2º Fase – Transformações Geométricas

Computação Gráfica – Universidade do Minho

2017/2018

Carlos Gonçalves – A77278

Jorge Oliveira – A78660

José Ferreira – A78452

Ricardo Peixoto – A78587

Índice

Índice de Figuras

1. Introdução

O presente relatório abordará a conceção da 2ª fase de um projeto que tem como objetivo a construção de um sistema que simule o sistema solar. Após uma primeira fase em que nos focamos essencialmente nas figuras geométricas que constituem o nosso sistema, nesta segunda fase o principal objetivo era aplicar transformações geométricas de modo a posicionar os planetas nos locais indicados.

Explicaremos o método que utilizamos para aplicar as transformações a cada planeta, de modo a que este ficasse na posição indicada com uma escala ideal. Para além disto, também será abordado o motor responsável pela geração das figuras e ainda as estruturas, já que tivemos de adicionar mais algumas em relação à primeira fase.

Para além de efetuarmos as transformações, também apresentamos o conceito da câmara, de uma forma, para já, mais simplificada. Decidimos ainda acrescentar mais duas figuras novas, o *torus* (que será essencial para uma fase posterior) e o cilindro (que nos permite desenhar novos elementos para acrescentar no sistema solar).

2. Transformações Geométricas

Visto este ser um projeto que visa representar uma realidade, decidimos ser o mais rigorosos possível no contexto da dimensão, distância ao sol e inclinação dos planetas que compõem o sistema solar.

2.1. Dimensões dos corpos celestes

Para determinar a escala que iriamos aplicar ao Sol e a cada planeta fizemos uma pesquisa e obtivemos valores que nos levaram a concluir que não poderíamos aplicar as escalas de forma muito rigorosa. Por exemplo, o diâmetro do Sol é cerca de 300 vezes maior que o raio de Mercúrio não sendo possível visualizar mercúrio caso pretendêssemos obter uma imagem de todo o sistema solar. O mesmo acontecia para Vénus, Terra e Marte. Deste modo decidimos reduzir a escala do sol em cerca de 100000000 vezes, a escala dos planetas telúricos (Mercúrio, Vénus, Terra e Marte) em cerca de 1000000 vezes e a escala dos planetas gasosos (Júpiter, Saturno, Úrano e Neptuno) em cerca de 20000000 vezes. Deste modo foi possível manter a ideia de superioridade de dimensão entre os diferentes corpos celestes sem sacrificar a capacidade de representação de todo o sistema solar. Às escalas obtidas anteriormente aplicamos apenas algumas atenuantes a cada planeta em particular para tornar mais realista esta comparação de dimensões como é possível observar na seguinte tabela que representa os seus raios.

Corpos Celestes	Dimensões reais (m)	Dimensões adaptadas
Sol	696.000.000.000	6,963
Mercúrio	2.440.000	0,314
Vénus	6.052.000	0,645
Terra	6.371.000	0,688
Marte	3.390.000	0,437
Júpiter	69.911.000	2,984
Saturno	58.232.000	2,614
Úrano	25.362.000	1,528
Neptuno	24.622.000	1,448

2.2. Distâncias dos planetas ao Sol

À semelhança do que verificamos nos raios, as distâncias dos planetas ao Sol variam drasticamente à medida que nos afastamos do mesmo. Por exemplo a distância de Neptuno ao Sol é cerca de 6500 vezes superior ao raio do sol, impossibilitando assim a representação do sistema solar. Deste modo, para a representação das distâncias, a nossa abordagem foi apenas representar que à medida que nos afastamos do Sol, as distâncias entre os planetas são cada

vez maiores. A seguinte tabela mostra como convertemos as distâncias dos diferentes planetas ao Sol.

Corpos Celestes	Distâncias reais (m)	Distâncias adaptadas
Mercúrio	57.910.000.000	2,781
Vénus	108.200.000.000	4,733
Terra	149.600.000.000	7,483
Marte	227.900.000.000	11,263
Júpiter	778.500.000.000	25,477
Saturno	1.429.000.000.000	39,676
Úrano	2.871.000.000.000	55,167
Neptuno	4.495.000.000.000	71,925

3. Câmera

De maneira a conseguir uma aplicação o mais modular possível decidimos que a manipulação da camara seria feita numa classe à parte do *motor*. Assim sendo criamos uma classe *Camera* que tinha as seguintes funções:

1. Movimento horizontal (esquerda e direita)
2. Movimento vertical (para cima e para baixo)
3. Aproximação/afastamento
4. Velocidade nos movimentos regulável.

Para que este componente conseguisse realizar estas manipulações precisava de guardar os seguintes parâmetros:

- Distancia à origem (raio)
- Ponto com as coordenadas atuais da camara
- Ângulo para a movimentação horizontal (alpha)
- Ângulo para a movimentação vertical (beta)
- Valor que define a variação do angulo (angulo)
- Variáveis que controlassem o movimento do rato (mouseX, mouseY, mouseTracking).

Após definirmos os parâmetros necessários construímos os métodos necessários.

Sempre que efetuamos alterações em algum parâmetro que altere a posição da câmera recorremos ao método *atualizaPosicao*, que calcula as novas coordenadas da posição da câmera e atualiza a variável de instância *posicao*, responsável por guardar essa mesma posição.

O método *getPosicao* é constantemente chamado na função *renderScene* do *motor* e devolve a posição atual da câmera. Apesar de atualizarmos a posição da câmera sempre que pressionamos as teclas, essa posição só é atualizada no programa quando a imagem é impressa, ou seja, através da função renderScene.

Como o tratamento das teclas pressionadas e do movimento do rato são controlados nesta classe precisávamos de alguns métodos que processassem ações do teclado e outros métodos que processassem ações do rato.

3.1. Ações do teclado

Para o teclado decidimos criar os métodos *normalKeys* e *specialKeys.* O primeiro é chamado quando deteta que o sinal mais/menos é pressionado e aumenta/diminui a variável raio para termos um efeito de aproximação/afastamento em relação ao foco (origem). O segundo método é utilizado para controlar as rotações e velocidade dessas mesmas rotações. As setas esquerda e direita são utilizadas para controlar o movimento horizontal (angulo alpha) e consequentemente quando são pressionadas aumentam/diminuem o valor desse angulo. O mecanismo utilizado para controlar o movimento vertical é idêntico, mas neste caso o angulo a alterar é o beta. Além disso utilizamos também as teclas PAGE_DOWN e PAGE_UP para definir a velocidade de rotação. Sempre que estas são clicadas a variável que guarda a variação do angulo é multiplicada por um fator que o aumenta/diminui criando um efeito de aceleração/desaceleração.

3.2. Ações do rato

Para monitorizar o rato recorremos aos métodos *mouseButtons* e *mouseMotion*.

O método mouseButtons é chamado quando se deteta que algum dos botões do rato é pressionado. Como cada botão do rato vai realizar uma ação diferente, registamos qual o botão clicado através da variável *mouseTracking*. Para calcular o deslocamento do rato no ecrã também é necessário registar as posições horizontal e vertical do rato no momento em que se clica num dos botões. Esses valores são registados nas variáveis *mouseX* e *mouseY*. Quando o botão é largado, é executada uma ação tendo em conta o botão que estava a ser pressionado. O botão esquerdo é usado para rodar a câmera em torno do foco (ponto (0,0,0)) logo, quando for largado, as variáveis *alpha* (movimento horizontal) e *beta* (movimento vertical) terão de ser atualizadas. Para tal, utilizamos o deslocamento (horizontal e vertical) do rato no ecrã. Já o botão direito é usado para aproximar / afastar a câmera do foco, ou seja, após ser largado é necessário atualizar a variável *raio* (distância da câmera ao foco). Para tal utilizamos apenas o deslocamento vertical do rato no ecrã.

Como o método mouseButtons apenas atualizava as variáveis *raio* (botão direito) ou *alfa* e *beta* (botão esquerdo) quando o botão era largado, mantendo os valores das mesmas

constantes enquanto o botão estava a ser pressionado, criamos o método mouseMotion. Este método tem como função monitorizar continuamente a posição do rato no ecrã e atualizar constantemente a posição da câmera à medida que se move o rato ao longo do ecrã, enquanto que algum dos botões está a ser pressionado. Para tal, se algum botão estiver a ser pressionado, determinamos o deslocamento do rato desde que o botão foi pressionado até à sua posição atual e calculamos o valor das variáveis locais *rAux*, *alphaAux* e *betaAux* tendo em conta qual o botão que está a ser pressionado do mesmo modo que no método *mouseButtons*. De seguida é chamado o método *atualizaPosicao* para atualizar a posição usando os valores das variáveis locais referidas.

A diferença entre o mouseButtons e o mouseMotion é que o mouseButtons apenas atualiza as variáveis *raio*, *alpha* e *beta* quando o botão do rato é largado enquanto que a mouseMotion calcula um novo valor para as suas variáveis locais rAux, alphaAux e betaAux sempre que o método é chamado. Como o método mouseMotion está constantemente a ser chamado no *motor*, temos a garantia que a posição da câmera estará sempre a ser atualizada enquanto se move o rato com alguma tecla pressionada.

4. Figuras

4.1. Torus

Para que pudéssemos desenhar os anéis de saturno o grupo decidiu que seria necessário desenhar uma nova figura, neste caso, o *torus*. O *torus* é uma figura peculiar em forma de “donut”, mas que tinha as propriedades necessárias para representar de uma forma fidedignas os anéis de saturno.

Para desenhar o *torus* era necessário um conjunto de parâmetros: o raio do tubo, define o raio do “aro”, o raio do anel, define o raio do “buraco” do *torus*, o número de fatias e o número de camadas. Para além disto, foi necessário efetuar uma transformação de coordenadas esféricas para coordenadas cartesianas. Aplicando os conhecimentos de trigonometria obtivemos as seguintes fórmulas:

z = r * sin(beta)

x = (R + r * cos(beta)) * cos(alpha)

y = (R + r * cos(beta)) * sin(alpha)

Onde r é o raio do tubo e R o raio do anel. Os ângulos alpha e beta pertencem ao intervalo 0 < ângulo < 2*PI. De salientar, que com o uso destas fórmulas, a “altura” do *torus* estaria em z, pelo que este nos aparece em “pé” no ecrã ao invés de “deitado”.

Os valores iniciais de cada ângulo são os seguintes:

alpha = 2*PI / fatias

beta = 2*PI / camadas

Isto pressupunha que o ângulo alpha era usado para controlar o movimento circular do *torus* e o ângulo beta era usado para formar parte do tubo.

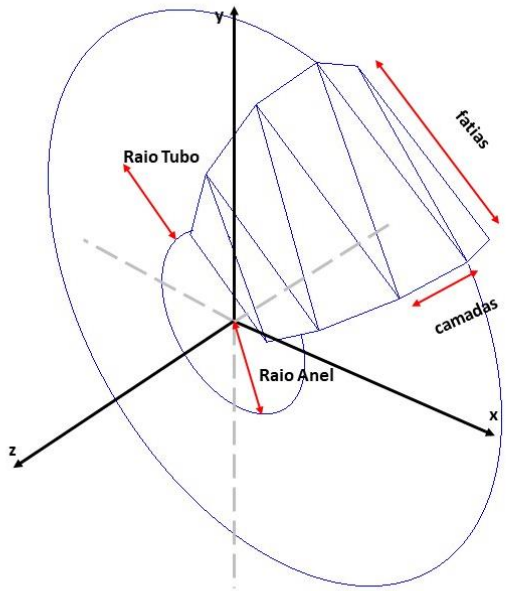


Figura 1 - Orientação do Torus

A estratégia adotada pressupunha que para cada fatia eram desenhadas as N camadas, recebidas como parâmetro. Mas ao contrário da esfera em que por cada fatia só se desenhava "metade" da esfera, no *torus* por cada fatia o ângulo beta rodava totalmente para que pudéssemos ter, naquela fatia, um tubo completamente fechado. Por cada camada eram desenhados 2 triângulos seguindo a regra da mão direita, para que ficasse orientado para o exterior. A iteração foi realizada através de 2 ciclos for, um para a deslocação horizontal e outro para a rotação total do tubo. O incremento dos ângulos (o que verdadeiramente nos permitia mover) era feito através da fórmula i*alpha e j*beta, sendo que tanto i como j estavam delimitados, respetivamente, ao número de fatias e de camadas. Ao aumentar o valor de alpha e beta seguindo uma operação de multiplicação, permitiu diminuir a percentagem de erro que poderia aparecer nas operações de virgula flutuante quando se usam somas.

Assim os pontos seriam:

```
z1 = raioTubo*sin(beta*j);
z2 = raioTubo*sin(beta*(j+1));
y1 = (raioMaior + raioTubo*cos(beta*j))*sin(alpha*i);
y2 = (raioMaior + raioTubo*cos(beta*j))*sin(alpha*(i+1));
y3 = (raioMaior + raioTubo*cos(beta*(j+1)))*sin(alpha*i);
```

```
y4 = (raioMaior + raioTubo*cos(beta*(j+1)))*sin(alpha*(i+1));
x1 = (raioMaior + raioTubo*cos(beta*j))*cos(alpha*i);
x2 = (raioMaior + raioTubo*cos(beta*j))*cos(alpha*(i+1));
x3 = (raioMaior + raioTubo*cos(beta*(j+1)))*cos(alpha*i);
x4 = (raioMaior + raioTubo*cos(beta*(j+1)))*cos(alpha*(i+1));
```

P1 = (x1, y1, z1)

P2 = (x2, y2, z1)

P3 = (x3, y3, z2)

P4 = (x4, y4, z4)

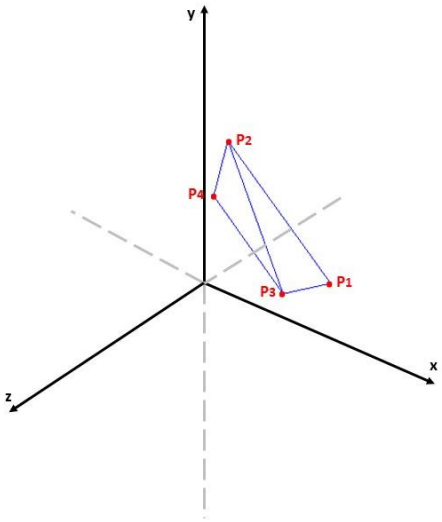


Figura 2 - Pontos Torus

Os pontos estão orientados segundo a regra da mão direita.

4.2. Cilindro

Decidimos construir uma nova figura geométrica, o cilindro (que nos vai ser útil para um desenho que iremos abordar posteriormente neste relatório). Para poder desenhar esta figura geométrica, são precisos 3 parâmetros: o raio, altura e o número de fatias. O raio define a "largura", a altura, como o próprio nome indica, define a altura do cilindro e o número de fatias assenta no grau de detalhe com que o cilindro é desenhado.

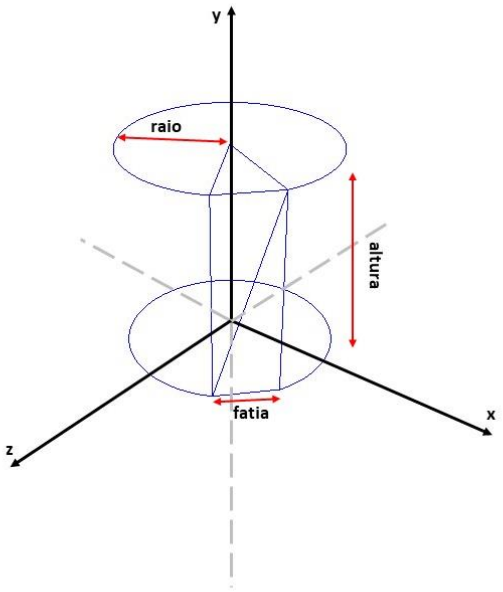


Figura 3 - Orientação Cilindro

Para desenhar o cilindro era necessário desenhar 4 triângulos por cada fatia iterada, 1 da base inferior, 2 para a face lateral (em conjunto formavam um retângulo) e 1 para a base superior. Esta iteração foi efetuada usando um ciclo for, que inicializava uma variável i a zero, e o caso de paragem era quando i fosse menor que N, sendo N o número de fatias passadas inicialmente.

Trabalhámos com um ângulo alpha, que inicialmente tomava o valor de 2*PI / (número de fatias). O y não nos apresentou grande problema, já que ou tinha o valor de 0 ou então o valor da altura do cone. Porém o x e o z não foi tão trivial. Para estes foi necessário recorrer a fórmulas trigonométricas, onde:

x = raio * sin(alpha)

z = raio * cos(alpha)

De salientar que mais uma vez, para conseguir avançar no eixo dos xx e dos zz, era necessário incrementar o ângulo, que foi feito à custa de uma multiplicação pela variável i, para que os erros referentes à virgula flutuante fossem diminutos.

Sendo assim os pontos usados são:

```
pxA = radius * sin(alpha*i);
pxD = radius * sin(alpha*(i+1));
pzA = radius * cos(alpha*i);
pzD = radius * cos(alpha*(i+1));
```

P1 = (0, 0, 0)

P2 = (pxA, 0, pzA)

P3 = (pxD, 0, pzD)

P4 = (pxA, altura, pxA)

P5 = (pxD, altura, pzD)

P6 = (0, height, 0)

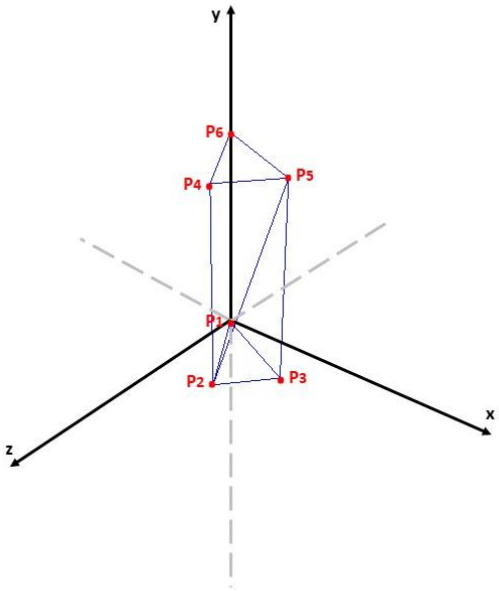


Figura 4 - Pontos Cilindro

De salientar que os pontos estão orientados segundo a regra da mão direita para que a face ficasse voltada para o ecrã. De notar que nesta imagem alterámos a orientação da base para que se notasse a formação do triângulo, em circunstâncias normais, com esta câmera não víamos o triângulo.

4.3. Foguetão

Decidimos criar um foguetão. Apesar deste objeto não ser um elemento natural do espaço, hoje-em-dia é bastante usual encontrá-lo no nosso sistema solar, sendo então um objeto mítico e representativo do universo. Quando se fala em espaço, muitas das vezes veem a cabeça o tal “foguetão”. Posto isto, e também como um elemento extra no trabalho e estar relacionado com as figuras geométricas, decidimos criar um foguetão.

Para desenhar este foguetão utilizamos as figuras que que já tínhamos feito previamente. O foguetão é então constituído por um cilindro, que será o corpo do foguetão, e mais 5 cones, um dos cones será a cabeça enquanto que os outros 4 vão formar a “cauda”.

Começamos por desenhar então o cilindro, para termos o corpo do foguetão, este foi desenhado como sendo o centro do foguetão, ou seja, não foi preciso aplicar nenhuma transformação geométrica. Após ter o corpo desenhado, tínhamos de desenhar a cabeça, ou seja, um cone em cima do cilindro. Para realizar esta operação tivemos de efetuar um translate no eixo dos yy com valor igual à altura do cilindro. Assim, só nos faltava a cauda.

Os 4 cones que formam a cauda, tinham de ser colocados em redor do cilindro, ou seja, era necessário efetuar uma translação (para colocar o cilindro na base do cone) e de 2 rotações. Antes destas transformações decidimos aplicar uma pequena translação para que toda a base do cone esteja em contacto com a face do cilindro.

Para desenhar os 4 cones, criámos um ciclo for, que iterava com um valor inicial i=0 e parava quando i=4. Também temos um ângulo alpha era inicializado com o valor de 360/4. A primeira transformação (após a translação inicial) foi uma rotação em torno do eixo dos yy, com o valor i*alpha. Efetuando esta rotação em torno do eixo y tínhamos exatamente o eixo do xx na posição onde iria ser desenhado o cone, assim, de seguida era só aplicar uma translação no eixo dos xx com um valor igual ao raio do cilindro. Através destas duas transformações tínhamos então o cone, que fazia parte da cauda, sobre a face do cilindro, faltava então colocá-lo com uma orientação correta com a face. Assim, efetuamos a última rotação que seria em torno do eixo dos zz com um valor de -135 para que ficasse orientado para fora.

Ainda tentamos efetuar 1 única rotação em torno de 2 eixos, mas não conseguimos os resultados pretendidos, por isso decidimos efetuar 2 rotações. Desta forma simples, conseguimos então criar um foguetão que pode viajar pelo nosso sistema solar.

5. Aplicação

5.1. Motor

Como já foi referido no diagrama de classes, o **motor** contém um apontador para uma estrutura **Grupo** (correspondente à **cena**) e **Câmera**.

Inicialmente faz o *parsing* do ficheiro passado como argumento, com o auxílio do **parser**, carregando para a **cena** o apontador para **Grupo** devolvido pela função de *parsing*. Depois, sempre que é chamada a função **renderScene** este chama uma função **imprimeGrupo** que tem como finalidade apresentar as imagens correspondentes ao grupo passado como argumento (com as devidas transformações).

A função **imprimeGrupo** percorre as transformações geométricas do mesmo, aplicando a operação para cada uma. Depois percorre as **Figuras** que o grupo contém apresentando-as com o auxílio da função **imprimeFigura**. Depois percorre os **grupos** filhos presentes no mesmo e aplica a mesma função a esses **grupos**. Para que as transformações geométricas sejam realizadas a todo o grupo (modelos e grupos) e apenas a esse grupo, no início é realizada a operação *glPushMatrix()* e no final *glPopMatrix()*.

```
void imprimeGrupo(Grupo* g){

     glPushMatrix();

    vector<Operacao*> operacoes = g->getOperacoes();

    for(Operacao op: operacoes){

        op->aplicaOperacao();

    }

    vector<Figura*> figuras = g->getFiguras();

    imprimeFigura(figuras);

    vector<Grupo*> gruposFilhos = g->getGrupos();

    for(Grupo* filho: gruposFilhos){

        imprimeGrupo(filho);

    }

    glPopMatrix();

}
```

A função **imprimeFigura** recebe como argumento um apontador para uma **Figura** e percorre cada ponto dessa **figura**, desenhando-os (a ordem é importante) com a função *glVertex3f*.

```
void imprimeFigura(vector<Figura*> figuras){

    for(Figura* f: figuras){

        vector<Ponto*> pontos = f->getPontos();
```

```
        for (Ponto *p: pontos) {
            glVertex3d(p->getX(), p->getY(), p->getZ());
        }
    }
    glEnd();
}
```

5.2. Estruturas

Nesta fase necessitamos de mais estruturas para além das definidas na fase anterior. Por forma a realizar as transformações geométricas e como estas são de 3 tipos diferentes necessitamos de uma classe **Operação** que corresponde a uma transformação geométrica. No entanto esta terá de ser abstrata pois nunca pode ser concretizada, ou seja, uma operação é um caso geral, mas a transformação em si é um caso particular desta, pelo que criamos 3 subclasses da classe **Operação**: **Rotação, Translação, Escala**, que correspondem às transformações que necessitamos. Para além desta, e como o ficheiro *XML* está organizado por grupos decidimos também ter uma classe **Grupo**, que corresponde a um *group* do ficheiro *XML*. Por forma a ter uma câmera móvel, decidimos também implementar uma classe **Camera.**

5.2.1. Operação

A classe operação corresponde a uma transformação geométrica, pelo que é abstrata (como referido anteriormente). Esta classe apenas tem 3 variáveis do tipo *float* que correspondem respetivamente aos valores relativos aos eixos **x,y** e **z** pelo que têm exatamente esse nome (todas as transformações contém estas variáveis). Para além disto necessitamos de criar um método para que a transformação fosse criada, pelo que temos o método **aplicaOperação** (método abstrato que apenas é realizado nas subclasses).

```
class Operacao {
    float x;
    float y;
    float z;
public:
      Operacao();
      Operacao(float,float,float);

      float getX();
```

```
        float getY();
        float getZ();

        void setX(float);
        void setY(float);
        void setZ(float);

        virtual void aplicaOperacao() = 0;
        virtual string toString() = 0;
}
```

5.2.2. Rotação

Esta classe é uma subclasse da **Operacao** e corresponde a uma rotação, pelo que necessita de mais uma variável, o **angulo** da rotação. Para além disto a função **aplicaOperacao** realiza uma rotação (*glRotate*) segundo as variáveis da classe.

```
class Rotacao: public Operacao {
private:
    float angulo;
public:
        Rotacao();
        Rotacao(float,float,float,float);
        float getAngulo();
        void setAngulo(float);
        void aplicaOperacao();
        string toString();
}
void Rotacao::aplicaOperacao(){
        glRotatef(angulo,getX(),getY(),getZ());
        }
```

5.2.3. Translação

Esta classe é uma subclasse da **Operacao** e corresponde a uma translação, pelo que não necessita de mais nenhuma variável das correspondentes ao **x, y** e **z**. A função **aplicaOperacao** realiza uma translação (*glTranslate*) segundo as variáveis da classe.

```
void Translacao::aplicaOperacao(){

	glTranslatef(getX(),getY(),getZ());

}
```

5.2.4. Escala

Esta classe é uma subclasse da Operacao e corresponde a uma escala, pelo que não necessita de mais nenhuma variável das correspondentes ao x, y e z. Ainda, a função aplicaOperacao realiza uma escala (glScale) segundo as variáveis da classe.

```
void Escala::aplicaOperacao(){

	glScalef(getX(),getY(),getZ());

}
```

5.2.5. Grupo

A classe **Grupo** corresponde a um *group* presente no ficheiro *XML*. Como identificamos no ficheiro, um grupo pode conter várias operações (ordenadas), várias figuras e vários grupos (filhos), pelo que esta classe contém:

- um *vector* de apontadores para **Operacao**, sendo que cada operação é inserida no fim desse vector (por causa da ordem)
- um *vector* de apontadores para **Figura**, correspondente às figuras
- um *vector* de apontadores para **Grupo**, correspondendo aos filhos

Para além disto, para identificar um grupo decidimos ter um inteiro que é o **id** do mesmo.

```
class Grupo {

private:

	int id;

	vector<Operacao*> operacoes;

	vector<Figura*> figuras;

	vector<Grupo*> grupos;

public:

	Grupo(int);
```

```
        vector<Operacao*> getOperacoes();
        vector<Figura*> getFiguras();
        vector<Grupo*> getGrupos();
        void adicionaOperacao(Operacao*);
        void adicionaFigura(Figura*);
        void adicionaGrupo(Grupo*);
        string toString();
}

void Grupo::adicionaOperacao(Operacao* a){
        operacoes.push_back(a);
}
```

Outros casos são semelhantes.

5.2.6. Diagrama de classes

De seguida apresentamos o diagrama de classes da aplicação **motor.**

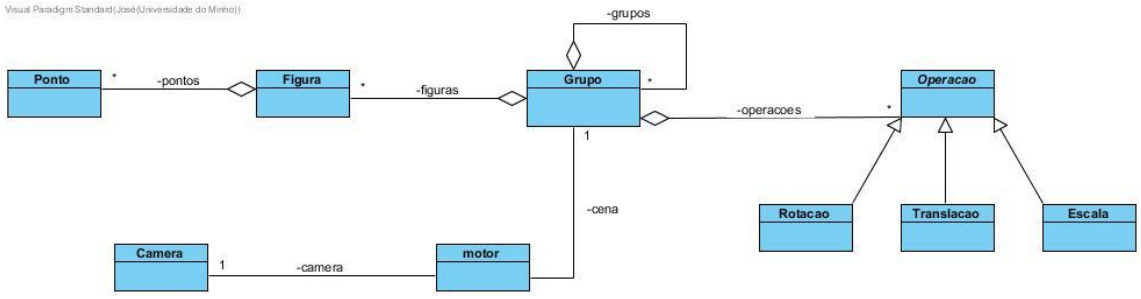


Figura 5 - Diagrama de classes do motor

Podemos verificar que a aplicação contém um apontador para um **Grupo** e para uma **Camera**. Cada **Grupo** contém uma lista de **Figuras**, **Operações** e de **Grupos** filhos. A classe **Operação** é abstracta e as suas subclasses são: **Rotação, Translação** e **Escala**. Uma **Figura** é constituída por uma lista de **Pontos**.

5.3. XML

5.3.1. Parsing

Para realizar o *parsing* do ficheiro XML primeiro analisamos as *tags* que este continha, sendo que delimitamos as seguintes:

- Uma principal chamada *scene*
- Uma secundária chamada *group* que está contida na principal, em cada uma destas podem estar também:
  - *translate/rotate/scale* que correspondem às transformações geométricas a um determinado grupo
  - *models* que são os ficheiros que o grupo contém (as figuras que irão ser desenhadas)
  - *group*, outros grupos

Posto isto, decidimos que a função principal teria de retornar um apontador para uma estrutura do tipo **Grupo** que seria a cena principal. Para conseguirmos ter acesso ao grupo principal, teríamos de aceder ao primeiro elemento filho do ficheiro com a tag *scene* e ao primeiro filho desse com a tag *group.* Depois chamávamos uma função que fazia o *parsing* de um grupo, ou seja:

```
XMLElement * elemento = xmlDoc.FirstChildElement("scene")->FirstChildElement("group");
        if(elemento != nullptr){
            XMLElement * elementoFilho = elemento->FirstChildElement();
            if(elementoFilho != nullptr){
                grupos = parseGrupo(elementoFilho);
            }
        }
```

Parsing de um grupo:

Para realizar o *parsing* de um grupo reparamos que o ficheiro vem sempre ordenado da seguinte forma: transformações geométricas, modelos e os grupos restantes. Com isto, definimos uma ordem para realizar o *parsing* do grupo e criamos funções auxiliares, para realizar o *parsing* às operações geométricas e aos modelos, por forma a organizar melhor o código.

Ou seja, primeiro realizamos o *parsing* às operações geométricas a e aos modelos e só depois aos restantes grupos:

Nas operações temos de passar o apontador para o elemento para que este conteúdo seja alterado.

```
parseOperacoes(&elemento,res);
	cout << elemento->Name() << endl;


	if(strcmp(elemento->Name(),"models") == 0){
		parseModelos(elemento, res);
		elemento = elemento->NextSiblingElement();
	}


	while((elemento) && (strcmp(elemento->Name(),"group") == 0)){


		XMLElement *elementoFilho = elemento->FirstChildElement();
		if(elementoFilho){
			Grupo* filho = parseGrupo(elementoFilho);
			res->adicionaGrupo(filho);
		}
		elemento = elemento->NextSiblingElement();
	}
```

Parsing de Operações Geométricas

Para realizar a leitura das operações geométricas tivemos de separar em três casos: rotações, escalas e translações. Para tornar mais fácil a compreensão, decidimos criar funções auxiliares para cada uma das operações. Ao realizar a leitura, comparamos o nome do elemento XML com *rotate/scale/translate* e caso fosse um dos mencionados chamamos a função auxiliar correspondente e avançamos para o próximo elemento, caso não fosse abandonávamos a função, pois já seria um modelo ou grupo.

```
if(strcmp((*elemento)->Name(),"rotate") == 0){
		parseRotacao(*elemento,grupo);
```

```
}


else{
	if(strcmp((*elemento)->Name(),"scale") == 0){
		parseEscala(*elemento,grupo);
	}


	else{
		if(strcmp((*elemento)->Name(),"translate") == 0){
			parseTranslacao(*elemento,grupo);
		}


		else{
			return;
		}
	}
}


(*elemento)=(*elemento)->NextSiblingElement();


if(*elemento){
	parseOperacoes(elemento,grupo);
}
```

Para realizar o *parsing* da rotação/escala/translação apenas teríamos de veiricar se continham os atributos (*X,Y,Z* no caso das últimas e *angle,axisX,axisY,axisZ* no caso da primeira) e caso contivessem então líamos o valor do número correspondente. Para inicializar as variáveis teríamos de inicializar ao valor identidade para a operação em causa, ou seja, para as rotações e translações inicializamos a 0 e para as escalas a 1.

Depois adicionávamos cada operação ao grupo correspondente.

```
if(elemento){
            float x=0,y=0,z=0,angulo=0;

            if(elemento->Attribute("angle")){
                  const     char*     anguloAux     =     elemento-
>Attribute("angle");
                  angulo = atof(anguloAux);
            }

            if(elemento->Attribute("axisX")){
                  const char* xAux = elemento->Attribute("axisX");
                  x = atof(xAux);
            }
            if(elemento->Attribute("axisY")){
                  const char* yAux = elemento->Attribute("axisY");
                  y = atof(yAux);
            }
            if(elemento->Attribute("axisZ")){
                  const char* zAux = elemento->Attribute("axisZ");
                  z = atof(zAux);
            }

            Rotacao* r = new Rotacao(x,y,z,angulo);
            grupo->adicionaOperacao(r);
      }
```

Os casos da translação e escala são análogos.

Parsing dos Modelos

Antes de realizar a leitura verificamos que cada *tag models* contém um conjunto de *tag model* que corresponde ao ficheiro. Para aceder a este necessitávamos de obter o primeiro filho

do elemento XML com a *tag model* e depois continuar a iterar sobre estes elementos até que não houvesse mais nenhum.

Depois reparamos que cada modelo tem um atributo *file* que contém o nome do ficheiro, pelo que para obter o nome do mesmo acedemos a esse atributo. Após termos o nome do ficheiro (o **3D** criado anteriormente) chamamos a função que extrai a informação desse ficheiro e nos dá um apontador para uma estrutura de **Figura** e adicionamos essa mesma ao **grupo**.

```
XMLElement * pElement = elemento->FirstChildElement("model");

      while (pElement != nullptr){

            const char* cenas = pElement->Attribute("file");

            if (cenas == nullptr) return;

            string fich(cenas, strlen(cenas));

            Figura* f = extraiFicheiro(fich);

            g->adicionaFigura(f);

            pElement = pElement->NextSiblingElement("model");


            }
```

Extrair informação do ficheiro para a figura

Para extrair a informação do ficheiro **3D** previamente criado, apenas necessitamos de obter o inteiro que se encontra na primeira linha, correspondente ao número de vértices (e linhas) que estão nesse mesmo ficheiro. Depois iteramos cada linha, até chegarmos ao número obtido (que corresponde ao final do ficheiro) e para cada linha obtemos 3 valores de vírgula flutuante, correspondentes ao x,y e z de cada vértice, respetivamente. Após obtermos esses 3 pontos criávamos um novo objecto da classe **Ponto** e adicionávamos esse objeto à **Figura** correspondente. No final retornávamos o apontador para a **Figura**.

```
int count;

      string line;

      getline(inputFileStream, line);

count = stoi(line);

for(int i=0; i < count; i++){

            getline(inputFileStream, line);

      stringstream ss(line);
```

```
        vector<float> numbers;
        for(int k = 0; k < 3; k++) {
              string aux;
              ss >> aux;
        float j = stof(aux);
                    numbers.push_back(j);
        }
Ponto *p = new Ponto(numbers.at(0),numbers.at(1),numbers.at(2));
        figura->adicionaPonto(p);
    }
```

5.3.2. Criação do Ficheiro XML

Para criar o ficheiro *XML* decidimos criar um programa que o gerava automaticamente. Decidimos criar este programa devido à cintura de asteroides existente entre marte e saturno, pois seria complicado gerar todos estes asteroides manualmente.

Para gerar o ficheiro criamos uma função **geraFicheiro** que inicialmente cria as *tags scene* e *group* e no final do ficheiro fecha cada uma destas *tags*. Depois chama uma função para gerar cada um dos grupos. Consideramos que cada planteta (com as restantes transformações e objetos do mesmo) seria um grupo, a cintura de asteroides seria outro e o foguetão outro.

.

6. Conclusão

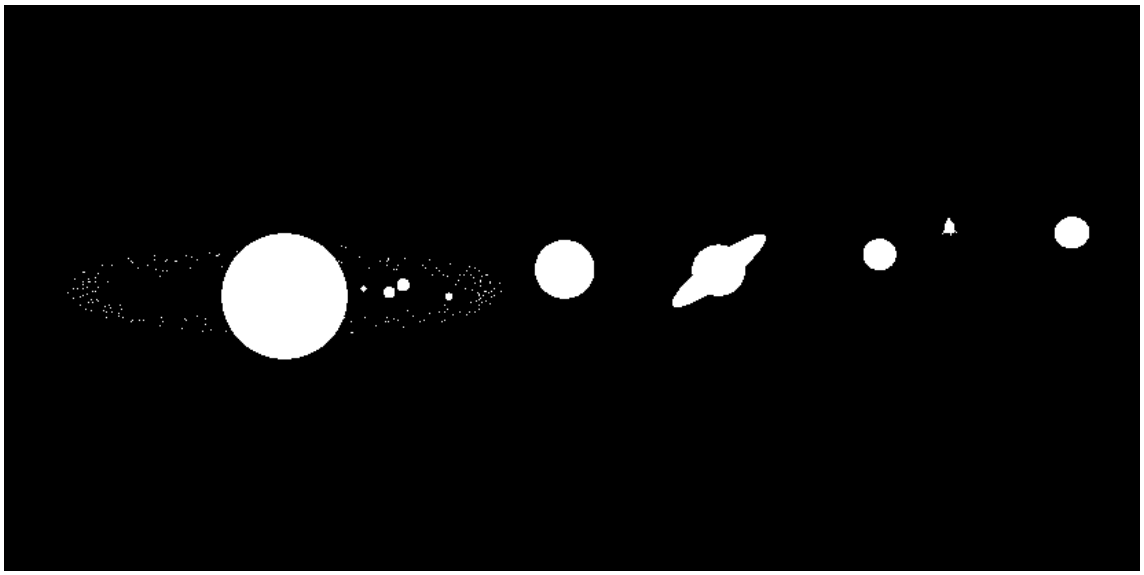
Esta fase do projeto foi concluída sem grandes dificuldades, no entanto pensamos que poderíamos ter adicionado alguns satélites naturais aos planetas, nomeadamente os mais conhecidos. Para além disso, poderíamos ter introduzido alguma noção de diferentes cores aos planetas, mas isso será realizado posteriormente quando iniciarmos a fase das texturas.

Nas próximas fase vamos introduzir *VBO's* por forma a melhorar o desempenho da aplicação. Iremos também tornar o sistema dinâmico, ou seja, irão ser representados os movimentos de rotação e translação dos planetas, bem como de outros intervenientes. Como referido, vamos também trabalhar com texturas por forma a aproximar o modelo à realidade.

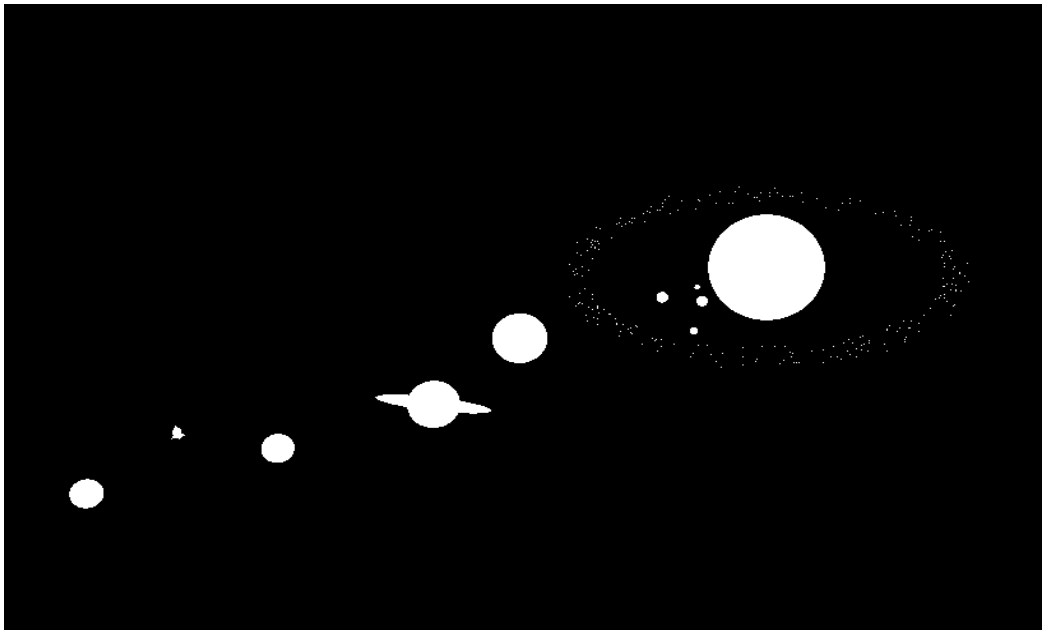
Anexo I

Segue em anexo algumas fotos do nosso sistema solar:

Vista lateral sistema solar:



Vista traseira sistema solar:



Foguetão: