Universidade do Minho
Escola de Engenharia

Departamento de Informática
Licenciatura em Engenharia Informática

29 de Maio de 2015

Mini Motor 3D

Trabalho Prático - Fase 4
Computação Gráfica

Trabalho realizado por:

Mariana Imperadeiro Carvalho a67635
Orlando José Gomes Martins Costa a67705
Paulo Ricardo Cunha Correia Araújo a58925
Rui Pedro Azevedo Oliveira a67661

Grupo 33

Índice

Introdução

Neste relatório é apresentado o trabalho desenvolvido para a criação de um mini motor 3D e de um gerador de figuras. Estas aplicações têm como propósito a concepção e apresentação ao utilizador de figuras e cenas em 3 dimensões (teóricamente), sendo independentes uma da outra. Cada aplicação é capaz de receber input do utilizador e responder de acordo com as primitivas implementadas.

Os objetivos da 1ª fase consistem na implementação da capacidade de criação de figuras sem o auxílio dos mecanismos GLUT, mantendo no entanto os parâmetros de funcionalidade que este disponibiliza. Nesse sentido, foram desenvolvidos os algoritmos necessários para gerar os pontos respetivos a cada figura, tendo sido utilizados os mecanismos de criação de sólidos de revolução.

Na 2ªfase deste projeto foi adicionado o suporte para leitura de grupos de figuras declaradas no ficheiro XML e transformações geometricas tais como rotações, translações e escalas. Estas transformações são então interpretadas pelo motor, tendo sido criadas algumas cenas XML como exemplo, nomeadamente um modelo do Sistema Solar, um boneco de neve e um gato.

Na 3º fase deste projeto foram adicionadas novas funcionalidades à aplicação, nomeadamente a geração de figuras recorrendo a superficies de Bézier, a utilização de VBOs com índices, rotações e translações com algoritmos da curva de Catmull-Rom associadas a tempo. Às cenas criadas na segunda fase foram acrescentadas animações, e criou-se também uma nova cena (serviço de chá) recorrendo a superfícies de Bézier.

Finalmente, na 4ºfase o projeto foi concluído com a adição das funcionalidades referentes às texturas e iluminações dos objetos e cenas, sendo estas novamente atualizadas. Durante todo o projeto a arquitetura do sistema sofreu pequenas alterações (adições) graças à solida base criada inicialmente e cumpre todos os requisitos impostos, possuindo ainda diversas funcionalidades extras descritas neste documento.

Arquitetura do Sistema

Este projeto está dividido em duas aplicações com funcionalidades distintas, o **gerador** e o **mini-motor**. O gerador é a aplicação responsável por criar as primitivas, enquanto que o motor é o responsável por interpretar um ficheiro de configuração XML onde estão as primitivas.
A arquitetura de cada uma destas aplicações é similar e irá ser explorada seguidamente.

Gerador

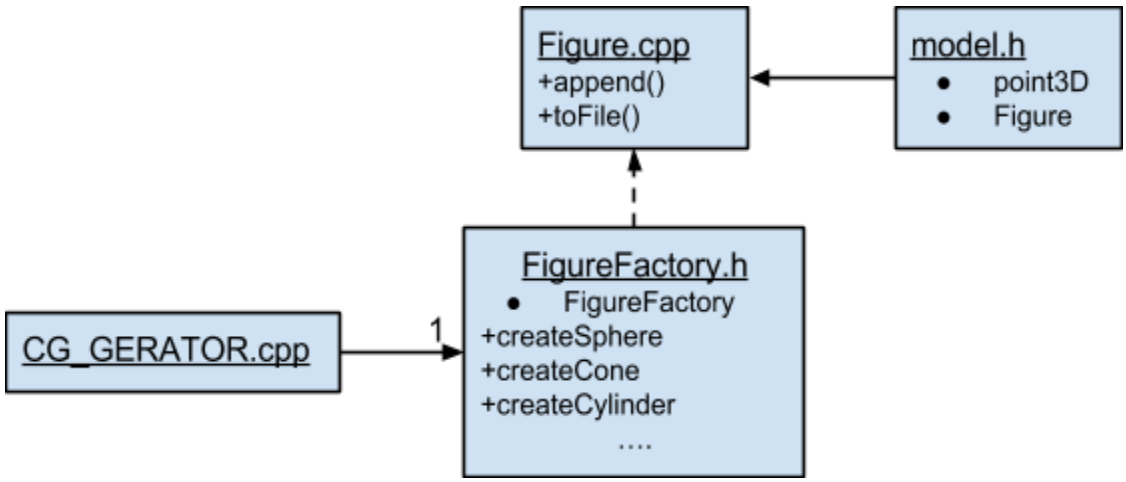


***Figura**: Arquitetura da aplicação Gerador*

A aplicação geradora dos pontos possui uma arquitetura definida em diversos ficheiros, sendo estes:

model.h

Neste header estão declaradas as definições da estrutura que representa um ponto tridimensional e a classe que representa uma Figura constituida por pontos tridimensionais. Os Pontos são constituidos por 3 inteiros (x,y,z), sendo que a Figura é constituida por um "vector" de Pontos que a definem, e um vetor de índices que define a forma pela qual os pontos serão desenhados. Este ficheiro é incluido no ficheiro "Figure.cpp". As estruturas estão declaradas da seguinte forma:

```
typedef struct point3D {
	float x;
	float y;
	float z;
} Point3D;

class Figure {
	vector<Point3D> triangles;
	vector<unsigned int> indices;
	vector<Point3D> normal;
	vector<Point3D> texture;
public:
	void toFile(string  file);
	void toFileVBO(string  file);
	void append(Point3D p);
	unsigned int appendIndice(unsigned int p);
	void appendNormal(Point3D p);
	int appendPoint(Point3D p);
	void appendPointTexture(Point3D p);
	vector<unsigned int> getIndexes();
	vector<Point3D>* getPoints();
	vector<Point3D>* getNormals();
};
```

Figure.cpp

Neste ficheiro estão implementadas as funções correspondentes a adicionar um Ponto3D a uma dada Figure (append), adicionar um índice a uma Figure (appendIndice) e a escrever num dado ficheiro todos os Pontos3D de uma dada Figure, sendo estes mais tarde interpretados pela aplicação Motor. Estão também implementados todas as funções correspondentes à adição das normais e de pontos de textura de uma dada Figure.

FigureFactory.h

Neste header está declarada a classe FigureFactory responsável por "fabricar" as figuras pretendidas. Esta implementação é baseada em Factory Pattern, e aumenta a simplicidade de criação de novas figuras. Assim, na classe FigureFactory estão implementadas as funções capazes de criar os pontos de cada figura. Estas funções estão divididas por diversos ficheiros .cpp (1 figura por ficheiro), aumentando a simplicidade em termos de

capacidade de expansão do projeto. Cada uma das funções do *FigureFactory* retorna uma Figura(“Figure”), à qual pode ser aplicada a função *toFile()* de forma a guardá-la num ficheiro.

CG_GERATOR.cpp

Neste ficheiro reside a maioria da lógica da aplicação Geradora. Consiste num parser de comandos passados pelo utilizador, decompondo a mensagem recibida do stdin e executando a acção pretendida. O parser possui um sistema de controlo de erros, pelo que a inserção de um comando inválido não afetará o estado de execução da aplicação. Neste ficheiro está também presente a declaração do ficheiro onde irão ser armazenados os pontos gerados. Seguidamente apresentaremos um exemplo de utilização da aplicação.

Input de comandos:

As várias figuras poderão ser criadas através da inserção dos diversos comandos na linha de comandos:

- **Esfera**: gerador esfera <raio> <fatias> <camadas> <outputfile>
- **Cubo**: gerador <lado> <outputfile>
- **Cone**: gerador <raiobase> <altura> <fatias> <camadas> <outputfile>
- **Paralelipipedo**: gerador <altura> <comprimento> <largura> <outputfile>
- **Plano**: gerador <comprimento> <largura> <outputfile>
- **Cilindro**: gerador <raiobase> <altura> <fatias> <camadas> <outputfile>
- **Anel:** gerador <raiodentro> <raiofora> <camadas> <fatias><outputfile>
- **Circulo**: gerador <raio> <fatias> <outputfile>
- **Patch:** gerador patch <ficheiro.patch> tesselação <outputfile>
- **PatchZ:** gerador patchZ <ficheiro.patch> tesselação <outputfile>

Exemplos de utilização

Criação de uma esfera de raio 5, com 10 fatias e 15 camadas:

- gerador esfera 5 10 15 output.3d

Criação de uma superfície de bezier utilizando um ficheiro denominado ex.patch com um grau de tesselação 15:

- gerador patch ex.patch 15 output.3d

Motor 3D

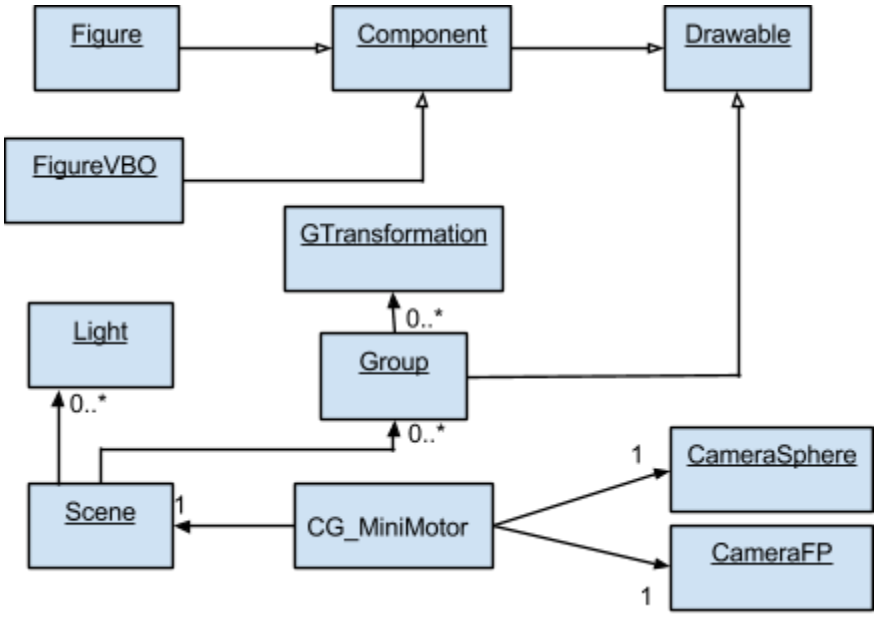


***Figura**: Arquitetura da aplicação Motor.*

O CG_MiniMotor é o módulo onde está a *main* e todas as funções dos eventos relacionados com o GLUT. Quando iniciado, o motor lé o ficheiro XML utilizando o módulo Scene que irá utilizar o Figure e Group para carregar as figuras e os grupos. Após a leitura do XML, todas as figuras são guardadas em memória em Scene e todas as transformações definidas são associadas com Group.
Esta aplicação tem duas cameras usáveis: CameraSphere e CameraFP. Além disso, após o desenho da cena, é possível, através de um menu, alterar o modo como é ilustrada a figura. Clicando com o botão direito do rato em cima da figura, é possível escolher entre GL_POINT, GL_LINE e GL_FILL para modo como é desenhada a figura.

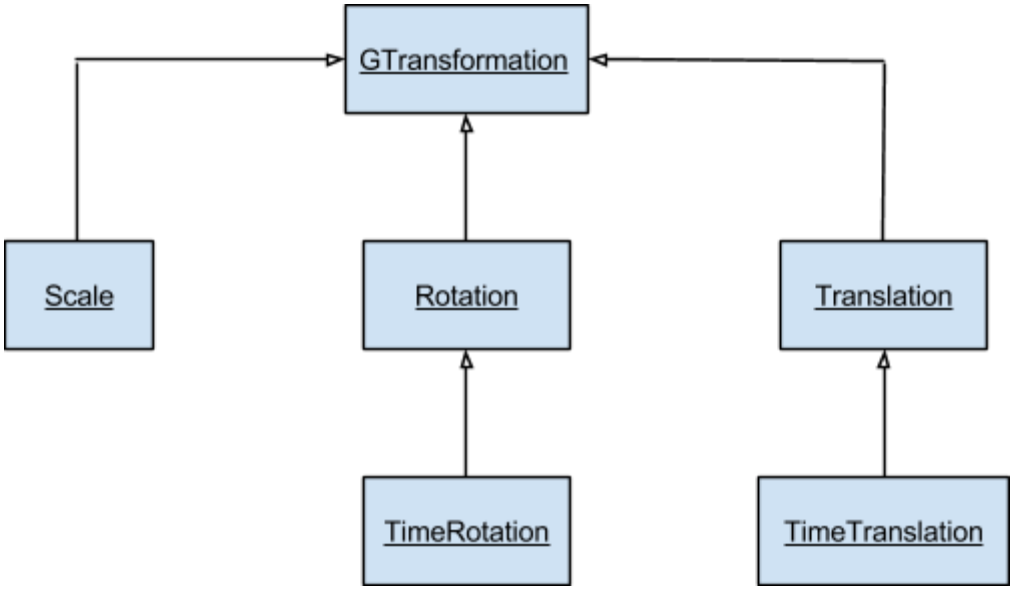


***Figura**: Classes que implementam GTransformation.*

Cada transformação pertencente a um dado grupo pode possuir diversas variantes, correspondentes a modificações na escala de uma figura, assim como rotações e translações efetuadas sobre estas (relacionadas com tempo ou não).

Cameras.h

Neste header estão declaradas as classes CameraSphere e CameraFP. A classe CameraFP é útil na implementação da câmera que vai permitir ao utilizador assumir a posição desta e com o rato, aproximar-se e afastar-se da figura. A classe CameraSphere implementa uma câmera que, com o auxílio do teclado, permite ao utilizador mover-se sempre com a câmera centrada num ponto fixo, podendo aproximar-se e olhar de diferentes ângulos.

CameraFp.cpp

Neste ficheiro estão implementadas todas as funções de movimento, e também uma função que interpreta um determinado caracter do teclado e executa a ação correspondente.
O vetor de deslocamento é calculado atravez através de duas variáveis, o *pitch* e o *yaw*, que representam o ângulo da camara. O valor destas variáveis varia com o movimento do rato.
Existem também três variáveis que representam a posição atual da câmera no universo.
As funções de movimento com mais destaque são a **moveLeft** e **moveRight** as quais calculam o vetor de movimento lateral através da multiplicação dos vectores referentes ao eixo Oy e vetor de deslocamento atual.

CameraSphere.cpp

Este ficheiro implementa funções semelhantes às do ficheiro CameraFp.cpp, com a diferença que esta câmera não interpreta os movimentos do rato.
A forma como calcula a posição actual da câmera é utilizando dois ângulos, *pitch* e *yaw,* e um raio, estando a câmera sempre direccionada para a origem.

Model.cpp

Este ficheiro implementa funções essenciais ao funcionamento do Motor, nomeadamente a função que permite ao motor ler o ficheiro criado anteriormente pelo gerador e guardar todos os pontos em memória, assim como a função que permite desenhar a figura/cena no ecrã. Além destas funções, também implementa a função que faz parse do ficheiro XML.

CG_MiniMotor.cpp

Neste ficheiro reside a maioria da lógica da aplicação Motora. A função main recebe do stdin o nome do ficheiro XML a ler e de seguida invoca a função que trata de ler e fazer parse do ficheiro. Além disso, implementa funções responsáveis pela renderização das figuras. Funções de auxílio, como a interpretação de teclas premidas no teclado e criação do menu também se encontram neste ficheiro. Como já foi referido anteriormente, o menu possui 3 entradas (GL_POINT, GL_LINE e GL_FILL) e está associado ao botão direito do rato, ou seja, quando este for clicado, irá aparecer um pequeno menu na janela. Escolhendo uma das entradas do menu, a função *viewOptions* recebe um identificador da entrada selecionada, interpreta-o e trata de atualizar a figura de acordo com a opção escolhida.

Exemplo de utilização

Iniciado o motor, deve ser inserido o nome do ficheiro XML onde estão definidos os dados da cena a desenhar:

- ficheiro.xml

Leitura XML

De forma a implementar a leitura dos ficheiros XML onde estão definidas as cenas, foi utilizado um parser já existente, o tinyXML. O parser foi implementado na função *parserXML*, declarada no ficheiro Model.cpp. Esta função é chamada recursivamente e recebe como parâmetro um nodo do documento e o grupo actual, sendo que na primeira invocação lhe é passado a cena.
Para cada filho deste nodo ir-se-à identificar qual é a sua tag e efetuar o seu processamento:

- Caso seja a tag **'modelo'**, verifica se o nome do atributo dessa tag é 'ficheiro'. Se tal se verificar, é chamada a função *fromFile()* com o valor do atributo, i.e, o nome do ficheiro, que vai criar uma Figura e adicionar ao grupo actual (recebido por parametro).
  Caso o modelo tenha uma textura, esta será carregada para posteriormente ser desenhada na cena

- Caso seja a tag **'grupo**' irá chamar recursivamente *parserXML,* para continuar o parsing.

- Caso seja a tag **'rotacao'**, **'translacao'** ou **'escala'** irá adicionar a respetiva translação ao grupo actual (ver secção Transformações Geométricas)

- Caso seja a tag **'luzes'**, serão guardadas as luzes declaradas na Scene atual

Durante a interpretação do ficheiro XML é verificada a existência de erros, e caso existam a interpretação do ficheiro pára e a aplicação termina a sua execução.
Foi também criada uma tag extra que facilita a definição da posição inicial da camera na cena a produzir:

- Tag **'Camera'** permite controlar a posição inicial da camera.

De forma a activar a funcionalidade de iluminação e texturização, assim como permitir a especificação de cores e texturas, foi necessária a utilização de novos atributos na tag 'modelo':

- '**Textura**', que permite a aplicação de uma textura num determinado modelo;
- '**emitR**', '**emitG**', '**emitB**', '**emitA**', que permitem criar iluminação a partir de um determinado modelo;

Exemplos de ficheiros XML

Exemplo 1

```
<?xml version="1.0" encoding="UTF-8" ?>
<xml>
   <cena>
      <modelo ficheiro='esfera.3d'/>
   </cena>
</xml>
```

Exemplo 2

```
<?xml version="1.0" encoding="UTF-8" ?>
<xml>
   <cena>
         <camera X='-70' Y='45' Z='3'/>
         <grupo>
                  <translacao Y='10'/>
                  <escala Y='0.5'/>
                  <modelo ficheiro='exemplos/cone.3d' />
         </grupo>
         <grupo>
                  <translacao X='-1'/>
                  <rotacao angulo='90' eixoX='1' eixoY='0' eixoZ='1' />
                  <modelo ficheiro='exemplos/cubo.3d' />
         </grupo>
   </cena>
</xml>
```

Exemplo 3

```
<?xml version="1.0" encoding="UTF-8" ?>
<xml>
   <cena>
   <luzes>
     <luz tipo="POINT" posX="0" posY="0" posZ="0"/>
   </luzes>
   <camera X='-10' Y='5.5' Z='0.3'/>
```

```
    <grupo>
        <escala X='0.1' Y='0.1' Z='0.1'/>
        <grupo>
          <escala X='13.92' Y='13.92' Z='13.92'/>
          <rotacao angulo='360' eixoX='0' eixoY='1' eixoZ='0' tempo='70'/>
          <modelo ficheiro='solarSystem/planeta.3d' emitR='1' emitG='1' emitB='1' emitA='1'
textura='ss_textures/texture_sun.jpg'/>
        </grupo>
    </grupo>
    </cena>
</xml>
```

Figuras

Seguidamente são exploradas as figuras implementadas, a lógica por trás da criação dos seus pontos, e uma breve explicação sobre como as funções associadas interagem umas com as outras.

Área Plana

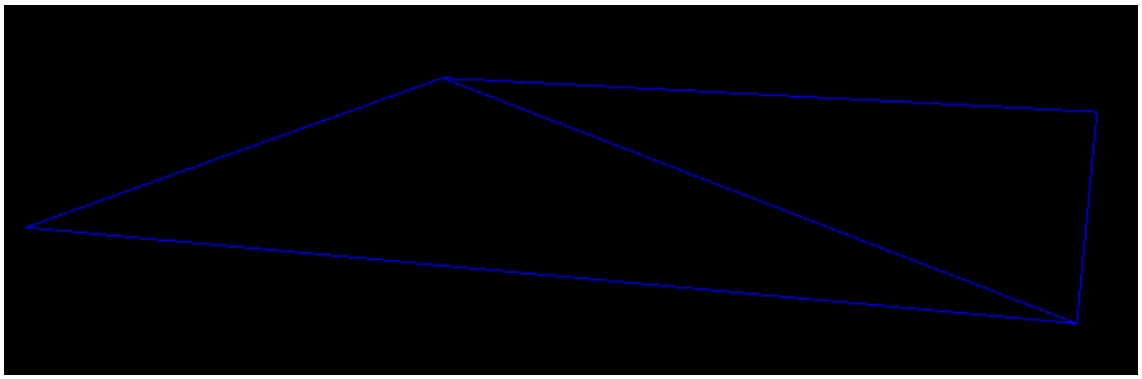


***Figura**: Exemplo de área plana*

```
Figure FigureFactory::createPlane (float x, float z);
void FigureFactory::createPlaneAux (Figure* f, float p[3], int axisFirst, int axisSecond);
```

A criação dos pontos da área plana é realizada por duas funções. A função “createPlane” recebe como argumentos o comprimento e largura do plano a criar, e invoca a função “createPlaneAux” com os argumentos necessários dependendo da situação. A primeira função corresponde ao desenho mais geral de uma área plana, neste caso com o centro na origem do plano XoZ, enquanto que a segunda função permite especificar algumas particularidades no desenho, nomeadamente a direcção e orientação da área. Esta funcionalidade foi implementada com o objetivo de o desenho da área plana poder ser utilizado noutras figuras geométricas, nomeadamente o paralelipípedo.

Paralelipípedo

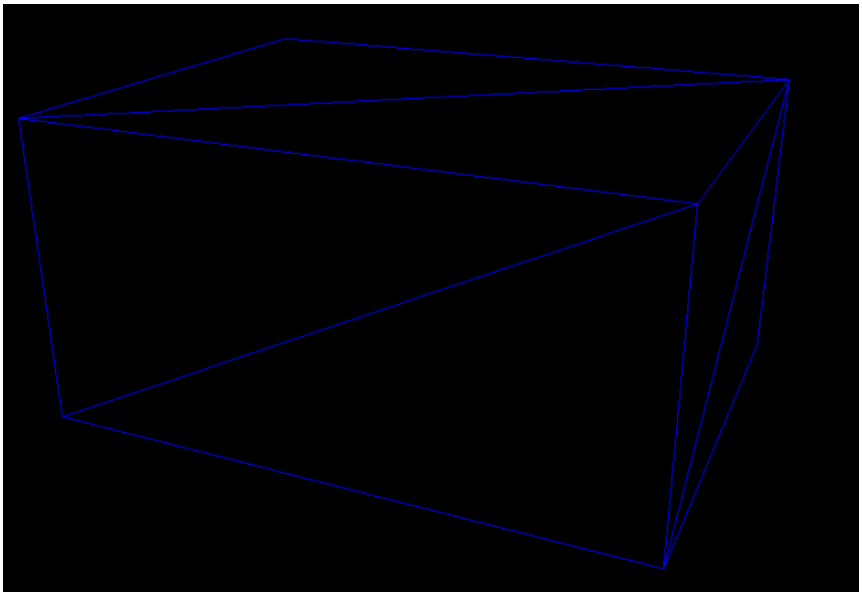


***Figura**: Exemplo de paralelipípedo*

Figure **FigureFactory::createParallelepiped** (float x, float y, float z);

O paralelipípedo é formado por 6 planos criados com a ajuda da função “createPlaneAux” referida anteriormente. A função “createParallelepiped” recebe como argumento as diversas medidas da figura (comprimento, largura e altura), e através de 6 chamadas (6 planos) à função “createPlaneAux”, cria os pontos necessários à sua representação.

Circulo

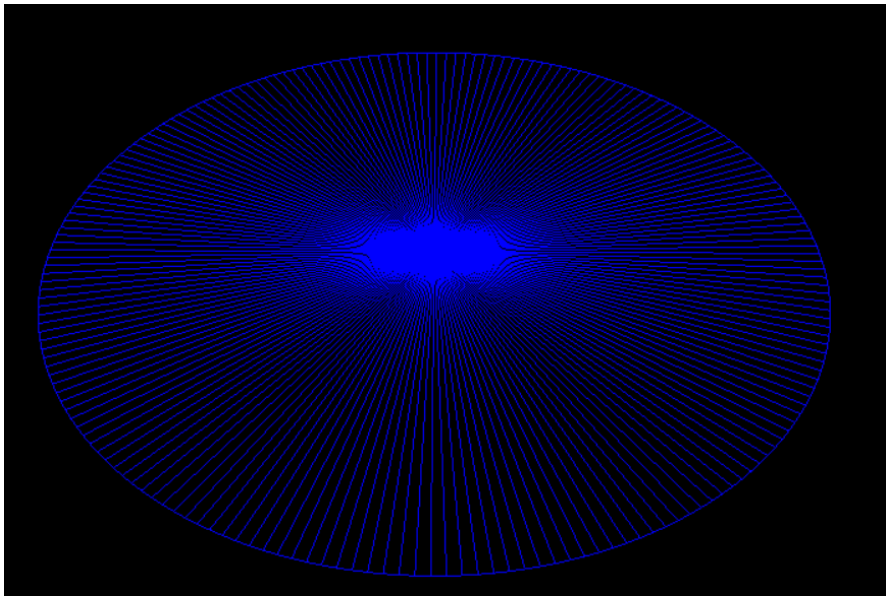


***Figura**: Exemplo de circulo*

```
Figure FigureFactory::createCircle (float raio, int fatias);
void FigureFactory::createCircleAux (Figure* f,float raio, int fatias, float altura, int orient);
```

A criação dos pontos de um círculo é realizada por duas funções. A função “createCircle” recebe como parâmetros o raio e as fatias do circulo a criar, e corresponde ao desenho mais geral de um círculo (basicamente efetua uma chamada à função “createCircleAux” com os argumentos apropriados de forma a criar um círculo com o centro na origem do plano cartesiano (0,0,0)), enquanto que a segunda função (“createCircleAux”) recebe como parâmetros não apenas o raio e as fatias mas também a altura e orientação do círculo, o que permite especificar algumas particularidades no desenho, nomeadamente a altura a que esse circulo está relativamente ao plano XoZ, e para onde este está orientado (cima ou baixo). Esta funcionalidade foi implementada com o objetivo de o desenho de um círculo poder ser utilizado em outras figuras geométricas, nomeadamente o cone (base) e o cilíndro (base e topo).
Para especificar os pontos na circunferência foram utilizadas coordenadas polares, sendo estas calculadas num ciclo com o um número de iterações igual à quantidade de fatias especificada.

Sólidos de revolução

As seguintes figuras, ao contrário das anteriores, foram implementadas seguindo a metodologia de sólidos de revolução. Estes são sólidos gerados pela revolução (rotação) de uma figura plana ao redor de um eixo, por exemplo:

- esfera: rotação de um círculo.
- cone : rotação de um triângulo.
- cilindro: rotação de um retângulo.
- anel: rotação de um círculo em torno de um ponto a uma distância fixa.

Esfera

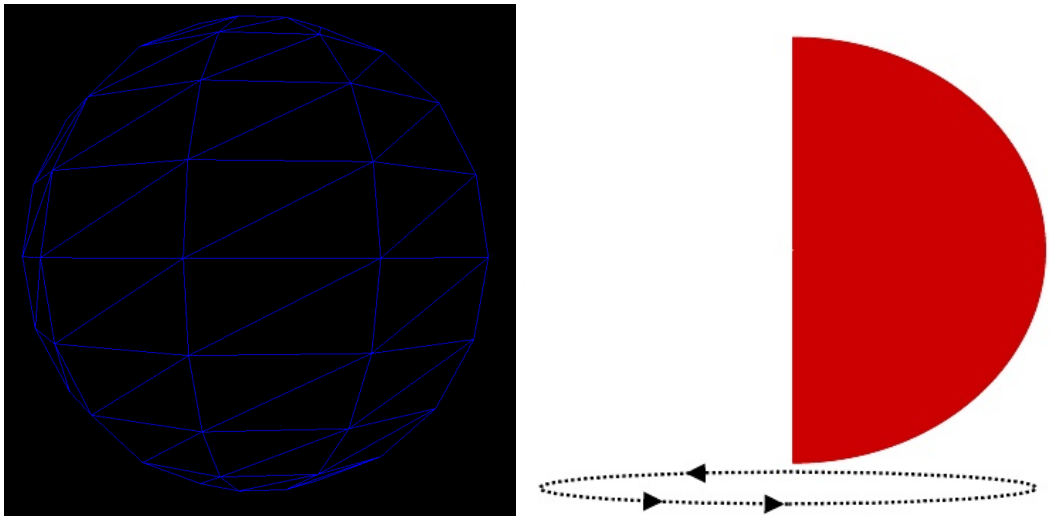


***Figura**: Exemplo de esfera*

```
Figure FigureFactory::createSphere (float raio, int camadas, int fatias);
void FigureFactory::createRotate(Figure* f, Point3D points[], int camadas, int fatias);
```

Utilizando a revolução como método de desenho torna-se bastante simples desenhar uma esfera, para isso bastando passar um conjunto de pontos que formam uma meia lua.
A criação da meia lua num plano de 2 dimensões é feita utilizando um ciclo que para cada camada cria o ponto respetivo utilizando trignometria. A função “createSphere” recebe como argumentos o raio da esfera, e o número de camadas e fatias que esta vai ter, responsáveis pelo número de pontos(i.e. da complexidade) que irá constituir a esfera.

Cone

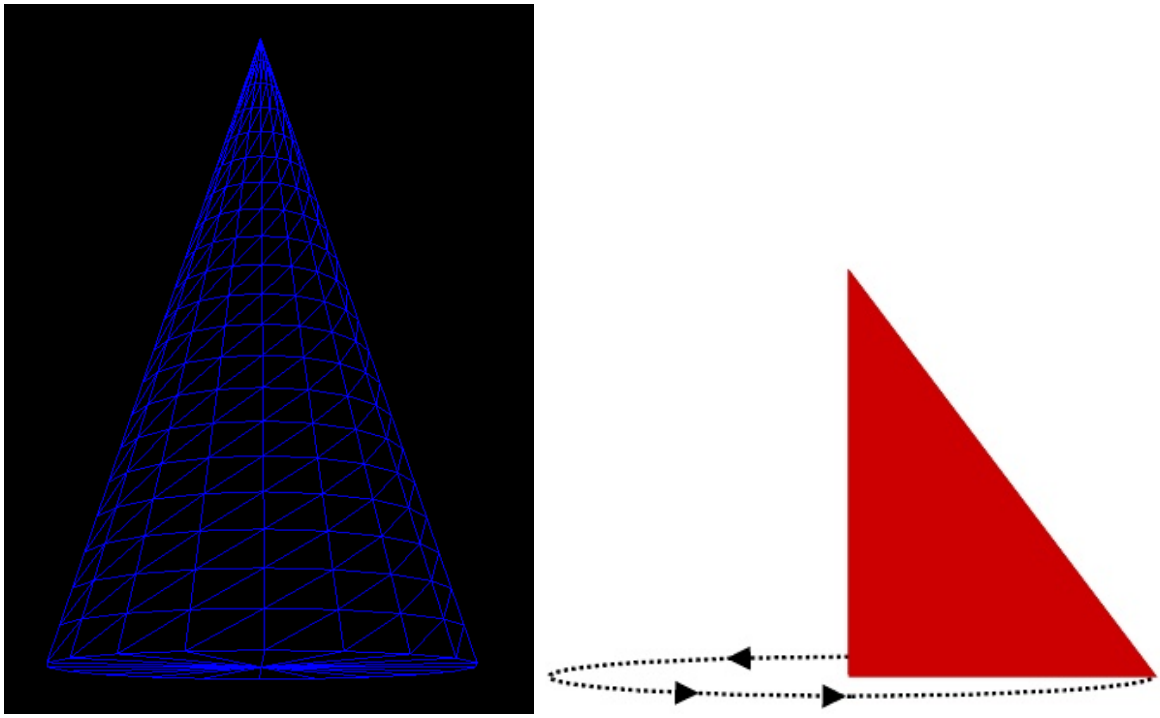


***Figura**: Exemplo de cone*

Figure **FigureFactory::createCone** (float raiob, float altura, int fatias, int camadas);
void **FigureFactory::createRotate**(Figure* f, Point3D points[], int camadas, int fatias);

O cone é feito através da revolução de um conjunto de pontos que formam meio triângulo. A função recebe como parâmetros o raio da base, a altura do cone, e a quantidade de fatias e camadas que irão definir o nível de complexidade do desenho da figura. Para isso começa-se por desenhar o ponto de cima, e mediante o número de camadas vai-se decrementado o valor em Y e aumentando o valor em X até que chega à base com o valor em X igual ao raio. Para formar a base é adicionado um ponto no seu centro, por forma a formar a reta que irá, através da revolução dos seus pontos, criar a circunferência necessária.

Cilindro

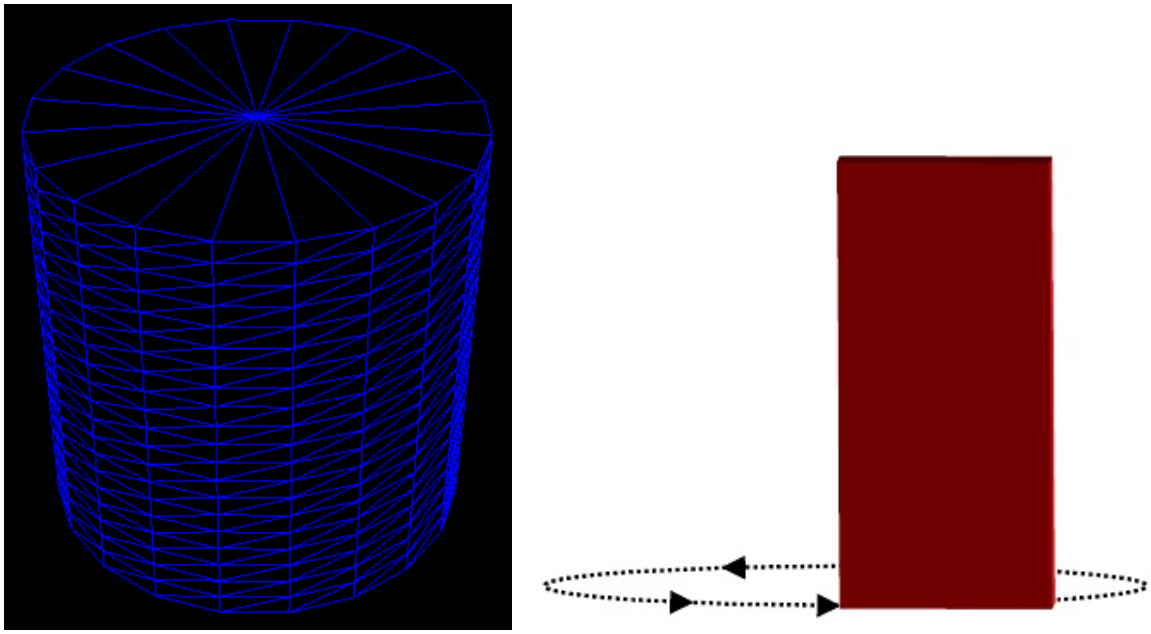


***Figura**: Exemplo de cilindro*

Figure **FigureFactory::createCylinder**(float raiob, float altura, int fatias, int camadas);
void **FigureFactory::createRotate**(Figure* f, Point3D points[], int camadas, int fatias);

O cilindro é formado por um conjunto de pontos que formam um retangulo, onde as camadas são o número de pontos na reta lateral. A função recebe como parâmetros o raio da base/topo, a altura, e as fatias e camadas, que irão definir a quantidade de pontos (i.e. complexidade) da figura.

Anel

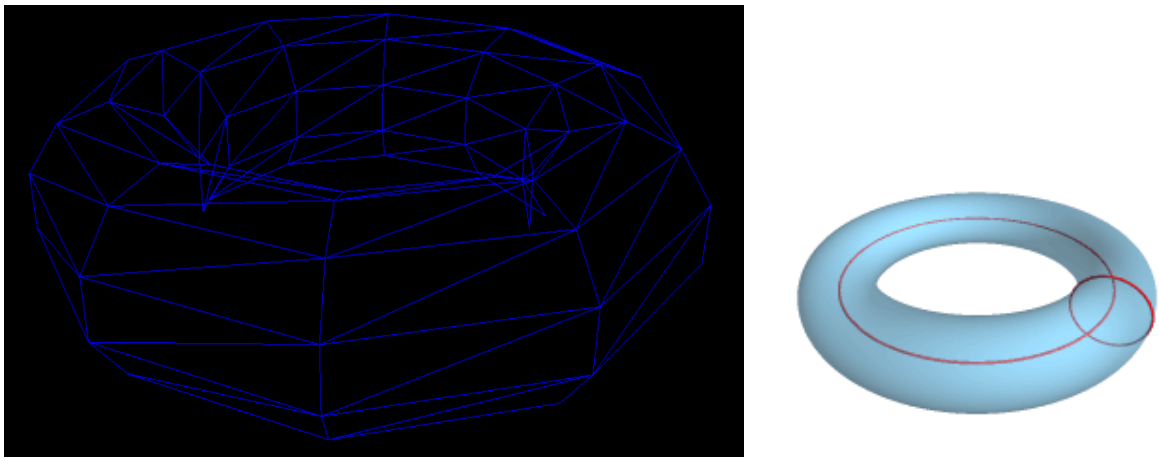


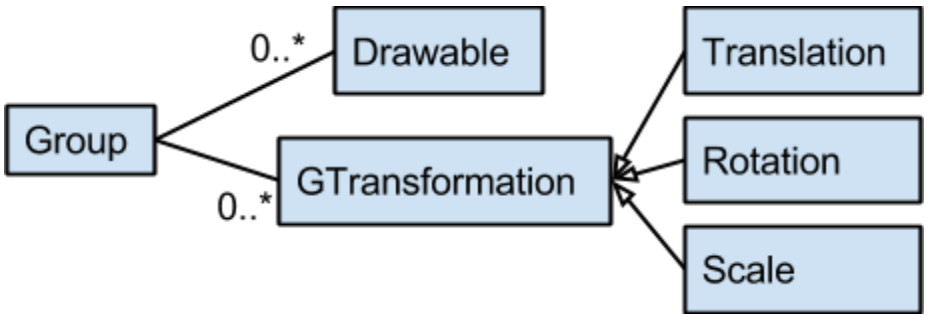
***Figura**: Exemplo de anel*

Figure **FigureFactory::createTunnel**(float insideRadius, float outsideRadius, int nsides, int rings);
void **FigureFactory::createRotate**(Figure* f, Point3D points[], int camadas, int fatias);

O Anel é feito a partir da revolução de um círculo trasladado para o lado com valor do raio interno. Ou seja criam-se os pontos correspondentes a uma circunferência utilizando trigonometria e para a circunferência ficar no sitio correto, soma-se em X o valor do raio interno mais o valor do raio da circunferência que está a ser desenhada.

Transformações geométricas

As transformações geométricas suportadas pelo motor 3D são: Translação, Rotação e Escala. Cada uma destas transformações deve ser executada pela ordem na qual se encontram definidas no ficheiro de cena XML, e deve estar ativa apenas no grupo e subgrupos onde está inserida.
Estas transformações foram implementadas numa classe Group, na qual para além dos elementos do tipo Drawable implementados na 1ºfase, possui agora também um vetor com todas as transformações a serem executadas pela ordem definida no ficheiro XML referido.



A classe que representa a transformação denomina-se **GTransformation**. Esta classe é herdada por cada uma das implementações das transformações geométricas: Translation, Rotation e Scale (e respetivas subclasses - TimeTranslation e TimeRotation).
Cada classe implementa a função **doTransformation**, sendo que esta executa a respetiva primitiva do openGL, utilizando os parâmetros que foram inicializados durante a leitura do XML.

Translation

Para criar um objeto do tipo Translation é necessário um vetor que indica a translação a realizar. Este vetor é constituido pelos parâmetros definidos no ficheiro XML(float x, float y, float z). A primitiva usada no método doAction é glTranslatef.

Rotation

Para criar um objeto do tipo Rotation é necessário um vetor e um ângulo referente à rotação. Este vetor é constituido pelos parâmetros definidos no ficheiro XML(float x, float y, float z) e o ângulo de rotação consiste num float angle. A primitiva usada no método doAction é glRotatef.

Scale

Para criar um objeto do tipo Scale é necessário uma constante para cada eixo com a escala a ser usada que afetará a forma/tamanho dos elementos envolvidos no grupo(float x, float y, float z). A primitiva usada no método doAction é glScalef.

Execução das transformações

Como referido anteriormente, a figura e o grupo implementam a Interface Drawable, que os obriga a implementar o método Draw(). É de notar também que um grupo tem um vetor de Drawables e um vetor de GTransformations.
Com estes dois pontos em mente, para desenhar um grupo começa-se por salvar a matriz do estado do openGL(pushMatrix).
Seguidamente executam-se todas as transformações de desenho, e consoante a definição, irá ser executada uma translação, rotação ou escala. Estas são executadas pela ordem que foram definidas no XML.
Após serem efetuadas as transofmações necessárias na cena, são desenhados os objetos Drawable, que podem tratar-se de Figuras ou (sub) Grupos., sendo esta alternativa indiferente dado que ambos serão afetados pelas transformações efetuadas anteriormente.
Finalmente é retomado o estado da matriz de estado inicial do openGL(popMatrix).

Curvas de Catmull

As translações de objetos podem também ser definidas através da declaração de pontos que servirão para calcular o trajeto dos objetos, utilizando para o efeito os algoritmos de cálculo de curvas Catmull-Rom. Este necessita de pelo menos quatro pontos para o fazer, e permite a obtenção dos pontos da trajectória pretendida.

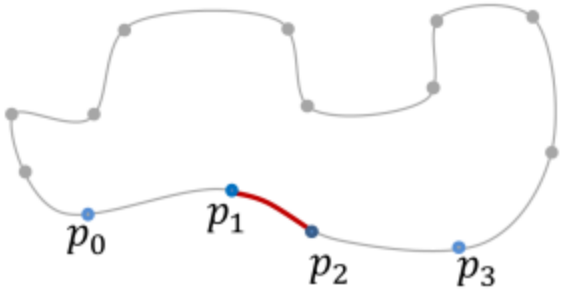


Figura: *Exemplo de cálculo de curvas através de Catmull-Rom.*

A função responsável por efetuar o movimento dos objetos seguindo os pontos definidos pela curva de Catmull-Rom está implementada na classe TimeTranslation e denomina-se TimeTranslation :: doTransformation, e com o auxílio de funções como TimeTranslation :: calculateTransformation (calcula os 4 pontos a utilizar em cada iteração do cálculo da curva, e devolve a componente P(t) da matriz de transformação ou a componente P'(t), consoante o valor passado no parâmetro bool definido) é possível cálcular a matriz de transformação a usar com a primitiva glMultMatrixf(matrix). Outras funções usadas para este cálculo denominam-se normalizeVector e crossProduct (denominadas de acordo com a sua utilidade), e auxiliam no cálculo das componentes r(t) e up(t).

As translações implementadas nas diversas cenas estão atualizadas de forma a efetuar o cálculo através dos pontos declarados no ficheiro XML.

Rotação com movimento

A rotação com movimento herda da rotação normal e tem o atributo "tempo total", referente ao tempo que a rotação deve demorar (implementada na classe TimeRotation). Quando o método doGtranslaction é executado, este calcula o atual ângulo de rotação utilizando o tempo atual que é dado pelo GLUT (glutGet(GLUT_ELAPSED_TIME)).

Excerto do código de cálculo:

```
float elapsedNow = glutGet(GLUT_ELAPSED_TIME);
float deltaTime = elapsedNow - this->elapseBefore;
float anglePerMili = 360 / (this->time * 1000);
this->angle = (deltaTime * anglePerMili + this->angle);

while(this->angle > 360){
    this->angle -= 360;
}

glRotatef(angle, p.x, p.y, p.z);
elapseBefore = elapsedNow;
```

A utilização de VBO's, relativamente ao modo imediato, aumenta grandemente a performance do motor3D, como é possível verificar, sendo as FPS significativamente superiores. O facto de anteriormente terem sido implementadas figuras de revolução facilita a tarefa de implementação de VBO's através de indíces, dado que apenas é necessário alterar a função responsável pela revolução dos pontos propostos.

Superfícies de Bezier

Na renderização do bule de chá utilizando patches de Bézier, foi criada a função FigureFactory::createBezierSurface (na aplicação geradora), que recebe como argumentos o ficheiro .patch a processar, e uma Figure f onde serão guardados os índices e os triângulos a serem desenhados.
A função createBezierSurface lê os índices e os pontos do ficheiro .patch e guarda-os numa Figure (que contem um vetor de pontos e um vetor de índices, como referido na implementação de VBO's).
Processados todos os pontos e índices do ficheiro, é chamada a função definePatches. Para cada 16 índices do patch, vai guardar os respetivos pontos de controlo numa matriz/array [4][4]. Cada ponto na superfície de Bézier é calculado recorrendo a funções auxiliares que aplicam a fórmula para o cálculo de superfícies de Bézier

$$P(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} p_{ij} * m_i(u) * m_j(v),$$

,

na qual u e v são valores no intervalo [0,1] e calculados a partir da tesselação recebida.
Tomando cada linha da matriz dos pontos de controlo como uma curva de Bézier, um ponto na superfície Bézier é calculado a partir da soma dos 4 pontos de controlo de cada linha, devidamente ponderados pelos polinómios de Bernstein, sendo estes:

$$m_0(t) = (1-t)^3$$
$$m_1(t) = 3t(1-t)^3$$
$$m_2(t) = 3t^2(1-t)^2$$
$$m_3(t) = t^3$$

Patch a patch, os pontos da superfície gerados vão sendo guardados numa figura, sendo depois gerados os índices que definem como os triângulos serão desenhados. Estes pontos e índices são então armazenados num ficheiro a ser interpretado pela aplicação motora.

Implementação de Vertex Buffer Objects

De forma a adaptar o desenho de figuras através do uso de VBO's, torna-se necessário efetuar alterações nas aplicações Gerador e Mini-Motor.
No âmbito da aplicação geradora, o objeto Figura é constituido agora por 2 vectores, servindo estes para armazenar os pontos da figura e para armazenar os índices utilizados no desenho.
Na aplicação motora, está definido um novo objecto denominado FigureVBO, mantendo-se no entanto a implementação da classe Figure responsável pelo desenho em modo imediato.

Ficheiro 3d

Os ficheiros .3d estão restruturados de forma a albergar os índices do VBO. Neste sentido, possuem agora uma lista com todos os pontos e índices referentes aos triângulos da figura em causa. A função Figure::fromFile, no caso de se tratar da leitura de um ficheiro referente a desenho sem o uso de VBO's, converte a lista de índices num vetor de pontos para desenho imediato (2ºetapa), de forma a poder ser comparada a performance a nivel de Fps.
A função FigureVBO::fromFile guarda estes índices num vetor, e coloca na memória da placa gráfica os pontos necessários ao desenho da figura.

Gerador

A implementação do desenho das figuras utilizando os índices, obriga a modificar a função de criação dos pontos de figuras de revolução. Esta encontra-se agora dividida, de forma a separar a adição dos pontos da formação dos triângulos. Assim, os triângulo são agora definidos a partir dos índices. Para figuras formadas através da revolução, apenas foi necessário alterar a função de revolução.

Minimotor

De forma a poder comparar as FPS do modo imediato com o modo de VBO, existe no menu da janela uma opção que permite modificar uma flag que em caso positiva, na leitura do XML utiliza o Objeto FigureVBO e caso negativa, utiliza a Figure.
Como qualquer um dos dois implementa Drawable, ambas tem um método draw, que no caso da Figura permanece igual à 2º etapa, mas no caso da FigureVBO utiliza as funções de desenho dos VBO.

Cenas implementadas

Boneco de neve

O boneco de neve é composto por 3 esferas que constituem o tronco, 2 braços, 2 olhos, 1 boca, 1 nariz e 1 chapéu.
A base do boneco de neve é desenhada como uma esfera. De seguida, através de uma translação vertical (eixo Y) e a utilização de uma escala, define-se a esfera que constitui o "peito" do boneco. A partir da zona que foi modelado o "peito", foram criados os braços. Através de cilindros, com a utilização de translações simétricas no eixo do X e rotações de 45º e 135º no braço esquerdo e direito, respetivamente.
A cabeça é constituida por uma esfera, tendo esta sofrido uma translação vertical similar à esfera do peito. Ainda neste grupo, são criados os seus vários constituintes como a boca, os olhos, o nariz e a base do chapéu.
A modelação dos olhos é feita recorrendo a esferas, utilizando uma escala muito menor e translações com componentes em todos os eixos de forma a colocar os olhos na parte frontal da cabeça. O nariz é modelado através de um cone rodado 90º no eixo do X e de uma translação no eixo do Z de forma a colocá-lo na face do boneco. Através da utilização de um anel, foi modelada a boca do boneco. De forma a desenha-la no sítio correto, é necessário o uso de uma translação nos eixos Y e Z, assim como uma rotação de forma a alinhar com a zona inferior da face. A base do chapéu é modelada através de um cilindro. É necessária uma escala de forma a diminuir a sua altura e a aumentar o seu raio. Além disso, é necessária uma translação vertical de forma a colocar o cilindro acima da cabeça do boneco. A partir da base do chapéu, foi modelada a parte superior do chapéu através de outro cilindro. Para isso, apenas é necessária uma translação vertical para colocar o cilindro no topo do anteriormente criado.

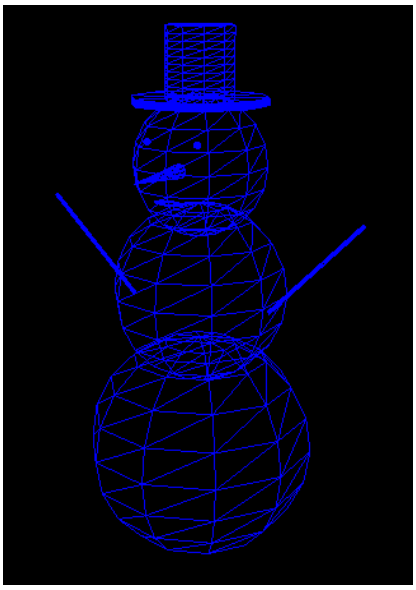


Figura: *Cena do boneco de neve*

Gato

O gato criado é composto por um cilindro que constitui o corpo, quatro pernas sob a forma de cilindros, uma cauda, uma cabeça, um pescoço, duas orelhas, nariz, seis bigodes e dois olhos.
Inicialmente foi modelado o corpo como um cilindro que sofreu uma rotação de 90º e que serviu de ponto de referência para a modelação dos restantes elementos. A partir do corpo, foram criadas as quatro pernas. Cada perna sofreu uma rotação de 90º, de forma a que o cilindro se mantivesse na vertical, e uma translação, que varia de acordo com a posição da perna. Em seguida foi criado o pescoço recorrendo também a um cilindro, sendo este posicionado numa das bases do corpo através de uma translação no eixo de X e do Y.
A seguir ao pescoço foi modelada a cabeça, estando esta sob a forma de uma esfera, e à qual foi aplicada uma escala para ficar mais achatada. Depois de posicionada a cabeça criaram-se os olhos, conseguidos com a modelação de dois tórus aos quais foi aplicada uma escala de modo a ficarem ligeiramente ovais, e assim mais semelhantes aos olhos de um gato. Para simular a íris dos olhos foram usadas esferas, também estas sob uma escala que as tornou mais ovais e comprimidas. Na superfície da cabeça, um pouco abaixo do centro, foi colocado o nariz, com a modelação de um cone virado do avesso e ligeiramente achatado. O gato tem apenas três bigodes de cada lado do nariz, que foram feitos recorrendo a cilindros escalados de modo a ficarem mais finos.
As orelhas do gato, no topo da cabeça, são formadas por cones. A cauda é a junção de dois cones, sobrepostos, para simular uma cauda mais felpuda. Um dos cones é mais comprido e fino, enquanto que o outro é mais largo e curto.

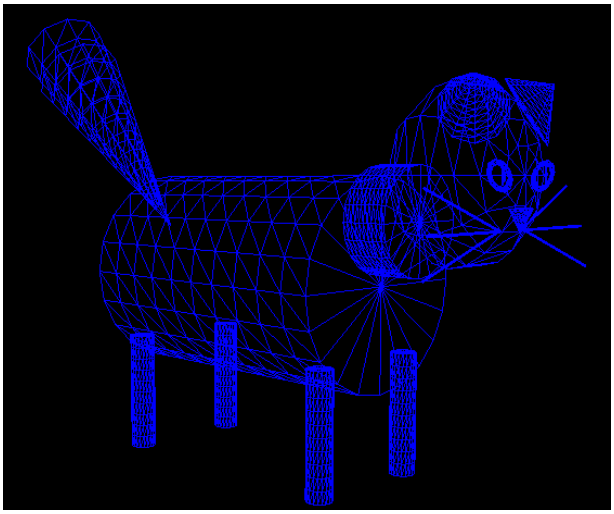


Figura: Cena do gato

Bule de Chá

Na figura é possível observar o bule de chá gerado através de superfícies de Bezier, a ser implementado na cena correspondente ao Sistema Solar, tomando o papel de asteróide.

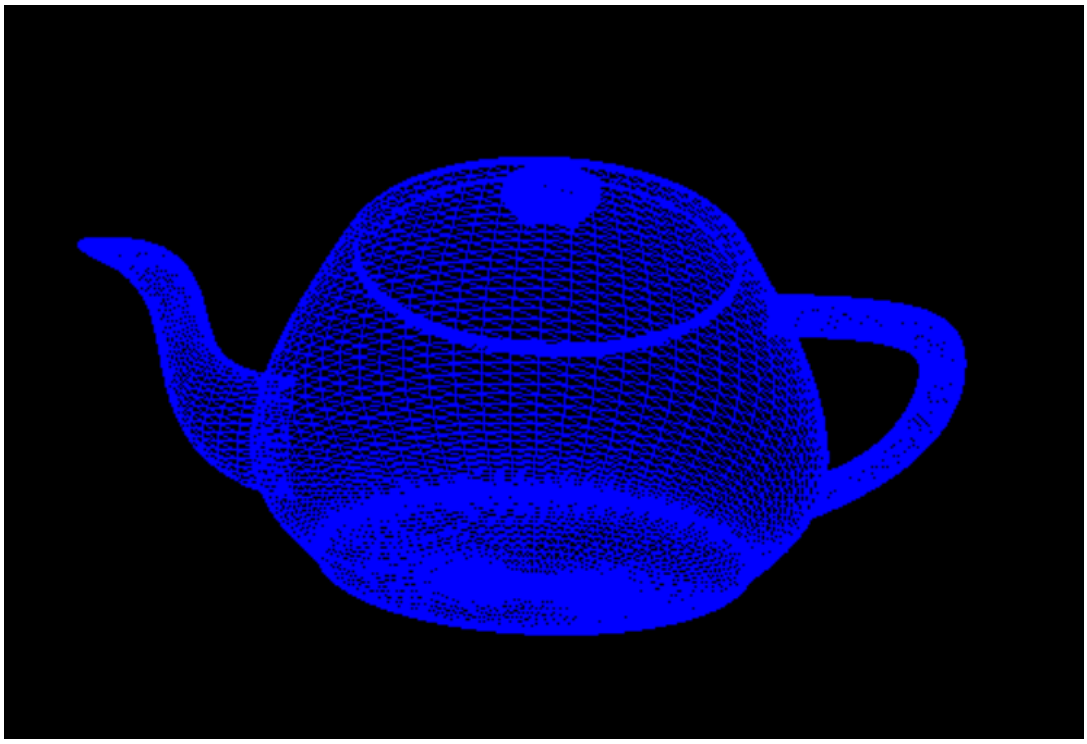


Figura: *Bule de chá gerado através de superfícies de Bezier*

Hora do Chá

Recorrendo ao dataset original do serviço de chá completo, criado por Martin Newell em 1975 (pode ser encontrado em: ftp://ftp.funet.fi/pub/sci/graphics/packages/objects/teaset.tar.Z), foi criada uma cena em que com superfícies de bézier foram renderizados o bule, 4 chávenas e 4 colheres, dispostos na cena através de diversas rotações e translações.

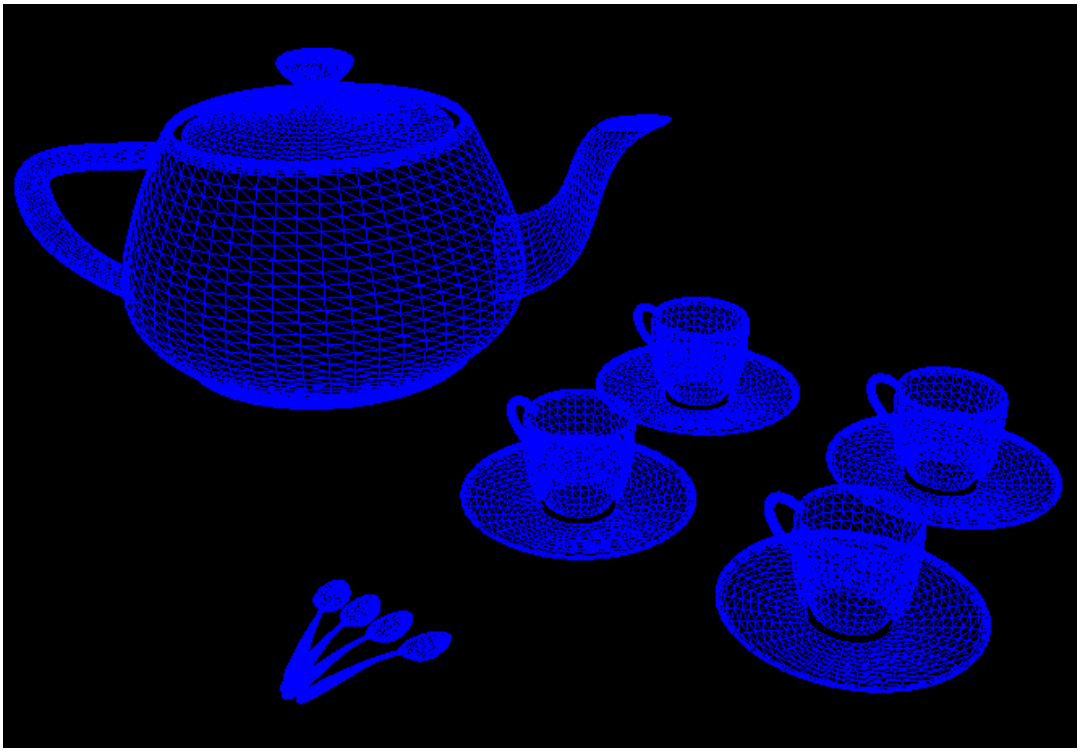


Figura: Serviço de chá gerado através de superfícies de Bézier

Sistema solar

De forma a facilitar a visualização, as distâncias entre os planetas não foram implementadas de acordo com uma escala linear/logaritmica, mas sim com os "planetas" relativamente cerca uns dos outros, não o suficiente para causar colisões quando futuramente forem implementados os movimentos de translação e rotação correspondentes a cada um.
Foram desenvolvidas assim duas versões do problema, tendo sido implementadas duas versões diferentes para calcular o tamanho dos planetas (escala linear e escala logarítmica).
Assim, na escala linear, cada planeta pertence a um grupo ao qual é aplicado uma escala relativamente ao tamanho do "Sol", numa medida de 10000 -> 1, e uma translação relativamente ao mesmo. Cada "planeta" consiste numa esfera, e os "planetas" que possuem anéis à sua volta (p.e. Saturno) encontram-se num grupo que possui um subgrupo referente a esses anéis, os quais são implementados através de um tórus ao qual é aplicado uma escala por forma a serem visualizados como estando "espalmados", proporcionando um efeito mais afeto à realidade. Foi também implementado o satélite natural da Terra (Lua), como sendo um subgrupo deste planeta ao qual é aplicada uma escala relativamente a este, assim como uma translação no eixo dos Z (de forma a representar de forma distinta aos planetas).

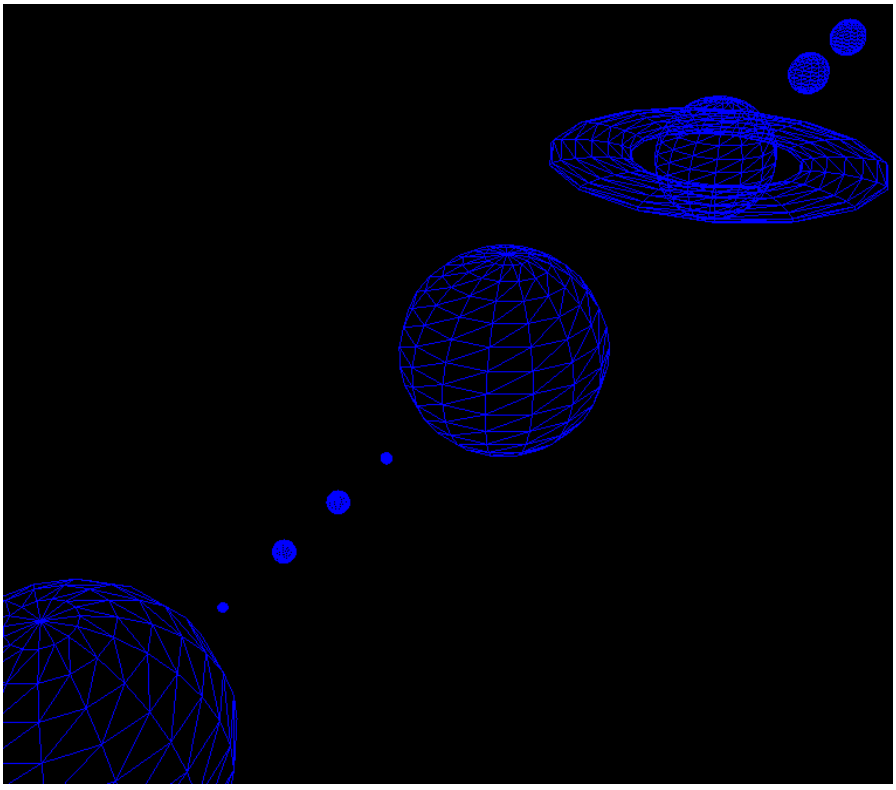


Figura: *Cena de sistema solar com escala linear*

Nota: Na imagem apresentada, a esfera referente ao “Sol” foi também reduzida relativamente ao resto da cena (100000 -> 1).

Relativamente à escala logarítmica, esta torna mais homogénea a diferença entre os tamanhos dos planetas, podendo visualizar-se mais fácilmente a Lua. Esta escala é obtida através do logarítmo de base 10 do tamanho real em kilómetros dos planetas.

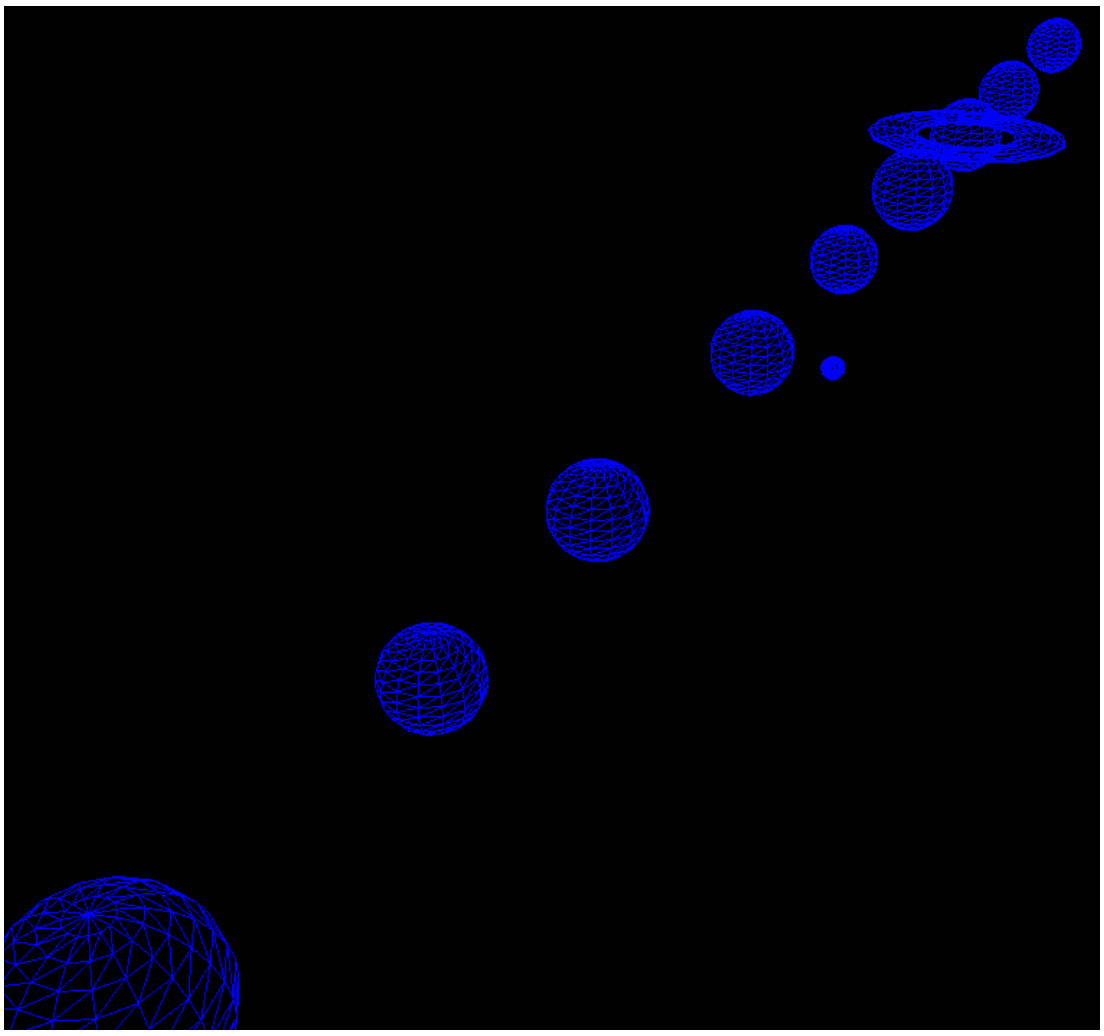


Figura: *Cena de sistema solar com escala logarítmica*

Movimento das cenas

Para “dar movimento” ao sistema solar criado, tornando-o num sistema solar animado, foi necessário um conjunto de alterações ao ficheiro XML que definia a cena. Foram calculados, para todos os planetas do sistema solar, um conjunto de oito pontos pertencentes às suas órbitas, mantendo as distâncias do sistema solar criado anteriormente.
Para calcular os tempos de translação dos planetas, optou-se por determinar um tempo de translação de 10 segundos ao planeta com o menor tempo de translação real, sendo este Mercúrio. Proporcionalmente, foram calculados todos os tempos de translação do resto dos planetas, assim como dos seus satélites naturais. Quanto aos tempos de rotação dos planetas, foi aplicado o mesmo tipo de cálculos, com a excepção dos planetas cuja rotação sobre si próprios é muito mais rápida que o resto. Nesses casos, foi aplicado um tempo de rotação de 3 segundos.
Através dos pontos e tempos de translação calculados, foi possível atribuir a cada elemento da cena o seu respetivo movimento.
Dado que é requerido que o sistema solar possua um cometa animado em forma de bule de chá, foram calculados 4 pontos de translação relativos a esse elemento, seguindo como referência o movimento de translação do cometa “Halley”. Quanto ao seu tamanho, foi escolhido um tamanho grande o suficiente para que seja possível a sua observação e pequeno o suficiente para que seja visualmente coerente. Quanto à sua trajetória, os pontos foram calculados de modo a evitar colisões com os planetas e de forma a seguir uma órbita mais inclinada. Relativamente ao seu tempo de translação, foi atribuído tempo que torna fácil a sua observação e detecção.

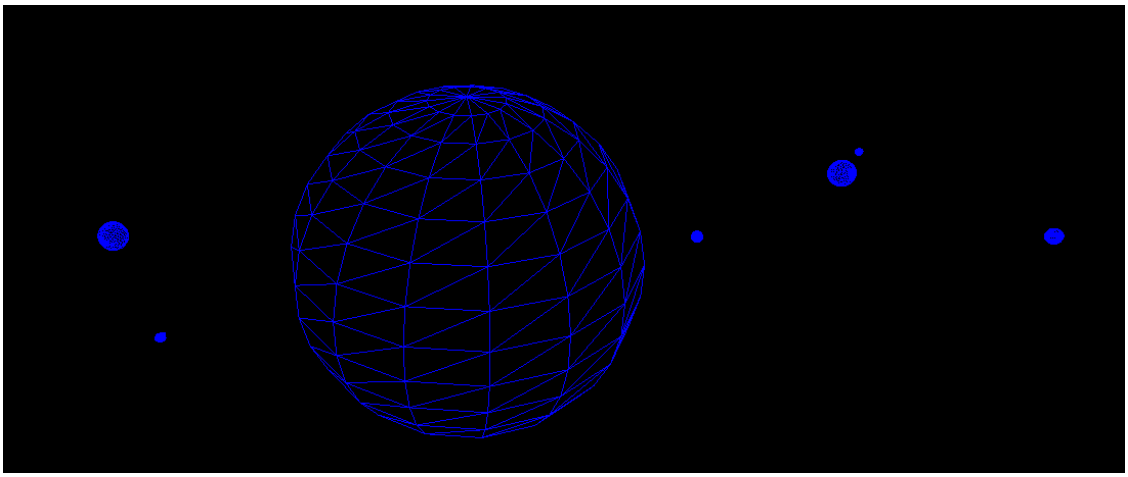


Figura: *Cena animada de Sistema Solar*

Nas restantes cenas estão também implementados diversos movimentos, sendo estes:

Boneco de Neve:

- Mover os braços.
- Rodar o chapéu.

Gato:

- Abanar a cauda.
- “Saltitar”.

Iluminação

De forma a implementar a capacidade de adição de luz na aplicação Motor, é necessário modificar a aplicação Gerador de forma a calcular as normais de cada vértice. Assim, as funções responsáveis pela criação das coordenadas de cada objeto foram modificadas de forma a calcular também as normais associadas a estes. Estes valores são então adicionados juntamente às coordenadas no ficheiro produzido.

Relativamente ao cálculo das normais para as curvas/superficies de Bézier, estas são cálculadas utilizando a fórmula:

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

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

Para as figuras que utilizam o algoritmo de revolução na criação dos seus pontos, é apenas necessário alterar o algoritmo de revolução, sendo que este passou a calcular as normais sobre a curva 2D que é passada para o algoritmo como parâmetro. À medida que se "roda" a curva 2D e se calculam as coordenadas, também se "rodam" e calculam as normais.

De forma a armazenar em memória as luzes declaradas no XML foi adicionado um vetor de *Lights* na classe Scene, sendo as luzes declaradas da seguinte forma:

```
class Light {
public:
	float posCoords[4];
	int number;
	int property;
	Light (float x, float y, float z, float type, int number, int property);
};
```

As luzes são colocadas no decorrer do desenho da cena, e apenas são ativas quando necessárias. A class Component, responsável pelas propriedades das luzes e texturas (referidas seguidamente), tem acesso à *Scene* onde está inserida, de forma a que sempre que esta é desenhada sejam tidas em conta as propriedades das luzes.

Texturas

Similarmente à implementação de iluminação no motor, devem ser realizadas alterações no Gerador de forma a calcular as coordenadas de textura correspondentes a cada figura. Estas coordenadas são também calculadas simultâneamente ao cálculo das coordenadas de cada ponto e as suas normais, sendo estes valores também inseridos no ficheiro criado.

Relativamente aos sólidos de revolução, as coordenadas Y das texturas são calculadas dividindo a soma actual da curva desenhada com a soma total da curva. As coordenadas X consistem simplesmente no inverso do nº de fatias actuais.
No caso do paralelípipedo, as coordenadas de textura são calculadas manualmente em cada um dos pontos. Finalmente, nas curvas/superfícies de Bezie, os pontos da textura são as variáveis u e v utilizadas também no cálculo dos pontos espaciais.

Para o desenho das texturas a inicialização é feita na class Component. As texturas declaradas no ficheiro XML são carregadas através da função loadTexture() recorrendo a funções da biblioteca *DevIL*, sendo depois gerada a textura OpenGL. Quando é desenhada a cena são também as texturas correspondentes a cada modelo.

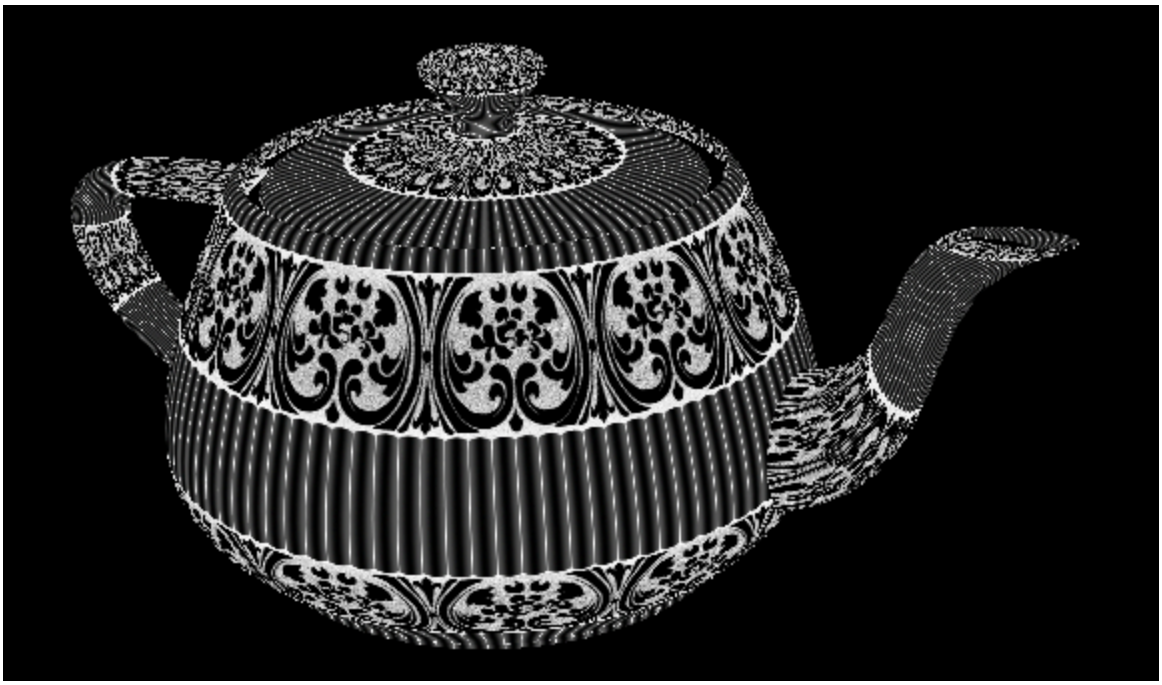


Figura: *Texturização de um objeto (Bule)*

Texturização e iluminação das cenas

Após a implementação de iluminação e texturas nas aplicações, as cenas são finalmente atualizadas de forma a consistirem numa representação gráfica de todas as funcionalidades apresentadas. Assim, a cada planeta/objeto foram atribuidas as texturas correspondentes à sua superfície, sendo também atribuido ao bule (cometa) uma textura similiar à encontrada nas “porcelanas de Viana”. Relativamente à iluminação, esta consiste numa unica luz do tipo POINT presente no centro do objeto que representa o Sol.
O jogo de chá foi atualizado similarmente ao cometa, sendo adicionadas as texturas e luzes adequadas.

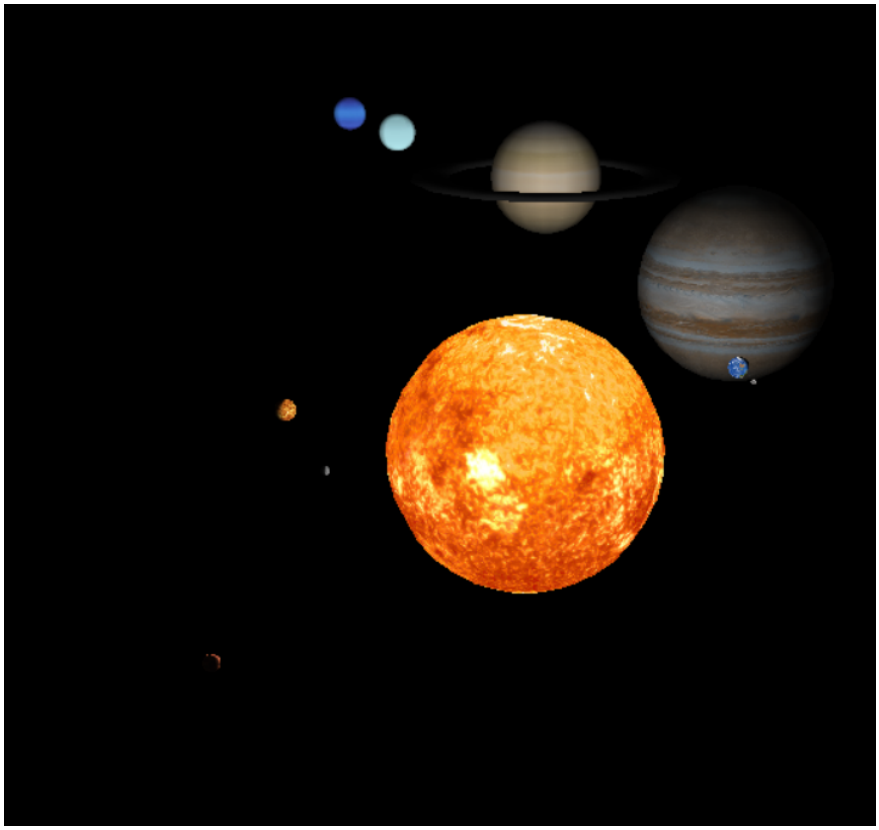


Figura: *Sistema solar com texturas e iluminação*

Figura: *Jogo de chá com texturas e iluminação*

Conclusões e pontos chave

Dada a conclusão da 1ª fase, foram criadas duas aplicações (Gerador e Motor) modulares e fácilmente extensíveis. A criação de uma classe (Figure) capaz de abstrair a criação de uma dada figura proporciona facilidade na "comunicação" entre os diversos módulos do Motor, assim como no momento da criação dos pontos da figura pelo Gerador.
O motor fornece uma experiência de visualização dinâmica das figuras, sendo capaz de mudar a aparência dos sólidos em tempo de execução e a pedido do utilizador, assim como a forma como este "navega" no espaço tridimensional (diversas câmeras).
Foram implementadas todas as figuras requeridas e com os requisitos referidos sendo também implementadas algumas figuras complementares. Certas figuras foram também implementadas de forma a poderem ser reutilizadas e implementadas na criação de outras figuras mais complexas, providenciando ferramentas para a fácil criação futura de uma grande variedade de outras figuras.
De notar que figuras tais como a esfera, cone, cilindro e anel foram implementadas como sólidos de revolução, simplificando não apenas os cálculos necessários à sua criação, mas também o código necessário para o fazer.

Suportados pela base sólida desenvolvida na 1ª etapa, na 2ª etapa foi modificada a estrutura de dados de forma a interpretar os grupos e as diferentes transformações associadas. Ocorreu um aumento na complexidade do trabalho realizado previamente, não descuidando a capacidade de evolução do projeto geral para as próximas fases, tendo sido cumpridos todos os requisitos mencionados no enunciado e tendo sido adicionados vários extras que complementam a funcionalidade das aplicações e a forma como são utilizadas.

Na 3ªetapa foram implementadas diversas novas funcionalidades, tais como a geração de figuras através da utilização de superfícies de Bezier, a translação e rotação de figuras através dos algoritmos de curvas de CatMull-Rom assim como a inclusão de noção de tempo. A performance do motor foi incrementada através da utilização de Vertex Buffer Objects com índices, sendo que a implementação prévia de figuras de revolução tornou-se útil dado ter facilitado a inclusão desta funcionalidade. No final, todas as cenas foram atualizadas para incluir o movimento agora definido, tendo sido cumpridos todos os objetivos referentes a esta fase.

Na 4ª e última fase, todo o trabalho desenvolvido anteriormente foi complementado com a possibilidade de adição de iluminação a uma dada cena (por vetores ou pontos), assim como texturas aos sólidos nesta presentes. Como funcionalidade extra é possível observar agora também as normais associadas aos vértices dos diferentes sólidos. Todas as cenas foram novamente atualizadas com as diversas texturas e iluminações (quando aplicável). Em forma de conclusão final, creemos ter realizado um trabalho com um nível consideravelmente acima

das expetativas, possuindo este diversas funcionalidades extra não presentes nos requisitos, sendo estes também satisfeitos com sucesso.

Bibliografia

[1] Ramires, António, University of Minho, CURVES AND SURFACES [2015-04-20]