

UNIVERSIDADE DO MINHO

DEPARTAMENTO DE INFORMÁTICA

Computação Gráfica
Fase 4 - Grupo 7

José Ferreira (A83683)

João Teixeira (A85504)

Miguel Solino (A86435)

31 de Maio de 2020

Conteúdo

1	Introdução	3
2	Generator	4
2.1	Plano	4
2.2	Esfera	5
2.3	Torus	7
2.4	Cilindro	9
2.5	Cone	12
2.6	Caixa	13
2.7	Patches de Bezier	14
3	Engine	16
3.1	Buffers	16
3.1.1	Buffers de Textura	16
3.1.2	Buffers de Modelos	16
3.1.3	Buffers de Terrenos	16
3.1.4	Conjuntos de Buffers	16
3.2	Objects	17
3.3	Iluminação	17
3.4	Parsing de XML	17
3.4.1	Cor	17
3.4.2	Include	18
3.4.3	Luz	18
3.4.4	Object	18
3.4.5	Error Handling	18
3.5	Keybinds	19
3.6	Window Title	19
4	Scenes	20
4.1	Terrain Generation	20
4.2	Castle in the lake	20
4.3	Solar System	21
5	Conclusão	24

Capítulo 1

Introdução

Este trabalho foi desenvolvido no âmbito da unidade curricular de computação gráfica e está dividido em várias fases de entrega distintas sendo que esta é a terceira.

O projeto foi dividido em 4 entregas faseadas sendo que esta é a última. Ao longo deste relatório iremos descrever a metodologia e as soluções criadas para solucionar todos os critérios propostos assim como descrever as funcionalidades extra implementadas.

Primeiro adicionamos a todas as primitivas desenvolvidas anteriormente as coordenadas de textura e as normais. Em seguida modificamos o engine para conseguir ler os ficheiros gerados.

Em seguida adicionamos a possibilidade de definir as propriedades de um objeto no *XML*. Adicionamos ainda a possibilidade de definir luzes no *XML*.

Finalmente criamos mais *scenes* e melhoramos as já existentes de forma a fazer uso das novas funcionalidades do *engine*.

Capítulo 2

Generator

Para facilitar o cálculo dos novos pontos foi criada uma nova classe chamada *ModelPoint*. Esta contém um *Point* que representa as coordenadas do ponto no espaço, um *Vector* que representa a normal do ponto e dois floats que representam as coordenadas da textura. Como métodos apenas foram definidos um conjunto de construtores que recebem os vários tipos de pontos e vetores e um *toString* para facilitar a escrita dos pontos para um ficheiro.

O formato escolhido para este ficheiro é relativamente simples e estende o que já foi implementado anteriormente. Cada linha do ficheiro .3d representa um ponto. Em cada linha estão escritas as coordenadas dos pontos, as coordenadas da normal e as coordenadas de textura, separados por vírgulas. Assim, um ponto que seja caracterizado por estar nas coordenadas (1, 2, 3), que tenha como normal o vetor (0, 1, 0) e coordenada de textura (1, 1) fica representado no ficheiro como:

```
1 1 2 3 0 1 0 1 1
```

Assim, todas as primitivas desenvolvidas anteriormente foram alteradas:

- Plano
- Esfera
- Torus
- Cilindro
- Cone
- Caixa
- Patches de Bezier

Todos os exemplos apresentados ao longo deste capítulo têm o seu *XML* na pasta *scenes/report* com o nome [*PRIMITIVA*].*xml*.

2.1 Plano

Para desenhar um plano com dois triângulos sabendo o lado do plano basta conseguir obter as coordenadas dos cantos do plano.

A normal do plano é sempre a mesma em todos os pontos e aponta para cima.

As coordenadas da textura são mapeadas diretamente, sendo que cada ponto no plano corresponde aos pontos nos cantos da imagem.

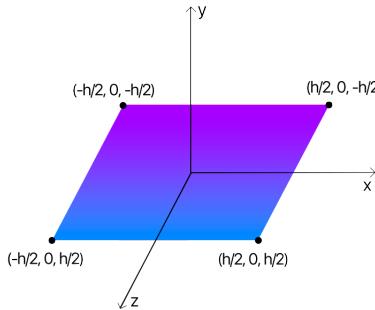


Figura 2.1: coord dos pontos

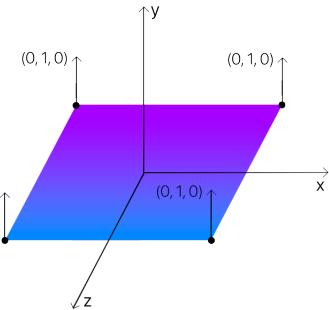


Figura 2.2: coord das normais

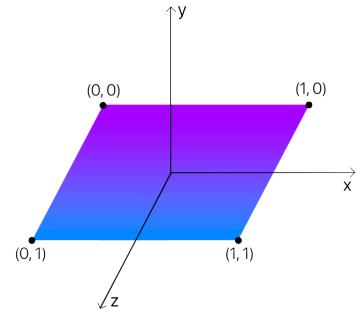


Figura 2.3: coord das texturas



Figura 2.4: exemplo de textura



Figura 2.5: resultado da textura

2.2 Esfera

Para desenhar a esfera é necessário saber o seu raio (`_radius`), o número de slices (`_slices`) e o número de stacks (`_stacks`).

Fazendo uso do sistema de coordenadas esférica basta calcular qual é o azimute e qual é a inclinação de cada ponto (visto que o raio do ponto é igual ao raio da esfera).

A normal de uma esfera num dado ponto é igual ao vetor normalizado do vetor que vai do centro da esfera a um dado ponto. Para agilizar esta conta foi criado um método na classe `Point` que realiza estas contas (`normalized_vector`).

Para facilitar estas contas são calculados dois valores, o ângulo entre cada slice (`a_slice`) e o ângulo entre cada stack (`a_stack`).

As coordenadas de textura de uma esfera são semelhantes às de um mapa, sendo que para calcular as coordenadas de textura de uma esfera basta ver em que slice e em que stack está um dado ponto e dividir esses valores pelo número total de slices e de stacks respectivamente.

```

1 float curr_t_x = slice / _slices;
2 float curr_t_y = stack / _stacks;
3 float next_t_x = (slice + 1) / _slices;
4 float next_t_y = (stack + 1) / _stacks;
```

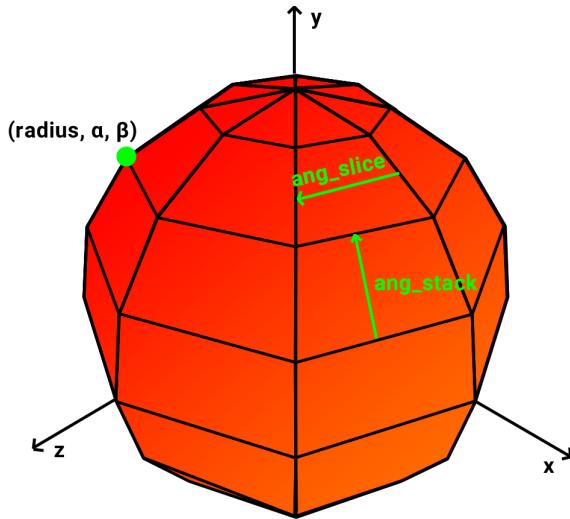


Figura 2.6: coord dos pontos

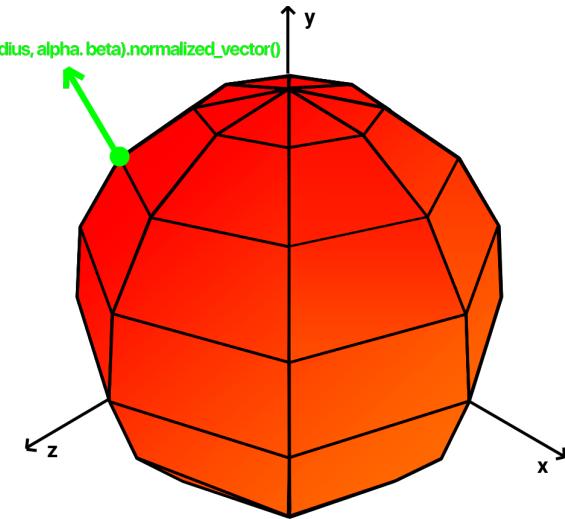


Figura 2.7: coord das normais

Para guardar os pontos calculados é criado um vetor de *ModelPoint* chamado *coords*.

```

1   for (i32 slice = 0; slice < _slices; slice++) {
2       for (i32 stack = 0; stack < _stacks; stack++) {
3           auto p0 = PointSpherical(_radius, a_stack * stack, a_slice * slice);
4           auto p1 = p0.add_inclination(a_stack);
5           auto p2 = p0.add_azimuth(a_slice);
6           auto p3 = p2.add_inclination(a_stack);
7
8           auto v0 = p0.normalized_vector();
9           auto v1 = p1.normalized_vector();
10          auto v2 = p2.normalized_vector();
11          auto v3 = p3.normalized_vector();
12
13          if (stack != 0) {
14              // 1st triangle
15              coords.emplace_back(p2, v2, next_t_x, curr_t_y);
16              coords.emplace_back(p0, v0, curr_t_x, curr_t_y);
17              coords.emplace_back(p3, v3, next_t_x, next_t_y);
18          }
19          if (stack != _stacks - 1) {
20              // 2nd triangle
21              coords.emplace_back(p3, v3, next_t_x, next_t_y);
22              coords.emplace_back(p0, v0, curr_t_x, curr_t_y);
23              coords.emplace_back(p1, v1, curr_t_x, next_t_y);
24          }
25      }
26  }
```

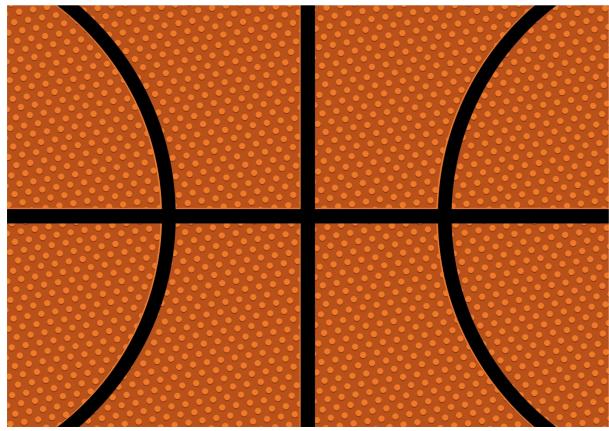


Figura 2.8: exemplo de textura



Figura 2.9: resultado da textura

2.3 Torus

Para desenhar uma Torus é preciso indicar o raio interior, o raio exterior, o número de stacks(*_stacks*) e o número de slices(*_slices*).

Para facilitar os cálculos estes valores são internamente convertidos para o raio que vai do centro da torus ao centro do anel da torus (*_radius*) e o raio do anel propriamente dito (*_ring_radius*).

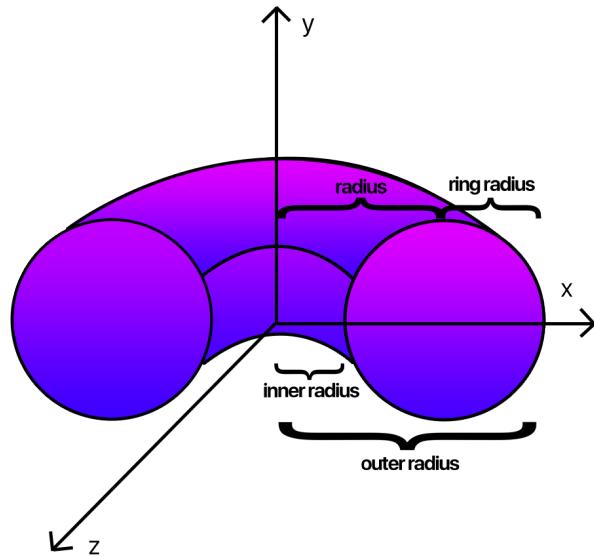


Figura 2.10: Esquema da conversão

Primeiro é definido o ângulo entre cada slice e cada stack com:

```
1 float a_slice = 2.0f * M_PI / _slices;
2 float a_stack = 2.0f * M_PI / _stacks;
```

Para calcular as coordenadas de um ponto central da torus usamos coordenadas esféricas. O raio é igual ao *_radius* e a inclinação é igual a $\pi/2$. Logo, o único valor que varia é o azimute, que é igual a *a_slice*

vezes o índice da slice do ponto.

Com base nisto conseguimos calcular a coordenadas do ponto central de qualquer *slice*. Para calcular os triângulos do torus vai ser necessário ter o ponto *center* (que está na slice i) e o ponto *n_center* (que está na slice *i + 1*).

Para calcular as coordenadas do ponto que está na superfície do torus precisamos de calcular um vetor que vai do ponto calculado anteriormente até à superfície. O raio deste vetor é fixo e é igual a *_ring_radius* e o azimute deste vetor é igual ao *a_stack* vezes a stack atual mais π (para começar de baixo e assim corrigir as texturas). Logo, o único valor que varia é a inclinação, que é igual a *a_stack* vezes o índice da stack atual. Este vetor calculado se for normalizado com recurso ao método *normalize* é a normal do ponto.

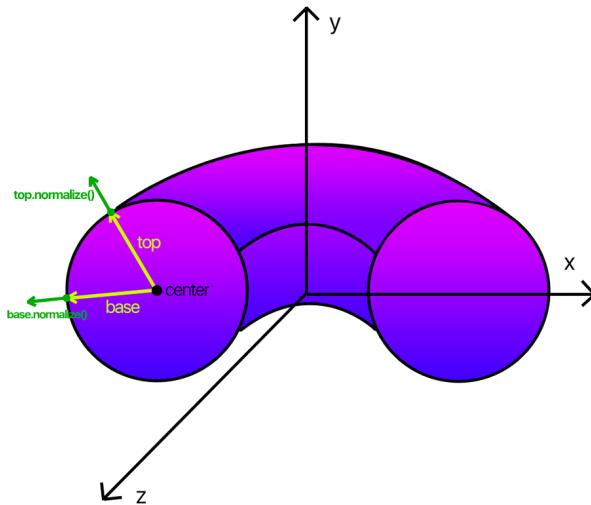


Figura 2.11: Esquema da conversão

Somando para um dado ponto no centro do torus ao vetor correspondente obtemos o ponto na superfície. A maneira como calculamos as texturas do torus é semelhante à que usamos na esfera mas rodada 90°. Assim fica mais fácil criar e editar as texturas.

```

1 float curr_t_x = stack / _stacks;
2 float curr_t_y = slice / _slices;
3 float next_t_x = (stack + 1) / _stacks;
4 float next_t_y = (slice + 1) / _slices;
```

Assim, para calcular os *ModelPoint* e criar um vetor (*coords*) com estes utilizamos o seguinte código.

```

1 for (i32 slice = 0; slice < _slices; slice++) {
2     for (i32 stack = 0; stack < _stacks; stack++) {
3         // vetores do centro do torus ate a superficie
4         float offset = a_stack * stack + M_PI;
5         auto base = VectorSpherical(_ring_radius, offset, center.azimuth());
6         auto n_base = base.add_azimuth(a_slice);
7         auto top = base.add_inclination(a_stack);
8         auto n_top = n_base.add_inclination(a_stack);
9
10        // pontos na superficie
11        auto p0 = center + base;
12        auto p1 = center + top;
13        auto p2 = n_center + n_base;
14        auto p3 = n_center + n_top;
15
16        // 1st triangle
17        coords.emplace_back(p1, top.normalize(), top_t_x, curr_t_y);
```

```

18     coords.emplace_back(p2, n_base.normalize(), base_t_x, next_t_y);
19     coords.emplace_back(p0, base.normalize(), base_t_x, curr_t_y);
20
21     // 2nd triangle
22     coords.emplace_back(p1, top.normalize(), top_t_x, curr_t_y);
23     coords.emplace_back(p3, n_top.normalize(), top_t_x, next_t_y);
24     coords.emplace_back(p2, n_base.normalize(), base_t_x, next_t_y);
25 }
26 }
```

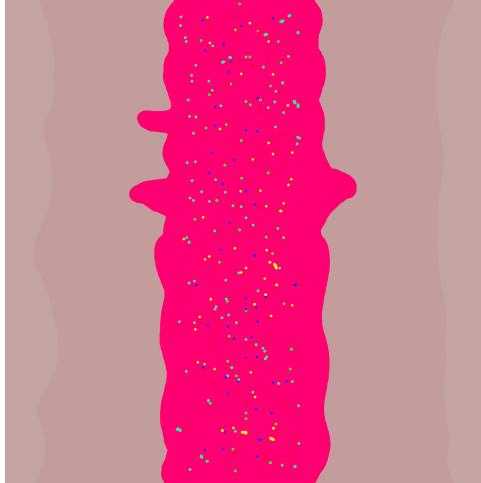


Figura 2.12: exemplo de textura



Figura 2.13: resultado da textura

2.4 Cilindro

Para desenhar um cilindro é necessário indicar o raio (*_radius*), a altura (*_height*), o número de slices (*_slices*) e o número de stacks (*_stacks*).

Com esta informação é calculado um vetor que vai da base do cilindro ao topo do cilindro (*top*). Dividindo este vetor pelo número de stacks obtemos um vetor com o tamanho de cada uma das stacks. Em seguida dividimos 360° pelo número total de slices (*a_slice*).

Assim, para um ponto da aresta da base numa slice é dado por (*radius*, $\pi/2$, *a_slice * slice*). Somando a este ponto o número da slice do ponto vezes o vetor *step* podemos calcular qualquer ponto na face lateral do cilindro.

```

1 Vector top = Vector(0, _height, 0);
2 Vector step = top / _stacks;
3 float a_slice = 2 * M_PI / _slices;
```

Em seguida definimos um ponto com coordenadas (0, 0, 0) denominado de *central*. Este ponto é o centro da base do cilindro. Para obter o ponto central do topo do cilindro basta somar o vetor *top* ao ponto *central*.

Para calcular um ponto ao longo da aresta da base (*base*) do cilindro utilizamos coordenadas esféricas. O raio da coordenada é igual ao raio do cilindro e a inclinação é igual $\pi/2$ logo o único fator que varia é o azimute. Para calcular este valor basta multiplicar o ângulo entre cada slice pelo índice da slice onde o ponto se encontra. Desta forma, para obter as coordenadas do ponto no índice seguinte (*n_base*) basta somar o ângulo entre cada slice ao azimute do ponto.

O vetor normal é sempre o mesmo para todos os pontos da base do cilindro e tem coordenadas $(0, -1, 0)$ (*bottom_normal*). De forma análoga, as normais dos pontos do topo do cilindro têm coordenadas $(0, 1, 0)$ (*top_normal*).

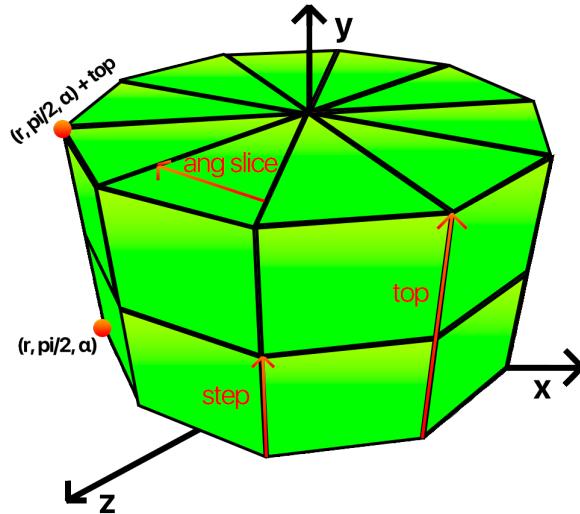


Figura 2.14: Vetores e ângulos dos cilindros

Para calcular um ponto na lateral do cilindro primeiro utilizamos o que foi definido no paragrafo anterior para criar um ponto na aresta da base do cilindro na slice do ponto, depois somamos a este o vetor *step* vezes o indice de stack do ponto. O vetor normal a este ponto é o vetor que vai do ponto *central* ao ponto calculado na aresta da base do cilindro (*base_v*).

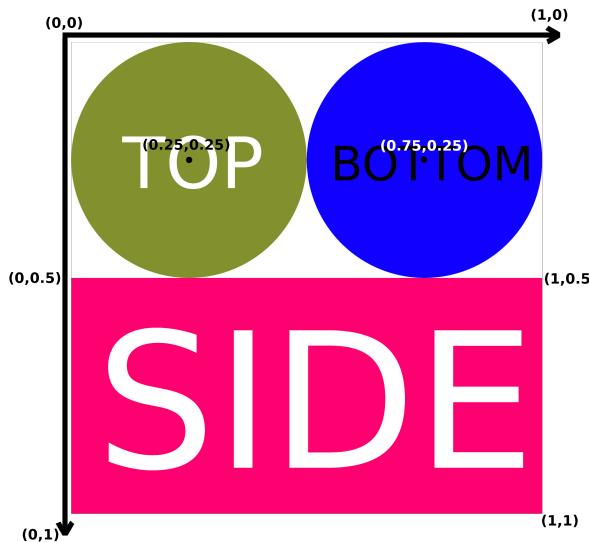


Figura 2.15: coordenadas de textura

Para facilitar o calculo das coordenadas de textura utilizamos os pontos 3D mas apenas utilizando as coordenadas x e z.

Primeiro definimos um ponto no centro do círculo top, que tem coordenadas $(0.25, 0, 0.25)$, denominado de *t_center* e outro no centro do circulo, com coordenadas $(0.75, 0, 0.25)$, denominado de *b_center*. Estes pontos correspondem às coordenadas de textura dos pontos que estão no centro do topo e da base do

cilindro, respetivamente.

Para calcular as coordenadas de textura para os pontos da aresta do topo do cilindro (*t_coor*) somamos às coordenadas do centro da textura o vetor *base_v* multiplicado por 0.25. Desta forma, o vetor passa a ter o comprimento do raio do círculo na imagem. Para a base do cilindro, o cálculo é exatamente igual mas o vetor é espelhado no eixo dos z de forma a compensar o facto de ser desenhado ao contrário (*b_coor*).

Finalmente, para calcular as coordenadas de textura no eixo do y para um dado ponto dividimos o índice da slice do ponto em questão pelo número total de slices (*curr_t_x*) e o eixo dos x é calculado por calcular 1 menos o índice atual da stack a dividir pelo número total de stacks vezes 0.5. Desta forma compensamos o facto de a lateral do cilindro ocupar metade da altura da textura.

Assim, juntando tudo o que foi calculado, criamos os pontos e adicionamos a um *vector* de *ModelPoint* chamado *coords*.

```

1 for (i32 slice = 0; slice < _slices; slice++) {
2     // bottom
3     coords.emplace_back( n_base, bottom_normal, n_b_coor.x(), n_b_coor.z());
4     coords.emplace_back(   base, bottom_normal,   b_coor.x(),   b_coor.z());
5     coords.emplace_back(central, bottom_normal, b_center.x(), b_center.z());
6
7     // top
8     coords.emplace_back(central + top, top_normal, t_center.x(), t_center.z());
9     coords.emplace_back(base    + top, top_normal,   t_coor.x(),   t_coor.z());
10    coords.emplace_back(n_base + top, top_normal, n_t_coor.x(), n_t_coor.z());
11
12    // side
13    for (i32 stack = 0; stack < _stacks; stack++) {
14        // 1st triangle
15        coords.emplace_back(pivot, base_v, curr_t_x, curr_t_y);
16        coords.emplace_back(n_pivot + step, n_base_v, next_t_x, next_t_y);
17        coords.emplace_back(pivot + step, base_v, curr_t_x, next_t_y);
18
19        // 2nd triangle
20        coords.emplace_back(pivot, base_v, curr_t_x, curr_t_y);
21        coords.emplace_back(n_pivot, n_base_v, next_t_x, curr_t_y);
22        coords.emplace_back(n_pivot + step, n_base_v, next_t_x, next_t_y);
23    }
24 }
```

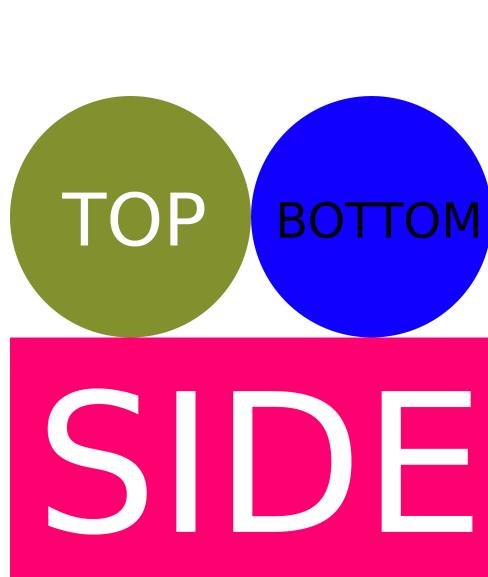


Figura 2.16: exemplo de textura

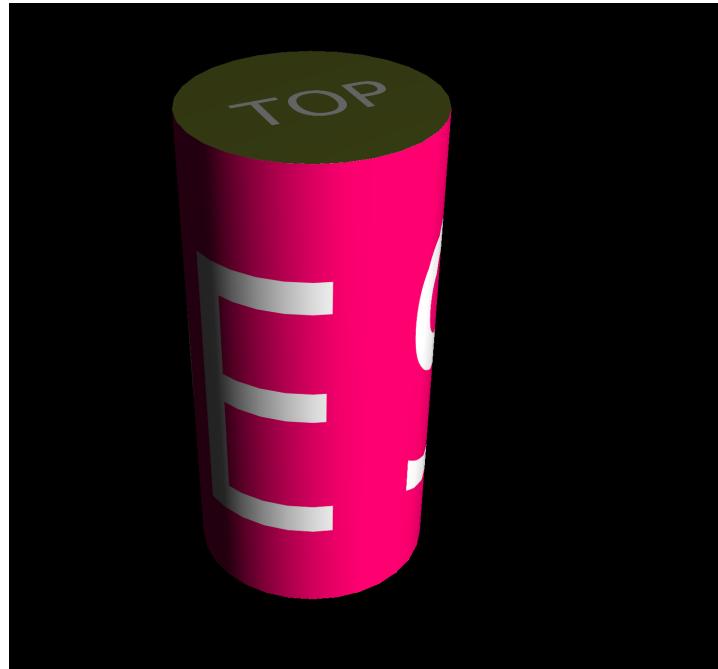


Figura 2.17: resultado da textura

2.5 Cone

Para desenhar um cone é preciso saber o raio (`_radius`), a altura (`_height`), o número de slices (`_slices`) e o número de stacks (`_stacks`).

Com base nisto podemos calcular o ponto do topo do cone (`top_point`) e o ângulo entre cada slice (`a_slice`). Qualquer ponto ao longo da aresta/2 que se encontra com coordenadas esféricas. O raio da coordenada é o raio do cone, a inclinação é sempre $\pi/2$ e o único valor que varia é o azimute. Este varia com base na fórmula $a_slice * slice$, sendo que slice é o índice da slice do ponto. Com estes pontos é possível desenhar a base do cone, sendo que a normal é igual em todos e aponta para baixo.

Para desenhar um ponto na lateral do cone numa determinada stack e numa determinada slice primeiro calculamos o ponto na aresta da base que partilha a mesma slice. Em seguida calculamos um vetor que vai da base do cone ao topo do cone e dividimos esse vetor pelo número total de stacks. Finalmente multiplicamos o vetor pelo índice da stack do ponto que queremos calcular e somamos o vetor obtido ao ponto na base.

A normal em todos os pontos da face lateral do cone tem raio igual a 1 e o azimute é igual ao azimute do ponto. Para além disso partilham todos da mesma inclinação, que é dada pela fórmula $\text{atan}(\text{_height} / \text{_radius})$ devido às propriedades de semelhança de triângulos que se podem observar no seguinte esquema:

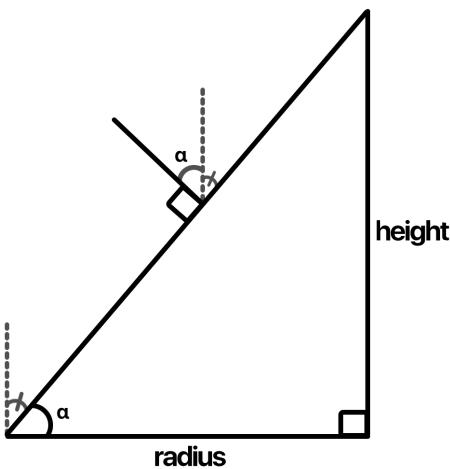


Figura 2.18: fórmulas para patches

Para calcular as coordenadas de textura usamos o mesmo método usado nas coordenadas de textura da base e do topo do cilindro mas com as coordenadas centrais alteradas para a imagem das texturas.

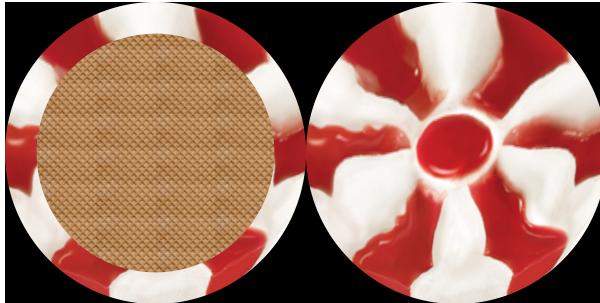


Figura 2.19: exemplo de textura



Figura 2.20: resultado da textura

2.6 Caixa

Para gerar uma caixa é necessário saber as 3 dimensões ($-x$, $-y$ e $-z$) e o número de divisões da caixa ($_slices$).

Primeiro calculamos 3 vetores distintos. Cada um alinhado com um dos eixos no sentido positivo e com comprimento o tamanho de uma divisão da caixa.

Em seguida calculamos mais 3 vetores com bases nestes três que têm exatamente o mesmo comprimento mas sentido inverso.

Em seguida criamos um conjunto de 6 pontos que indicam as coordenadas onde começam cada textura. Para calcular cada face foi criada uma função auxiliar que recebe o ponto no espaço onde começa a face, as coordenadas da textura equivalentes a esse ponto e 2 vetores dos calculados anteriormente de forma a que estejam alinhados com a aresta da face e que o seu *cross product* seja normal à face.

Tal como já foi referido basta fazer o cross product dos dois vetores para obter a normal de todos os pontos da face.

Para obter um ponto (n, m) na face da caixa basta multiplicar por n um dos vetores e multiplicar por m o outro, e somar estes vetores ao ponto de origem da face.

Da mesma forma, para calcular as coordenadas da textura de um ponto sabendo que ele está na posição (n, m) utilizamos a fórmula para obter o *offset* relativamente ao ponto de origem passado como parâmetro. As multiplicações por 3 e por 2 servem para compensar o facto de só se usar $1/6$ da área da textura para cada face.

```

1 float xt = i / (_slices * 3);
2 float yt = j / (_slices * 2);
```



Figura 2.21: exemplo de textura

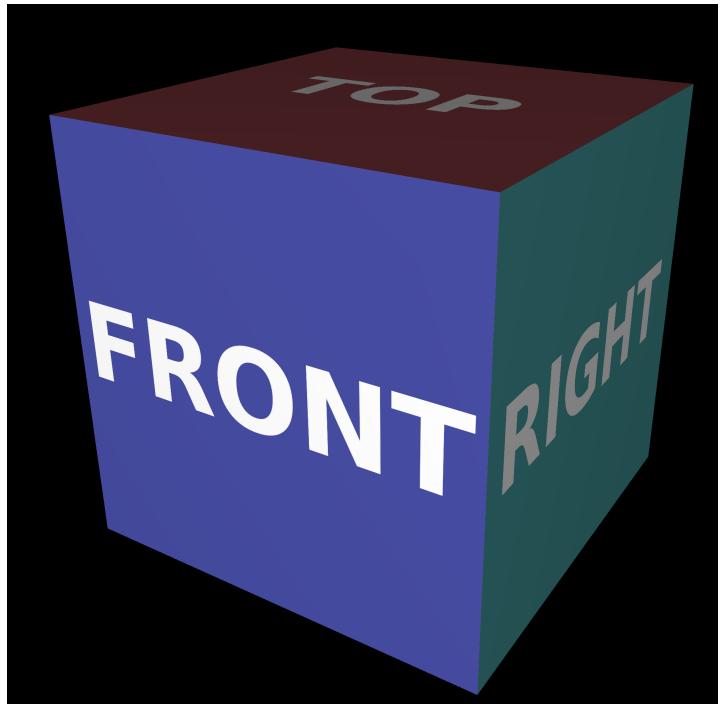


Figura 2.22: resultado da textura

2.7 Patches de Bezier

Para calcular um *patch de bezier* basta indicar o nome do ficheiro onde este se encontra e o nível de *tesselation*.

O parsing do ficheiro em questão não foi alterado para esta fase. A única parte que foi alterada foi a parte de desenhar.

Desta forma, convertemos as fórmulas para patches de Bezier de forma a fazer uso da biblioteca de pontos definida para este trabalho.

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

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

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

Figura 2.23: fórmulas para patches

A primeira calcula diretamente as coordenadas do ponto. As duas seguintes calculam as tangentes no ponto. Desta forma, para obter a normal no ponto é ainda preciso calcular o *cross product* entre as duas tangentes e normalizar o resultado.

As coordenadas de textura são mapeadas diretamente com base nas coordenadas do patch que se está a calcular. Assim, para um ponto (u, v) as coordenadas de textura são dadas por:

```
1 float x = u / _tesselation_level;
```

```
2 float y = v / _tesselation_level;
```



Figura 2.24: exemplo de textura

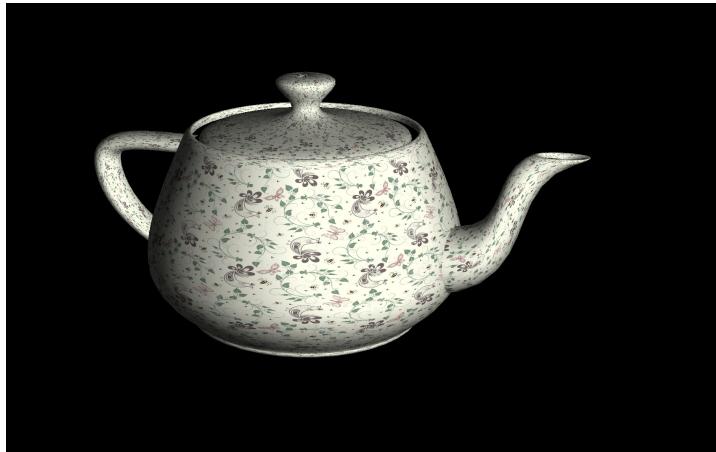


Figura 2.25: resultado da textura

Capítulo 3

Engine

Para facilitar a escrita de scenes mais complexas foram criadas dois extras para o *engine*. Primeiro é possível passar como argumento múltiplos ficheiros. Fazendo isto as várias *scenes* serão desenhadas em simultâneo no ecrã.

3.1 Buffers

3.1.1 Buffers de Textura

De forma a guardar a informação de um textura foi criada uma nova classe chamada *TextureBuffer* que carrega uma textura com base no nome do ficheiro da textura.

Para utilizar esta textura foi criado um método *bind_texture*.

3.1.2 Buffers de Modelos

De forma análoga, para guardar a informação de um Modelo foi criado uma classe denominada de *ModelBuffer* que carrega um modelo com base no nome de um ficheiro .3d definido com o *Generator*. A função *make* devolve um par com uma instância de *ModelBuffer* e uma *Axis-Aligned Bounding Box* que engloba o modelo carregado.

Para desenhar o modelo foi criado o método *draw_model* que encapsula aplicar os 3 buffers no modelo.

3.1.3 Buffers de Terrenos

Para guardar a informação de um Terreno foi criada a classe *TerrainBuffer* que carrega para memória uma imagem e que a converte para um conjunto de pontos que depois são desenhados por strips. Esta classe, à semelhança do *ModelBuffer*, contém a função *make* que devolve um par de *TerrainBuffer* e uma *BoundingBox* com as mesmas propriedades definidas acima.

O código utilizado para computar as normais e as coordenadas de textura é semelhante ao que foi utilizado na ficha prática número 10 mas modificado para utilizar o módulo *Point*.

Para desenhar o terreno foi criado o método *draw_terrain* que encapsula aplicar os 3 buffers no modelo.

3.1.4 Conjuntos de Buffers

Visto que pode na mesma *scene* pode ser usado o mesmo modelo, a mesma textura ou até o mesmo terreno várias vezes não faria sentido estes serem carregados mais do que uma vez para memória. Desta forma foi criado um buffer intermédio chamado *GroupBuffer*.

Esta classe contém quatro *maps*. O primeiro guarda pares nome de ficheiro *ModelBuffer*, o segundo guarda pares nome de ficheiro *TerrainBuffer*, o terceiro guarda pares nome de ficheiro *TextureBuffer* e o último guarda pares nome de ficheiro *BoundingBox*.

Assim, se se quiser adicionar um Modelo é chamado o método *insert_model*. Primeiro verificamos se o Modelo em questão já foi adicionado. Se ainda não foi adicionado chamamos a função *make* do *ModelBuffer*, adicionamos o buffer e a *bounding box* aos sítios respetivos e, por fim, devolvemos a *bounding box*. Se o buffer já tiver sido adicionado, procurasse no *map* para *bounding boxes* a respetiva para este modelo e devolve-se.

```

1 auto search = _model_buffers.find(model_name);
2 if (search == _model_buffers.end()) {
3     auto [model, bb] = ModelBuffer::make(model_name);
4     _model_buffers.insert(std::make_pair(model_name, model));
5     _bounding_box.insert(std::make_pair(model_name, bb));
6     return bb;
7 } else {
8     return _bounding_box[model_name];
9 }
```

O código é igual para os outros dois buffers, com a diferença que no caso de uma textura nem é inserido nem é devolvido uma *bounding box*.

3.2 Objects

Foi criada uma classe *Object* (com um template para indicar o tipo do objeto) com o objetivo de guardar informação sobre um dado objeto 3D. Esta classe contem o nome do ficheiro onde o objeto está guardado, pode conter o nome do ficheiro da textura caso o objeto tenha textura definida, a cor do objeto, as cores dos diferentes componentes do material (difuso, especular, emissivo e ambiente) e a Bounding Box do objeto.

Assim, para desenhar um dado objeto basta utilizar o método *draw*. Este método recebe um *GroupBuffer* e um booleano a indicar se se está a desenhar em modo de *debug*. Primeiro aplica-se a cor, depois as propriedades do material, em seguida aplica-se a textura caso exista, depois desenha-se o modelo com base no valor do template e por fim retira-se a textura. Caso ainda se esteja em modo de *debug*, desenha-se as *bounding boxes* do objeto.

3.3 Iluminação

Para guardar os diversos tipos de luz foi criada uma classe para cada um.

A classe *PointLight* guarda a localização da luz e a cor.

A classe *DirectionalLight* guarda a direção e a cor.

A classe *SpotLight* guarda a localização, direção e cor da luz.

Quando uma nova luz é criada é ainda adicionado um identificador único a cada luz, o que permite saber se se excedem as 8 luzes em simultâneo. Caso isso acontece é atirado um erro.

Cada classe possui ainda um método *on* que permite ativar a luz.

3.4 Parsing de XML

Para facilitar a construção de um parser e eventual expansão do mesmo foram criadas pequenas funções que lêem partes mais simples de forma a poder criar um *Parser Combinator*.

Por exemplo, criamos uma função que dado um nodo do *XML*, o nome do atributo e um valor *default*, verifica se o atributo existe e tem um valor válido de um float. Caso o valor não seja válido atira um erro com informação sobre a localização do erro. Caso o atributo não exista devolve o valor *default*.

Este parser é utilizado para a função que lê um ponto três vezes de forma a conseguir ler as coordenadas do ponto. Segundo a mesma lógica a função para ler um vetor utiliza a função de ler um ponto.

3.4.1 Cor

Um outro exemplo destes parsers modulares é um parser para ler cor. Este parser suporta dois formatos distintos: O formato hex que é usado normalmente em design gráfico (#RRGGBBAA e #RRGGBB)

e o formato proposto no enunciado. Visto que o primeiro é o mais usado nas nossas *scenes* devido à facilidade de escrita o *parser* dá prioridade de leitura a esse formato.

3.4.2 Include

Para facilitar ainda mais o processo de separar *scenes* em vários ficheiros decidimos criar uma extensão do *XML* chamada de *include*. Com esta elemento é possível incluir um ficheiro de *XML* dentro de outro e quando for desenhado irá herdar as propriedades definidas anteriormente, nomeadamente a cor. Tal como acontecia previamente, o *GroupBuffer* é partilhado entre os vários ficheiros. Desta forma mantém-se a premissa que um ficheiro apenas é carregado uma e uma só vez, independentemente de ser utilizado várias vezes. Assim, para incluir um ficheiro chamado *lighting.xml* basta colocar a linha:

```
1 <include file="lighting.xml">
```

Esta extensão foi usada em quase todos os exemplos ao longo deste relatório. Como o sistema de iluminação é igual para quase todos bastou definir uma *scene* onde está definida a iluminação e depois fazer *include* onde foi necessário utilizar.

3.4.3 Luz

Para definir uma luz utilizamos o elemento *light*.

A cor da luz é lida com base no parser de cor definido anteriormente com o prefixo *colour* e caso nenhuma cor seja definida o default é branco.

Para indicar o tipo da luz é utilizado o atributo *type*. *POINT* é para luz pontual, *DIRECTIONAL* para luz direcional, *SPOT* para luz de foco.

Para cada luz é ainda necessário indicar a posição e/ou a direção, que utilizam o prefixo *pos* e *dir* respetivamente. Por default, se estes não forem indicados, assumimos que são 0.

Assim, para definir uma luz de foco com posição (0,1,0), direção (1,0,0) e cor #909090 fazemos:

```
1 <light type="SPOT" colour="#909090" posY="1" dirX="1" />
```

3.4.4 Object

Para definir um modelo ou um terreno usa-se a elemento *model* e *terrain* respetivamente.

Dentro desta elemento definem-se o ficheiro a carregar com o atributo *file*, o ficheiro de textura a carregar com o atributo *texture* e as propriedades definem-se com *diff* para o difuso, *spec* para specular, *emis* para emissivo e *ambi* para ambiente. Para ler as propriedades do objeto é usado o parser de cores. Assim, para definir a propriedade emissiva de um objeto, a linha:

```
1 emisR="1" emisG="1" emisB="1"
```

é equivalente a:

```
1 emis="#FFFFFF"
```

3.4.5 Error Handling

Para facilitar a correção de erros dos ficheiros criamos mensagens de erro que indicam o ficheiro, a linha e a coluna onde ocorreu o erro. Ainda acrescentamos uma descrição do erro e, quando relevante, a indicação do atributo.

Assim, se ao ler o ficheiro *solar.xml* ocorrer um erro de parsing num *translate* porque o valor não representa um float o erro terá este aspeto:

```
~/Miei/CG-1920 ./target/release/bin/engine scenes/solar.xml
Error while parsing file: scenes/solar.xml
59:13: error: Invalid float value in Attribute: "X"
```

Figura 3.1: Exemplo de erro do praser

3.5 Keybinds

Keybinds relacionadas com camera:

modo de camera	keybind	keybinds contextuais	
		keybind	descrição
Orbit	1	hjkl	orbitar um dado ponto
		wasd	mover o ponto no plano xOz
		r	reset ponto para a origem
FPV	2	hjkl	olhar em redor
		wasd	mover a camera
		r	reset camera

Outras keybinds:

keybind	descrição
p	toggle pausa
[]	abrandar/acerclar tempo
g	toggle modo de debug (eixos, <i>bounding boxes</i> e caminhos de transformações)
\	toogle lights

3.6 Window Title

De forma a facilmente apresentar a informação ao utilizador decidimos usar o titulo da janela. Assim temos uma barra que mostra o modo atual da camera, o *TIME SCALE* atual, o número de *frames per second*, se as luzes estão ligadas. Também indica se se está em debug mode ou em pausa.



Figura 3.2: barra em modo normal



Figura 3.3: barra em modo debug

Capítulo 4

Scenes

4.1 Terrain Generation

Tal como foi referido acima, a forma como são guardados os buffers garante que apenas é guardada uma instância de cada ficheiro em memória.

Para demonstrar as vantagens deste método criamos um script em python que gera uma grelha de cubos com altura variável com base numa função de *Perlin Noise* fazendo com que o terreno final se aproxime bastante do aspeto do jogo *Minecraft*.

O ficheiro gerado em XML encontra-se em *scenes/not_a_game.xml*.

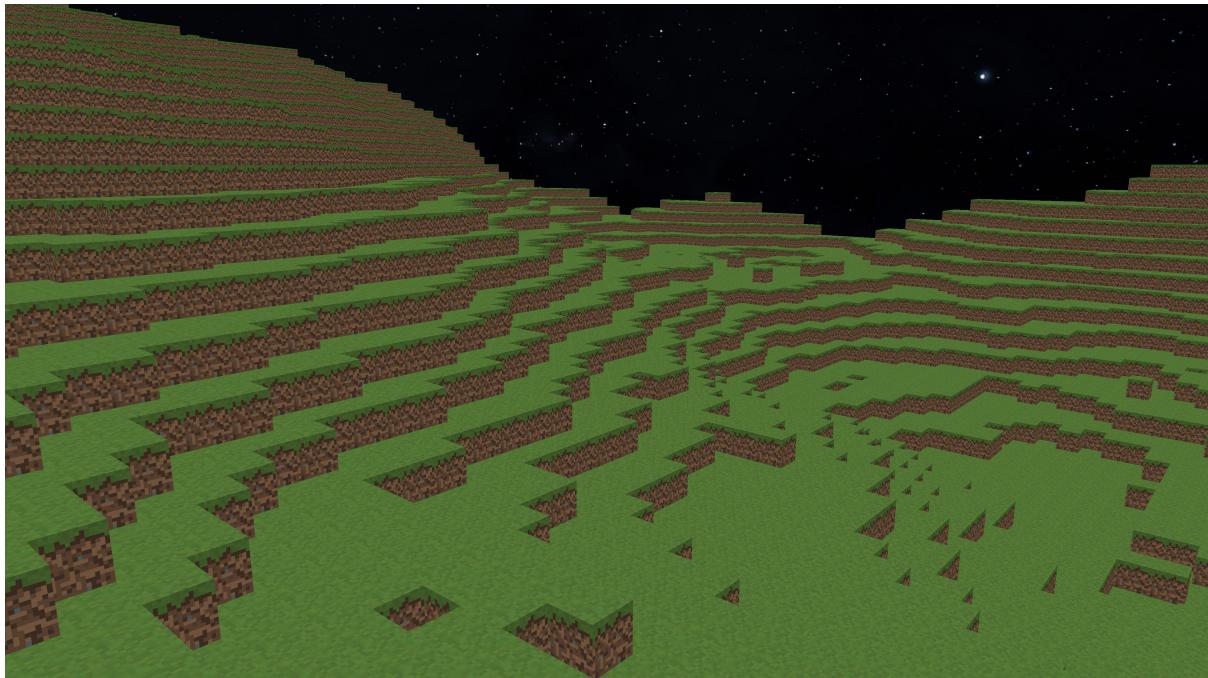


Figura 4.1: Exemplo de terreno gerado

4.2 Castle in the lake

Para demonstrar a capacidade das carregar imagens como terreno diretamente no XML criamos uma *Scene* nova. Nesta desenhamos um castelo numa ilha no centro de um lago. O Terreno onde se assenta o castelo é desenhado com base num *HeightMap*.

O ficheiro XML encontra-se em *scenes/castle.xml*.

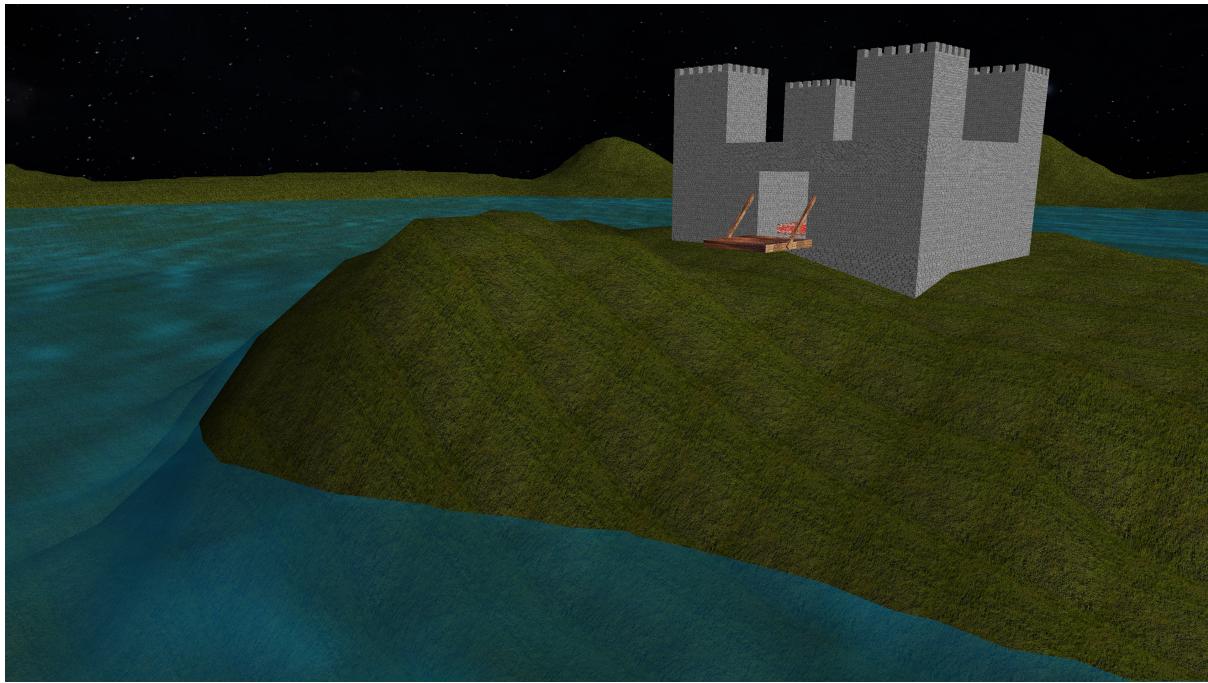


Figura 4.2: render do XML do castelo

4.3 Solar System

Para facilitar desenhar o sistema solar decidimos criar um script em Python.

Nesta fase melhoramos o script desenvolvido na fase anterior para tirar partido das alterações feitas ao *engine*.

Primeiro adicionamos ao *CSV* obtidos na fase anterior uma coluna a indicar qual é a textura de cada planeta.

As texturas utilizadas para cada planeta é a que foi possível encontrar com mais resolução e variam entre 4k e 8k, sendo que a grande maioria está nesta última.

Os anéis de Saturno passaram a ser semi-transparentes e por isso são desenhados em último lugar. Adicionamos ainda uma skybox com a textura da via láctea.

O ficheiro gerado em XML encontra-se em *scenes/solar.xml*.

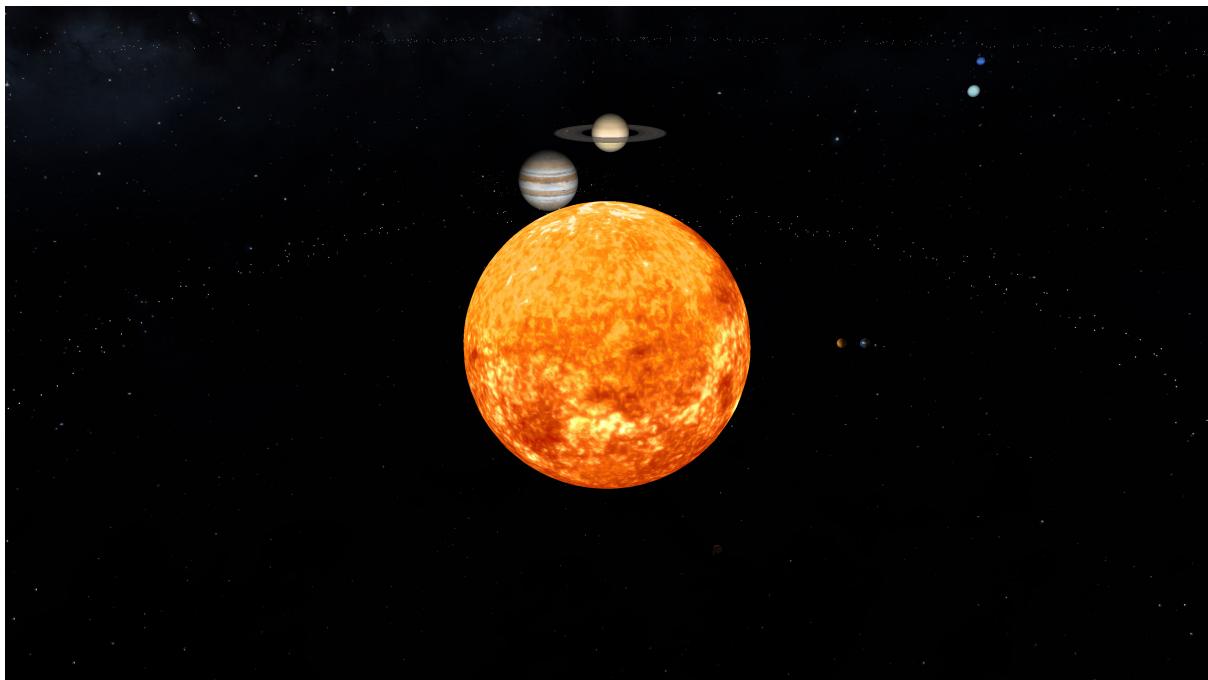


Figura 4.3: Sistema solar



Figura 4.4: Anéis de Saturno transparentes



Figura 4.5: Cometa a usar *patches de bezier*

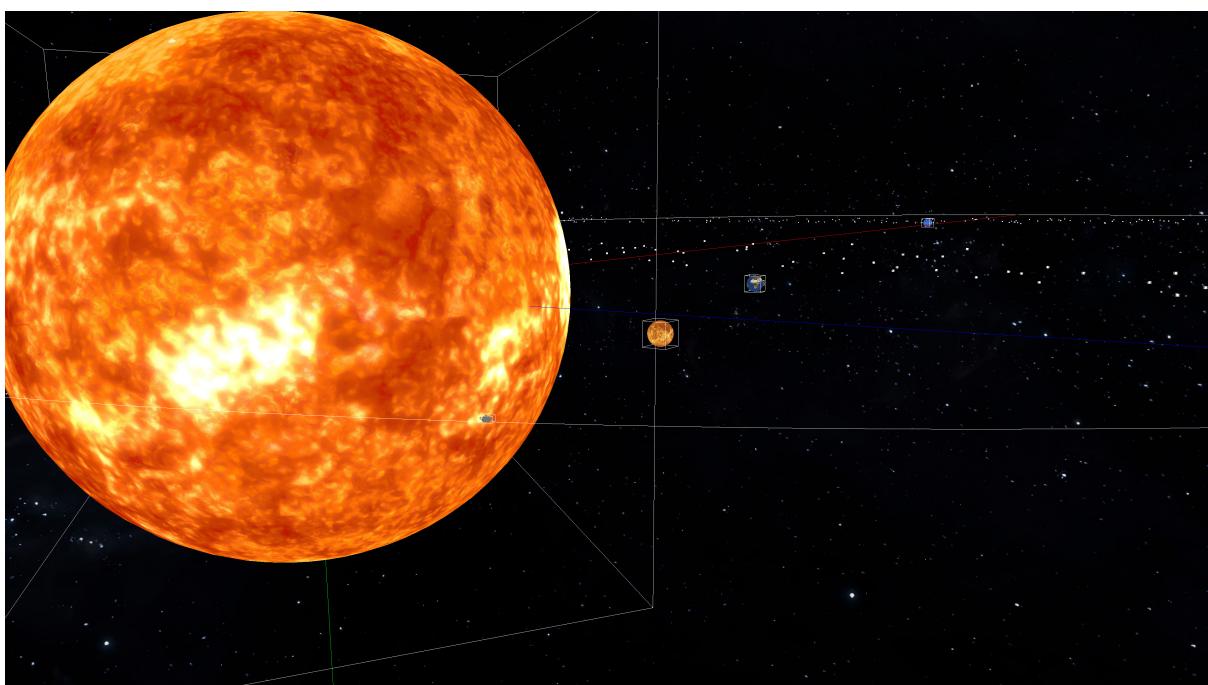


Figura 4.6: Sistema solar em modo de debug com as luzes desligadas

Capítulo 5

Conclusão

Com este trabalho prático podemos aplicar os conhecimentos leccionados até agora nas aulas da unidade curricular de computação gráfica permitindo assim aprofundar os nossos conhecimentos de openGl. Como trabalho futuro gostaríamos de acabar de implementar *View Frustum Culling* porque apenas tivemos tempo de começar a calcular as *Axis-aligned bounding boxes*.