Universidade do Minho
Escola de Engenharia

Computação Gráfica

Fase 3

Trabalho Prático

Grupo 9

João António Redondo Martins - a96215

João Pedro Antunes Gonçalves - a95019

Miguel de Sousa Braga - a97698

5 de maio de 2023

Conteúdo

1 Introdução **3**

2 Processo de Desenvolvimento e decisões tomadas **4**
2.1 Extensões das transformações 4
2.1.1 Implementação 4
2.1.2 Aplicação no Sistema Solar 6
2.2 Superfícies de Bezier 9
2.2.1 Implementação 9
2.2.2 Desenho do cometa 10
2.3 Implementação de VBOs 11
2.4 Extras 13
2.4.1 Modo 'minecraft' 13
2.4.2 Reimplementação do *ring* 17
2.4.3 *Teleports* dinâmicos 17

3 Resultados obtidos **18**
3.1 Testes fornecidos 18
3.2 Testes extra 19
3.3 Sistema Solar 19
3.4 Modo 'minecraft' 21

4 Conclusão e balanço da terceira fase **23**

Lista de Figuras

1 Imagem ilustrativa de um *patch* usado na esfera e os seus pontos de controlo. 11
2 Geração de um terreno através de uma imagem 14
3 Gradiente de cores 15
4 Órbitas luas de Júpiter 19
5 Imagem a bordo da nave espacial 20
6 Cometa 20
7 Creeper 21
8 Cena criada no modo 'Minecraft' 21
9 Cena mais complexa criada no modo 'Minecraft' 22

1 Introdução

Na terceira e penúltima fase do trabalho prático de Computação Gráfica, fomos levados a aplicar os conhecimentos adquiridos nas aulas teóricas e nas aulas práticas sobre superfícies de *Bezier* e curvas de *Catmull-Rom*. O objetivo era extender os comandos *XML* relativos a transformações para passar a incluir variações no tempo, permitindo assim gerar animações mais complexas. As translações passaram a ser parameterizadas com os parâmetros *time* e *align*, bem como alguns pontos que formam a curva por onde o objeto deve circular. Da mesma forma, as rotações podem agora ser parameterizadas com um tempo total de rotação, em vez de um ângulo. Já as superfícies de Bezier permitiram-nos criar objetos mais complexos, através de um conjunto de pontos de controlo. Para o Sistema Solar permitiu-nos, por exemplo, modelar um cometa. Para além destes objetivos, era também imperativo que todos os modelos fossem guardados em VBOs. Como trabalho extra, nesta fase, decidimos implementar um outro modo para a nossa *engine*, dando ao utilizador a liberdade de colocar e retirar objetos do mundo, podendo este definir as suas próprias cenas.

2 Processo de Desenvolvimento e decisões tomadas

2.1 Extensões das transformações

2.1.1 Implementação

Nesta fase do projeto, houve a necessidade de extender as transformações de modo a permitir realizar animações com o tempo. Por exemplo, ao aplicar a seguinte transformação sobre um qualquer objeto colocado em cima do eixo dos y, este deverá girar em torno do seu eixo, completando a rotação ao fim de 10 segundos.

```
<rotate time="10" x="0" y="1" z="0" />
```

Olhando para as translações, fornecendo um conjunto com mais de 3 pontos (necessidade das curvas de Catmull-Rom), podemos definir uma curva que une todos estes pontos. O tempo indicado determina o tempo necessário para percorrer a totalidade da curva.

```
<translate time="10" align="True" >
    <point x="1" y="0" z="1" />
    <point x="0.707" y="0.707" <="1""/>
    <point x="0" y="1" z="1" />
    ...
    <point x="-1" y="0" z="1" />
</translate>
```

Tivemos assim que acrescentar ao *parser* a capacidade de ler todos estes parâmetros (para rotações e para translações), atribuindo-os de seguida aos campos da transformação correspondente. Uma outra variável foi ainda acrescentada a cada uma das transformações. Esta variável marca o *timestamp* do início da contagem de um ciclo (quer de rotação, quer de translação).

Foi, por isso, necessário alterar o método *applyTransformation* em cada uma das transformações para considerar estes novos casos. De forma resumida, o algoritmo para aplicar uma dada translação é o seguinte:

- Calcula-se o tempo passado desde o início do ciclo atual;
- Determina-se um parâmetro t (entre 0 e 1) que dá a posição na curva;
- Caso o tempo passado desde o início do ciclo seja superior ao tempo estipulado para uma volta completa, reinicia-se a contagem;
- Com base no parâmetro t, calcula-se a posição e a derivada na curva;
  - Calcula-se o segmento da curva onde o objeto se encontra;

  - Define-se a ordem dos pontos (4) que vai determinar a próxima posição do objeto na curva;
  - A partir desses pontos define-se uma matriz que, multiplicada pela matriz que resulta da pré-multiplicação da matriz de Catmull-Rom pela matriz do parâmetro t, devolve a posição pretendida;
  - Para obter a derivada, o processo é semelhante, à exceção da matriz dos t's que deve ser derivada.

- Aplica-se a translação do objeto para a posição calculada;
- No caso de o atributo *align* estiver ativo, com base na derivada, calcula-se a matriz que define o referencial do objeto alinhado com a curva;
- Multiplica-se a matriz atual pela matriz calculada, com recurso à função *glMultMatrix*, efetivando a rotação.

Em resumo, o processo de cálculo das novas posições e derivadas do objeto, conforme o tempo, resume-se ao seguinte:

$$\begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P0 \\ P1 \\ P2 \\ P3 \end{bmatrix} \tag{1}$$

$$\begin{bmatrix} P'_x \\ P'_y \\ P'z \\ 1 \end{bmatrix} = \begin{bmatrix} 3t^2 & 2t & 1 & 0 \end{bmatrix} \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P0 \\ P1 \\ P2 \\ P3 \end{bmatrix} \tag{2}$$

Relativamente à matriz de rotação, sabemos que terá a seguinte estrutura:

$$\begin{bmatrix} X_x & Y_x & Z_x & 0 \\ X_y & Y_y & Z_y & 0 \\ X_z & Y_z & Z_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3}$$

onde as colunas da matriz são representativas dos eixos coordenados do referencial do objeto após a rotação.

Temos que, em cada momento:

$$\vec{X_i} = p'(t)$$
$$\vec{Z_i} = \vec{X_i} \times \vec{Y_{i-1}}$$
$$\vec{Y_i} = \vec{Z_i} \times \vec{X_i}$$

pelo que será necessário guardar o vetor do eixo dos y da iteração anterior.

Quanto às rotações, a primeira parte do processo é semelhante, isto é, a gestão dos *timestamps* e dos ciclos é a mesma. A aplicação da transformação propriamente dita torna-se mais simples, uma vez que usa a mesma primitiva (*glRotate*) que a transformação estática. A única diferença é que, neste caso, o ângulo depende do tempo passado desde o início do ciclo atual.

2.1.2 Aplicação no Sistema Solar

De forma a tornar mais realista a nossa cena, criamos um Sistema Solar dinâmico, adicionando, quer aos planetas quer às luas, animações que ilustram os movimentos orbitais em torno do Sol e os movimentos rotacionais em torno de si mesmos. Tomando como unidade 1 dia terrestre construímos a seguinte tabela:

	Período orbital (dias terrestres)	**Período de rotação (dias terrestres)**
Mercúrio	87.97	58.6
Vénus	224.7	243
Terra	365.25	1
Marte	687	1.03
Júpiter	4,332.59	0.41
Saturno	10,759.22	0.45
Úrano	30,685.4	0.72
Neptuno	60,190.03	0.67

Tabela 1: Períodos orbitais e rotacionais de cada um dos planetas

Relativamente às luas, as informações obtidas foram as seguintes:

	Planeta Originário	**Período orbital (dias terrestres)**	**Período de rotação (dias terrestres)**
Lua	Terra	27.32	27.32
Fobos	Marte	0.32	0.32
Deimos	Marte	1.26	1.26
Europa	Júpiter	3.55	3.55
Ganimedes	Júpiter	7.16	7.16
Calisto	Júpiter	16.69	16.69
Io	Júpiter	1.77	1.77
Titã	Saturno	15.95	15.95
Reia	Saturno	4.52	4.52
Tetis	Saturno	1.89	1.89
Titânia	Úrano	8.71	8.71
Oberon	Úrano	13.46	13.46
Tritão	Néptuno	5.88	5.88

Tabela 2: Períodos orbitais e rotacionais de cada uma das luas

Um outro pormenor tido em conta pelo grupo foi a forma da órbita de cada um dos planetas. A órbita de cada um dos planetas em torno do Sol é, na verdade, elíptica, pelo que o conjunto de pontos a fornecer deverá representar uma elipse. Para além disso, as órbitas não estão situadas no mesmo plano horizontal, tal como tinha sido apresentado no modo estático do Sistema Solar. Foi por isso necessário adicionar alguma inclinação às trajetórias elípticas dos objetos da cena. De modo a automatizar este processo de geração de pontos, acrescentamos uma nova extensão ao conjunto de tags disponíves:

```
<pointSet type="ellipse" num_points="16" a="10" b="7"
angleX="30" angleY="79" angleZ="0"/>
```

Esta nova primitiva permite gerar, na *engine*, um conjunto de n pontos pertencentes a uma elipse com semi-eixos de comprimento a e b, rodada sucessivamente em torno dos eixos x, y e z. O parâmetro *num_points* controla o número de pontos usados para gerar a elipse. Um maior valor deste parâmetro resulta numa elipse com maior resolução.

Para gerar os pontos, começamos por calcular os pontos da elipse assente no plano x0z. Estes pontos podem ser facilmente calculados através das fórmulas:

$$x = a * cos(angle) \tag{4}$$

$$z = b * sin(angle) \tag{5}$$

onde *angle* varia entre 0 e 2π e o seu incremento é controlado pelo número de pontos do conjunto. Quanto aos parâmetros a e b, é fácil notar que, caso estes sejam iguais, a elipse transforma-se numa circunferência de raio r, em que r=a=b.

Para visualmente o sistema solar ficar mais agradável também acrescentámos dois atributos ao *translate*. São eles o nível te tesselação, *tesselation*, e o *show*, para mostrar ou não a curva gerada.

Depois, para gerar os pontos finais, aplicamos, a cada ponto, as rotações indicadas em torno de cada um dos eixos coordenados. Com base nestes dados, atualizamos o ficheiro *XML*, adicionando estes novos tipos de translações e rotações a cada um dos objetos em que as mesmas fazem sentido.

Como exemplo, abaixo encontra-se a especificação para os movimentos da Terra e da Lua.

```
<group> <!-- Terra -->
    <transform>
        <translate time="365.25" align="false">
            <pointSet type="ellipse" num_points="16" a
                ="2420 b="2120" angleX="0" angleY="0"
                angleZ="0"/>
        </translate>
    </transform>
    <group>
        <transform>
            <rotate time="1" x="0" y="1" z="0"/>
            <scale x="10" y="10" z="10"/>
        </transform>
        <models>
            <model label="Terra" file="sphere_1_30_30.3d
                "/>
        </models>
    </group>
    <group> <!-- Lua -->
        <transform>
            <translate time="27.32" align="false">
                <pointSet type="ellipse" num_points="16"
                    a="38.7" b="35.7" angleX="4" angleY
                    ="0" angleZ="0"/>
            </translate>
            <rotate time="27.23" x="0" y="1" z="0"/>
            <scale x="2.7" y="2.7" z="2.7"/>
        </transform>
        <models>
```

```
            <model label="Lua" file="sphere_1_30_30.3d"/>
        </models>
    </group>
</group>
```

Podemos ver que tanto a terra como a lua sofrerão rotação em torno do seu eixo, bem como movimento de translação em torno de um centro orbital (no caso da terra, o Sol, no caso da Lua, a Terra). No caso da Terra, esse movimento é mais demorado, demorando 365.25 unidades (aqui 1 unidade = 1 dia). No caso da Lua, esse movimento demora cerca de 1 mês (27.32 unidades).

2.2 Superfícies de Bezier

2.2.1 Implementação

O programa *generator* também sofreu alterações nesta terceira fase. A primeira permitiu-nos gerar modelos bastante mais complexos, recorrendo a curvas/superfícies de Bezier. O nosso programa vai receber 3 argumentos:

- O ficheiro com os pontos de controlo para gerar a superfície;
- O nível de tesselação, que indica o detalhe que queremos dar ao objeto;
- O ficheiro onde vão ser escritos os pontos e os triângulos da figura (.3d).

Uma das primeiras tarefas a realizar foi o *parsing* do ficheiro com os pontos de controlo. Este tem a seguinte estrutura:

```
1: <n_patches>
2: patch1 (ind1, ind2, ...)
3: patch2 (ind1, ind2, ...)
...
2+n_patches: <n_points>
3+n_patches: point1 (x, y, z)
4+n_patches: point2 (x, y, z)
...
```

Recorrendo ao módulo *regex*, percorremos este ficheiro e guardamos os pontos de controlo e os índices em vetores.

Paralelamente, demos a possibilidade de o ficheiro conter apenas linhas com coordenadas de pontos de controlo e cujas coordenadas y e z estejam trocadas. Isto foi-nos muito útil pois permitiu-nos importar ficheiros gerados por *software* à parte, como o *Rhino 7*, ajudando-nos a desenhar superfícies cada vez mais complexas. O desenho do cometa foi, contudo, realizado a partir da manipulação dos pontos de controlo do *teapot*.

Assim, para gerar os vértices e os triângulos finais, começamos por percorrer cada um dos patches de 16 pontos, preenchendo uma matriz de pontos da seguinte forma:

$$\begin{bmatrix} P0 & P4 & P8 & P12 \\ P1 & P5 & P9 & P13 \\ P2 & P6 & P10 & P14 \\ P3 & P7 & P11 & P15 \end{bmatrix} \tag{6}$$

Em seguida, iteramos pelos diferentes valores de u e v (ambos entre 0 e 1 e dependentes do nível de tesselação). E criamos os vetores:

$$\begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \tag{7}$$

$$\begin{bmatrix} v^3 & v^2 & v & 1 \end{bmatrix} \tag{8}$$

Por fim, multiplicamos estes vetores e matrizes de forma a obter cada um dos pontos da figura da seguinte forma:

$$\begin{bmatrix} Px \\ Py \\ Pz \\ 1 \end{bmatrix} = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} M \begin{bmatrix} P0 & P4 & P8 & P12 \\ P1 & P5 & P9 & P13 \\ P2 & P6 & P10 & P14 \\ P3 & P7 & P11 & P15 \end{bmatrix} M^T \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix} \tag{9}$$

onde

$$M = M^T = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{10}$$

Para lidar com estas operações criamos um módulo auxiliar, *matrixOp*, onde estão definidas operações de multiplicação de matrizes de inteiros e matrizes de pontos (na verdade matrizes tridimensionais).

De seguida, foram construídos os triângulos com recurso a índices com localidade espacial (4 índices para cada conjunto de 4 pontos) e preenchidos os vetores que serão enviadas à função de escrita em ficheiro, da mesma forma que eram enviados os resultados da geração das outras primitivas.

2.2.2 Desenho do cometa

Conforme pedido no enunciado, o cometa é construído com *bezier patches*, mais especificamente com 8.

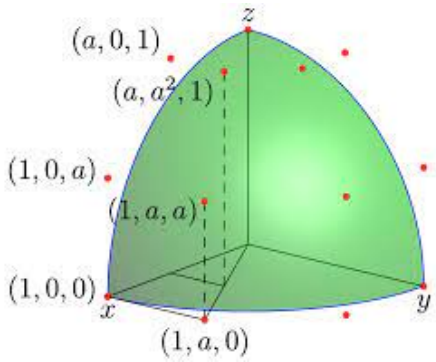


Figura 1: Imagem ilustrativa de um *patch* usado na esfera e os seus pontos de controlo.

Para a construção de cada *patch*, recorremos à imagem acima de forma a conhecer os pontos de controlo da mesma. Aqui, o *patch* é, na verdade, constituído por apenas 13 pontos, pois colapsa num único ponto (0,0,1). O parâmetro a controla a curvatura, variando entre 0 e 1. Um valor mais próximo de 1 irá originar um esfera de forma mais quadrangular, enquanto que um valor mais próximo de 0 irá gerar uma esfera mais 'encolhida'. Para gerar a figura completa, basta repetir o processo para as restantes porções da esfera.

Por fim, para ficar com um aspeto de cometa, moldamos a esfera, alterando alguns dos pontos de controlo, criando pequenas deformações à sua superfície. Para além disso, alteramos também o vértice (-1,1,0) para (-5,0,0), dando o efeito do rasto brilhante que o cometa deixa.

2.3 Implementação de VBOs

A renderização gráfica dos elementos da nossa cena com recurso a *VBOs* já tinha sido conseguida na fase 2, no entanto, devido a alguns erros na execução do programa, decidimos passar para a classe *Model* os buffers com os índices e com os vértices de cada objeto.

Mantivemos, contudo, a opção de simular o modo em que os objetos são renderizados como GL_TRIANGLES, entre os comandos *glBegin()* e *glEnd()*. Isto permitiu-nos comparar os desempenhos antes e depois da inserção de VBOs com localidade espacial, explicitados na fase anterior:

	FPS (média)	**Cena renderizada**
Sem VBOs	9	Sistema Solar com cintura de asteroides
Com VBOs	500	Sistema Solar com cintura de asteroides
Sem VBOs	1	Sistema Solar com cintura de asteroides e de Kuiper
Com VBOs	180	Sistema Solar com cintura de asteroides e de Kuiper
Sem VBOs	27	Sistema Solar sem cinturas
Com VBOs	800	Sistema Solar sem cinturas
Sem VBOs	630	Cubo com 2 de lado centrado na origem
Com VBOs	640	Cubo com 2 de lado centrado na origem

Tabela 3: Comparação entre renderização com ou sem VBOs

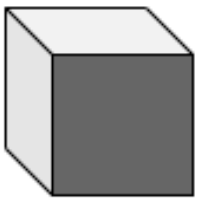
Estes testes foram realizadas num *pc* ASUS ROG com placa gráfica Nvidia GeForce GTX 1660 TI (with max-q design) e evidenciam o enorme proveito conseguido com o uso de buffers de vértices e índices internos à placa gráfica.

Podemos também verificar que, à medida que aumenta o número de triângulos a renderizar, aumentam também as diferenças de desempenho entre estes dois modos. Começando com apenas um cubo 2x2, de *grid* 3, verificamos que possuem desempenhos semelhantes. No entanto, à medida que vamos aumentando o número e a complexidade dos objetos a gerar, o desempenho com *VBOs* chega a ser cerca de 180(!) vezes superior (como no Sistema solar com cintura de asteroides e cintura de Kuiper).

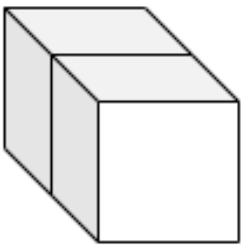
2.4 Extras

2.4.1 Modo 'minecraft'

Aproveitando a vasta gama de funcionalidades que o nosso motor gráfico já permite obter, decidimos criar um modo auxiliar que permite ao utilizador interagir com a sua própria cena, adicionando e removendo objetos a um terreno previamente gerado. Denominamos este novo modo 'minecraft' pelas semelhanças com o bem conhecido jogo *sand-box*. Optou-se aqui por limitar o tipo de objetos que podem ser colocados no mundo, simplificando o processo de cálculo da posição onde cada objeto deve ser colocado no mundo. Neste momento, apenas é possível colocar e remover cubos, apontando com o rato para a face de um determinado cubo que tem a orientação desejada.



(a) Bloco original



(b) Bloco colocado

Por exemplo, na figura acima, considerando a imagem da esquerda, se o utilizador tiver selecionado a face que se apresenta sombreada e premir o botão direito do rato, vai-se obter o resultado da imagem da direta. Por outro lado, se a partir da imagem da direita, for selecionado o bloco da frente e utilizador premir o botão esquerdo do rato, o resultado obtido será a imagem da esquerda.

Geração dos blocos

Uma das primeiras estratégias abordadas para a geração dos cubos consistiu na reutilização do código utilizado para o Sistema Solar, na base de grupos e transformações. Deste modo, cada cubo era representado por um grupo distinto e a sua posição era calculada recorrendo a translações aplicadas a um cubo colocado na origem, importado do ficheiro .3d gerado no *generator*. Isto implicava a criação de n VBOs, sendo n o número de cubos na cena. Para renderizar os cubos, era utilizada a função *drawGroup*, aplicada sobre cada um dos blocos.

Para um número reduzido de blocos (terreno 50x50), esta abordagem era minimamente viável. No entanto, à medida que aumentávamos o número de blocos, o desempenho tornava-se num *bottleneck* da nossa aplicação, devido ao elevado número de VBOs gerados.

Decidimos, por isso, usar apenas um VBO para desenhar todos os vértices de todos os cubos. Para isso, tivemos de abdicar das figuras geradas pelo *generator* e gerar os cubos internamente na *engine*, adicionando de seguida os pontos (e os índices) ao VBO respetivo.

De modo a facilitar o processo de seleção da face que é clicada pelo utilizador, cada bloco é desenhado como um conjunto de 6 planos de lado 1. Para isso, recorremos à função *drawCube*, que recebe como argumento a posição onde o cubo deve ser gerado. Esta função gera os 8 pontos que formam as faces do cubo e os 12 triângulos (2 para cada face). Os vértices e os índices são depois colocados em dois VBOs (um para todos os vértices e outro para todos os índices).

É possível gerar o terreno inicial como um plano de lado l ou importando um ficheiro de imagem e fazendo coincidir o valor do pixel (x,y) da imagem (entre 0 e 255) com a altura (possivelmente sofrendo uma escala) do bloco na posição (x,y) da cena. Na imagem seguinte, apresenta-se o resultado de gerar um terreno com base no ficheiro *terreno.jpg* disponibilizado nas aulas práticas.

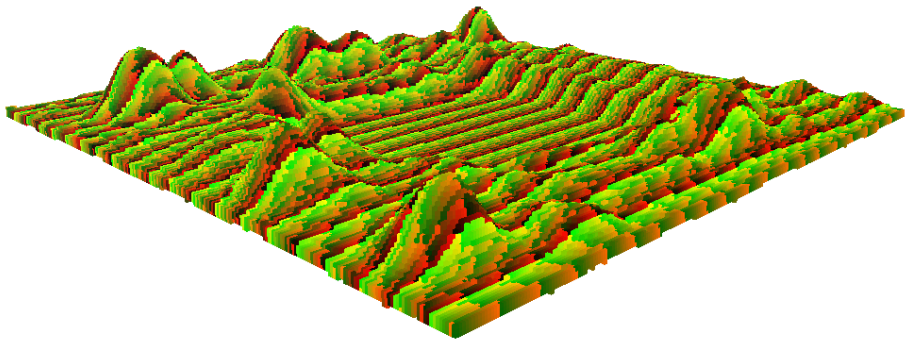


Figura 2: Geração de um terreno através de uma imagem

Nota: É necessário gerar, para cada posição (x,y) todos os blocos desde a altura desse pixel até n blocos abaixo dessa altura, fazendo com que não apareçam 'falhas' de blocos no terreno gerado.

Seleção da face

Após desenhar todos os blocos, o utilizador é colocado na cena em modo FPS, tendo a possibilidade de construir e destruir blocos, pressionando os botões direito e esquerdo, respetivamente. Sendo conhecida a face que o utilizador selecionou, é possível calcular a posição do próximo bloco a ser colocado ou remover o bloco ao qual pertence a face selecionada.

O algoritmo de seleção das faces é o algoritmo de *picking*, baseado no código *rgb*. É necessário distinguir cada face gerada com um código de cor, constituído por 3 valores de 1 *byte* (*red*, *green* e *blue*), que é incrementado sempre que é gerada uma nova face e guardado numa estrutura auxiliar global. Depois, sempre que o utilizador tenta colocar ou destruir um bloco, são desenhadas internamente todas as faces visíveis da cena, cada uma com uma cor diferente, de forma invisível para o utilizador. Por fim, com recurso à função *glReadPixels*, disponibilizada pelo *glut*, capturamos a cor do pixel onde o rato se encontra e mapeamos essa cor para a posição do próprio bloco (*array currPos*) ou para a posição do próximo bloco (*array nextPos*))

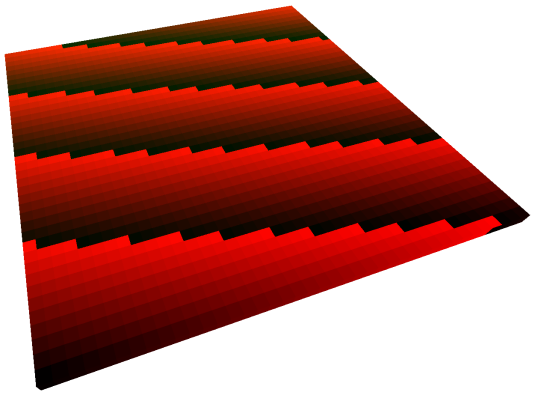


Figura 3: Gradiente de cores

Podemos ver, na figura acima, o resultado de gerar um plano 100x100, indicando, na função *render* o VBO com as cores que identificam cada face como o VBO das cores a desenhar. O resultado obtido é o de um gradiente de cores, que começa em 0 0 0 e é incrementado em 1 unidade, até eventualmente atingir o limite (255 255 255). Temos, por isso a possibilidade de desenhar até 16777216 faces diferentes (perto de 3 milhões de cubos).

Colocação de novos blocos e remoção de blocos

Ao ler a cor que identifica a face selecionada e ao ser calculado o bloco a ser removido ou o ponto do bloco a ser colocado, será necessário atualizar os VBOs para efetuar as alterações pretendidas. Para isso, recorremos à função *glBufferSubData*, fornecida pelo *glut*, que, dado um offset no buffer dos VBOs e o número de *bytes* a inserir, substitui essa porção do array pelos dados do novo cubo a inserir. Esta função é utilizada tanto para os vértices, como para os índices e para as cores. Dado que não é possível alocar mais memória de forma dinâmica para o mesmo VBO ao adicionar um novo objeto à cena, pre-alocamos algum espaço adicional no final de cada um dos arrays de modo a poder ser substituído por novos blocos, à medida que estes passam a fazer parte do mundo a renderizar.

De modo a permitir remover blocos da cena, era necessário remover os dados do bloco selecionado dos VBOs respetivos (vértices, índices e cores). Não havendo uma função de realocação da memória reservada para os VBOs na placa gráfica, tivemos que ser nós a controlar quais os dados, dentro do array, que seriam usados para renderizar a cena. Por isso, quando um bloco é removido, os *bytes* que representam esse bloco em cada um dos VBOs são preenchidos a 0s, o que impede que esse bloco seja renderizado de forma visível na cena.

Leitura e escrita em ficheiro

De modo a poder gravar, em ficheiro, as cenas criadas com recurso a este novo modo de criação, acrescentamos ao módulo *Creator* duas novas funções, *importScene* e *exportScene* que importam e exportam modelos de e para ficheiro, respetivamente. Os dados são gravados em modo texto, com a seguinte estrutura:

```
// Bloco 1
x1 y1 z1 red1 green1 blue1
// Bloco 2
x2 y2 z2 red2 green2 blue2
...
// Bloco N
xN yN zN redN greenN blueN
```

xi, yi e zi são as coordenadas do centro do bloco i e redI greenI e blueI as componentes vermelha, verde e azul da cor do bloco i guardado.

O utilizador tem, assim, a possibilidade de, ao entrar neste modo "criador", importar uma cena que tenha gravado anteriormente e continuar o trabalho produzido. Da mesma forma, ao sair deste modo para a cena principal do programa, é possível que este indique o nome do ficheiro para onde quer guardar a cena.

2.4.2 Reimplementação do *ring*

Tendo observado alguns problemas de desempenho na renderização de um elevado número de asteroides, problemas esses que eram acentuados quando se aumentava o nível de detalhe (slices e stacks) de cada esfera, decidimos repensar o sistema de geração de faixas (*rings*).

Na fase anterior, os *rings* estavam a ser gerados no programa *engine*, sendo reservado um grupo para cada um dos objetos gerados. A perda de *performance* observada era devido ao facto de estarem a ser gerados n *VBOs* adicionais (um para cada asteroide). No entanto, o facto de cada asteroide ter sido colocado num grupo diferente, permitia-nos fornecer transformações diferentes (rotações, translações e escalas) a cada um dos objetos, obtendo-se o resultado apresentado na fase anterior.

Para esta fase, procuramos resolver este problema criando um único VBO adicional, ao qual são adicionados os n objetos que constituem cada uma das cinturas. Para isso, geramos, no *generator*, cada um dos pontos dos n objetos a renderizar. Estes pontos começam por ser gerados relativamente à origem do referencial, sofrendo depois translações, rotações e escalas, para serem colocados na posição devida do Sistema Solar.

Esta alteração teve um impacto profundo no desempenho da nossa aplicação,

2.4.3 *Teleports* dinâmicos

Tendo terminado a modelação do novo Sistema Solar, agora, dinâmico, foi necessário alterar o sistema de teletransporte para considerar as posições em tempo real de cada um dos objetos.

Visto que é necessário calcular constantemente as posições dos objetos que possuem translações dinâmicas decidimos, a cada vez que a função *renderScene* é executada, calcular as posições dos obejtos. Para isso, começamos com a matriz identidade e quando uma transformação é aplicada, multiplicámos a matriz atual pela matriz da transformação. Para isso é importante conhecer as matrizes da translação 12, rotação sobre um qualquer eixo 13 e escala 11.

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (11) \qquad \begin{bmatrix} 0 & 0 & 0 & v_x \\ 0 & 0 & 0 & v_y \\ 0 & 0 & 0 & v_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (12)$$

$$\begin{bmatrix} x^2 + (1-x^2)cos(\theta) & xy(1-cos(\theta)) - z * sin(\theta) & z(1-cos(\theta)) + y * sin(\theta) & 0 \\ xy(1-cos(\theta)) + z * sin(\theta) & y^2 + (1-y^2)cos(\theta) & yz(1-cos(\theta)) + x * sin(\theta) & 0 \\ xz(1-cos(\theta)) - y * sin(\theta) & yz(1-cos(\theta)) + x * sin(\theta) & z^2 + (1-z^2)cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{13}$$

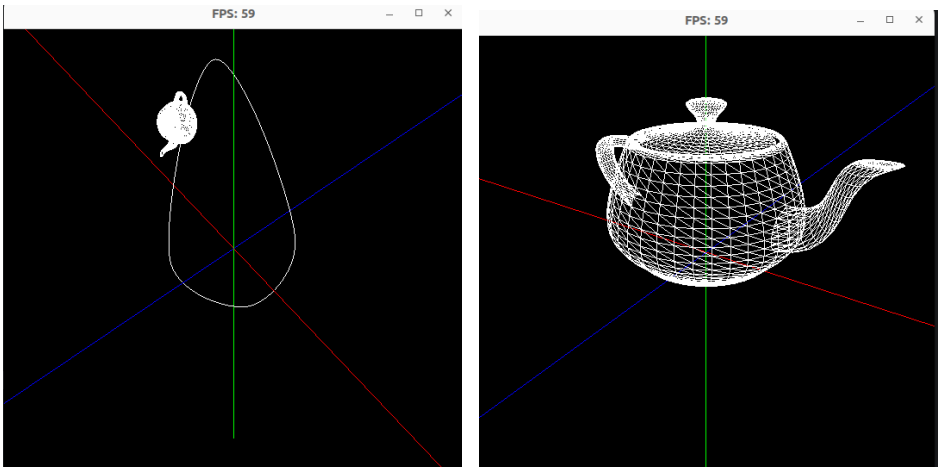
Para obter para os modelos pretendidos o ponto no qual ele está centrado basta retirar a compenente de translação da matriz.

Por fim, para obter o ponto em que queremos posicionar a câmera, temos de obter o raio. Para isso, basta multiplicar a matriz pelo ponto pertendido no sistema de coordenadas local e depois fazer a diferença do ponto resultante com o centro do objeto no espaço global, obtendo o vetor que nos permite calcular o raio, e assim posicionar a câmera corretamente no momento do teletransporte.

3 Resultados obtidos

3.1 Testes fornecidos

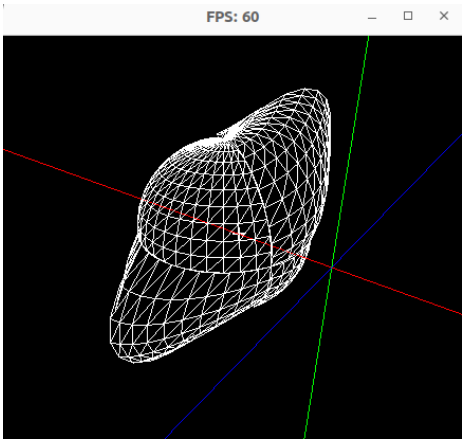
De forma a testar a correção do código para as translações, rotações e para as superfícies de Bezier, corremos os testes fornecidos, obtendo os seguintes resultados:



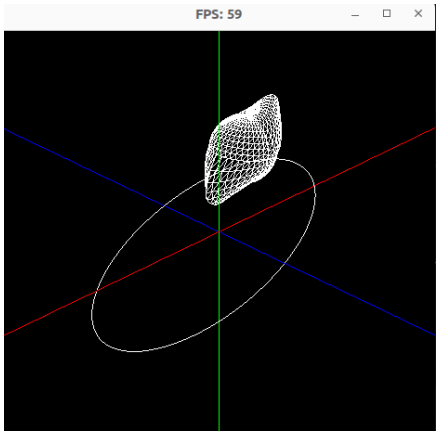
(a) Teste 1 (b) Teste 2

3.2 Testes extra

Para além desses testes, foram realizados outros, como forma de testar de forma mais profunda as funcionalidades do nosso sistema.



(c) Teste 3



(d) Teste 4

Nas imagens acima, podemos ver o exemplo de um outro modelo gerado com recurso a superfícies de *Bezier*. Para gerar os pontos de controlo das superfícies, recorremos ao programa *Rhino 7*, alterando alguns dos pontos de controlo da esfera, para adquirir a forma da figura c). Na figura d) apresenta-se o resultado de adicionar a esse modelo uma translação em formato elíptico.

3.3 Sistema Solar

Em seguida, mostra-se o resultado da colocação de órbitas em Júpiter.

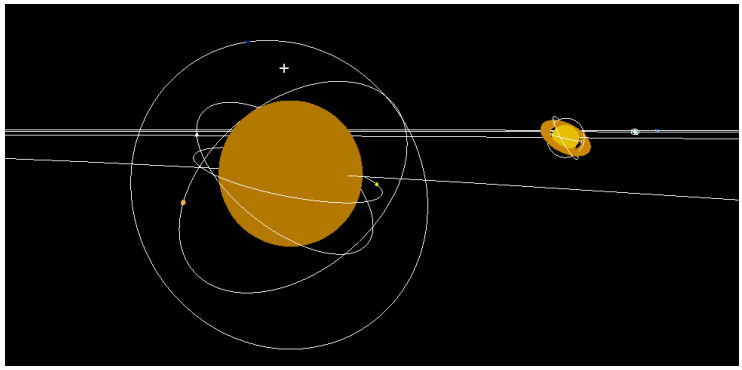


Figura 4: Órbitas luas de Júpiter

Agora, é também possível, graças aos *teleports* dinâmicos, capturar imagens panorâmicos, por exemplo a bordo da nave espacial:

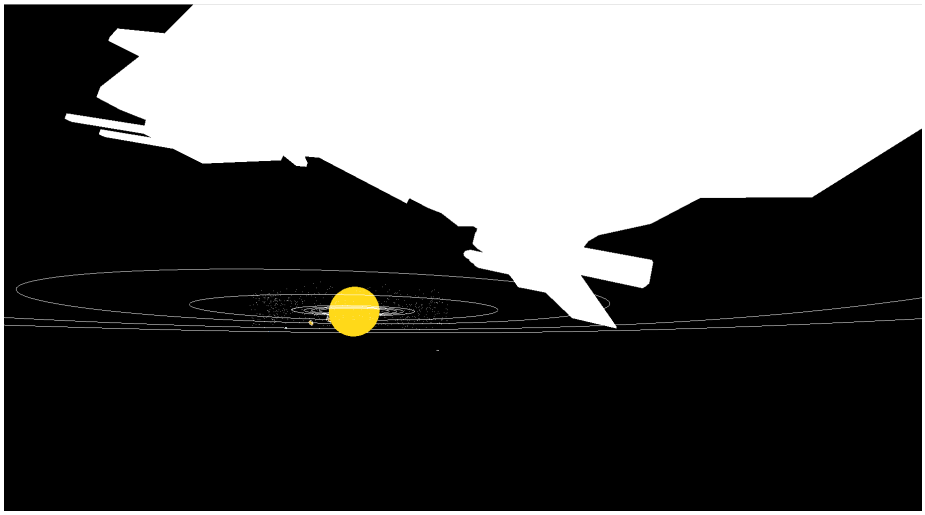


Figura 5: Imagem a bordo da nave espacial

A seguir, podemos ver também o cometa gerado com recurso a superfícies de *Bezier*, colocado numa órbita elíptica em torno do Sol (neste caso a órbita foi configurada como não visível):

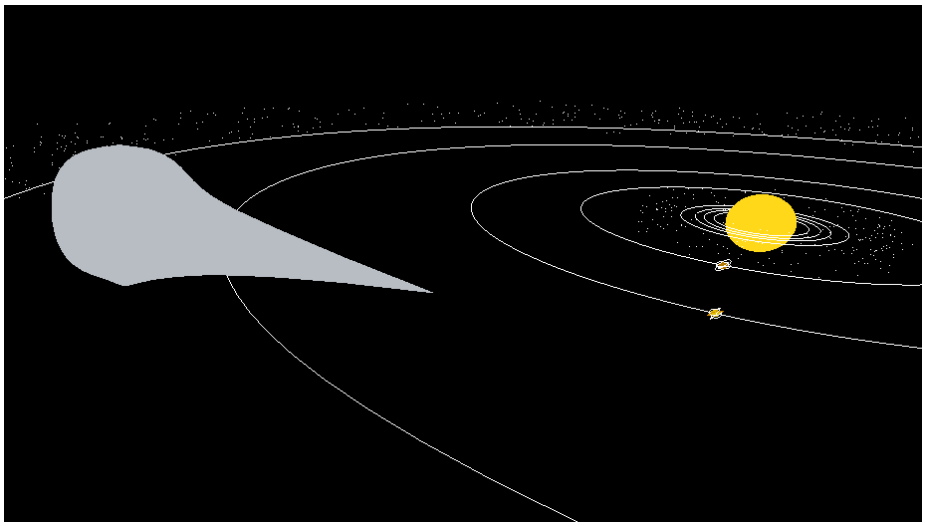


Figura 6: Cometa

3.4 Modo 'minecraft'

A seguir apresentam-se alguns exemplos de cenas criadas recorrendo a este modo 'criativo' de colocação e remoção de blocos.

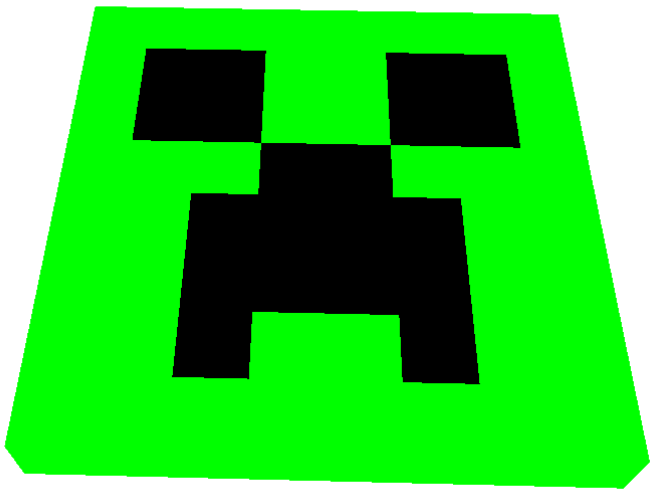


Figura 7: Creeper

Nesta imagem, podemos ver um plano verde, gerado com lado 8, de onde foram retirados alguns blocos e inseridos novos blocos de cor preto. A imagem pretende ilustrar a cara de um *creeper*, entidade existente no jogo *Minecraft*.

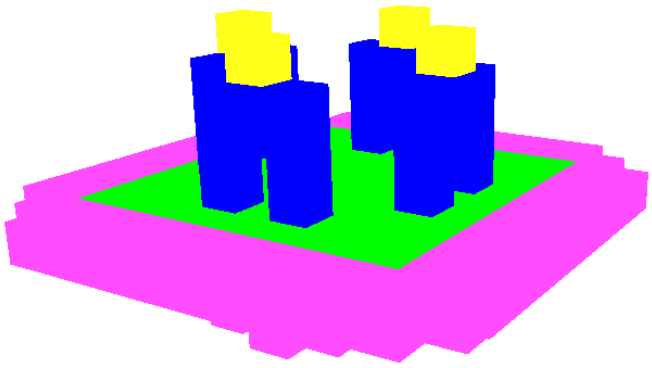


Figura 8: Cena criada no modo 'Minecraft'

Na cena acima ilustra-se uma dificuldade deste novo modo em ambientes 3d, relativo à falta de texturas dos blocos nesta fase. Para cenas com profundidade e com zonas preenchidas da mesma cor, torna-se difícil distinguir os contornos dos blocos e, assim, fica difícil saber quem está à frente e quem está atrás. Esta é uma dificuldade que procuraremos resolver na próxima fase.

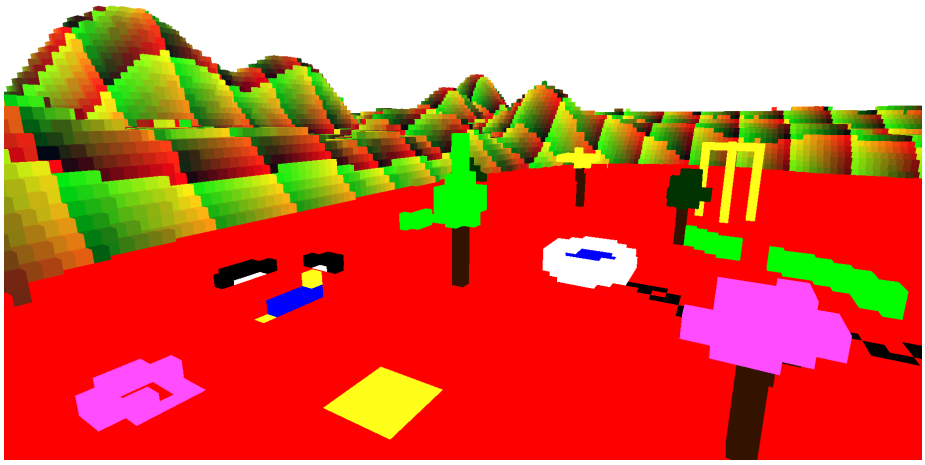


Figura 9: Cena mais complexa criada no modo 'Minecraft'

Por fim, elaboramos uma cena mais complexa de modo a mostrar as potencialidades deste modo da nossa aplicação. O tema para esta imagem foi um parque infantil, desenhado sobre o terreno gerado pela imagem disponibilizada. De forma a realçar o contraste entre as cores, o chão (todos os blocos com y=0) foi desenhado a vermelho.

Elaboramos ainda alguns vídeos que mostram, dinamicamente, os resultados obtidos. Estes encontram-se na pasta vídeos do *zip* enviado.

4 Conclusão e balanço da terceira fase

Fazendo a retrospetiva desta terceira e penúltima fase, avaliamos como positivo o trabalho desenvolvido. Tendo obtido os resultados esperados, o grupo conseguiu, de uma forma lúdica e divertida, colocar em prática os conhecimentos sobre animações, curvas e superfícies leccionados nas aulas. Para além disso, procuramos ir mais além e adicionar novas funcionalidades, tais como o modo 'minecraft', que não tinham sido previstas inicialmente. Isso ajudou-nos a explorar novas funcionalidades do OpenGL. Por fim, como ponto de partida para a próxima e última fase, para além dos objetivos estipulados no enunciados, tentaremos também aprimorar o modo 'minecraft' iniciado nesta fase, adicionando texturas mais realistas, limitando o alcance de colocação e remoção de blocos e adicionando novos tipos de câmara mais realistas.