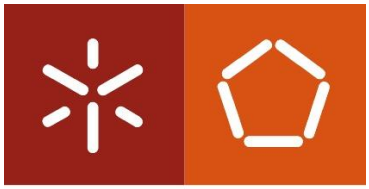

Universidade do Minho

Departamento de Informática

Computação Gráfica

Fase 3 - Grupo 7

José Ferreira (A83683)

João Teixeira (A85504)

Miguel Solino (A86435)

4 de Maio de 2020

Conteúdo

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.
Ao longo desta fase dividimos o trabalho em 4 partes:
Primeiro convertemos todos os modelos no *engine* para *VBO*s. Em seguida melhoramos o parser de *XML* desenvolvido na fase anterior de forma a suportar uma sintaxe mais abrangente. Também melhoramos algumas funcionalidades do *engine* de forma a o tornar mais *user friendly*.
Acrescentamos ainda ao *generator* a capacidade de processar ficheiros contendo *patches de Bezier*.
Finalmente criamos mais *scenes* e melhoramos as já existentes de forma a fazer uso das novas funcionalidades do *engine*.
Ao longo deste relatório irá ser explicado a metodologia e raciocínio usados para a realização desta fase.

Capítulo 2

Engine

Neste módulo foi desenvolvido um motor 3D simples.
Quando se executa o programa gerado, por defeito, é aberta a scene guardada em *scenes/config.xml*. Se alguém ficheiro for passado como argumento esse será o selecionado.

2.1 Conversão para VBOs

Nas fases anteriores os modelos eram desenhados com o modo imediato.
Para esta faze é necessário que estes sejam desenhados com VBOs.

2.1.1 Estruturas de Dados

Foi criada uma class denominada de *ModelBuffer* que contem um buffer (*_buffers*) e o número de vértices no buffer(*_n_vertices*).
Assim, para criar um *ModelBuffer* são primeiro lidos todos os pontos contidos no ficheiro (*fileName*) a carregar para um vetor. Depois o buffer é inicializado e os dados são copiados do vetor para o buffer.

```
float x, y, z;
std::vector<float> vec;
auto file = std::ifstream(fileName);
while (file >> x >> y >> z) {
    vec.push_back(x);
    vec.push_back(y);
    vec.push_back(z);
}
_n_vertices = vec.size();

glGenBuffers(1, _buffers);
glBindBuffer(GL_ARRAY_BUFFER, _buffers[0]);
glBufferData(
    GL_ARRAY_BUFFER,
    sizeof(float) * _n_vertices,
    vec.data(),
    GL_STATIC_DRAW);
```

Também nesta classe está o método para desenhar um modelo. Para tal o *_buffers[0]* é associado e depois o modelo é desenhado usando as funções de glut apropriadas.

```
glBindBuffer(GL_ARRAY_BUFFER, _buffers[0]);
glVertexPointer(3, GL_FLOAT, 0, 0);
glDrawArrays(GL_TRIANGLES, 0, _n_vertices);
```

Visto que em muitas *Scenes* o mesmo modelo 3D é repetido várias vezes concluímos que o ideal seria apenas carregar uma vez cada modelo. Assim, desenvolvemos uma classe denominada de *GroupBuffer* que contém um *unordered_map* que faz corresponder ao nome do ficheiro um *ModelBuffer*.
Quando se pretende adicionar um novo modelo utiliza-se o método *insert_model* que verifica se o modelo

já existe e caso não exista carrega-o. Da mesma forma. caso se pretenda desenhar um modelo utiliza-se uma instância da classe *GroupBuffer* e o método *draw_model* que caso o model exista chama o método definido para o *ModelBuffer*.

2.1.2 XML parsing

Para ler o XML continuamos a utilizar a biblioteca *TinyXML*.
Para garantir que o ficheiro contendo a *Scene* seja lido apenas uma vez foi criada na fase anterior uma estrutura recursiva. Para suportar VBOs alteramos apenas um dos campos desta estrutura.
Desta forma, a class *Group* continua a conter um vetor de transformações, a cor que foi definida nesse nodo (ou, se nenhuma foi definida, a cor do nodo pai) e um vetor de Group filhos.
A novidade é que passou a conter um vetor de uma classe denominada de *Object*. Esta classe apenas contém o nome do modelo (*_file_name*) e a cor do modelos (*_colour*).
Assim, quando se está a ler o *XML* e é encontrado um modelo, é colocado nesse nodo uma instância da classe *Object* e é carregado o ficheiro correspondente ao modelo para um *GroupBuffer* (gb) passado por referencia como argumento.

```
vMod.push_back(
    Object(elem->Attribute("file"), parse_colour(elem, colour)));
gb.insert_model(elem->Attribute("file"));
```

2.2 Transformações com tempo

De forma a ser possível animar as *Scenes* geradas foram implementadas transformações que podem receber o atributo tempo.
De forma a conseguir manter a estrutura usada anteriormente todas as transformações, mesmo aquelas que não dependem do tempo, recebem quando são aplicadas o tempo que passou desde o inicio do programa (*elapsed*)

2.2.1 Rotação

Para fazer com que a rotação suporte tempo foi adicionado um campo à classe *Rotate* que representa quanto tempo demora a completar uma rotação(*_time*). Por defeito, para ser uma rotação estática basta colocar o tempo a 0.
Assim, para calcular o ângulo de uma rotação não estática fazemos a conta: *_ang + elapsed * 360/_time*.
Assim, uma rotação estática e não estática pode ser calculada com o seguinte código.

```
float total_ang = _ang + elapsed * (_time ? 360.0f / _time : 0);
glRotatef(total_ang, _x, _y, _z);
```

2.2.2 Translação - Curvas de Catmull-Rom

Visto que uma translação com tempo e uma translação sem tempo são inerentemente diferentes, contrariamente ao que foi feito para as rotações, foi criada uma classe nova para representar as translações com tempo.
Assim, caso uma translação possua o atributo do tempo, a classe *Catmull-Rom* é utilizada. Esta contem o tempo que deve demorar a percorrer todos os pontos definidos (*_time*) e todos os pontos (*_points*).
Em seguida é criada uma Transformação e é adicionada ao vetor de translações.
Para ser possível aplicar a transformação foi definido o método *apply* que recebe quanto tempo passou desde o inicio do programa (t) e quatro pontos (p0, p1, p2 e p3).
Primeiro criamos uma função capaz de calcular um ponto e a tangente ao longo de uma curva definida por 4 pontos.
Para tal definimos a matriz com valores constantes (m) e a matriz com os valores das coordenadas dos pontos (pm).

```
// catmull-rom matrix
const float m[4][4] = {{-0.5f, +1.5f, -1.5f, +0.5f},
                       {+1.0f, -2.5f, +2.0f, -0.5f},
                       {-0.5f, +0.0f, +0.5f, +0.0f},
                       {+0.0f, +1.0f, +0.0f, +0.0f}};

// point matrix
const float pm[4][3] = {{p0.x(), p0.y(), p0.z()},
                        {p1.x(), p1.y(), p1.z()},
                        {p2.x(), p2.y(), p2.z()},
                        {p3.x(), p3.y(), p3.z()}};
```

Em seguida multiplicamos a matriz das constantes pela matriz dos pontos (a).
Para calcular o ponto ao longo da curva num instante de tempo multiplicamos a matriz tv pela matriz a (pv)
Para calcular o vetor tangente à curva multiplicamos a matriz tvd pela matriz a (dv). Por fim basta devolver esses dois valores como um par.
Para calcular que valores devem ser passados à função definida anteriormente foi criada a função *get_location*.
Para saber quantas vezes ocorreu a translação dividimos o tempo que passou pelo tempo que se demora a completar uma rotação. Depois para calcular o tempo global multiplicamos o valor obtido anteriormente pelo número de pontos.
Finalmente para saber qual é o primeiro index e qual é o tempo dentro do segmento (de 0 a 1) arrendamos o tempo por defeito (index) e subtraímos ao t o index.
Finalmente calculamos quais são os indices dos pontos a utilizar e chamamos a função definida anteriormente.

```
u64 indices[4];
indices[0] = (index + point_count - 1) % point_count;
indices[1] = (indices[0] + 1) % point_count;
indices[2] = (indices[1] + 1) % point_count;
indices[3] = (indices[2] + 1) % point_count;

return get_catmull_rom_point(
    t,
    _points[indices[0]],
    _points[indices[1]],
    _points[indices[2]],
    _points[indices[3]]);
```

Para finalmente ser possível definir a função *apply* basta apenas definir ainda a função auxiliar para calcular uma matriz de rotação com base em 3 vetores denominada de *build_rotation_matrix*.
Assim, para aplicar uma translação não estática primeiro calcula-se com a função *get_location* a posição e a tangente na curva de *Catmull-Ron*.
Para mover o objeto para essa posição basta aplicar um *glTranslatef* com base nas coordenadas calculadas anteriormente.
Para finalmente rodar o objeto na direcão certa calculam-se 3 vetores. O vetor X e o vetor unitario com a mesma direção e sentido que o vetor tangente a curva de *Catmull-Ron*. O vetor Z e o resultado de fazer *cross product* entre o vetor X e o vetor (0, 1, 0) e normalizar o vetor resultante. E finalmente o vetor Y e o resultado de fazer o *cross product* entre o vetor Z e o vetor X e normalizar o resultante. Estes 3 vetores são colocados na função *build_rotation_matrix* e a matriz resultante e aplicada com *glMultMatrixf*.

```
void CatmullRon::apply(bool draw, float elapsed) {
    if (draw) draw_curve();
    auto point_dir = get_location(elapsed);
    auto p = std::get<0>(point_dir);
    glTranslatef(p.x(), p.y(), p.z());

    auto X = get<1>(point_dir).normalize();
    auto Z = X.cross(Vector(0, 1, 0)).normalize();
    auto Y = Z.cross(X).normalize();

    glMultMatrixf(build_rotation_matrix(X, Y, Z).data());
}
```

Para facilitar a visualização das curvas calculadas criamos a função *draw_curve* que desenha a curva no espaço.

2.3 Extensões de XML implementadas

2.3.1 Cor

Para além das funcionalidades pedidas no enunciado também implementamos a possibilidade de indicar a cor que se pretende desenhar.
De forma a facilitar a escrita das cores decidimos utilizar a sintaxe de #RRGGBB e #RRGGBBAA. E esta pode facilmente ser definida em qualquer label que não represente uma transformação. Assim, é possível definir a cor no group (fazendo que com todos os modelos e subgrupos tenham a mesma cor) ou dentro do models (fazendo com que todos os subgroups e modelos tenham a mesma cor).

2.3.2 HeightMap

Para carregar imagens como *HeightMaps* criamos uma tag nova de XML denominada de *terrain*. Para permitir um maior controlo também foi introduzido o campo *min* e o campo *max* indicando qual o intervalo para onde são mapeados os valores de cada pixel da imagem.
Assim, para definir na *Scene* uma imagem a ser carregada basta fazer.

```
<scene>
    <terrain file="terrains/terreno.jpg" colour="#FF0000" max="60"/>
</scene>
```

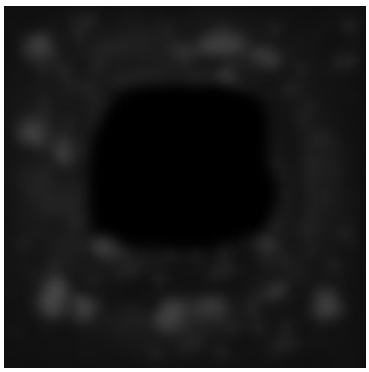

Figura 2.1: Imagem Original em terrains/terreno.jpg

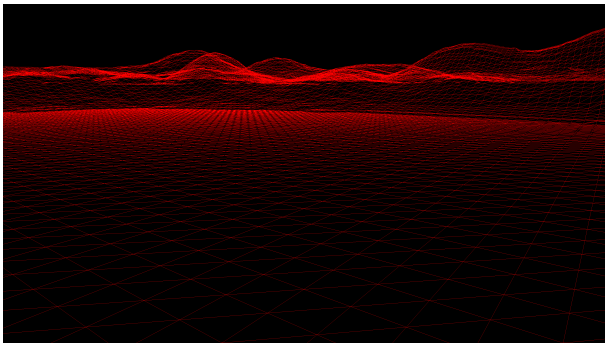

Figura 2.2: Resultado do XML

Para conseguir carregar o terreno foi expandido sobre o que tinha sido desenvolvido anteriormente para carregar um modelo.
À semelhança de um *ModelBuffer* desenvolvemos um *TerrainBuffer*. Este último contem um buffer, e as dimensões da imagem carregada em pixeis.
Assim, quando se quer carregar uma imagem para um VBO basta usar o construtor desta classe, sendo que o fileName é o caminho para a imagem e o *min_height* e *max_height* é o novo intervalo em que se encontram os valores de altura.
Primeiro a imagem é carregada usando *DevIL* para memória (*imageData*).
Depois são construídos os pontos num vetor.

```
//  Build the vertex arrays
std::vector<float> vec;
i32 new_interval = max_height - min_height;
for (i32 h = 0; h < _image_height; h++) {
    for (i32 w = 0; w < _image_width; w++) {
        float point = imageData[_image_height * h + w];
        float n_point = imageData[_image_height * (h + 1) + w];

        vec.push_back(w - (_image_width / 2.0));                        // x1
        vec.push_back(point * new_interval / 255.0f + min_height);      // y1
        vec.push_back(h - (_image_height / 2.0));                       // z1
        vec.push_back(w - (_image_width / 2.0));                        // x2
        vec.push_back(n_point * new_interval / 255.0f + min_height);    // y2
        vec.push_back(h + 1.0 - (_image_height / 2.0));                 // z2
    }
}
```

Por fim o buffer é inicializado e o vetor é copiados para o buffer.
Seguindo a mesma lógica, acrescentamos ao *GroupBuffer* outro *unordered_map* que associa o nome da imagem a um *TerrainBuffer* correspondente e criamos o método *draw_terrain* que procura nesse *unordered_map* e desenha.
Depois acrescentamos à estrutura recursiva que reflete o XML (*Group*) um vetor de *Objects* que representam os Terrenos carregados nesse nodo.

2.4 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/acelerar tempo
g	toggle modo de debug (eixos e caminhos de transformações)

Para implementar a pausa calculamos o tempo entre cada frame. Assim, só se incrementa um *static u64 elapsed* caso não se esteja em pausa.
Para suportar acelerar e abrandar o tempo basta definir um *TIME_SCALE* e incrementar e decrementar com base na tecla premida. Depois quando se incrementa o *elapsed* basta multiplicar o delta pelo *TIME_SCALE*.

```
1 //calculate current elapsed
2 static u64 elapsed = 0;
3 auto delta = frame_delta();
4 if (!PAUSE)
5     elapsed += delta * TIME_SCALE;
```

2.5 Window Title

De forma a facilmente apresentar a informação ao utilizador decidimos usar o titulo da janela.
Assim temos uma barra normal que mostra o modo atual da camera, o *TIME_SCALE* atual e o número de *frames per second* que se estão a obter. Também indica se se está em debug mode ou parado.

Figura 2.3: barra em estado normal

Figura 2.4: barra paused

Figura 2.5: barra em modo de debug

Capítulo 3

Generator

Neste módulo foram definidas nas fases anterior 6 primitivas:

- Plano
- Caixa
- Esfera
- Cone
- Cilindro
- Torus

Nesta fase implementamos a capacidade do generator interpretar *Patches de Bezier*.

3.1 Patches de Bezier

3.1.1 Parsing

Primeiro damos parse ao nível de *tesselation* (*_tesselation_level*).
Para ler os dados presentes no ficheiro primeiro lemos o número de patches presentes (*_n_patches*). Em seguida lemos os n patches seguintes e guardamos num vetor de vetores (*_index_patches*). Depois está presente o número de pontos de controlo (*_n_control_points*). Por fim lemos os n pontos seguintes para um vetor de pontos (*_control_points*).

3.1.2 Calcular pontos

Para calcular os pontos criamos uma função, *bezier_point*, que calcule um ponto ao longo de um conjunto de 4 pontos (p0, p1, p2, p3) com base na matriz de constantes de bezier e o tempo decorrido (t).
A parte de calcular as coordenadas do ponto é exatamente igual à parte de calcular um ponto numa curva de Catmull-Rom tirando a diferença na matriz de constantes.

```
1 // bezier matrix
2 const float m[4][4] = {
3     {-1.0f, +3.0f, -3.0f, +1.0f},
4     {+3.0f, -6.0f, +3.0f, +0.0f},
5     {-3.0f, +3.0f, +0.0f, +0.0f},
6     {+1.0f, +0.0f, +0.0f, +0.0f}};
```

Em seguida, utilizamos esta função para calcular as coordenadas de um ponto dentro de um patch. A função criada para isso recebe o índice do patch (*patch*) e o espaço relativo percorrido para cada lado do patch (u e v).
Primeiro criamos um vetor com todos os pontos que o patch contem. Ou seja, converter indices de pontos para os pontos em si.
Depois aplicamos a função calculada para obter o ponto com as coordenadas u dentro de cada bezier

patch de 4 em 4 pontos.
Depois aplicamos novamente essa conta aos 4 pontos calculados e utilizamos a mesma função mas desta vez com o v.

```
std::vector<Point> patch_control_points;
for (auto const& index : _index_patches[patch]) {
    patch_control_points.push_back(_control_points[index]);
}

std::vector<Point> new_patch_points;
for (auto i = patch_control_points.cbegin();
     i != patch_control_points.cend();
     i += 4) {
    new_patch_points.push_back(bezier_point(u, i[0], i[1], i[2], i[3]));
}

return bezier_point(
    v,
    new_patch_points[0],
    new_patch_points[1],
    new_patch_points[2],
    new_patch_points[3]);
```

Dividindo 1 pelo *_tesselation_level* obtemos o delta de cada ponto dentro de um patch.
Para obter triangulos válidos é necessário saber os 3 pontos que vêm imediatamente a seguir ao ponto que está a ser calculado. Para isso basta calcular as combinações de pontos para *ui + 1* e para *vi + 1*.

```
for (i64 p = 0; p < _n_patches; p++) {
    for (i64 ui = 0; ui < _tesselation_level; ui++) {
        const float uf = ui * delta;
        const float next_uf = (ui + 1) * delta;
        for (i64 vi = 0; vi < _tesselation_level; vi++) {
            const float vf = vi * delta;
            const float next_vf = (vi + 1) * delta;

            auto p0 = patch_generator(p, uf, vf);
            auto p1 = patch_generator(p, uf, next_vf);
            auto p2 = patch_generator(p, next_uf, vf);
            auto p3 = patch_generator(p, next_uf, next_vf);

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

            coords.push_back(p2);
            coords.push_back(p0);
            coords.push_back(p1);
        }
    }
}
```

Capítulo 4

Scenes

4.1 Terrain Generation

Tal como foi referido acima, quando fizemos a conversão do *engine* para VBOs fizemos de forma a que cada ficheiro contendo um modelo apenas fosse carregado para memoria uma vez mesmo que este aparecesse no XML varias vezes.
Para demonstrar as vantagens deste método criamos um script em python que gera uma grelha de cubos com tamanho 200 por 200. Visto que também queríamos tornar esta *Scene* visualmente apelativa decidimos variar a altura dos cubos com base em *Perlin Noise* fazendo com que o terreno final se aproxime bastante do aspeto do jogo *Minecraft*.
Visto que ainda não temos nem luz nem texturas, para ser possível distinguir cada caixa individualmente, o script atribui uma cor aleatória a cada um dos blocos.
O ficheiro gerado em XML encontra-se em *scenes/minecraft.xml*.

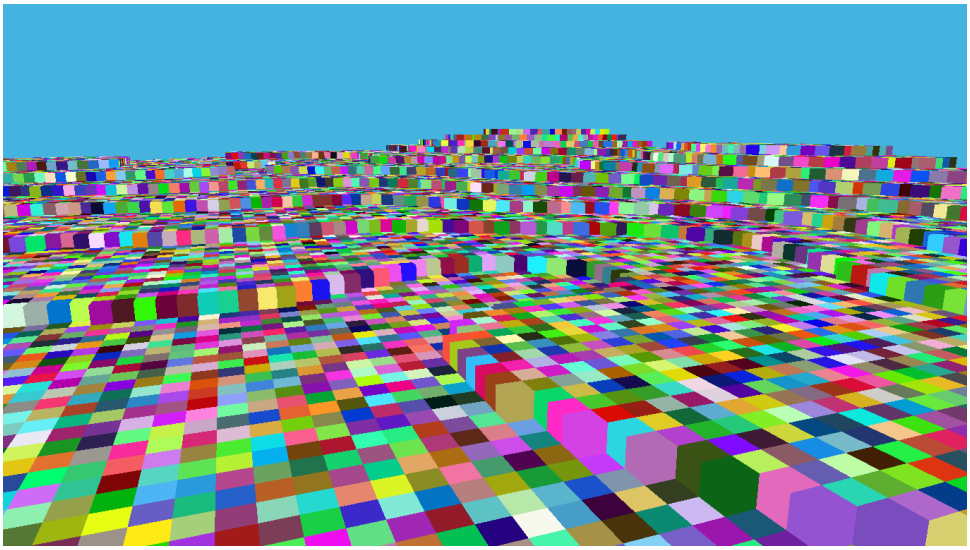

Figura 4.1: Imagem criada

4.2 Castle in the lake

Para demonstrar a capacidade das cores terem um fator de transparência e de 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* e a água é um plano com transparência.

Como ainda não temos luz nem texturas decidimos colocar vários ângulos para facilitar a percepção da *Scene* criada.
O ficheiro XML encontra-se em *scenes/castle.xml*.

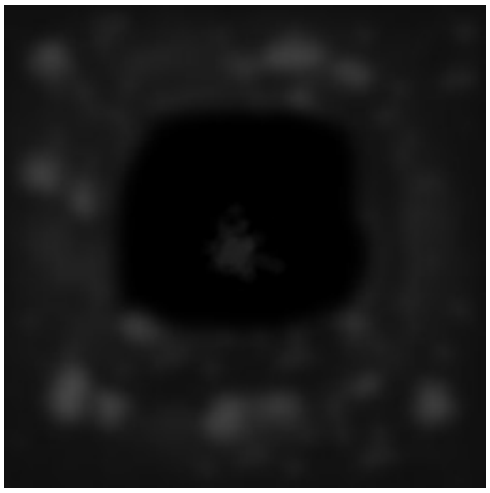

Figura 4.2: HeightMap

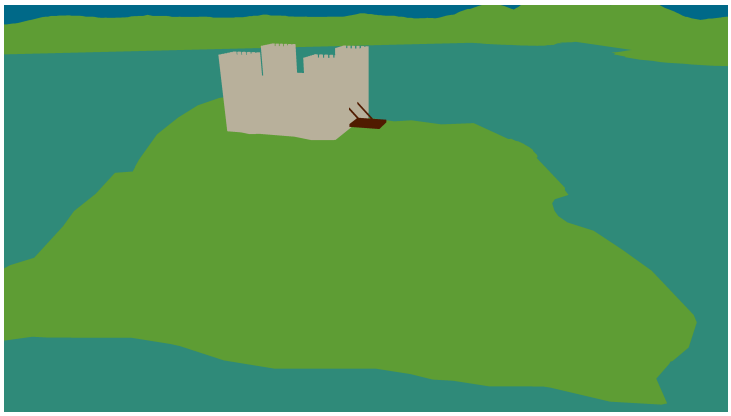

Figura 4.3: lado direito do castelo

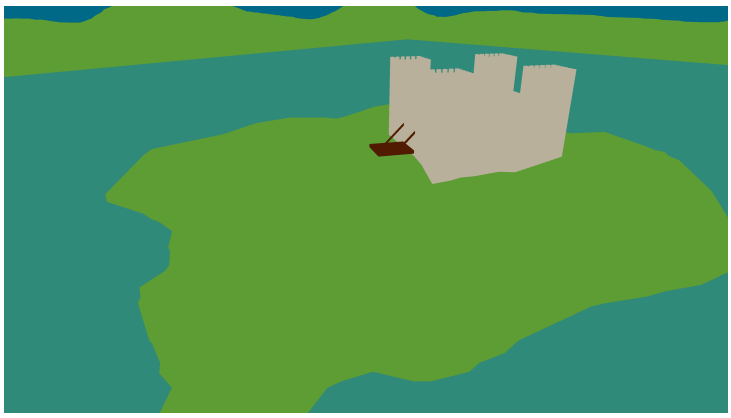

Figura 4.4: lado esquerdo do castelo

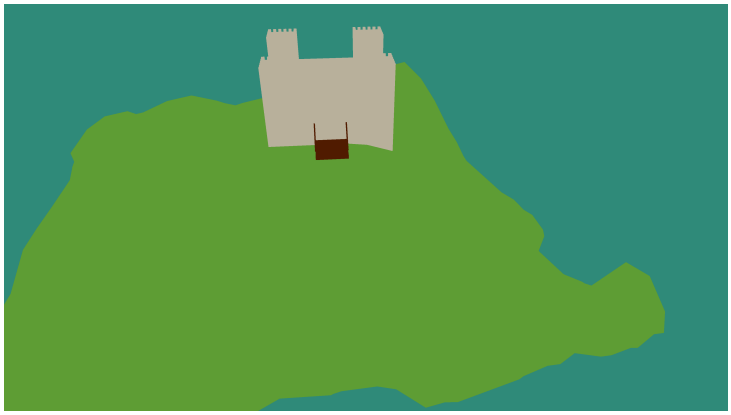

Figura 4.5: lado esquerdo do castelo

4.3 Solar System

Para facilitar desenhar o sistema solar decidimos criar um script em Python.
De forma a criar um modelo relativamente realista do sistema solar decidimos seguir um conjunto de regras simples, criando várias escalas permite que seja criado um modelo relativamente realista do sistema solar mas ao mesmo tempo visualmente apelativo.

- tamanho de planetas e luas está à escala entre planetas e luas
- distancia dos planetas e cinturas de asteróides ao sol está à escala
- o tempo de órbita e o tempo de rotação estão à escala mas acelerados por fatores diferentes

Os *CSV* obtidos na fase anterior forma expandidos para também incluir informação sobre o tempo de rotação e o tempo de órbita de cada planeta. Também acrescentamos Ceres, o maior asteróide na cintura de asteróides.
A informação acrescentada foi obtida na página de Wikipédia de cada planeta.
Para calcular a orbita do cometa decidimos aproxima-la a uma elipse, visto que esta é a mais próxima possível que pode ser facilmente computada. A fórmula mais comum de uma elipse é:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$

Assim, para calcular uma fórmula para a elipse de um cometa basta descobrir o a e o b. Os dados mais comuns que se encontram para a orbita de um cometa são a eccentricity, o perihelion e o aphelions. Assim podemos calcular os valores acima com.

$$a = \frac{aphelion + perihelion}{1 + eccentricity}$$

$$b = a * \sqrt{1 - eccentricity^2}$$

Agora basta calcular os pontos e mover a posição da elipse calculada centrada na origem.
À medida que o cometa se aproxima mais do sol mais depressa se desloca, e quanto mais se afasta mais devagar se desloca.
Visto que podemos representar a órbita de qualquer asteróide decidimos utilizar os dados do Halley.
O ficheiro gerado em XML encontra-se em *scenes/solar.xml*.

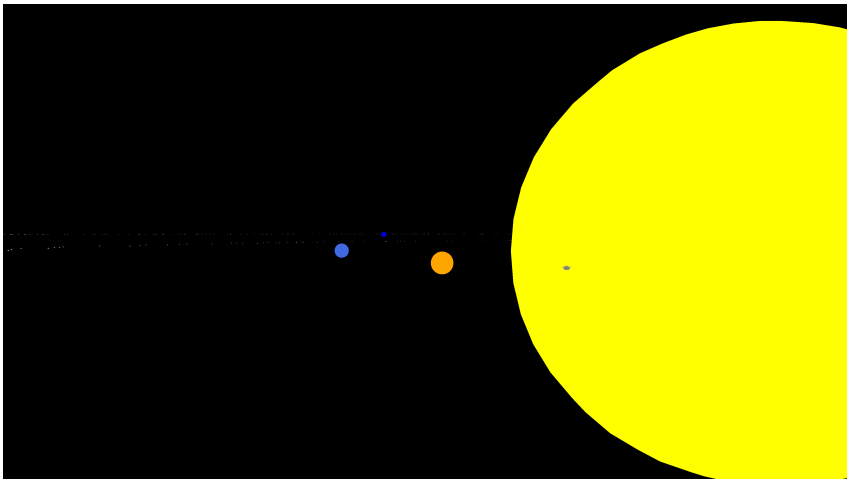

Figura 4.6: teapot em frente ao sol

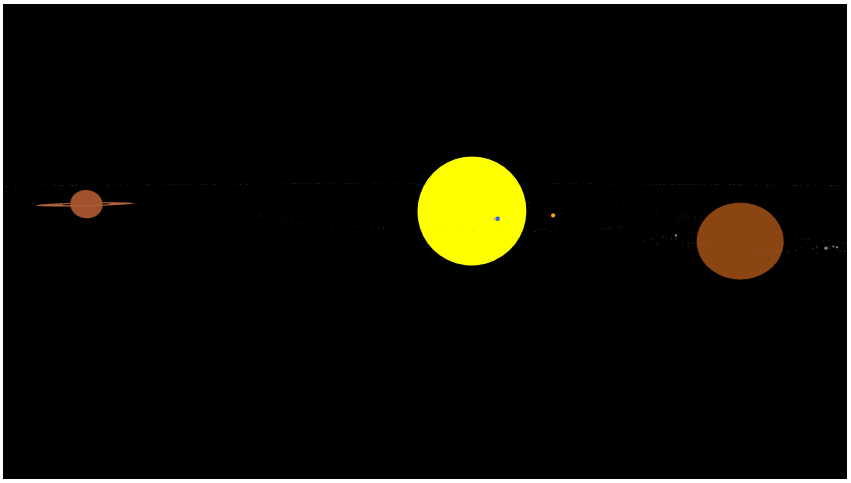

Figura 4.7: planetas interiores

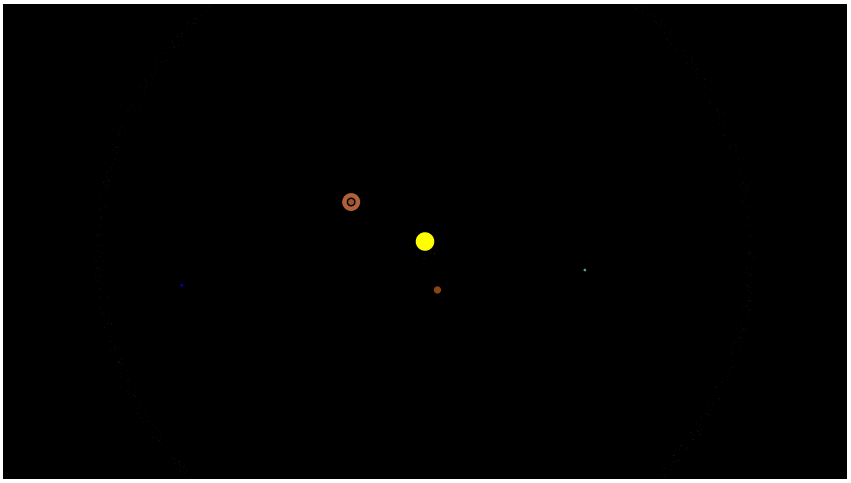

Figura 4.8: vista superior do sistema solar

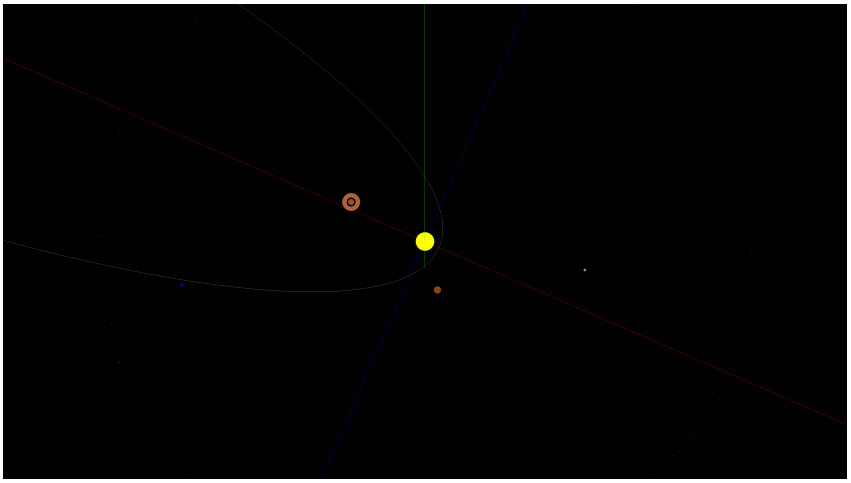

Figura 4.9: vista superior do sistema solar em debug mode

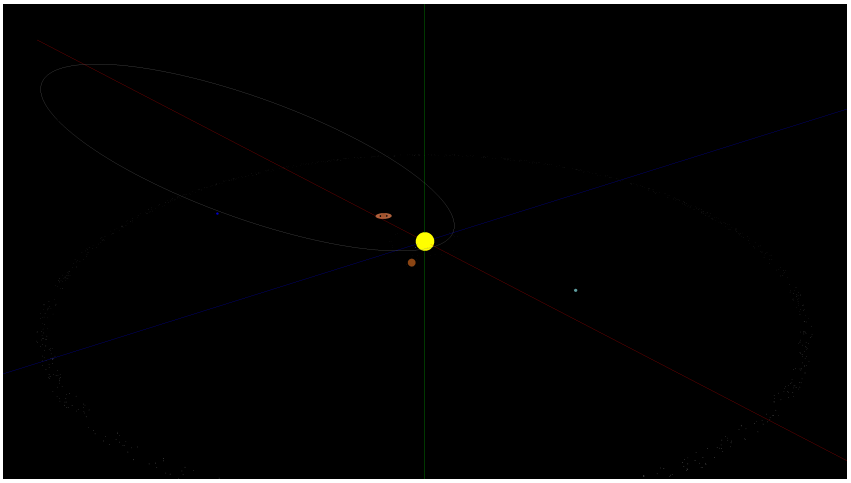

Figura 4.10: Vista de toda a órbita do Cometa em cebug mode

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 acrescentar iluminação e texturas, melhorar as *scenes* criadas até agora para usarem essas melhorias e acrescentar mais modos à camera.