

**Universidade do Minho**

Licenciatura em Ciências da Computação

**Unidade Curricular de**

**Computação Gráfica**

Ano Letivo de 2018/2019

**José Boticas A81241, Gonçalo Costa A82405, Ricardo Vieira A81640**

Maio, 2019

**Índice**

[1. Introdução 3](#_Toc7547505)

[1.1. Descrição do problema 3](#_Toc7547506)

[2 Aplicações 4](#_Toc7547507)

[2.1 *Generator* 4](#_Toc7547508)

[2.2 *Engine* 4](#_Toc7547509)

[3 Implementação 6](#_Toc7547510)

[3.1 Fase 1 6](#_Toc7547511)

[3.1.1 Plano 6](#_Toc7547512)

[3.1.2 Caixa 7](#_Toc7547513)

[3.1.3 Esfera 7](#_Toc7547514)

[3.1.4 Cone 8](#_Toc7547515)

[3.2 Fase 2 9](#_Toc7547516)

[3.2.1 Ficheiro XML 9](#_Toc7547517)

[3.2.2 Aplicação das transformações 10](#_Toc7547518)

[3.3 Fase 3 11](#_Toc7547519)

[4 Conclusões 12](#_Toc7547520)

## Introdução

O objetivo deste trabalho é o desenvolvimento de um motor gráfico 3D, mostrando vários exemplos comprovativos do seu potencial, utilizando o *OpenGL* e a biblioteca *GLUT*.

Foi-nos pedido o desenvolvimento de duas aplicações: um gerador de ficheiros com a informação de vários modelos (no caso, a aplicação gera apenas os vértices) e um motor que lê um ficheiro, escrito em XML, e que desenhe os modelos gerados previamente, tendo em conta qualquer transformação geométrica que se exija.

Numa **1ª fase**, para criar um ficheiro com um modelo, o gerador vai receber, como parâmetros, o tipo da primitiva gráfica e as restantes informações necessárias à sua criação. Como por exemplo, as dimensões e o ficheiro onde os vértices vão ser guardados.

As primitivas gráficas pedidas são:

* **Plano** (um quadrado no plano *XZ*, com centro na origem, feito com 2 triângulos);
* **Caixa** (requer *X*, *Y* e *Z* como dimensões e opcionalmente o número de divisões);
* **Esfera** (requer o raio e o número de *slices* e *stacks*);
* **Cone** (requer o raio da circunferência de baixo, altura e número de *slices* e *stacks).*

Na **2ª fase**, o motor deve ser capaz de implementar cenas hierárquicas e de aplicar transformações geométricas a um certo modelo, sendo elas: *rotate*, *scale* e *translate*. Para isso, cada cena é definida como uma árvore, onde cada nodo é composto por um conjunto de modelos e, opcionalmente, um conjunto de transformações geométricas. Cada nodo pode ainda ter nodos-filhos que herdam as suas transformações geométricas.

A **3ª fase** do projeto exige que o gerador seja capaz de tratar uma nova primitiva gráfica, as *patches de Bezier*, que o motor estenda o *rotate* e o *translate*, sendo esta última extensão feita com recurso a curvas *Catmull-Rom* e que os modelos sejam desenhados com recurso a VBOs.

Na **4ª** e **última fase** são implementadas as texturas e as luzes. Nesta fase, o gerador deve gerar as coordenadas de textura e as normais de cada vértice e o motor deve ter ativas as funcionalidades de texturização e iluminação, para além de aplicar as normais e as coordenadas de texturas dos ficheiros gerados.

## Aplicações

## *Generator*

O programa *generator* (gerador) gera os ficheiros dos modelos. Este pode ser invocado a partir do terminal, caso se use um sistema Unix ou Linux, executando o comando: ./generator figura param1 […paramN] ../../figura.3d. Aqui, “figura” é o nome da primitiva (*plane*, *box*, *sphere*, *cone* ou *Bezier*) a gerar, “param1 […paramN]” são os parâmetros, que variam consoante a figura e “figura.3d” é o ficheiro dos vértices que representam a figura.

Neste programa, criaram-se as funções *drawPlane*, *drawBox*, *drawCone*, *drawSphere* e *drawBezier*, que vão de encontro às primitivas gráficas pedidas e escrevem, nos ficheiros acima mencionados, as coordenadas dos pontos pertencentes a um conjunto de triângulos. Cada linha do ficheiro contém um conjunto de três números decimais (*floats*), correspondentes a um vértice. Cada triângulo é constituído por três vértices.

## *Engine*

O *engine* (motor) irá ler a configuração do ficheiro, escrito em XML, e mostrar o correspondente gráfico 3D. Para se invocar este programa, basta executar o comando: ./engine [ficheiro xml].

Recorremos à biblioteca *TinyXML2* para processar o ficheiro XML, que terá a seguinte estrutura:

<scene>

<lights>

<light1 type=[type] param1 […paramN]>

…

<lightN type=[type] param1 […paramN]>

</lights>

<group>

<[transformação1] param1 […paramN]>

…

<[transformaçãoN] param1 […paramN]>

<models>

<model file = figura1.3d/>

…

<model file = figuraN.3d/>

</models>

<group>

…

</group>

</group>

<group>

…

</group>

</scene>

O nº raiz deste ficheiro corresponde à *tag <scene>*, que contém os grupos, representados pela *tag <group>*. Estes elementos podem fazer parte de um outro elemento igual e podem ainda conter transformações geométricas, identificadas pelas *tags <translate>*, *<scale>* e *<rotate>*. Para além destas, os grupos devem ainda conter uma única *tag <models>*, sob a qual aparecerão as *tags* *<model>*, que correspondem às primitivas gráficas que se pretendem desenhar. Estas últimas possuem como atributo o caminho para o ficheiro de pontos a representar, gerado pela outra aplicação. De notar ainda que um subgrupo, ou seja, um grupo que esteja inserido noutro, herda as transformações geométricas deste.

Aquando da invocação desta aplicação, é certificado que o ficheiro XML foi passado como parâmetro e que o mesmo é válido. Prossegue-se com a criação de um objeto *XMLDocument* e invoca-se, sobre o mesmo, o método *LoadFile* com o nome do ficheiro recebido. De seguida, são extraídos e processados os elementos do mesmo, utilizando a abstração *XMLElement*. A cada *tag* é executada uma ação que a trata e adiciona as informações presentes na mesma à estrutura de dados correta.

O motor, depois de processar o ficheiro, deve representar o modelo correspondente. Para isso, aplica as transformações geométricas respetivas a cada um dos modelos que se querem representar.

## Implementação

Neste capítulo, são abordadas as implementações essenciais ao desenvolvimento de cada uma das fases do projeto.

### Fase 1

A primeira fase prende-se pela implementação de todas as primitivas pedidas. De notar que sempre que se falar em desenhar um triângulo, quer-se, na verdade, falar em guardar os vértices de um triângulo, já que o desenho é feito pelo motor desenvolvido. De destacar a existência da estrutura de dados que guarda os vértices lidos do ficheiro de cada modelo.

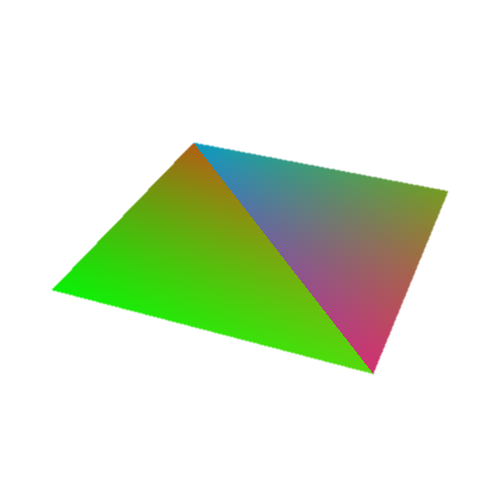
typedef std::tuple<float, float, float> vertice;

typedef std::vector<vertice> Vertices;

### Plano

void drawPlane (int x, int z)

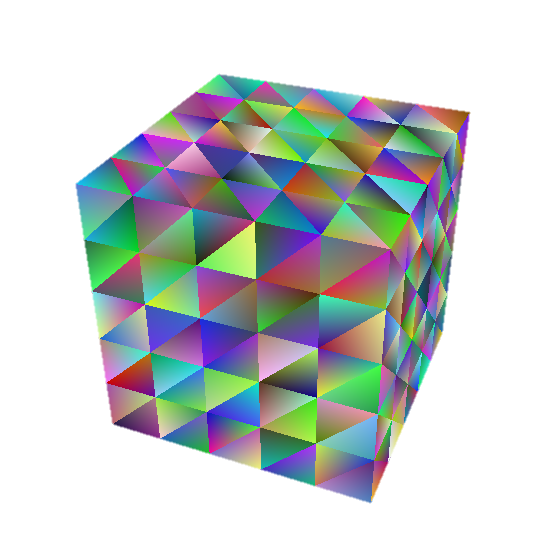
Para desenhar um quadrado/retângulo no plano *XZ*, desenham-se dois triângulos. Estes têm dois vértices em comum (os vértices de uma das diagonais do quadrado). Para que o plano possa ser visto quer de cima, quer de baixo, decidimos desenhá-lo por duas vezes, sendo assim criados quatro triângulos.

A função *drawPlane* recebe como parâmetros a distância máxima do plano ao longo dos eixos dos *xx* e dos *zz*. O centro do plano coincide com o centro do referencial. Portanto, a primeira coisa a fazer é dividir as componentes *x* e *z* por 2 e desenham-se os pontos com estes novos valores, respeitando a regra da mão direita.

### Caixa

void drawBox(float x, float y, float z, int divisions)

Para desenhar a caixa, escreveram-se dois ciclos que iteram até ao número de divisões pedidas (1 por *default*). Vendo o exemplo da face superior, mantém-se o *y* constante e calculam-se as coordenadas de *x* e *z*. Para isso, estas são iniciadas com o seu valor mínimo, ou seja, *-x/2* e *-z/2*, ficando assim a caixa centrada com o referencial, tal como acontece com o plano. O valor de *x* é incrementado em *x/divisões* a cada iteração do ciclo interior e restaurado para o seu valor inicial, mencionado atrás, a cada iteração do ciclo exterior. O valor de *z* é incrementado em *z/divisões* a cada iteração do ciclo exterior. Assim, conseguimos desenhar a face superior com as divisões pedidas, sendo que, a cada iteração do ciclo interior, são desenhados dois triângulos. A cada iteração do ciclo exterior fica completa uma linha da face em questão.

O desenho das restantes faces é análogo. Importa referir que, por comodidade, são guardadas duas variáveis correspondentes a *x*, *y* e *z*.

### Esfera

void drawSphere(float radius, int slices, int stacks)

No desenvolvimento da esfera, começou por se pensar apenas na metade superior da mesma. O processo iniciou-se com o desenho de tantas circunferências quantas as *stacks* pedidas e depois ajustaram-se as suas alturas e raios, de forma a termos uma estrutura e percebermos quais seriam os vértices dos triângulos, a cada *stack*.

A partir daqui tudo ficou mais facilitado e tivemos apenas de ir desenhando uma a uma as *stacks*, guardando, para isso, dois raios e duas alturas (do topo e da base da *stack*).

O raio varia de forma decrescente e a sua variação é igual…(completar)

As *slices* determinaram quantos triângulos seriam desenhados por cada *stack*.

Foram então desenvolvidos dois ciclos *for* aninhados, um para controlar o número de *stacks* e outro para se desenharem os triângulos de cada uma delas. Os raios e as alturas eram atualizados a cada iteração do primeiro ciclo.

Uma imagem com balão, aeronaves, transporte

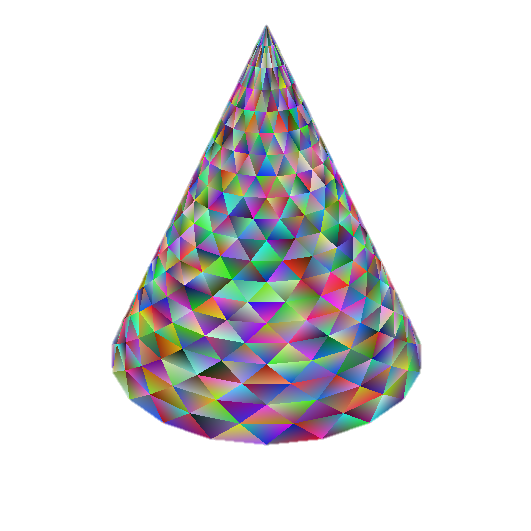
Descrição gerada com confiança muito altaA metade inferior da esfera é uma cópia da metade superior em que se alteram apenas os valores de *y* para o seu simétrico, ou seja, para *-y*. Tem também de se ter em conta a ordem por que são escritos os vértices.

### Cone

void drawCone(float radius, float height, int slices, int stacks)

Para o desenho do cone, foi utilizada a mesma estratégia usada no desenvolvimento da esfera. Começou-se então por se pensar em várias circunferências. Desta vez, o raio de cada uma das circunferências decresce de forma constante. Esta diminuição é igual à divisão do raio da base pelo número de *stacks*. As alturas das circunferências calculam-se da mesma forma (começa-se em 0 e aumenta-se de forma constante e igual à divisão entre a altura e o número de *stacks*). Mais uma vez, o número de slices é igual ao número de triângulos de cada *stack*.

Assim, começamos por desenhar a base do cone, que consiste numa circunferência desenhada com o número de triângulos igual ao número de *slices*. De seguida, desenham-se as *stacks*, uma a uma, como explicado acima.

Foi então escrito um ciclo *for* para o desenho da base e outros dois ciclos *for* aninhados, seguindo a mesma estratégia do desenho da esfera. De realçar que a diferença entre estas duas figuras se prende pela forma como eram calculados os raios das circunferências que serviram de base para o pensamento.

### Fase 2

A segunda fase passa por tratar as transformações geométricas, permitindo a construção de um sistema solar estático.

### Ficheiro XML

As principais alterações nesta fase dão-se a nível do ficheiro XML. Este passa a conter a informação de cada planeta num grupo, desde as transformações nele aplicadas (*scale*, *rotate*, *translate*) até outros grupos que podem representar os seus satélites, naturais ou artificiais. De notar que a ordem das transformações associadas a cada grupo tem influência no resultado e que estas também são aplicadas nos subgrupos do respetivo grupo.

O ficheiro XML foi então escrito de forma a que representasse um sistema solar. Para isso foram criados nove grupos, um por cada planeta e ainda um para o Sol. Um dos grupos, que, no caso, é o grupo que representa o planeta Terra, contém um subgrupo, que corresponde à Lua. Cada grupo tem um *translate*, excetuando o grupo que representa o Sol, de forma a que os modelos não sejam desenhados no mesmo local. Têm ainda um *scale*, por forma a que as representações dos planetas tenham diferentes tamanhos, tornando o resultado final mais agradável à vista. De referir que não estão representadas as luas de Júpiter, por serem muitas. Tanto os planetas como o Sol são desenhados a partir da mesma primitiva, uma esfera, com recurso à *tag <model file=”../../sphere.3d”/>*. Assim, as diferentes representações são conseguidas com recurso às transformações geométricas referidas.

Importa ainda realçar que, a cada modelo, pode ser associada uma cor diferente, através da *tag <color R=r G=g B=b>*.

### Aplicação das transformações

Foi alterada a aplicação *Engine* de modo a guardar as informações do ficheiro XML. Para isso criaram-se novas estruturas de dados: uma para guardar transformações e outra para guardar os grupos, sendo que esta última inclui, para além da estrutura de vértices já existente, a primeira.

typedef std::tuple<char, float, float, float, float> transformation;

typedef std::vector<transformation> Transformations;

struct group {

    Transformations trans;

    Vertices v;

    std::vector<struct group> subGroups;

} Group;

Esta leitura é feita linha a linha, através da função *addGroup*, identificando as *tags* e guardando as transformações, vértices e subgrupos, grupo a grupo.

A cada transformação geométrica encontrada, cria-se um tuplo *transformation* e é feito um *push\_back* para o vetor de transformações do grupo. O primeiro elemento de cada tuplo é um *char* que identifica o tipo de transformação, sendo que T identifica um *translate*, R identifica um *rotate* e S identifica um *scale*. Os restantes elementos são *floats* e são usados para guardar os detalhes de cada uma delas. Quando é encontrado um modelo, guardam-se os vértices do ficheiro correspondente no vetor de vértices do grupo. A cada subgrupo, repete-se o processo normal de um grupo, com a particularidade de que estes são guardados no vetor de subgrupos do grupo “pai”, em vez de serem guardados na variável global que guarda os grupos.

O desenho do gráfico 3D é feito através da estrutura de dados que guarda os grupos. A cada entrada desta estrutura de dados é executada uma função para o desenho de um grupo (*drawGroup*). Esta função inicia-se com a execução da função *glPushMatrix*. De seguida, aplicam-se as transformações pela ordem em que são lidas e desenham-se os vértices, sendo estes afetados pelas transformações geométricas referidas. No final, é executada a função *glPopMatrix*, de modo a que o sistema de coordenadas seja restaurado e as transformações geométricas até aqui aplicadas, não interfiram com outros grupos que não sejam subgrupos do mesmo. Para o desenho dos subgrupos segue-se a mesma lógica, uma vez que se trata de um grupo, sendo que estes aplicam, no entanto, as transformações do grupo “pai”. Portanto, antes de ser executada a função *glPopMatrix* e se seguir para o próximo grupo, são desenhados os subgrupos do grupo que está a ser desenhado. A estrutura é a seguinte:

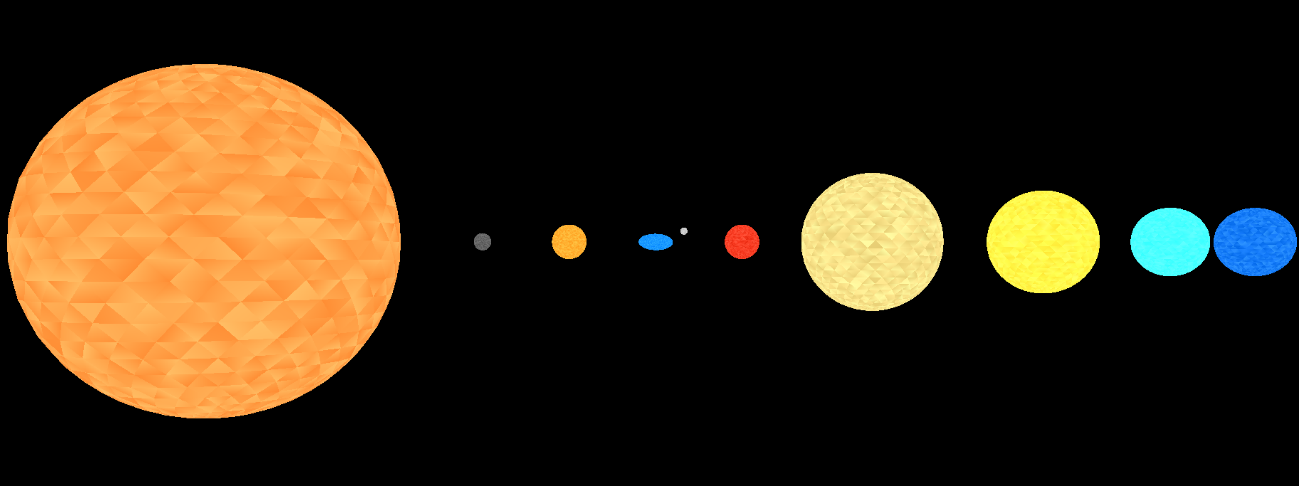
glPushMatrix();

aplicar transformações;

desenhar vértices;

desenhar subgrupos;

glPopMatrix();

**3.3 Fase 3**

A terceira fase passa por mudar o gerador de modo a que possa criar um novo tipo de modelo baseado nas *Bezier patches*. Para além disso, implementaram-se duas novas formas de transformação geométrica, *rotate* e *translate* temporais.

* + 1. **Ficheiro XML**

Foram feitas mudanças no ficheiro XML ao adicionar, para cada grupo, um *rotate* *time* para a rotação de cada planeta à volta do sol (movimento de translação) e outro *rotate* *time* para a rotação sobre si próprio. Adicionou-se ainda um novo grupo, que contém, como modelo, o *teapot* gerado pelo ficheiro com pontos de controlo fornecido, associado a um *translate time* e alguns pontos de controlo, por forma a que o mesmo descreve uma curva, em órbita do Sol.

* + 1. ***Generator***

Nesta fase, foi pedido que o generator fosse capaz de criar uma nova primitiva baseada em *Bezier patches*. Este recebe como parâmetros o ficheiro com os pontos de controlo e o nível de tesselação pretendido.

Para o tratamento deste ficheiro, foi criada uma estrutura que guarda os pontos de controlo, e outra que guarda as ditas *patches*, que não são mais do que índices que indicam quais os pontos de controlo a utilizar.

Para o tratamento do ficheiro, lê-se o mesmo linha a linha. A primeira linha indica o número de *patches*, que é usado para a alocação de uma matriz com *numPatches* linhas e 16 colunas, em que *numPatches* é o número lido da primeira linha do ficheiro. De seguida, lêem-se as *numPatches* linhas seguintes e guardam-se os índices na matriz. O ficheiro repete o mesmo padrão, mas, desta vez, com pontos de controlo. O processo para guardar os pontos de controlo é semelhante.

Após este tratamento, é processada a superfície curva. Itera-se por todas as *patches* e cada *patch* são usados 16 pontos de controlo. É repetido o processo de avaliar a superfície de *Bezier* (t+1) \* (t+1) vezes, por cada *patch*, sendo *t* a tesselação indicada. O resultado de cada avaliação é um ponto de controlo. Por fim, é gerado um vetor de índices que regerá a forma como se escrevem os pontos de controlo gerados no ficheiro do modelo.

* + 1. ***Engine***

Para esta fase, era pretendido que o motor fosse capaz de animar as rotações e as translações dos planetas e que se usassem VBOs. Uma das diferenças mais importantes desta fase é a alteração da estrutura de dados *group* e a criação de novas estruturas, *model* e *curva*.

struct group {

    Transformations trans;

    Curva c; // para guardar uma curva, caso exista

    std::vector<struct model> models; // para guardar os modelos

    std::vector<struct group> subGroups;

} Group;

struct model {

    int indice;

    int numVertices;

    std::vector<vertice> vertices;

} Model;

struct curva {

    int valid;

    float time;

    Vertices pontos;

} Curva;

* + - 1. **Animação do *rotate* e do *translate***

A animação do rotate é definida indicando, no ficheiro XML, o tempo de duração da mesma e os eixos na qual será feita a rotação. Essas informações são guardadas no vetor de transformações do grupo correspondente. O tuplo criado para tal, é identificado pelo *char* I. Este *rotate* é tratado como um normal, com a diferença de que o ângulo de rotação é calculado com base no tempo dado e no tempo decorrido até ao momento, usando para tal a função glutGet(GLUT\_ELAPSED\_TIME) do *OpenGL*. Tomemos por *t* o tempo passado no ficheiro XML e por *td* o tempo decorrido até ao momento. A fórmula usada para calcular o ângulo de rotação é a seguinte: fmod(td/1000, t)\*(360/t). É usada a função *fmod* para que *td* esteja entre 0 e *t*; multiplica-se o resultado por 360/*t* ,porque 360 representa uma volta completa.

De notar que, no ficheiro XML, o primeiro *rotate* *time* (do movimento de translação) é feito antes do *translate*, que posiciona o objeto na posição correta, e o segundo (do movimento de rotação) é feito depois. Deste modo, podemos ter a mesma definição para ambos os movimentos.

A animação do *translate* recorre a curvas de *Catmull*-*Rom*. Para tal, são fornecidos os pontos de controlo no ficheiro XML que são guardados numa estrutura, *curva*, que, por sua vez, é guardada na estrutura do grupo. O tempo que demora a percorrer a curva é baseado no tempo dado no parâmetro *time* do *translate*. A estrutura *curva* tem ainda uma variável *valid* que indica se uma dado grupo tem uma curva ou não. Assim, quando se estão a desenhar os grupos, basta verificar essa variável para ser tomada a decisão de correr o código que corresponde à aplicação de uma curva ou não.

Para o cálculo da curva, é chamada a função criada *getGlobalCatmullRomPoint* com todos os pontos de controlo e o tempo decorrido até ao momento. Essa função, dependendo do tempo que recebe, junta os quatro pontos que lhe correspondem e chama a função *getCatmullRomPoint* com esses quatro pontos. Esta última, aplica a matriz de Catmull-Rom a cada um dos pontos, obtendo-se um novo ponto. É então a aplicado um *translate* com o ponto obtido.

* + - 1. **VBOs**

De modo a melhorar a performance do programa fizemos a transição para VBOs. Assim, reduzimos o número de chamadas a funções e o uso redundante de vértices comuns permitindo uma renderização mais rápida da imagem.

Para a implementação dos VBOs, alterou-se a estrutura *Group*, de forma a que a mesma pudesse guardar um vetor de modelos. Portanto, substituiu-se o vetor de vértices existente por um novo vetor. Este novo vetor tem por base uma nova estrutura, *Model*, que guarda dois inteiros, *indice* e *numVertices*, bem como um vetor de vértices, que são os vértices do ficheiro indicado pela *tag <model>*. O *indice* indica qual o índice do *buffer* que representa o VBO do modelo; *numVertices* indica a quantidade de vértices do modelo, o que é útil para a geração do próprio VBO.

Enquanto se faz a leitura do ficheiro XML, mantém-se a contagem do número de modelos já lidos e, a cada um destes, gera-se um *Model*, com as informações presentes na respetiva *tag*, e adiciona-se o mesmo ao respetivo grupo.

No final dessa leitura, aloca-se o espaço para os *buffers* dos VBOs, tantos quanto o número de modelos, do qual, tal como referido atrás, se manteve uma contagem. De seguida, geram-se os mesmos com o seguinte pedaço de código: glGenBuffers(numModels, buffers), em que *numModels* é o número de modelos e *buffers* é a variável onde se guardou o espaço alocado anteriormente. Posteriormente, preenchem-se os VBOs. Para isso, percorrem-se os grupos e, a cada modelo, corre o seguinte código:

glBindBuffer(GL\_ARRAY\_BUFFER, buffers[m.indice]);

glBufferData(GL\_ARRAY\_BUFFER, sizeof(float) \* m.numVertices \* 3, m.vertices.data(), GL\_STATIC\_DRAW);

De notar que *m* representa o modelo em questão.

O mesmo processo é aplicado para os subgrupos de um grupo.

* 1. **Fase 4**

A quarta e última fase prende-se pela aplicação das texturas e cores ao modelos existentes.

Durante a realização do trabalho deparamo-nos com alguns problemas que nos deram algumas dores de cabeça. Um destes, e talvez o pior, foi o facto de que, aparentemente e inexplicavelmente, as luzes e os planetas rodavam consoante a câmara, apesar de a câmara ser colocada antes das luzes e dos planetas. Depois de muito pensar, e sem encontrar uma solução lógica, acabamos por conseguir resolver o problema. Tentamos explicar a seguir o processo.

Começamos por inverter o eixo dos *zz* com a instrução *Scalef(1, 1, -1)* no *renderScene(),* após o *gluLookAt()*, pois achamos que este estava, de alguma maneira, ao contrário. Isto trouxe um outro problema: o eixo dos *xx* inverteu também. Assim, decidimos inverter a coordenada *x* em todos os *translates* e o ângulo de todos os *rotate*s.

Apesar de esta não ser a melhor solução para o problema, foi aquela que nos possibilitou a resolução do mesmo.

* + 1. ***Generator***

A aplicação generator foi alterada por forma a serem geradas as normais e as coordenadas de textura de cada modelo.

Para isso, como as coordenadas de textura são num espaço 2D, ao invés do espaço 3D dos modelos, criou-se uma nova estrutura.

struct p\_2d {

    float x;

    float y;

};

Também foram criadas novas funções, em particular uma função que percorre um vetor de pontos de duas coordenadas (*p\_2d*) e os escreve num ficheiro (*drawPoints2D*); uma função que cria um *p\_2d* quando recebe dois floats (*make\_ponto*).

* + - 1. **Plano**

Para o cálculo das normais, como se trata de um plano, todas elas são iguais. O facto de o mesmo ser desenhado no plano *xOz* facilita o processo, já que a normal, como é perpendicular ao plano, aponta diretamente para cima, paralelamente ao eixo dos *yy*, ou seja, a normal é igual a (0, 1, 0). Os vértices dos triângulos voltados para baixo, têm normal (0, -1, 0), seguindo a mesma lógica descrita anteriormente.

Para o cálculo das coordenadas de textura, o processo é igualmente simples. O plano é constituído por dois triângulos, logo, basta dividir a imagem da textura em dois triângulos iguais aos do plano.

* + - 1. **Caixa**

A caixa revela-se trabalhosa no que toca ao cálculo das coordenadas de textura, mas é simples no que toca ao cálculo das normais. Isto porque as normais de cada face, tal como no caso do plano, são todas iguais. Assim, tomando o exemplo da face superior, as normais são iguais a (0, 1, 0), apontando para cima. Podemos seguir a mesma lógica para todas as faces.

As coordenadas de textura são calculadas por face, sendo que a textura se repete por todas as faces. Vejamos o exemplo de uma face, a superior, sendo que as outras serão análogas. O processo será idêntico ao tomado para o desenho da própria caixa, com a diferença de que os valores iniciais são 0 e 1, em vez de -*x/2* e -*z/2*, para o caso da face superior. Então, acompanhando os mesmo ciclos descritos no ponto 3.1.2, o valor que inicia a 0 é incrementado a cada iteração do ciclo interior em *1/divisões* e restaurado a cada iteração do ciclo exterior. O valor que inicia a 1 é decrementado, a cada iteração do ciclo exterior, em *1/divisões*.

* + - 1. **Esfera**

Para o cálculo das normais, basta normalizar o ponto.

As coordenadas de texturas são calculadas *stack* a *stack*, tal como o desenho da esfera. De realçar que, como se desenha a esfera em espelho, ou seja, desenham-se duas *stacks* de cada vez, uma na parte superior e outra na parte inferior da esfera, existe uma variável *texOffset*, iniciada a 0,5. Usa-se o mesmo método que no desenho da própria esfera, sendo que, tal como acontece com a caixa, os valores estão compreendidos entre 0 e 1.

* + - 1. **Cone**

As normais da base do cone são todas (0, -1, 0), já que a base está virada para baixo.

As normais da parte superior do cone são iguais a (x, y – y\_stack, z), sendo que x, y e z são os valores do próprio ponto e y\_stack é a altura da stack anterior ao ponto.

As coordenadas de textura seguem o mesmo raciocínio da esfera. São calculadas *stack* a *stack*. No entanto, o valor mínimo da coordenada *y* da imagem é iniciado a 0.25, sendo que este é o valor do diâmetro do círculo presente na mesma imagem, que tem a textura da base do cone.

* + 1. ***Engine***

A aplicação *engine* foi alterada por forma a tratar a iluminação e aplicar texturas.

* + - 1. **Cores**

Para além das texturas, um modelo pode ter, associado a ele, componentes de cor difusas, especulares, emissivas e ambientes. Para isso, alteramos o nosso *addGroup* de maneira a fazer a leitura destas componentes, guardando os valores numa transformação associada ao respetivo grupo. Nestas transformações, que são representadas por tuplos de *floats*, o último valor diz respeito ao tipo de componente (0 -> difusa, 1 -> especular, 2 -> emissiva, 3 -> ambiente). No case de as componentes não estarem presentes, definimos como valores *default* para difusa e emissiva (0, 0, 0) e (0.5, 0.5, 0.5) para especular e ambiente.

Estas transformações são posteriormente aplicadas no *drawGroup*. Aqui, guardamos os valores RGB numa lista *cor* e utilizamos *glMaterialfv(GL\_FRONT, \*\*, cor)*, onde *\*\** é substituído por *GL\_DIFFUSE, GL\_SPECULAR, GL\_EMISSION* ou *GL\_AMBIENT* consoante o tipo respetivo.

* + - 1. **Iluminação**

As luzes são lidas do ficheiro XML e só são ativadas se existir, no mesmo, a *tag* *<lights>*. Se a mesma aparecer, é executado o comando *glEnable(GL\_LIGHTING)*. Depois, a cada *tag <light>*, faz-se um *glEnable* e guarda-se a posição, passada como parâmetro no ficheiro, na nova estrutura *luzes*(std::vector<float\*> luzes).

No *renderScene* ativa-se cada uma das luzes, com o seguinte pedaço de código:

glLightfv(GL\_LIGHT0 + i, GL\_POSITION, luzes.at(i));

* + - 1. **Normais e texturas**

Alterou-se a estrutura *model* por forma a que se pudessem tratar as normais e as texturas.

typedef std::tuple<float, float> v\_2d;

struct model {

    int indice;

    int numVertices;

    int texture = 0;

    Vertices vertices;

    Vertices normals;

    std::vector<v\_2d> texCoords;

} Model;

Deste modo, guardam-se as normais de cada modelo, bem como as coordenadas de textura.

Na função *main*, para além do espaço já alocado para os *buffers* dos VBOs, aloca-se também espaço para os *buffers* das normais e das texturas. Os passos são os mesmos, sendo que se repetem para cada uma das categorias: vértices, normais e coordenadas de textura. Assim, repetem-se os seguinte passos (o exemplo é o dos vértices):

1. Aloca-se o espaço para os *buffers*, vertices = (GLuint \*)malloc(numModels \* sizeof(GLuint));
2. Geram-se os *buffers*, glGenBuffers(numModels, vertices);
3. Preenchem-se os VBOs, por cada modelo. Para as coordenadas de textura, só se aloca o espaço de *sizeof(float) \* m.numVertices \* 2*, no segundo parâmetro da função *glBufferData*:

glBindBuffer(GL\_ARRAY\_BUFFER, vertices[m.indice]);

glBufferData(GL\_ARRAY\_BUFFER, sizeof(float) \* m.numVertices \* 3, m.vertices.data(), GL\_STATIC\_DRAW);

1. De cada vez que se desenham os grupos, percorrem-se os modelos e desenham-se os VBOs, recorrendo a todos os *buffers*, vértices, normais e coordenadas de textura.

## Extras

Foram desenvolvidas a *third person camera* e uma *skybox*.

**4.1. *Third Person Camera***

Desenvolveu-se uma *third person camera* simples mas funcional, em que é possível tomar todas as direções (cima, baixo, frente, trás, esquerda e direita) e em que é possível rodar em todas as direções (cima, baixo, esquerda, direita).

Para tal, usam-se as teclas *w*, *a*, *s* e *d* para que a câmera se mova para a frente, esquerda, trás ou direita, respetivamente. Usam-se as teclas *e* e *q* para que a camera se mova para cima e para baixo, respetivamente. Usam-se as setas do teclado para que se gire a camera no sentido indicado pelas mesmas.

A camera é posicionada, no *renderScene*, com recurso à função *gluLookAt(px, py, pz, lx, ly, lz, ux, uy, uz).*

O movimento giratório é providenciado pela função *processSpecialKeys*, que: altera uma variável global *beta* se se premirem as setas para cima e para baixo; altera uma variável global *alfa* se se premirem as setas direita e esquerda. Os valores alterados são incrementados ou decrementados em 0,1. De seguida, recalculam-se os valores de *look at* da camera, da seguinte forma:

lx = dist\*cos(beta)\*sin(alpha);

      ly = dist\*sin(beta);

      lz = dist\*cos(beta)\*cos(alpha);

sendo que *dist* é outra variável global.

Os restantes movimentos são obtidos com recurso à função *processKeys*. Nesta função, quando uma tecla é premida, os valores de *p* e de *l* são recalculados, sendo que a componente *y* só é alterada no caso de movimentos ascendentes ou descendentes e as componentes *x* e *z* só são alterados para os restantes movimentos.

**4.2. *Skybox***

Para a construção da *skybox*, usou-se uma pequena “batota”. Em vez da tradicional *skybox*, acrescentou-se um novo modelo, a partir do ficheiro XML, que cobre toda a cena. Este modelo é uma esfera, com a diferença de que é desenhada voltada para dentro e contém a textura da via láctea. O modelo tem ainda uma componente emissiva, que lhe permite ser visto.

## Conclusões

O desenvolvimento deste pequeno trabalho permitiu a criação de algumas primitivas gráficas, essenciais para o projeto final. Permitiu ainda, aos elementos deste grupo, compreender algumas noções da computação gráfica e desenvolver capacidades na utilização das ferramentas utilizadas para a aprendizagem da mesma.

Por outro lado, os códigos de geração dos modelos foi onde surgiu uma maior dificuldade, sobretudo na construção dos algoritmos, em especial no da esfera. Contudo, após debate de grupo, conseguiu-se finalizar esta tarefa.

A segunda fase exigiu algumas alterações. Nesta fase, todas as decisões tomadas tiveram em conta as necessidades e alterações que poderão surgir em fases futuras. Por isso, foram criadas as estruturas de dados que guardam todas as informações necessárias ao desenho do gráfico 3D. Esta tarefa exigiu ainda muitas tentativas até que ficássemos satisfeitos com a representação do sistema solar. Por isso, foi bastante pedagógica na hora da escrita do ficheiro XML.

A terceira fase revelou-se a mais desafiante até ao momento, não só pelo volume de trabalho necessário como também pela sua dificuldade. Tal como nas fases anteriores, foram necessárias novas estruturas. As novas formas de transformação geométricas, *rotate* e *translate* temporais, bem como, a implementação de curvas de *Catmull-Rom*, permitiram a animação do sistema solar, o que o torna mais interessante.

A quarta fase foi o concluir de um projeto que se estendeu por cerca de três meses. Portanto, todos os erros cometidos até ao momento tiveram impacto nesta fase e tiveram de ser retificados ou contornados. As luzes foram, sem dúvida, a mais exigente das tarefas, não pela sua extensão, mas pelos problemas que apresentou, como explicado acima. Apesar de ter um impacto, no que toca a linhas de código, mínimo, esta tarefa foi, sem dúvida, a que nos tomou mais tempo. Podemos ainda implementar uma *skybox* e mudar o movimento de camera, melhorando o efeito visual do nosso sistema.

Apesar de todos os problemas, o grupo orgulha-se do trabalho que desenvolveu, tendo, ainda assim, a perfeita noção de que o mesmo tem as suas falhas. No entanto, todos estamos confiantes de que poderíamos resolvê-las (e ainda tencionamos fazê-lo) se tivéssemos mais tempo para tal.