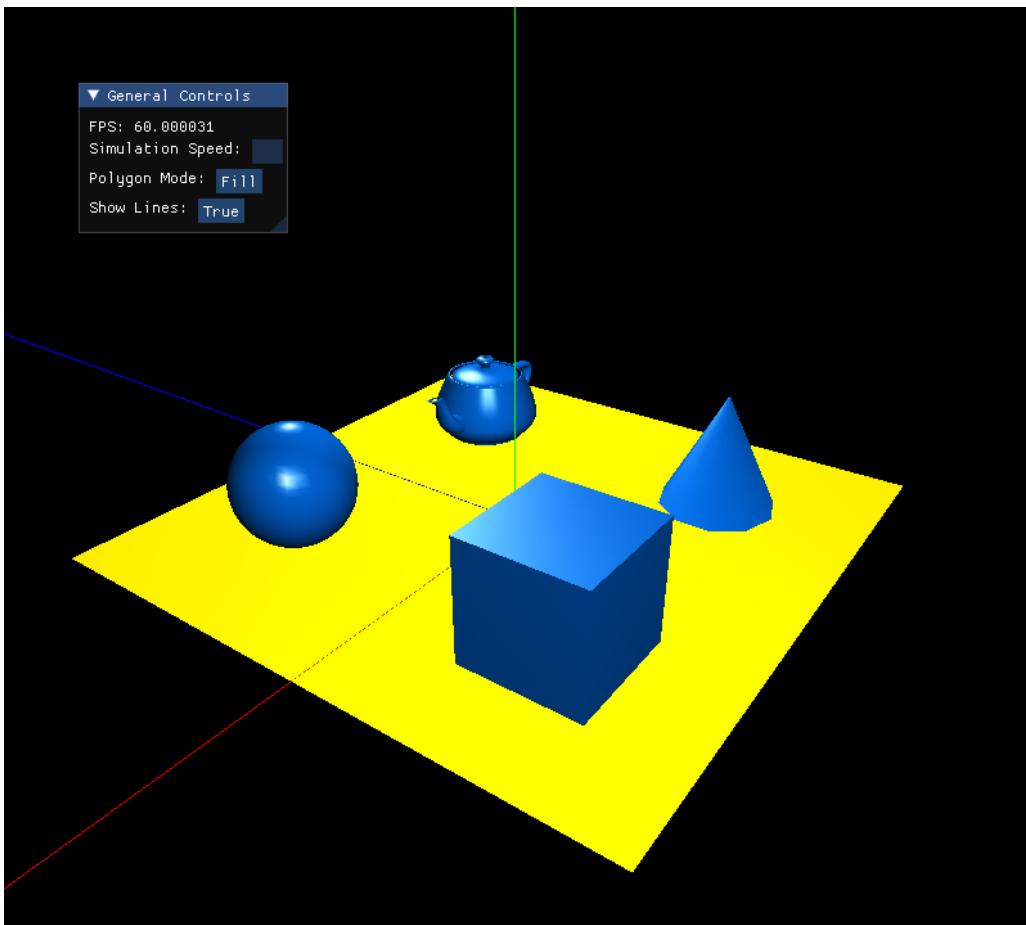


Figura 6.4: Teste 4

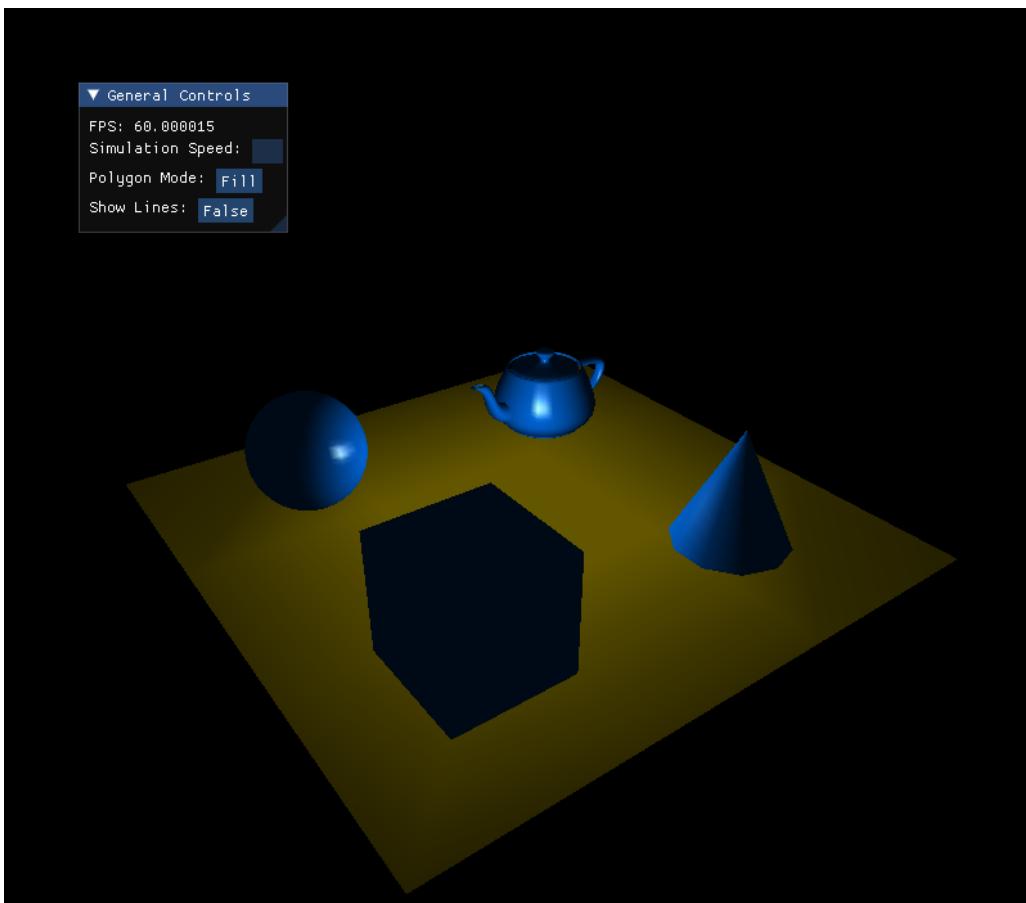


Figura 6.5: Teste 5

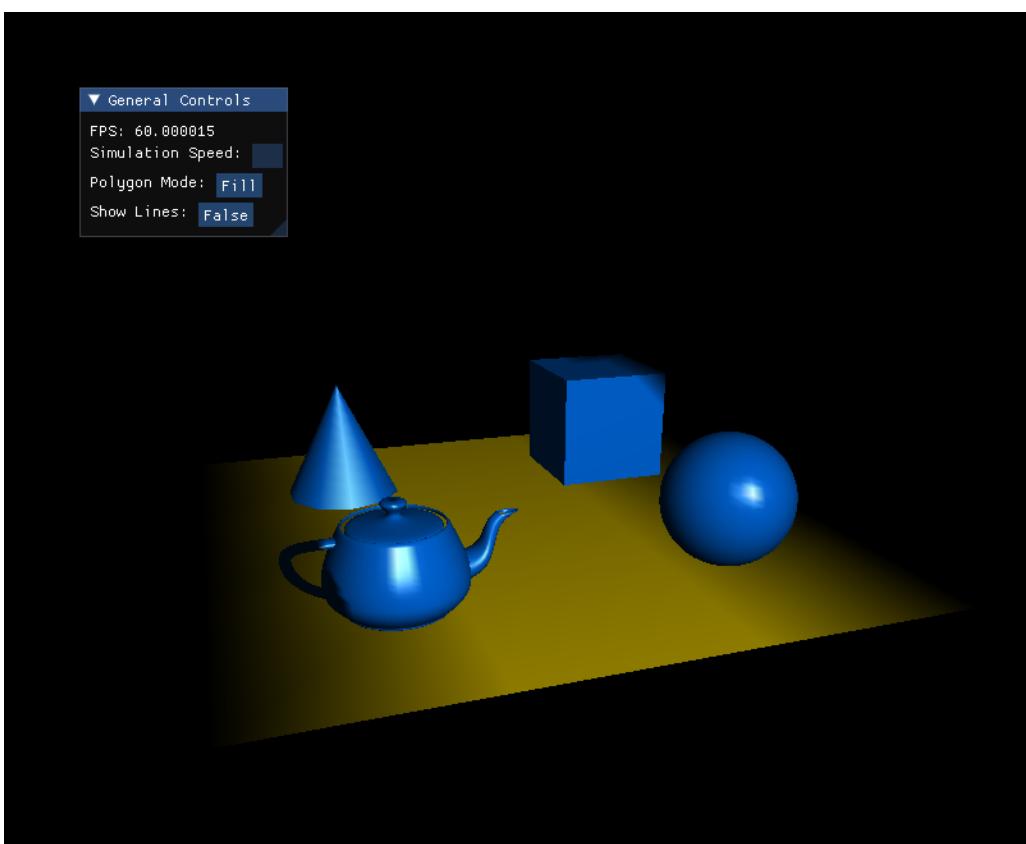


Figura 6.6: Teste 6

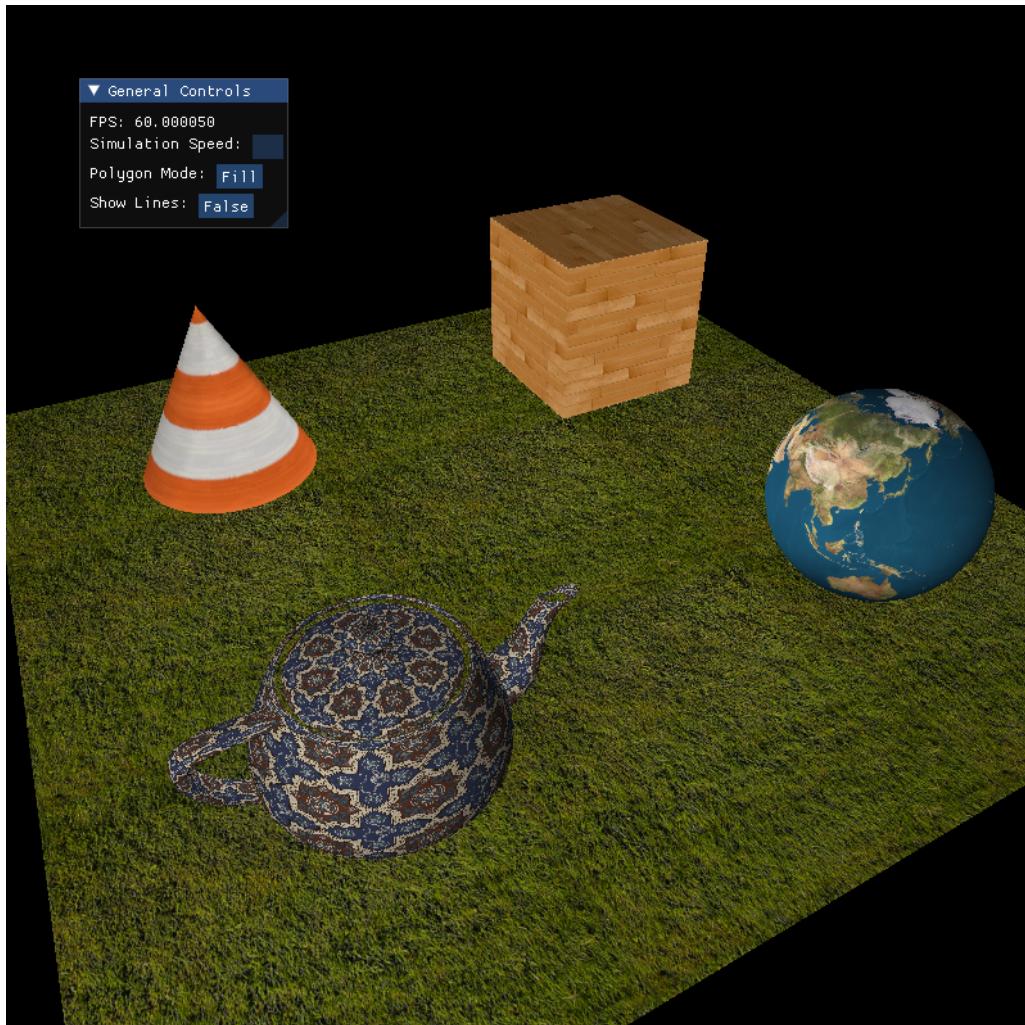


Figura 6.7: Teste 7

7. Conclusões

Considera-se que *Solaris* é um projeto bastante completo, com bastantes funcionalidades adicionais que tornam interessante a experiência de utilização. Para além disso, foram tomadas inúmeras medidas para tornar o programa o mais eficiente possível.

Através da sua execução, foi possível consolidar conhecimentos associados à área de Computação Gráfica e obter prática na utilização do *OpenGL*.

No entanto, considera-se que, no futuro, seria uma mais-valia a adição do mecanismo de *View Frustum Culling* e de terrenos. O projeto também beneficiaria de mais funcionalidades adicionadas ao seu menu.