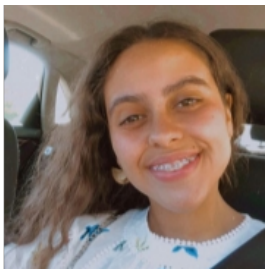
Universidade do Minho
Mestrado Integrado em Engenharia Informática

Computação Gráfica

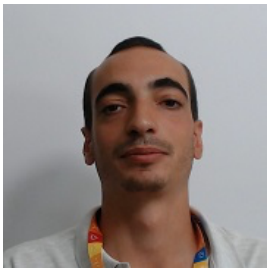
Curvas, superfícies cúbicas e VBOs (Fase 3)

Grupo 24



Ana Afonso (A85762)

Filipe Costa (A85616)



Manuel Carvalho (A69856)

Márcia Teixeira (A80943)

Índice

Introdução

O presente projeto enquadra-se na unidade curricular de Computação Gráfica, na qual foi proposta a conceção e desenvolvimento de um gerador de gráficos 3D (que os guarda em ficheiros) e um leitor dos mesmos que permita a apresentação gráfica, de forma a demonstrar o seu potencial.

Após uma análise detalhada, entende-se que esta primeira fase centra-se na elaboração de duas aplicações. Uma responsável por gerar ficheiros contentores da informação dos modelos (apenas os vértices) e outra que garante a leitura desses ficheiros e, posterior, apresentação gráfica dos mesmos.

A fim de assegurar uma melhor experiência na visualização dos modelos, implementaram-se funções capazes de efetuar translações e rotações.

Numa segunda fase, o principal objetivo assenta no aperfeiçoamento do motor gráfico indicado na primeira fase. Esta melhoria foi conseguida através da especificação e, consequente, interpretação de *scenes* hierárquicas em *xml*.

A terceira fase compreendeu a introdução de animações, superfícies cúbicas e utilização de VBOs. Mais especificamente no *Generator*, foram adaptadas as funcionalidades que já existiam de forma a poder ser lido o ficheiro *teapot.patch.* A tudo isto, foram adicionadas as funções correspondentes para serem efetuados os cálculos e aplicação lógica, permitindo a criação do ficheiro *teapot.3d*.

Fase 1 – Primitivas Gráficas

Numa primeira instância, desenvolveu-se um gerador com o intuito de gerar os vértices das primitivas plano, caixa, esfera e cone. A fim de garantir não só a leitura, mas também a representação das mesmas, tornou-se fundamental a elaboração de um visualizador que fosse responsável pela leitura dos ficheiros 3D concebidos, bem como pela sua ilustração gráfica.

1. Gerador

O gerador é a aplicação independente responsável pela leitura de comandos, criação de modelos 3D e escrita em ficheiro.

Segue-se o esqueleto da classe formada para satisfazer as necessidades do Gerador.

```
class Generator
{
    public:
        std::string* figura;
        std::vector<double> argsFig;
        std::string* fileName;
        bool isValid;

        Generator(int argc,char** argv);
        ~Generator();
        std::string modelo();
        void writeModelo();
};
```

Fig.1. Apresentação da classe *Generator*

O construtor do Gerador recebe como argumentos as variáveis passadas na main() através da linha de comandos, fazendo a sua interpretação e validação.

A aplicação é imediatamente terminada se detetar algum tipo de erro no comando, quer este esteja relacionado com a insuficiência no número de argumentos para a figura pedida, quer a figura seja a errada e/ou ocorra a tipagem de argumentos inválidos.

Relativamente ao nome do ficheiro desejado para escrita é interessante ressaltar que este é o único campo que não passa por qualquer tipo de filtro, possibilitando a liberdade ao utilizador para criar vários modelos para o mesmo tipo de figura, se este assim o desejar.

Sendo um comando válido, este é decomposto em várias componentes, designadamente o nome da figura, os argumentos da figura e o nome do ficheiro de escrita. Posteriormente, estes são interpretados pela função modelo(), que tratará de chamar o construtor para a figura e os argumentos desejados, devolvendo como valores *output,* a *string* correspondente aos pontos todos do modelo.

No que diz respeito, à função writeModelo(), esta é responsável pela chamada da função anterior (construção do modelo) e, consequente, escrita em ficheiro.

2. Primitivas Geométricas

2.1. Plano

Tal como descrito no enunciado, o plano não é mais do que um quadrado centrado na origem e desenhado no plano XoZ, ou seja, com todos os pontos por ele formados com coordenada em y igual a zero. Assim sendo, é imediato concluir que o plano será desenhado à custa de dois triângulos.

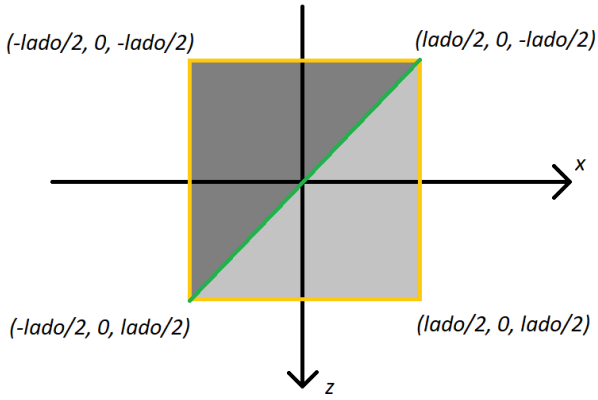


Fig.2. Ilustração esquemática do plano

Assumindo que, "lado" carateriza o tamanho das arestas do plano/quadrado desejado, as coordenadas dos quatro pontos que são, tanto vértices do quadrado, como vértices dos triângulos, estão descritas na imagem acima.

As duas tonalidades de cinzento permitem distinguir os dois triângulos formados, de modo a calcular os pontos correspondentes que constituem o desenho do plano.

2.2. Caixa

A caixa revela ser um paralelepípedo (eventualmente, um cubo, no caso de todas as arestas apresentarem a mesma dimensão). Como tal, será constituída por seis planos que correspondem a doze triângulos (não tendo em consideração as possíveis divisões opcionais).

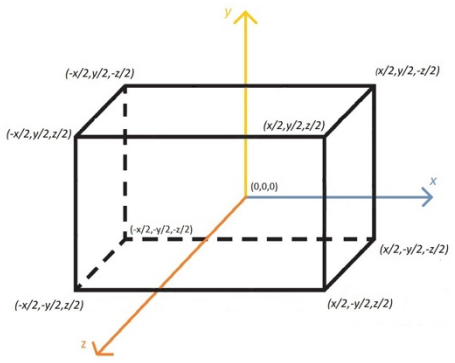


Fig.3. Ilustração esquemática da caixa

As coordenadas de cada vértice podem ser observadas na figura, assumindo que x, y e z são as dimensões pretendidas.

O raciocínio para o desenho dos doze triângulos, constituídos por 36 pontos diferentes, é em tudo semelhante ao raciocínio seguido para o desenho do plano. Contudo, é agora necessário ter em conta que tanto a base, como a face de trás e a face da esquerda (assumindo, o esquema na figura, que respeita a orientação habitual dos eixos e tendo em conta o ponto de vista apresentado) necessitam de ser desenhadas, segundo a regra da mão esquerda, dito de outra forma, seguindo a ordem inversa da regra da mão direita. Caso contrário, estas faces seriam apenas observáveis no interior da caixa.

No que diz respeito às divisões, o desenho da caixa revela um maior grau de compreensão, uma vez que já não se tem um número fixo de triângulos, as coordenadas dos vértices dos triângulos não são tão imediatas de calcular e é essencial ciclos que permitam representar os triângulos segundo qualquer número de divisões pretendidas.

É de realçar que, a interpretação do referido no enunciado leva a crer que no caso de não serem pretendidas quaisquer divisões, o utilizador deverá indicar que pretende zero ou uma divisão. E que, se porventura, pretenda desenhar a caixa dividida em duas partes em cada "direção", deverá indicar que tenciona obter duas divisões (da mesma forma, caso pretenda a caixa constituída por três partes em cada direção, deve indicar três divisões, e assim consecutivamente).

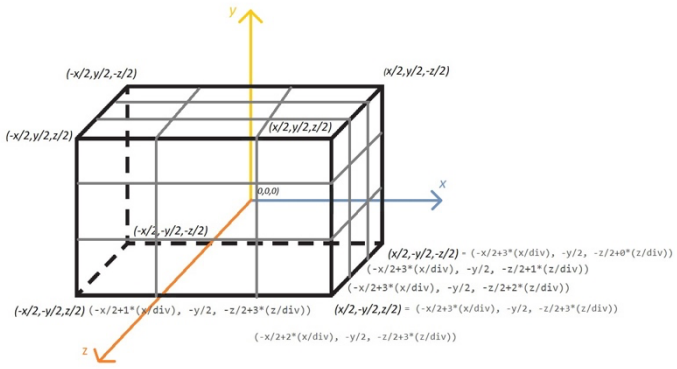


Fig.4. Ilustração esquemática da caixa (com divisões)

Para desenhar a caixa com divisões é fulcral refletir o impacto da variação em x, y e z, entre cada vértice, isto é, quando aumenta/diminui uma determinada coordenada de um vértice de um dos, possivelmente, imensos triângulos, quando se passa para o "próximo" a desenhar. Essa variação é facilmente calculada, uma vez que é "x/nº divisões", "y/nº divisões" e "z/nº divisões", respetivamente.

Como mencionado anteriormente, são necessários dois ciclos aninhados para desenhar todos os triângulos. Esses ciclos começam sempre por desenhar os triângulos que compõem os seis "mini planos" (das seis faces) que têm as coordenadas dos seus pontos menores e, assim o fazem para todos os outros, até desenharem os triângulos com as coordenadas "variáveis" de maior valor.

```
for (int i = 0; i < divisions; i++) {
    float xAtual = (-x / 2) + i * (x / divisions);
    float xAtualSup = (-x / 2) + (i + 1) * (x / divisions);
    float zAtualEsqDir = (-z / 2) + i * (z / divisions);   //no caso da face esquerda e direita, a variação em z, necessita de estar
    float zAtualSupEsqDir = (-z / 2) + (i + 1) * (z / divisions); //num ciclo diferente da variação em y, que se encontra no ciclo mais "interior"
    for (int j = 0; j < divisions; j++) {
        float zAtual = (-z / 2) + j * (z / divisions);
        float zAtualSup = (-z / 2) + (j + 1) * (z / divisions);
        float yAtual = (-y / 2) + j * (y / divisions);
        float yAtualSup = (-y / 2) + (j + 1) * (y / divisions);

        //base:
        s.append(to_string(xAtual) + " " + to_string(-hMax) + " " + to_string(zAtual) + "\n");
        s.append(to_string(xAtualSup) + " " + to_string(-hMax) + " " + to_string(zAtual) + "\n");
        s.append(to_string(xAtualSup) + " " + to_string(-hMax) + " " + to_string(zAtualSup) + "\n");

        s.append(to_string(xAtual) + " " + to_string(-hMax) + " " + to_string(zAtual) + "\n");
        s.append(to_string(xAtualSup) + " " + to_string(-hMax) + " " + to_string(zAtualSup) + "\n");
        s.append(to_string(xAtual) + " " + to_string(-hMax) + " " + to_string(zAtualSup) + "\n");
```

Fig.5. Código representativo da explicação supramencionada

De modo a facilitar a compreensão do raciocínio efetuado, sugere-se a observação de como deve ser desenhada a base da caixa, isto é, o plano com y constante igual a -y/2, com três divisões (neste caso quadrada, apesar do comprimento das arestas ser indiferente). É essencial notar que a ordem de desenho dos mini planos não é, tendo em conta a totalidade da caixa, a explicitada na imagem abaixo, esta refere-se apenas à ordem de desenho dos mini planos da base da caixa, uma vez que em cada iteração do ciclo interior são, na verdade, desenhados seis mini planos, um em cada face.

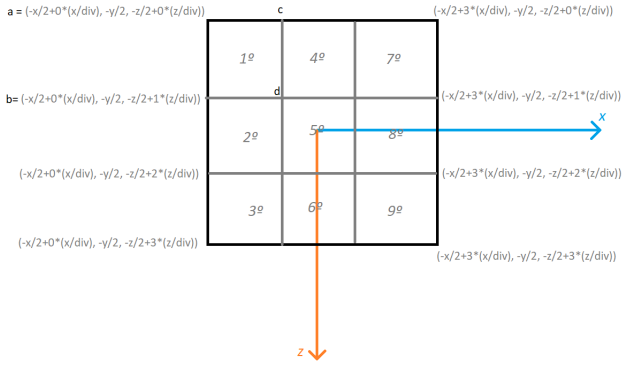


Fig.6. Ilustração explicativa da construção da base da caixa

Como é possível observar na imagem acima, e tendo em conta, apenas e só, o desenho da base da caixa, primeiramente desenha-se o mini plano, de menor valor de x e z (o mini plano com "1º" no interior). Tanto o valor de i (relativo no desenho da base, ao valor em x) e j (neste caso, relativo ao valor de z) de ambos os ciclos é ainda igual a zero e, portanto, tem-se como coordenadas dos dois triângulos que compõem esse mini plano as seguintes apresentadas:

a = (-x/2+i*(x/div), -y/2, -z/2+j*(z/div)) = (-x/2+0*(x/div), -y/2, -z/2+0*(z/div))

b = (-x/2+(i+1)*(x/div), -y/2, -z/2+j*(z/div)) = (-x/2+1*(x/div), -y/2, -z/2+0*(z/div))

c = (-x/2+i*(x/div), -y/2, -z/2+(j+1)*(z/div)) = (-x/2+0*(x/div), -y/2, -z/2+1*(z/div))

d = (-x/2+(i+1)*(x/div), -y/2, -z/2+(j+1)*(z/div)) = (-x/2+1*(x/div), -y/2, -z/2+1*(z/div))

Posteriormente (já com seis mini planos desenhados, um em cada uma das seis faces), são traçados todos os mini planos em que o valor de i se mantém igual e, portanto, neste situação, os valores de x mantêm-se inalterados e o valor de j vai sendo incrementado. Neste caso, com os três primeiros mini planos da base desenhados, incrementa-se o valor de i e volta-se a ter j igual a 0. E desenham-se mais três mini planos da base. O raciocínio é semelhante para os restantes mini planos (ainda por desenhar) sejam relativos a três ou mais divisões.

2.3. Esfera

A esfera tem como base a sua própria classe Esfera e foi elaborada com recurso a uma classe adicional gerada, Triangle.

Esta primitiva encontra-se centrada na origem e recebe como valores de *input* o raio, o número de *slices* e as *stacks.* Para além do referido, é constituída pelas variáveis e métodos de instância presentes na figura abaixo.

A estratégia deliberadamente estruturada para o desenvolvimento da esfera passou pelo cálculo dos pontos de interseção do plano YoZ com as *stacks*, seguido do cálculo de todos os pontos recorrendo a rotações a partir dos pontos obtidos anteriormente e, por fim, procedeu-se à construção dos triângulos.

```
typedef class Esfera
{
    private:
        double radius;
        int slices, stacks;

        std::vector<PPoint3D>* getYoZStackInterceptionPoints();
        PPoint3D rotatePoint_Yaxis(PPoint3D p,double angle);
        std::vector<std::vector<PPoint3D>*> getVerticalLinePoints();
        std::vector<PTriangle>* getSliceTriangles(std::vector<PPoint3D>* s1,std::vector<PPoint3D>* s2);
        void freeTriangles(std::vector<std::vector<PTriangle>*>& vec);

    public:
        Esfera(double r,int slices,int stacks);
        ~Esfera();

        double getRadius();
        int getSlices();
        int getStacks();
        double getZCoord(double x,double y);
        std::vector<std::vector<PTriangle>*> getTriangles();
        std::string toString();

}* PEsfera;
```

Fig.7. Apresentação de código referente à classe *Esfera*

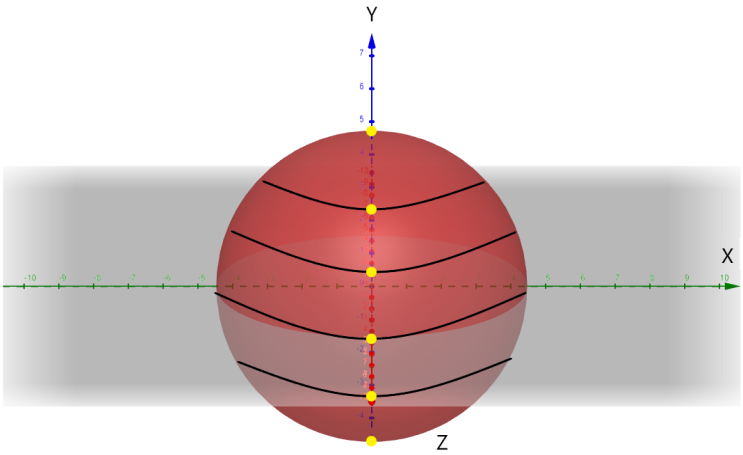


Fig.8. Ilustração esquemática da esfera

O cálculo destes primeiros pontos é crucial para a metodologia definida, visto que estes funcionarão como um pivot na obtenção de todos os outros.

A escolha da ordem é pouco relevante, no entanto foi convencionado começar no topo da esfera caminhando até à sua base. Assim sendo, para cada ponto, a componente x será sempre zero e y começará com o valor do raio da esfera, decrementado a cada iteração por "diâmetro/nº *stacks*", atingindo o valor mínimo de "-raio". Já a componente z é obtida com a equação geométrica da esfera, dando como argumento os valores de x e y.

```
double getZCoord(double x,double y);
std::vector<PPoint3D>* getYoZStackInterceptionPoints();
```

Fig.9. Representação de código referente a funções auxiliares usadas nos cálculos descritos acima

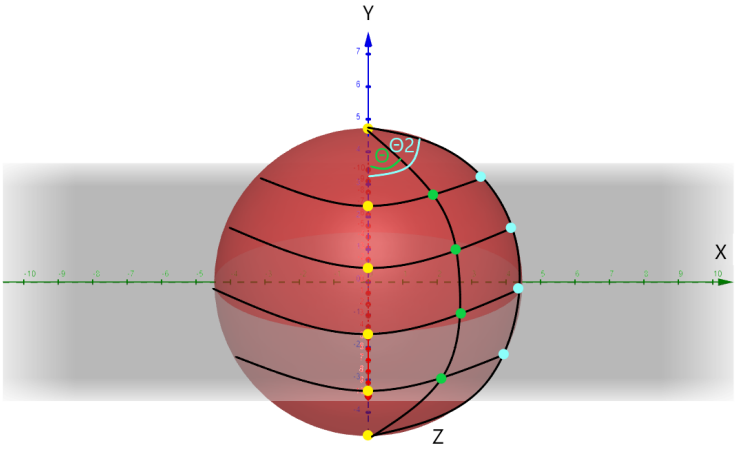


Fig.10. Ilustração esquemática da esfera (orientada a rotações)

Como refe arelo na figura acima) para o cálculo de todos os outros. A ordem dos pontos será linha a linha, na vertical, do topo até á base.

Para tal, foi essencial recorrer a rotações, assumindo cada linha da *stack* como uma circunferência, em que o valor do ângulo será incrementado a cada iteração e com raio igual à componente z do ponto inicial (amarelo).

O valor do ângulo de rotação será inicialmente "360º/nº *slices*" e incrementado por esse mesmo valor.

```
std::vector<std::vector<PPoint3D>*> getVerticalLinePoints();
```

Fig.11. Representação da função que possibilita os cálculos

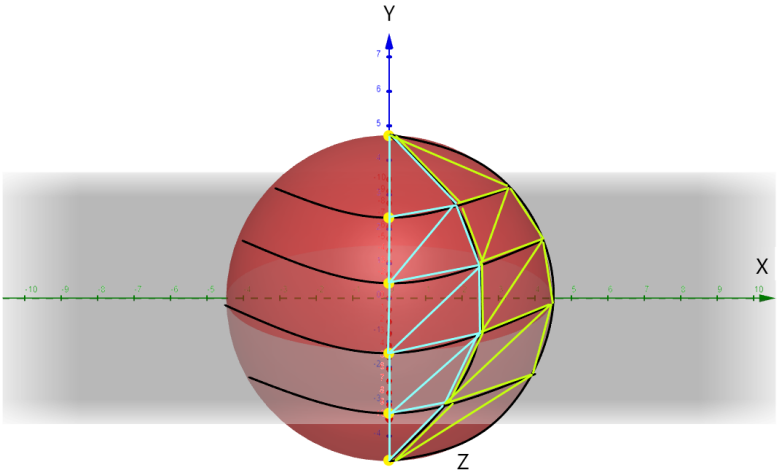


Fig.12. Ilustração esquemática da esfera
(orientada à construção dos triângulos)

Após o cálculo de todos os pontos foram definidas duas funções apresentadas de seguida.

```
std::vector<PTriangle>* getSliceTriangles(std::vector<PPoint3D>* s1,std::vector<PPoint3D>* s2);
std::vector<std::vector<PTriangle>*> getTriangles();
```

Fig.13. Representação de duas funções responsáveis pelo cálculo de todos os pontos

A primeira é uma função de auxílio, que recebe como argumento dois vetores de pontos, sendo que cada um representa as linhas verticais de fronteira de uma *slice* e constrói os triângulos no sentido contrarrelógio (regra da mão direita).
A função getTriangles() aplica a primeira a todas as *slices*, obtendo no final um vetor com todos triângulos para cada *slice*.

2.4. Cone

O fio condutor de raciocínio que conduziu à conceção do cone assenta na compreensão que o mesmo é constituído pela base e pela face lateral. Desta forma, faseou-se a construção desta primitiva geométrica em duas partes.
Primeiramente, tornou-se imprescindível entender o papel/conceito das *slices* na elaboração da base. Posteriormente, procedeu-se à implementação da base recorrendo a um ciclo for, que garante o desenho das várias *slices* essenciais para a composição da base.
De seguida, apresenta-se uma esquematização passível de facilitar a perceção do pensamento arquitetado.

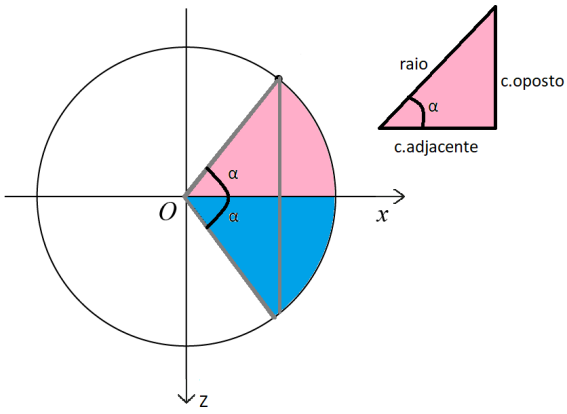


Fig.14. Ilustração explicativa da construção da base do cone (comportando trigonometria)

Através da figura acima é possível deduzir as seguintes fórmulas:

- $\alpha = (2\pi/slices) * i$ corresponde ao alfa representado na figura. Ao resultado proveniente do α tem-se ainda que aumentar em incrementos equivalentes a $2\pi/slices$, a fim de garantir que são abrangidos todos os ângulos da base circular.

- $catadj = \cos\alpha * raio$ obtém-se a partir da transformação das coordenadas polares em cartesianas. Através do recurso a fórmulas trigonométricas como $\cos\alpha = \frac{cateto\ adjacente}{hipotenusa}$.

- $\boxed{catopt = \sin\alpha * raio}$ obtém-se a partir da transformação das coordenadas polares em cartesianas. Através do recurso a fórmulas trigonométricas como $\sin\alpha = \frac{cateto\ oposto}{hipotenusa}$.

- $\boxed{nextcatadj = \cos(\alpha + 2\pi/slices) * raio}$ obtém-se a partir da transformação das coordenadas polares em cartesianas. Através do recurso a fórmulas trigonométricas como $\cos\alpha = \frac{cateto\ adjacente}{hipotenusa}$. Porém, utiliza-se o ângulo seguinte para calcular o terceiro ponto, uma vez que este é partilhado com o triângulo seguinte e o cálculo é assim mais acessível.

- $\boxed{nextcatopt = \sin(\alpha + 2\pi/slices) * raio}$ obtém-se a partir da transformação das coordenadas polares em cartesianas. Através do recurso a fórmulas trigonométricas como $\sin\alpha = \frac{cateto\ oposto}{hipotenusa}$. Porém, utiliza-se o ângulo seguinte para calcular o terceiro ponto, uma vez que este é partilhado com o triângulo seguinte e o cálculo é assim mais acessível.

Relativamente à construção da face lateral, o pensamento permaneceu similar. Todavia, foi fundamental a introdução de um novo conceito, as *stacks*.

Deste modo, fizeram-se dois ciclos for aninhados, um para controlar as *stacks* (isto é, vai subindo de *stack* em *stack*) e o outro que é praticamente idêntico ao ciclo realizado para a base.

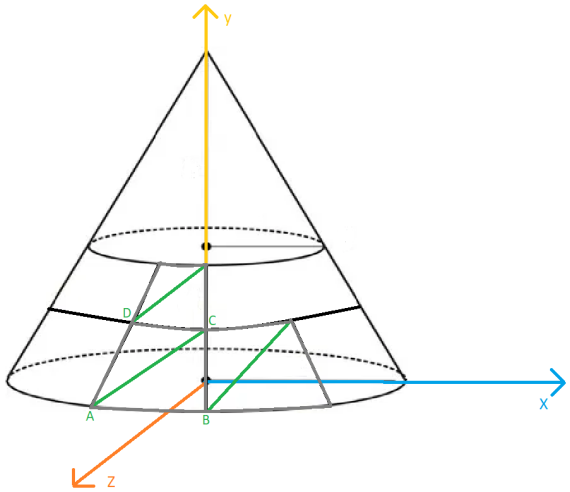


Fig.15. Ilustração explicativa da construção da face lateral do cone

É de salientar ainda que no for exterior é tratado o tamanho do raio mediante a *stack* onde se encontra (à medida que se vai subindo de *stack*, o raio vai diminuindo) e calculada a altura dessa mesma *stack*. Já o for interior garante que em cada iteração são desenhados dois triângulos, o primeiro é constituído por dois pontos que são obtidos de forma igual aos dois pontos da base (A e B), que não o ponto do centro (com coordenadas x=0, y=0 e z=0) e com a altura apropriada à *stack*. O ponto C é calculado de maneira idêntica ao ponto que utiliza o ângulo seguinte (ponto B), porém com o raio e a altura da *stack* seguinte. O segundo triângulo desenhado pelo ciclo interior apresenta dois pontos em comum com o anterior (sendo estes, o ponto A e C) e gera um quarto ponto novo (o D) com o x e z do ponto A e o y do ponto C.

Para o cume do cone, a estratégia abordada centra-se no desenho de um triângulo com os pontos A e B e outro em tudo equivalente à origem (x=0 e z=0), mas com altura máxima.

As fórmulas utilizadas na face lateral constituem um acréscimo às mencionadas e explicadas à priori, assim sendo:

- $raio1 = raio - \left(\left(\frac{raio}{stacks}\right) * i\right)$ deduz-se que o cálculo do raio para a *stack* que está a ser desenhada é feito com recurso à subtração do raio total por uma fração do raio que vai ser aumentada de *stack* em *stack.*

- $raio2 = raio - (\left(\frac{raio}{stacks}\right) * (i + 1))$ deduz-se que o cálculo é equivalente ao referido acima, no entanto a fração de raio a retirar é da *stack* superior.

- $altura1 = \left(\frac{altura}{stacks}\right) * i$ este cálculo é usado de modo a conhecer a altura de cada *stack,* sendo este aumentado a cada iteração.

- $\alpha = (2\pi/slices) * j$

- $catadj = \cos\alpha * raio1$

- $catopt = \sin\alpha * raio1$

- $nextcatadj = \cos(\alpha + 2\pi/slices) * raio1$

- $nextcatopt = \sin(\alpha + 2\pi/slices) * raio1$

- $cadj = \cos\alpha * raio2$

- $copt = \sin\alpha * raio2$

- $nextcadj = \cos(\alpha + 2\pi/slices) * raio2$

- $nextcopt = \sin(\alpha + 2\pi/slices) * raio2$

2.5. Classes auxiliares

```
typedef class Point3D
{
    public:
        double x,y,z;

        Point3D();
        Point3D(double x,double y,double z);
        Point3D(Point3D *p);
        ~Point3D();

        std::string toString();

}* PPoint3D;
```

Fig.16. Representação de um *Ponto* em três dimensões

```
typedef class Triangle
{
    private:
        std::vector<PPoint3D> points; //ordem de desenho index 0->1->2

    public:
        Triangle(double x1,double y1,double z1,double x2,double y2,double z2,double x3,double y3,double z3);
        Triangle(PPoint3D p1,PPoint3D p2,PPoint3D p3);
        Triangle(Triangle* t);
        ~Triangle();

        PPoint3D getPoint(int i);
        void setPoint(int i,PPoint3D p);
        std::string toString();

}* PTriangle;
```

Fig.17. Representação de um triângulo como um conjunto de três pontos

3. Visualizador

O *Visualizator* é a aplicação responsável por receber o ficheiro de configuração de uma *scene* em XML com todos os ficheiros que contêm os modelos a serem carregados para serem gerados através do *OpenGL*.

Cada ficheiro com os modelos presentes no ficheiro de configuração é analisado pelo programa, para dele serem extraídos os vários pontos que são posteriormente adicionados a uma estrutura de dados.

Esses pontos são depois lidos dessa mesma estrutura pelo render do *OpenGL*.

De forma a complementar o projeto, foram implementadas algumas funcionalidades que permitem a rotação e movimentação das figuras, nomeadamente a alteração da composição da figura (preenchimento da cor, linhas e pontos).

configuration.xml

```
<scene>
    <model file= "cone.3d" />
</scene>
```

plane.3d

```
6
1.500000 0.000000 1.500000
1.500000 0.000000 -1.500000
-1.500000 0.000000 1.500000
1.500000 0.000000 -1.500000
-1.500000 0.000000 -1.500000
-1.500000 0.000000 1.500000
```

Fig.18. Demonstração de um exemplo da configuração de um ficheiro 3D

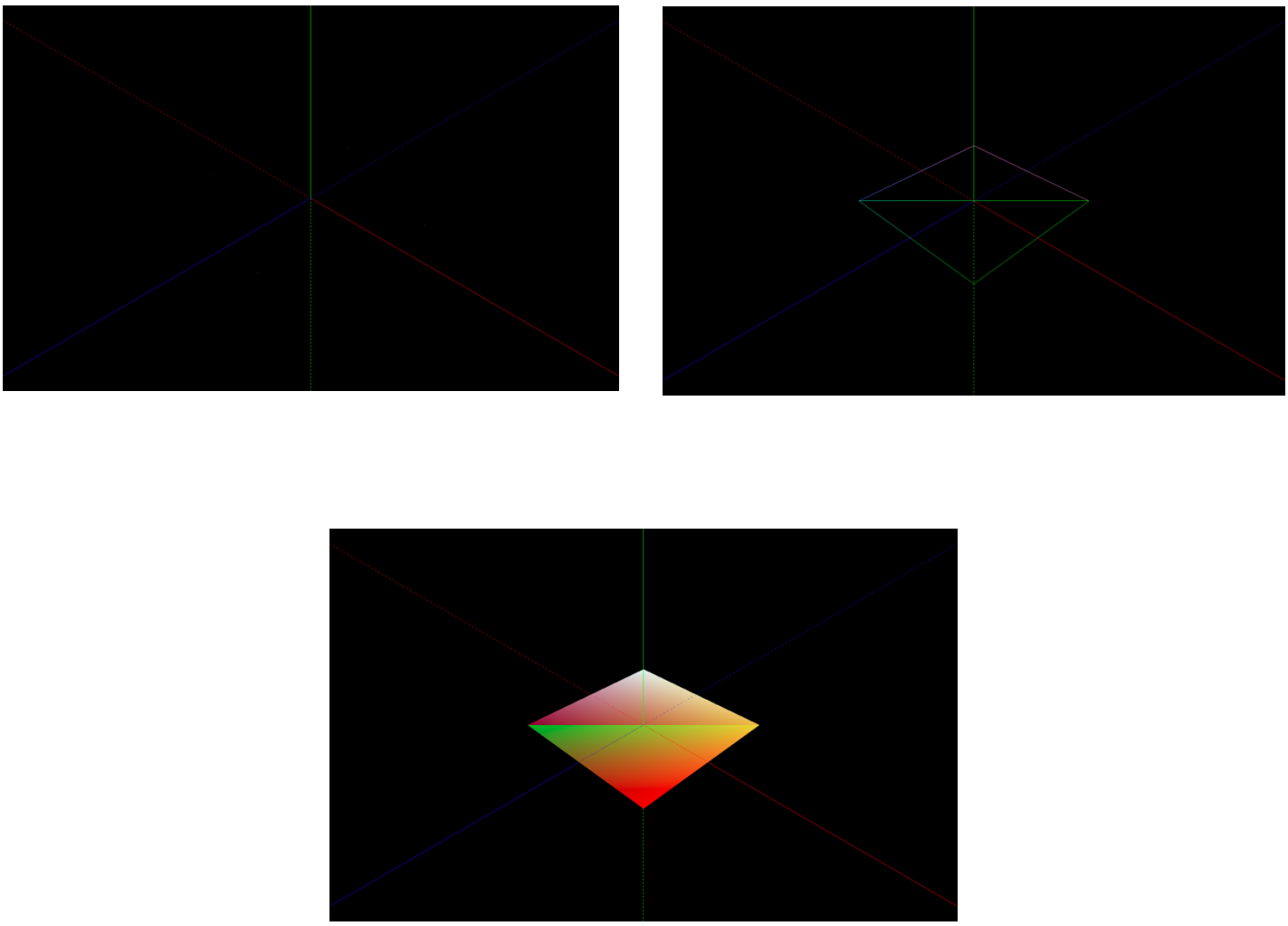


Fig.19. Representação gráfica visualmente mais apelativa do plano

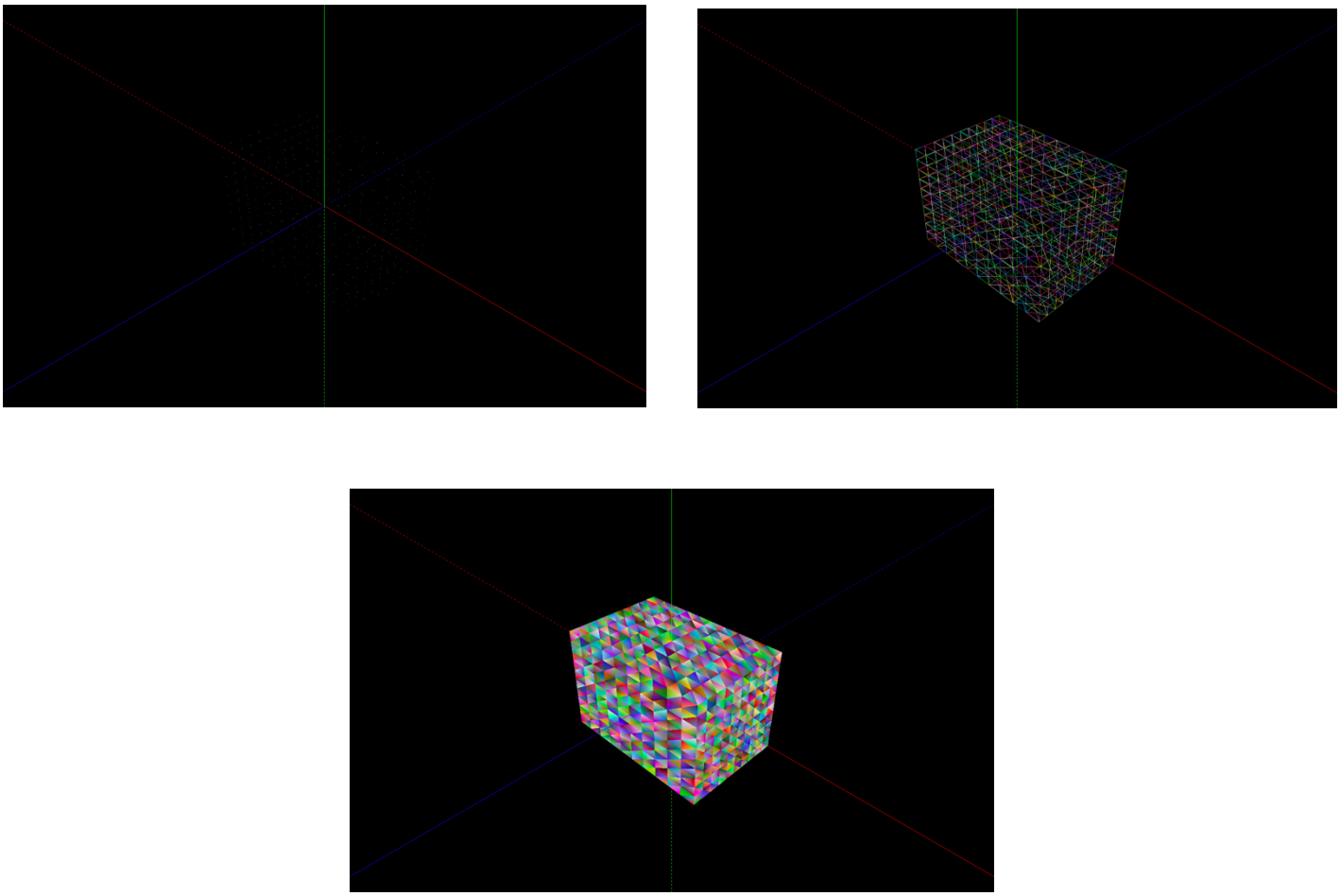


Fig.20. Representação gráfica visualmente mais apelativa da caixa

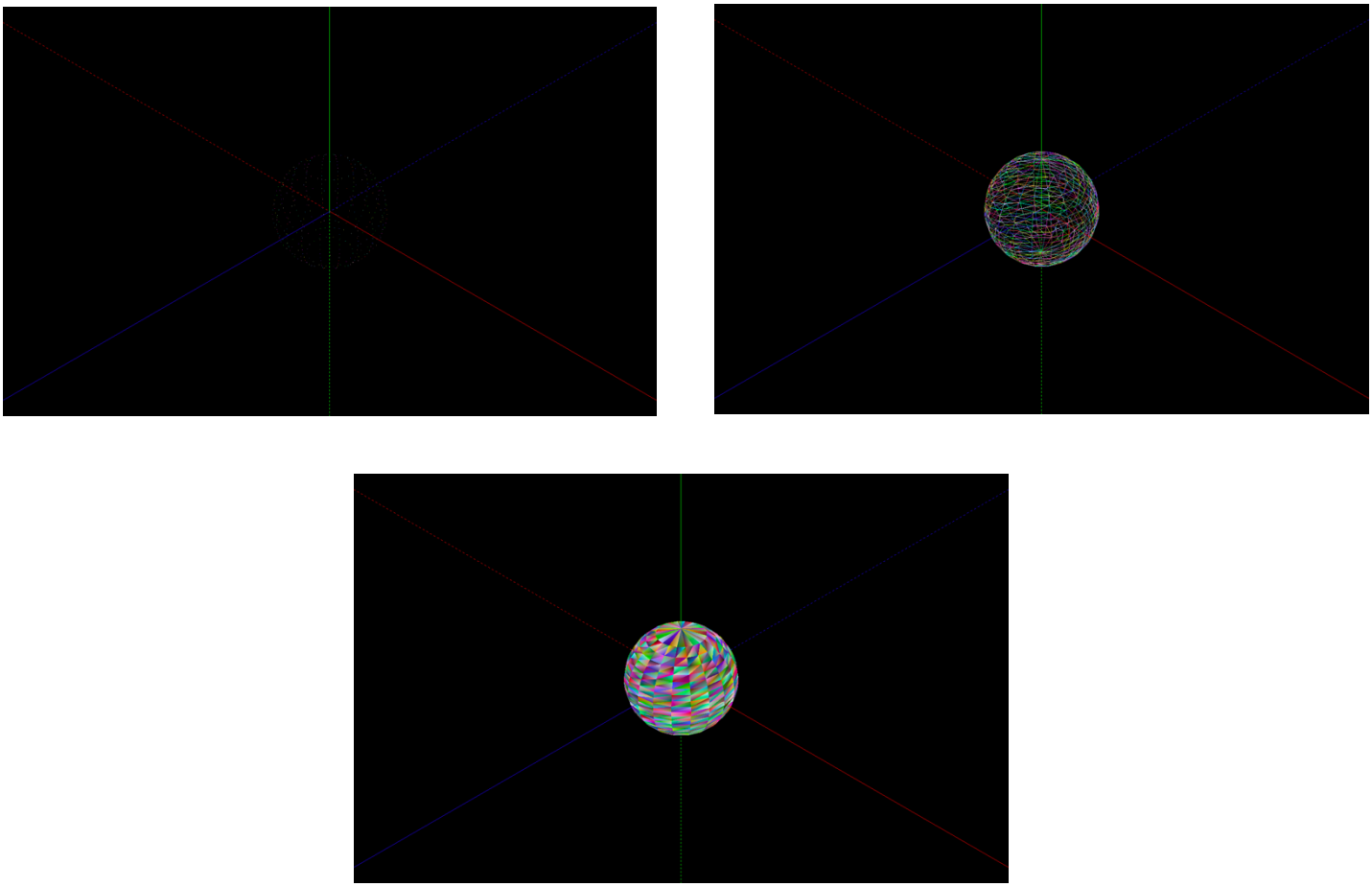


Fig.21. Representação gráfica visualmente mais apelativa da esfera

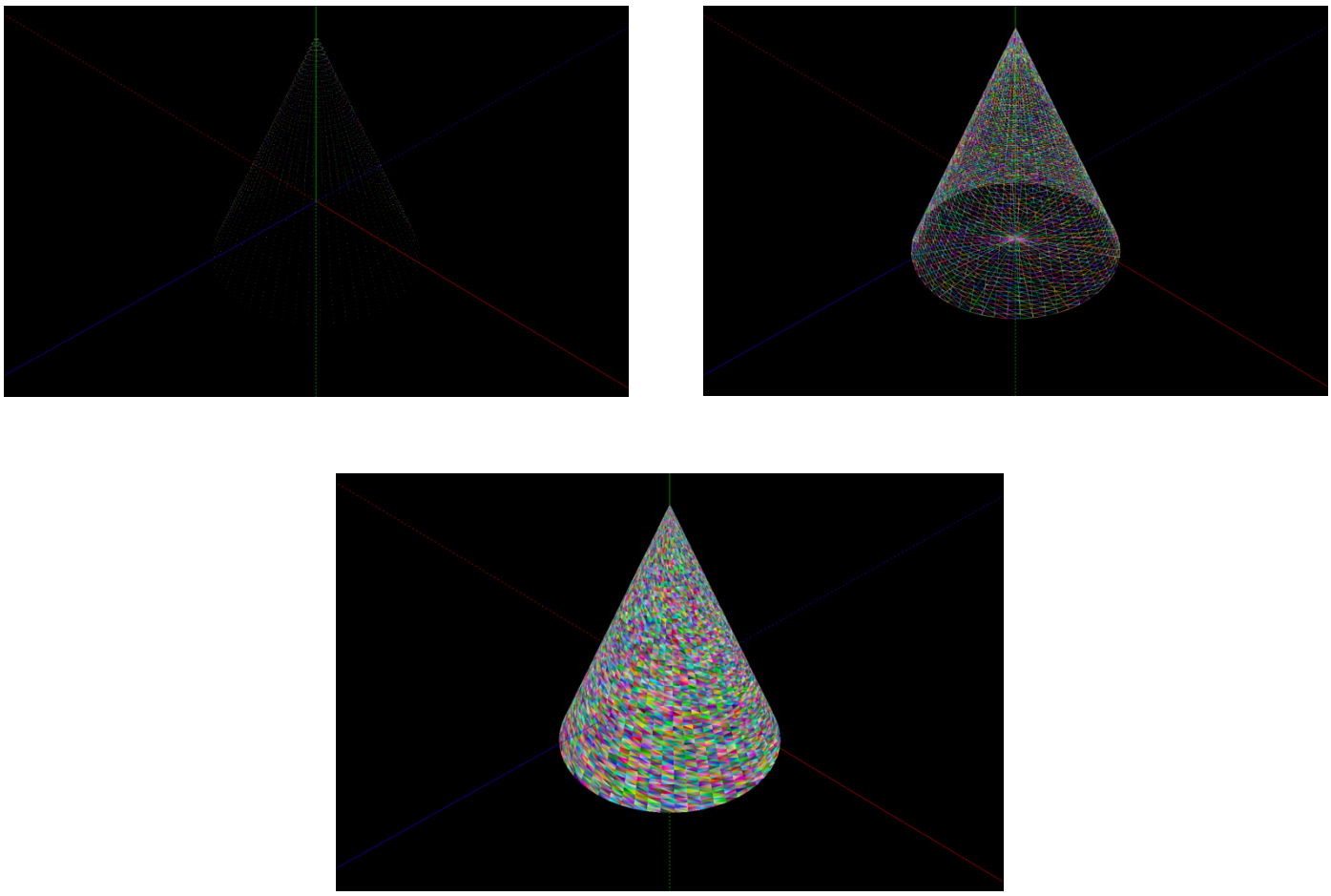


Fig.22. Representação gráfica visualmente mais apelativa do cone

Fase 2 – Transformações Geométricas

Primeiramente, introduziu-se a criação de *scenes* hierárquicas utilizando as transformações geométricas (*translate, rotate and scale*), ambas definidas num ficheiro *xml* que deve ser processado pelo programa. Para tal, adaptou-se o motor gráfico existente, de modo a que fosse possível a aplicação de transformações geométricas, bem como à atribuição de cores distintas aos diferentes modelos.

De seguida, foi tomada a decisão de criar estruturas de dados para armazenar as transformações geométricas e respetivos modelos. Posteriormente, essas transformações serão aplicadas hierarquicamente aos modelos.

Desta forma, e a título de exemplo, o visualizador permitirá a apresentação do desejado Sistema Solar.

1. Estrutura do ficheiro de configuração *xml*

A estrutura do ficheiro *configuration.xml* é constituída por:

- **<scene>** Define o início de uma *scene* e não apresenta qualquer atributo.
- **<group>** Demarca um conjunto de transformações a ser aplicadas a *models* e possíveis subgrupos nele incluídos.
- **<models>** e **<model>** Representam o conjunto de *models* e um *model*, respetivamente. O **<models>** alberga uma lista de **<model>**. Este, por sua vez, contém um atributo file e três atributos (r, g, b) que definem as cores.
- **<translate>** Contém três atributos referentes às coordenadas a ser aplicadas.
- **<rotate>** Contém quatro atributos referentes ao ângulo de rotação e às coordenadas a ser aplicadas.
- **<scale>** Contém três atributos referentes aos multiplicadores a serem aplicados às coordenadas.

2. Estruturas de dados

De modo a melhorar a implementação realizada na primeira fase, incluindo as novas funcionalidades, tomou-se a decisão de criar estruturas de dados contentoras das diversas informações consideradas pertinentes.

Segue-se uma explicação detalhada de todas e cada uma das estruturas desenvolvidas:

- **struct point:** Esta estrutura foi imprescindível, visto que reúne todas as coordenadas necessárias para a constituição de um ponto.
- **struct rotate:** Esta estrutura foi criada com o objetivo de armazenar não só todas as coordenadas (sob a forma da **struct point**), mas também o ângulo de rotação.
- **struct color:** Esta estrutura foi capaz de facilitar o processamento de cores no formato RGB, armazenando cada um dos valores separadamente.
- **struct model:** Esta estrutura é constituída tanto por um conjunto de pontos como pela cor em que será representada graficamente. Como tal, decidiu-se que a mesma contivesse um vetor de **struct point** e a **struct color.**
- **struct action:** Esta estrutura alberga todas as ações que possam existir no conjunto *group,* incluindo um nome da ação e possíveis subgrupos (**struct group**).
- **struct group:** Esta estrutura contém todas as ações passíveis de existirem num *group,* reunidas num vetor.
- **struct config:** Esta estrutura reúne todos os grupos existentes no ficheiro de configuração num vetor.

3. Processamento do ficheiro de configuração xml

Com o principal objetivo de facilitar o processo de *parsing*, separou-se a leitura do ficheiro de configuração nos possíveis grupos nele existentes. Para tal, criou-se uma função que permitisse a leitura inicial do ficheiro *xml*, intitulada de *readConfig*, esta é responsável por entrar em cada *scene* e respetivos *groups*.

Posteriormente, é chamada a função *readGroups,* que se foca no *parsing* do conteúdo armazenado nas tags *<groups>*. É nesta função que é feita uma passagem por todas as linhas existentes dentro do *groups*, verificando em que ramo se encaixa cada uma. No caso de se tratar de *models* é chamada a função *readModels*, que distingue os *translates*, *rotates* e/ou *scales* e, mediante o tipo de ação, são processadas as funções *readTranslate*, *readRotate* e *readScale*, respetivamente. E consequentemente, adicionados às *actions* que serão aplicadas à posteriori. Já caso se trate de outro *group* (subgrupo ou grupo independente), a própria função é chamada recursivamente.

A função *readModels* seleciona os atributos e chama a função *readModel,* que invoca as funções *readFile* e *readColor* para processar individualmente um ficheiro .3d, bem como as cores a serem aplicadas à representação gráfica.

Por sua vez, a função *readFile* lê cada linha do ficheiro .3d guardando os pontos que constituem o modelo numa *action*.

Relativamente às funções *readColor*, *readScale*, *readRotate* e *readTranslate,* estas apresentam um funcionamento semelhante, dado que todas passam pela leitura dos atributos que lhe são correspondentes e o seu armazenamento também é feito numa *action*.

4. Representação gráfica e *Motion Keys*

Partindo dos resultados reunidos pelas funções de *parsing*, o passo seguinte será então a representação gráfica das figuras correspondentes. Deste modo, e utilizando a função *renderScene* que, por sua vez, desenha o referencial e chama a função *drawScene*, é feita uma passagem iterativa por todos os grupos existentes neste ponto na **struct config.** Para cada um é chamada a função *drawGroup* que itera sobre as ações e processa-as, com recurso à função *drawAction*.

A função *drawAction*, tendo um funcionamento semelhante à *readGroups*, verifica se é um *model* (para o qual chama a função *drawModel*), um *translate*, *rotate* ou *scale* (chamando as funções *drawRotate*, *drawTranslate* e *drawScale*, respetivamente).

Por fim, são utilizadas as funções pré-definidas de *OpenGL* com o intuito de efetuar cada uma das ações supramencionadas.

Com principal foco de auxiliar no processo de visualização do resultado gráfico foram implementadas, para além das *keys* já existentes, as teclas para *camera motion*.

- **w e f:** movimentação da câmera para a esquerda e para a direita
- **z e x:** zoom-in e zoom-out da câmera
- **c, l e p:** alterar a figura para que apareça sob a forma de figura colorida, linhas e pontos
- **Seta direita:** rotação da câmera para a direita
- **Seta esquerda:** rotação da câmera para a esquerda
- **Seta para cima:** rotação da câmera para cima
- **Seta para baixo:** rotação da câmera para baixo

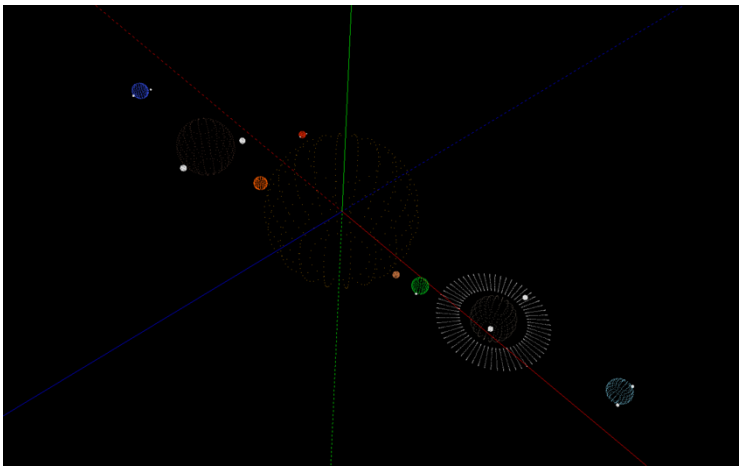
5. Sistema Solar

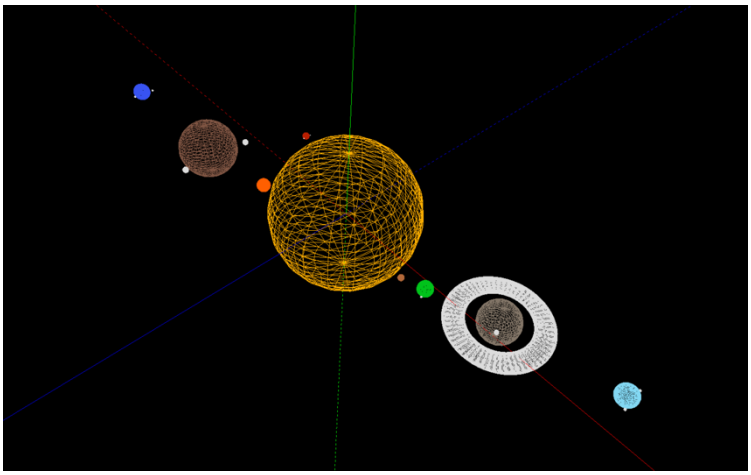
Como é de conhecimento geral, o sistema solar é constituído pelo sol e oito planetas, alguns acompanhados de uma ou mais luas. A título de exemplo foram apenas representadas no máximo duas luas por planeta.
Os planetas foram representados da forma mais próxima possível à escala (levianamente seguida), à exceção dos planetas gasosos, que dada a sua dimensão em comparação com os restantes planetas, não permitiriam que o sistema solar fosse completamente visualizado mantendo a capacidade de observar com qualidade e facilidade os planetas mais pequenos.
As distâncias entre os planetas foram calculadas manualmente de modo a que não ocorressem colisões entre os mesmos e as suas posições foram escolhidas aleatoriamente, permitindo a apresentação de uma imagem mais apelativa do sistema solar como um todo.

Como é possível a partir de uma observação atenta das figuras abaixo, Saturno apresenta um anel em seu redor, tal como acontece na realidade espacial. Para tal, foi fundamental delinear uma estratégia que conduzisse até à conceção desse anel.

Assim sendo, o desenho do anel assenta num raciocínio semelhante ao do cone, utilizando dois ciclos for (um para o cálculo das *slices* e outro para o cálculo das várias *stacks*). Todavia, neste caso, as *stacks* são calculadas da mesma forma que as *slices*, ou seja, a altura é calculada através da divisão de 2π pelas *stacks*, sendo esta aumentada em cada iteração do ciclo.

No que concerne aos raios, para a altura atual e para a seguinte, fazem-se de igual forma, porém no final, é subtraído o raio interno para criar o espaço onde vai permanecer o planeta.





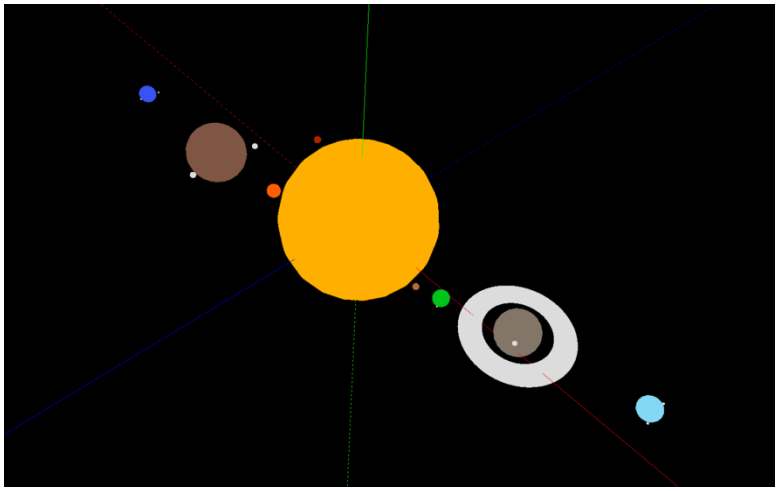


Fig.23. Representação gráfica visualmente mais apelativa do sistema solar

Fase 3 – Curvas, Superfícies cúbicas e VBOs

1. Gerador

Primeiramente, antes de qualquer modificação no gerador, impõem-se como fundamental dominar algum conhecimento subjacente a *patches* e curvas de *Bezier*. Desta forma, inicia-se este procedimento de transformação faseada pelo esclarecimento de que as *patches* de *Bezier* permitem a construção de *meshes* mais suaves com um reduzido número de pontos, quando comparada a simples *meshes* de triângulos. Contudo, devido a uma maior complexidade de cálculo, apresentam penalizações a nível de performance. Tal como acontece com as curvas de *Bezier*, estas também são definidas por um conjunto de pontos de controlo.

Para esta terceira fase o desafio proposto passou pela inclusão de uma nova funcionalidade ao *Generator*, com principal foco na geração de modelos baseados em *patches Bezier*. Para tal, é fornecido ao gerador (via linha de comandos) um ficheiro *PATCH* e o nível de tesselação desejado. Por sua vez, este ficheiro conterá a quantidade de *patches* presentes, os seus pontos de controlo, o número total de pontos e as coordenadas dos mesmos.
O objetivo do *Generator*, para esta etapa, consiste essencialmente na leitura do ficheiro input, na geração dos pontos da superfície mediante a tesselação, seguida da sua escrita em ficheiro num formato adequado à leitura do visualizador. Tudo isto tem, particularmente como base de implementação a classe ***Surface***.

A fim de proporcionar uma melhor compreensão da estrutura descrita acima, apresenta-se de seguida um possível exemplo para a sua representação.

```
2 <- number of patches
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
3, 16, 17, 18, 7, 19, 20, 21, 11, 22, 23, 24, 15, 25, 26, 27
28 <- number of control points
1.4, 0, 2.4 <- control point 0
1.4, -0.784, 2.4 <- control point 1
0.784, -1.4, 2.4 <- control point 2
0, -1.4, 2.4
1.3375, 0, 2.53125
1.3375, -0.749, 2.53125
0.749, -1.3375, 2.53125
0, -1.3375, 2.53125
1.4375, 0, 2.53125
1.4375, -0.805, 2.53125
0.805, -1.4375, 2.53125
0, -1.4375, 2.53125
1.5, 0, 2.4
1.5, -0.84, 2.4
0.84, -1.5, 2.4
0, -1.5, 2.4
-0.784, -1.4, 2.4
-1.4, -0.784, 2.4
-1.4, 0, 2.4
-0.749, -1.3375, 2.53125
-1.3375, -0.749, 2.53125
-1.3375, 0, 2.53125
-0.805, -1.4375, 2.53125
-1.4375, -0.805, 2.53125
-1.4375, 0, 2.53125
-0.84, -1.5, 2.4 <- control point 26
-1.5, -0.84, 2.4 <- control point 27
```

indices for the first patch

indices for the second patch

Fig.24. Representação exemplificativa da estrutura descrita

1.1. Operações aritméticas de matrizes, pontos e fórmula de *Bezier*

A interpretação da fórmula presente na figura seguinte resulta na dedução que de que esta consistia em sucessivas multiplicações de matrizes, fator este que conduziu à implementação de uma função capaz de executar esta operação. É de salientar ainda que a transposta da matriz M é igual a ela própria e que ambas as variáveis *u* e *v* apenas tomam valores racionais compreendidos entre 0 (zero) e 1 (um).

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

Fig.25. Equação para o cálculo do ponto numa *patch Bezier*

$$\mathbf{M} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Fig.26. Representação da matriz de constantes *Bezier*

```
template<typename T>
vector<vector<T>> Surface::multiplyMatrices(const vector<vector<T>> &m1, const vector<vector<T>> &m2)
```

Apesar desta assinatura abstrata permitir a multiplicação de matrizes de qualquer tipo, irá ser usada somente no contexto de *Point3D*, que é a classe que representa pontos no espaço.

A próxima questão que surge é como multiplicar matrizes de constantes com matrizes de pontos. Para resolver esse problema, tomou-se a decisão de não ter matrizes de constantes, mas sim transformá-las em matrizes de pontos, nas quais ambas as componentes de X, Y e Z tomam o mesmo valor da constante em questão, visto que a multiplicação de um ponto por um escalar representa a multiplicação de cada coordenada por esse valor.

De modo a facilitar operações algébricas entre pontos foram também acrescentados *overloads* de operadores à classe *Point3D*.

Assim, temos as ferramentas necessárias para computar a equação matemática referida previamente, que dado um *u*, *v* e a matriz de pontos de controlo, devolve o ponto correspondente.

```
Point3D Surface::bezierPatch(const double u,const double v,const vector<vector<Point3D>> &controlPoints)
```

1.2. Obtenção dos pontos de uma *patch Bezier*

A obtenção de todos os pontos de uma determinada *patch* é conseguida através de chamadas sucessivas da função anterior. Este número de chamadas está dependente do nível de tesselação.

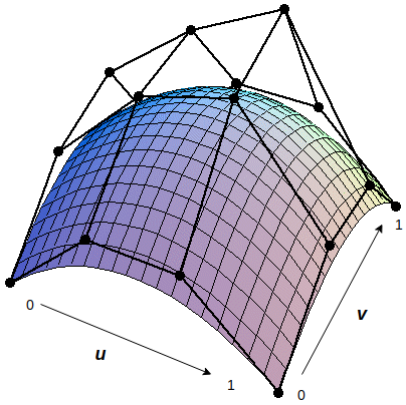


Fig.27. Representação da superfície de *patch Bezier*

A figura acima representa a superfície obtida através dos dezasseis pontos de controlo assinalados e tesselação 19 (número de secções em cada direção). Os pontos que serão calculados são os correspondentes aos vértices de todos os quadrados. Desta forma, é de fácil interpretação chegar-se a uma conclusão no que concerne aos valores de *u* e *v*. Dado que em ambas as direções se utiliza o mesmo nível de detalhe, ambas começam por tomar o valor de 0 (zero) e são incrementadas até 1 a um fator de ***1/(tesselação)***.

Por fim, repete-se o processo anterior para todas as *patches* lidas do ficheiro seguida da sua organização e ordenação em pontos três a três, definindo a primitiva gráfica (o triângulo) utilizada pelo visualizador.

Para além do supramencionado, acrescenta-se a informação de que a partilha dos cálculos entre o Gerador e o Visualizador é efetuada a partir de ficheiros, logo para concluir escreveremos os pontos representantes de triângulos num ficheiro de output (designado surface).

2. Visualizador

Ao longo desta fase pretendeu-se não só alargar as funcionalidades do translate e rotate de forma a suportarem animações, mas também implementar VBOs.

Nesta nova etapa, adaptou-se o antigo translate para um que é capaz de receber como parâmetros um conjunto de pontos que definem a curva cúbica *Catmull-Rom*, bem como o número de segundos necessários para percorrer a curva na totalidade. De grosso modo, o objetivo centra-se na apresentação de animações com base nestas curvas, pelo que os modelos estão habilitados a ter transformações dependentes de tempo ou estáticas (como nas fases anteriores).

No que concerne ao rotate, a estratégia utilizada passou pela substituição do ângulo de rotação por uma variável tempo, esta representa o número de segundos necessários para efetuar uma volta de 360 graus em torno do eixo especificado. De modo a medir o tempo foi utilizada a função glutGet(GLUT_ELAPSED_TIME).

É de salientar ainda que, tendo por base a definição de curvas *Catmull-Rom*, a sua utilização requer sempre um ponto inicial antes do segmento de curva e outro ponto depois do último segmento, sendo quatro o número mínimo de pontos a utilizar.

O resultado final é então um sistema solar dinâmico que inclui a rotação dos planetas e respetivos satélites naturais, assim como um cometa (representado por um *teapot*) com uma trajetória definida através de curvas *Catmull-Rom*. Para além do mencionado, é de realçar que o ficheiro *teapot.3d* foi desenvolvido recorrendo ao *teapot.patch*, tal como descrito acima no Gerador.

2.1. Estruturas de dados

De forma a permitir a utilização da variável tempo na construção e transformação de modelos foram efetuadas alterações à estrutura de dados rotate e action, assim como adicionada um nova estrutura com o intuito de capacitar o translate a receber outros parâmetros.

- **struct rotate:** Esta estrutura foi alterada de forma a substituir o ângulo de rotação pelo tempo necessário para a rotação completa (360 graus).
- **struct action:** A esta estrutura foi adicionada a ação *move.*
- **struct move:** Esta estrutura de dados foi criada com o intuito de armazenar o novo tipo de translate. Este por sua vez, conterá um array que define o eixo de translação, o tempo de translação e o vetor de pontos referentes à curva.

2.2. VBOs

De modo a melhorar a forma como os modelos são desenhados, deixando de efetuar os pedidos linha a linha à placa gráfica e passando a pedir modelo a modelo, introduziu-se a utilização de *Vertex Buffer Objects*. Esta alteração passou por substituir os *glVertex3f(args)* por *push_back(args)* para o vector correspondente, efetuando apenas o gl*DrawArrays(args)* para cada *Model*. Em suma, o desenho foi então feito, Modelo a Modelo em oposição ao modo imediato usado nas fases anteriores.

2.3. Translações e Rotações

Com o objetivo final de apresentar os planetas e satélites naturais animados com rotação e translação, bem como um cometa animado com uma rota de translação, recorreu-se inteiramente aos cálculos referentes às curvas de *Catmull-Rom* lecionadas em contexto de aula.

Tanto a translação como a rotação, contendo a componente de tempo, foram processadas de modo a que esta representasse o tempo que o cometa demoraria a efetuar uma translação e, no caso, dos planetas e luas uma rotação e translação completa. Para tal, foi necessária a utilização de um *ticker* que permitisse a dependência temporal destes dois movimentos, calculando a cada momento o ângulo em que o modelo em causa deveria estar.

Este *ticker* trata-se então da função *glutGet(GLUT_ELAPSED_TIME)*, sugerida pela equipa docente para o efeito e a fórmula final que determina o ângulo é "*(ticker * 360) / time*".

Adicionalmente, teve-se o cuidado de determinar tempos de rotação e translação para os planetas consistentes com os reais, ainda que levianamente calculados com base na comparação das velocidades destes, uns com os outros.

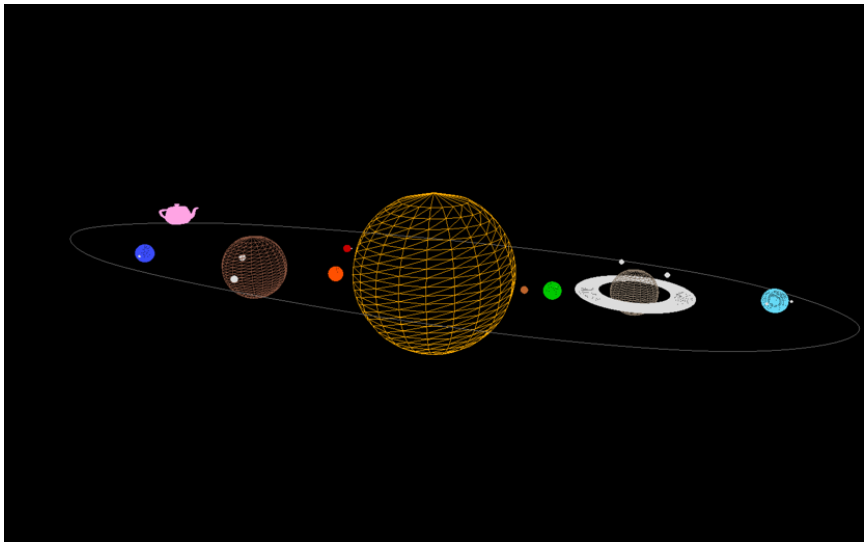


Fig.28. Representação gráfica visualmente mais apelativa do sistema solar, juntamente com o cometa (*teapot)* e a sua órbita

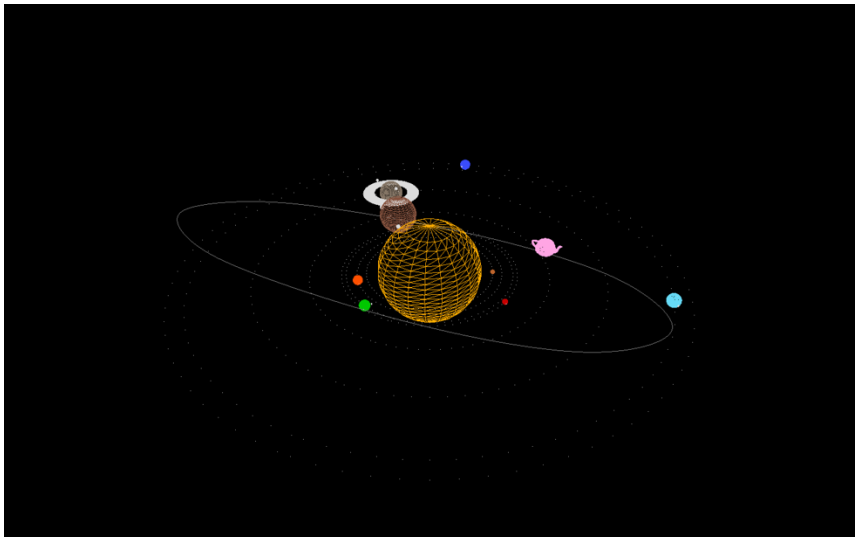


Fig.29. Representação gráfica visualmente mais apelativa do sistema solar, juntamente com o cometa (*teapot)* – com órbitas do cometa e dos planetas

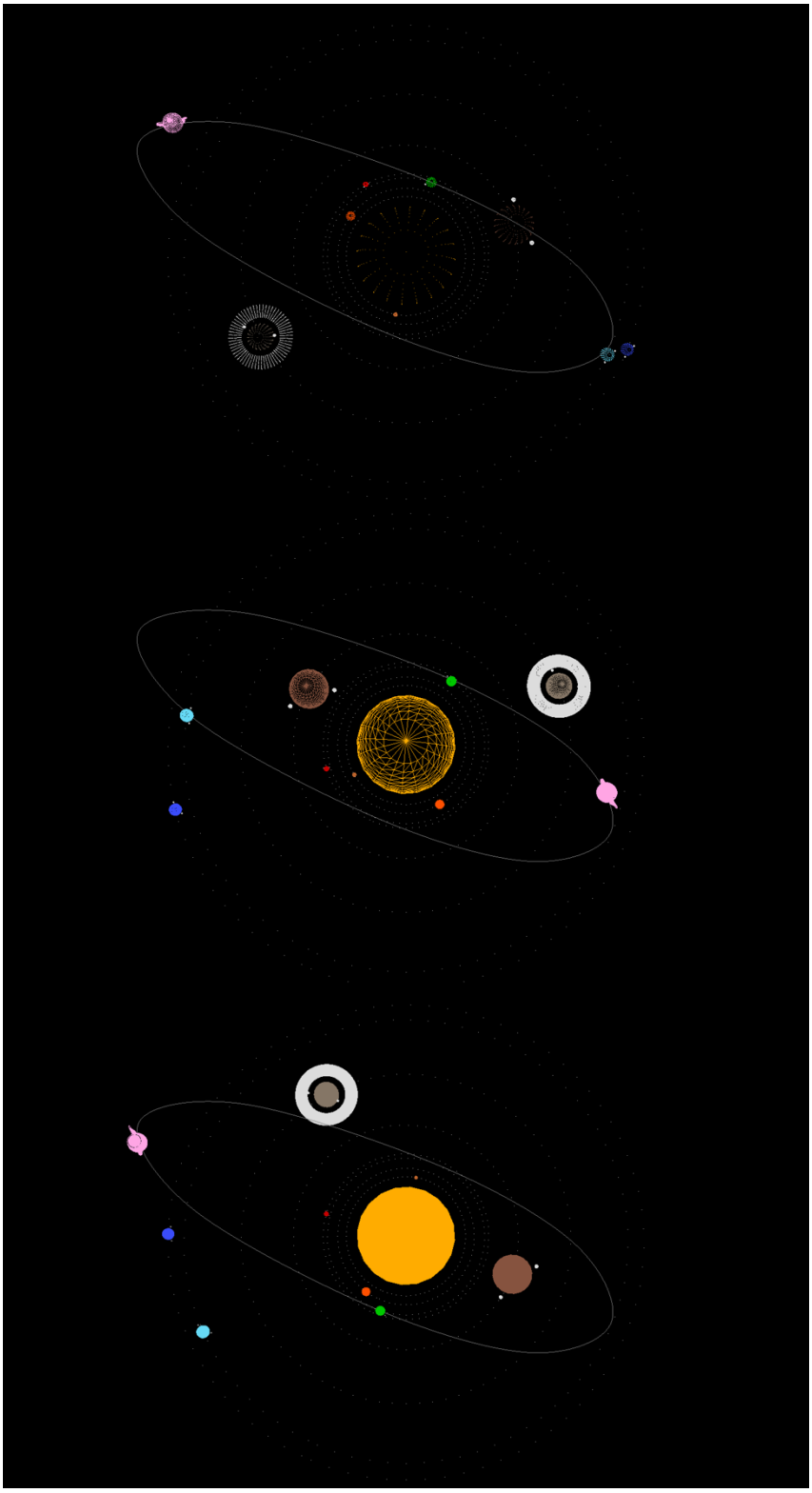


Fig.30. Representação gráfica visualmente mais apelativa do sistema solar, juntamente com o cometa (*teapot)* – com órbitas do cometa e dos planetas (de uma outra perspetiva)

Conclusão

Em virtude do que foi mencionado, o grupo considera que estas primeiras fases do projeto vão de encontro ao perspetivado aquando do início minuciosamente ponderado do mesmo.

Durante a realização desta fase, teve-se em consideração o principal objetivo que visa sensibilizar e motivar os alunos para a utilização das ferramentas *Glut*, derivadas da biblioteca *OpenGL,* juntamente com conhecimentos matemáticos. Nesse sentido, atingiram-se todos os objetivos definidos inicialmente para a primeira fase, nomeadamente a criação de um Gerador, capaz de receber por input do utilizador as dimensões de primitivas geométricas e gerar ficheiros com todos os pontos necessários para definir uma figura, bem como um Visualizador, capaz de pegar nos ficheiros criados pelo Gerador e representar graficamente os modelos correspondentes.

Na segunda fase foi feita uma reformulação do *Visualizator* para que este pudesse albergar o acréscimo de informação de forma mais eficiente, permitindo então que pudesse ser escrito, lido e apresentado, por exemplo, o ficheiro de configuração do sistema solar proposto no enunciado.

No que concerne à terceira fase, optou-se pela reestruturação do projeto de modo a albergar *Vertex Buffer Objects*, bem como a animação do Sistema Solar permitindo a rotação dos planetas e ainda a translação de um cometa sobre uma órbita definida com curvas de *Bezier*. Com o intuito de criar o ficheiro .3d correspondente ao cometa foi implementada uma nova classe denominada Surface, que calcula os pontos das *patches* (a partir de um ficheiro) e forma triângulos.

Deste modo, foi tirado proveito da capacidade de expansão do código a novas funcionalidades.

Para além do referido, foi tido em consideração o modo como a informação está a ser armazenada, evitando possíveis erros inerentes à adição de novas *tags*, permitindo assim a existência de mais funcionalidades, bem como a possibilidade de uma mais fácil expansão das mesmas.

De forma a responder aos objetivos propostos, foram explorados os meios fornecidos para o efeito de aprendizagem, tais como o material proveniente da unidade curricular e a bibliografia recomendada pela equipa docente.

Este projeto revelou-se bastante enriquecedor, uma vez que incentivou não só a aprendizagem, exploração e manuseamento relativo às ferramentas disponibilizadas, como também mitigou a capacidade de adaptação a uma nova linguagem de programação.

Por fim, não obstantes a futuros aprimoramentos, concebeu-se código de fácil compreensão para promover a vontade e ousadia necessária à realização de posteriores alterações.

Referências Bibliográficas

- Shreiner, D., Woo, M., Neider, J., Davis, T. (2006). OpenGL Programming Guide (5th ed.). Addison Wesley.Angel, E. (1999). Interactive Computer Graphics (2nd ed.). Addison Wesley.Lengyel, E. (2011). Mathematics for 3D Game Programming and Computer Graphics (3rd ed.). Cengage Learning PTR.