Universidade do Minho
Escola de Engenharia

COMPUTAÇÃO GRÁFICA

FASE 2

Alexandre Martins A77523

André Vieira A78322

Eduardo Rocha A77048

Ricardo Neves A78764

ÍNDICE

Introdução 3
Generator 4
Plano 4
Caixa 5
Esfera 6
Cone 8
Figura Adicional – Cilindro 9
Parser do XML 11
First 12
Second 12
Aux 12
Extração das Coordenadas 15
OpenGL 16
Sistema Solar Final 19
Conclusão e Trabalho Futuro 20

INTRODUÇÃO

Neste relatório iremos apresentar e discutir o trabalho realizado pelo grupo, no âmbito da Unidade Curricular de Computação Gráfica, do 3º ano do Mestrado Integrado em Engenharia Informática.

Nesta segunda fase do trabalho prático, tivemos como objetivo a criação de uma cena usando os modelos já gerados na fase anterior, aplicando transformações geométricas, entre as quais, translações, rotações e escalas.

Portanto, neste relatório, iremos apresentar detalhadamente todo o processo realizado, de modo a cumprir os objetivos previamente estabelecidos pelo docente da Unidade Curricular.

Para isto, apresentamos também neste documento, algumas linhas de pensamento que o grupo seguiu, ferramentas usadas, e excertos de código fonte utilizado (e respetiva explicação), de modo a suportar a perceção do trabalho realizado.

De relembrar que, na primeira fase, o grupo implementou um gerador de modelos geométricos 3D (plano, caixa, esfera, cone e uma figura adicional, um cilindro), bem como um motor que é capaz de ler um ficheiro XML de configuração e que desenhe os modelos mencionados nesse ficheiro.

Uma vez que sentimos que a explicação da geração dos modelos não foi a mais clara, iremos começar este documento com uma explicação mais detalhada do processo realizado na etapa anterior.

Depois disto, passaremos para a própria fase 2, as suas explicações, decisões e resultados, e, como conclusão, iremos apresentar um modelo estático do Sistema Solar utilizando todos os recursos desenvolvidos até este ponto do projeto.

GENERATOR

O gerador das figuras geométricas foi uma tarefa completada com sucesso na primeira fase do projeto. No entanto, decidimos voltar a abordar este tópico uma vez que o mesmo não ficou completamente esclarecido, sendo que, agora, iremos aprofundá-lo um pouco mais.

Para ir de encontro ao proposto pelo enunciado da fase 1, elaboramos as primitivas gráficas propostas recorrendo às posições relativas dos vértices dos diferentes modelos. Aqui, é onde são gerados os pontos que, unidos em triângulos, formam as figuras geométricas pedidas.

Plano

O plano foi a figura geométrica mais fácil de implementar, uma vez que é apenas a junção de dois triângulos simples. Para a sua geração, apenas é necessário referenciar a largura do plano quadrado, sendo que, com isto, são desenhados os dois triângulos constituintes. É a única figura geométrica desenvolvida onde não necessitamos de implementar um ciclo *for*, porque, como já disse, são apenas precisos dois triângulos para a sua construção.

```
file << "" << (-length/2) << " 0 " << (-length/2) << "\n";
file << "" << (-length/2) << " 0 " << (length/2) << "\n";
file << "" << (length/2) << " 0 " << (length/2) << "\n";

file << "" << (-length/2) << " 0 " << (-length/2) << "\n";
file << "" << (length/2) << " 0 " << (length/2) << "\n";
file << "" << (length/2) << " 0 " << (-length/2) << "\n";
```

Figura 1 - Algoritmo do plano

Este é o resultado do plano obtido. Aqui, confirma-se que são usados apenas dois triângulos retângulos, unidos pelas suas hipotenusas.

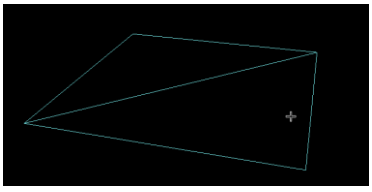

Figura 2 - Desenho do plano

Caixa

A segunda figura geométrica desenhada foi a caixa, sendo que é necessário especificar o seu comprimento, altura e largura e, opcionalmente, o número de divisões. Se nada em contrário for dito ao gerador, o número de divisões da caixa vai ser igual a 1, ou seja, cada base da caixa vai ser composta por um plano (como foi referenciado anteriormente, um plano é composto por 2 triângulos).

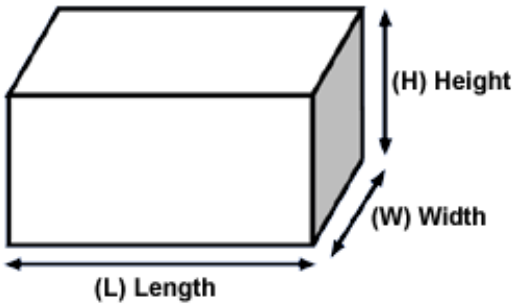

Figura 3 - Definição da caixa

Para o seu desenho, a melhor forma encontrada foi gerar uma face de cada vez. Foram necessários 6 ciclos *for* para a criação da caixa, um para cada face. O seu processo é muito semelhante ao do plano, acrescentando o número de divisões. Este é o exemplo do ciclo que desenha o topo da caixa.

```
x = length/2;
z = width/2;
for (int i = 0; i < div; i++) {
    y = height;
    for (int j = 0; j < div; j++) {
        file << "" << x << " " << y << " " << z << "\n";
        file << "" << x << " " << (y - divY) << " " << z << "\n";
        file << "" << x << " " << (y - divY) << " " << (z - divZ) << "\n";

        file << "" << x << " " << y << " " << z << "\n";
        file << "" << x << " " << (y - divY) << " " << (z - divZ) << "\n";
        file << "" << x << " " << y << " " << (z - divZ) << "\n";

        y -= divY;
    }
    z -= divZ;
}
```

Figura 4 - Algoritmo da caixa

Assim, o gerador percorre os ciclos das divisões especificadas, desenhado todos os triângulos que compões essa face. De salientar que o centro da caixa vai coincidir com a origem do referencial 3D (ponto (0, 0, 0)).

Este é o resultado final para a caixa, sendo que o grupo especificou um número de divisões igual a 5, como se pode constatar.

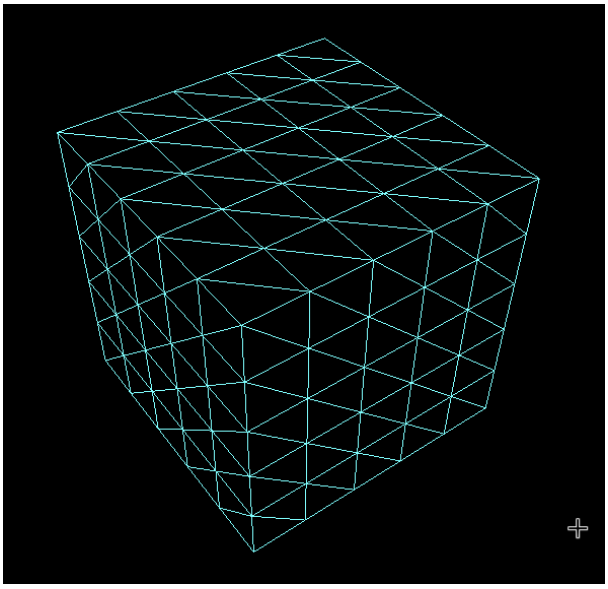

Figura 5 - Desenho da caixa

Esfera

Para a construção da esfera, temos de especificar primeiramente o raio da mesma, o número de *slices* e o número de *stacks*. Quanto maior o número destes dois últimos parâmetros, mais "redonda" ficará a esfera resultante.

Assim, percorrendo o número de *stacks* e de *slices*, e recorrendo às coordenadas polares, desenha-se as coordenadas correspondentes aos vértices dos triângulos.

Desta maneira, os pontos de todos os triângulos vão sendo obtidos à medida que o ângulo vai sendo alterado.

O primeiro ciclo percorre o total de *stacks* introduzidas pelo utilizador e tem a função de colocar o valor da variável *alpha* a 0. Já o segundo ciclo for encontra-se aninhado dentro do primeiro e percorre o número de *slices* também introduzido inicialmente.

```
double alpha = 0;
double deltaAlpha = (2 * M_PI) / slices;
double beta = 0;
double deltaBeta = M_PI / stacks;
```

Figura 6 - Inicialização das variáveis

Neste segundo ciclo, calculamos, através das já referidas coordenadas polares, os 4 pontos necessários para o desenho de um dos quadriláteros (formado por dois triângulos), como se pode ver pela imagem abaixo. Depois de calculadas as coordenadas para estes 2 triângulos, soma-se de *deltaAlpha* a *alpha*.

Quando uma linha inteira da esfera estiver completa, quer dizer que o ciclo de dentro terminou e soma-se *deltaBeta* a *beta*, de modo a passar para o nível seguinte da esfera.

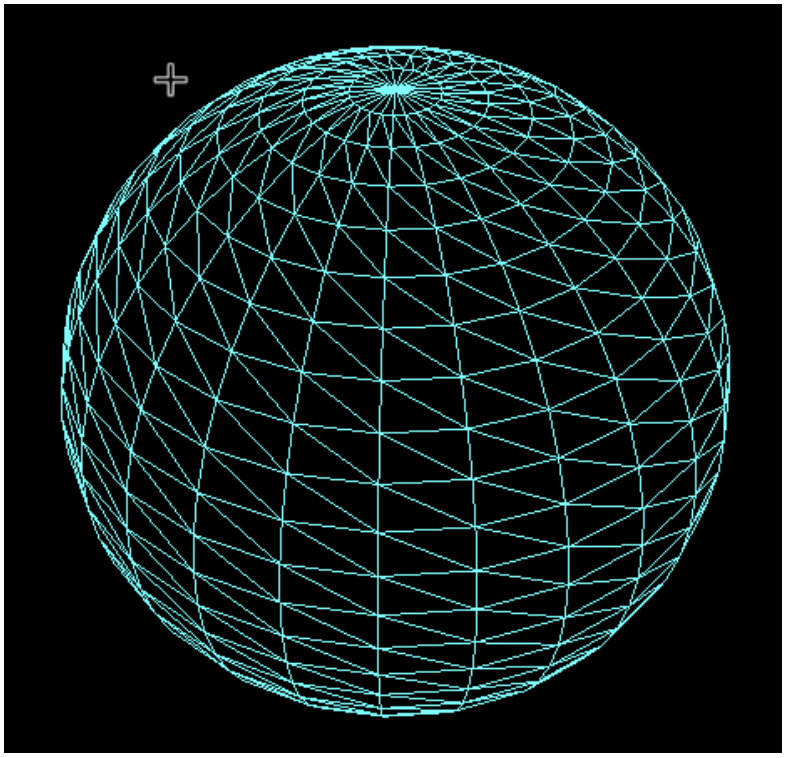

Figura 7 - Desenho da esfera

Cone

Para o desenho do cone, constatamos que podíamos utilizar novamente as coordenadas polares, uma vez que a sua base é uma circunferência. De salientar que os parâmetros necessários para a geração do cone são o raio da sua base, a sua altura e o número de *slices*.

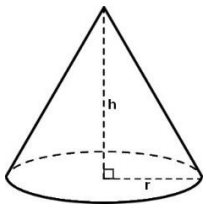

Figura 8 - Definição do cone

Assim, por cada iteração do ciclo *for*, é desenhada uma parte da circunferência base e uma das laterais. Quantas mais slices, mais iterações do ciclo vão haver e mais perfeito vai ficar o cone. O centro da base do cone está centrado na origem, ponto (0, 0, 0).

```
file << "" << "0" << " 0 " << "0" << "\n";
file << "" << (radius * sin(alpha + delta)) << " 0 " << (radius * cos(alpha + delta)) << "\n";
file << "" << (radius * sin(alpha)) << " 0 " << (radius * cos(alpha)) << "\n";

file << "" << "0 " << height << " 0" << "\n";
file << "" << (radius * sin(alpha)) << " 0 " << (radius * cos(alpha)) << "\n";
file << "" << (radius * sin(alpha + delta)) << " 0 " << (radius * cos(alpha + delta)) << "\n";

alpha += delta;
```

Figura 9 - Algoritmo do cone

As primeiras 3 linhas são referentes à base do cone. As 3 últimas desenham as "laterais" do cone. No fim de cada iteração, é somado um valor delta (2*M_PI / *slices*) ao valor de *alpha*. Como podemos observar, todas as laterais do cone irão coincidir no ponto (0, *height*, 0). O resultado final do cone é o seguinte:

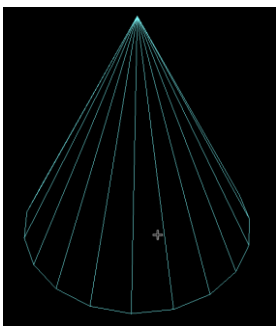

Figura 10 - Desenho do cone

Figura Adicional – Cilindro

Para além das figuras geométricas pedidas no enunciado, tomamos a iniciativa de criar uma outra figura a 3 dimensões. Assim, o grupo decidiu criar um cilindro.

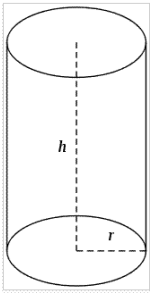

Figura 11 - Definição do cilindro

O método de criação do cilindro foi muito idêntico ao da figura anterior, o cone. Isto deve-se ao facto de ambas as suas bases serem círculos, o que permite o uso de coordenadas polares.

```
for (int i = 0; i < slices; i++) {
    file << "0 0 0" << "\n";
    file << (r * sin(alpha + delta)) << " 0 " << (r * cos(alpha + delta)) << "\n";
    file << (r * sin(alpha)) << " 0 " << (r * cos(alpha)) << "\n";

    file << (r * sin(alpha + delta)) << " " << height << " " << (r * cos(alpha + delta)) << "\n";
    file << (r * sin(alpha)) << " " << height << " " << (r * cos(alpha)) << "\n";
    file << (r * sin(alpha)) << " 0 " << (r * cos(alpha)) << "\n";

    file << (r * sin(alpha + delta)) << " " << height << " " << (r * cos(alpha + delta)) << "\n";
    file << (r * sin(alpha)) << " 0 " << (r * cos(alpha)) << "\n";
    file << (r * sin(alpha + delta)) << " 0 " << (r * cos(alpha + delta)) << "\n";

    file << "0 " << height << " 0" << "\n";
    file << (r * sin(alpha)) << " " << height << " " << (r * cos(alpha)) << "\n";
    file << (r * sin(alpha + delta)) << " " << height << " " << (r * cos(alpha + delta)) << "\n";
    alpha += delta;
}
```

Figura 12 - Algoritmo do cilindro

Para cada iteração do ciclo, desenha-se uma parte de cada uma das bases do cilindro (base e topo) e os dois triângulos que formam uma parte da sua lateral. Por sua vez, a variável *alpha* é responsável por ir dar uma forma redonda ao cilindro e, a cada iteração, é aumentada em *delta*. Como já mencionei em outras figuras geométricas desenvolvidas até aqui, quanto maior o número de *slices*, mais iterações do ciclo serão percorridas, aperfeiçoando cada vez mais o produto final.

O primeiro conjunto de 3 linhas do algoritmo correspondem à base do cilindro. Como podemos ver, a base do cilindro está centrada no ponto (0, 0, 0) do referencial.

O segundo conjunto diz respeito à base do topo do cilindro. Desta vez, temos de ter em atenção a *height* (altura) do cilindro, proporcionada pelo utilizador.

Os dois últimos conjuntos referem-se às laterais do cilindro. Estas laterais são simples planos, formados por dois triângulos.

Aqui vemos o resultado do cilindro final:

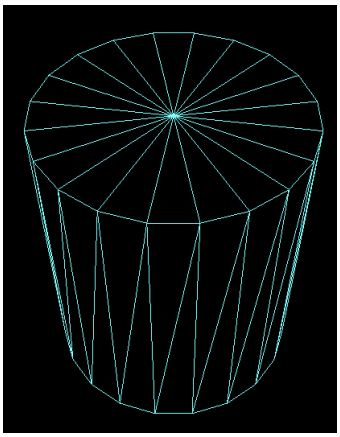

Figura 13 - Desenho do cilindro

O grupo pensa que função *main()* deste gerador foi explicada detalhadamente no relatório da fase anterior, pelo que não iremos repetir aqui essa mesma explicação.

Dito isto, passaremos para a segunda fase do projeto.

PARSER DO XML

Com a introdução das transformações geométricas a aplicar aos modelos, é natural que o Parser do ficheiro XML tivesse que ser alterado drasticamente. Isto deve-se ao facto da criação de novas *tags* para o *translate*, *rotate* e *scale*, mas também a possibilidade de um grupo herdar estas transformações do seu grupo "pai". Podemos afirmar que o verdadeiro desafio desta fase foi alterar e moldar o Parser de modo a satisfazer as necessidades propostas pelo docente.

O grupo constatou que isto iria depender na gravação de muitos dados porque cada uma destas transformações geométricas apresentam entre 3 e 4 parâmetros, para além do nome da própria transformação.

Assim, decidimos implementar as seguintes estruturas de dados, que guardarão grande parte dos dados necessários para o desenho da cena, neste caso, o Sistema Solar.

```
typedef struct aux {
    char* transf;
    double params[4];
    struct aux* next;
} Aux;

typedef struct second {
    char* transf;
    double params[4];
} Second;

typedef struct first {
    char* file;
    vector<Second*> transfs;
} First;
```

Figura 14 - Estruturas de dados usadas

Convém definir desde já que todos estes dados ficarão guardados num vetor de estruturas *First* denominado *files* (vector<First*> files).

Passaremos a explicar cada uma destas estruturas, na página seguinte:

First

Como disse, temos um *vector* que guarda, em cada uma das posições, uma destas estruturas. Vemos que esta estrutura *First* contém um espaço *file* para guardar o nome do ficheiro a abrir, como "sphere.3d", necessário para o *Extractor* das coordenadas. Para além disto, temos um vetor de estruturas Second denominado *tranfs*, que irei explicar de seguida.

Second

Aqui guardamos todas as transformações geométricas a aplicar a um certo modelo.

Guardamos, como é obvio, o nome da transformação, como "*translate*", "*rotate*" ou "*scale*", e os parâmetros necessários. Por exemplo, as transformações *translate* e *scale* apenas necessitam de 3 parâmetros (X, Y, Z), enquanto que o *rotate* necessita de 4 (*angle*, X, Y, Z). Aqui, o array *params* tem um tamanho definido de 4 *doubles*. No entanto, não há problema para as transformações que apenas necessitam de 3 parâmetros, uma vez que a quarta posição nunca irá ser acedida, nesse caso.

Deste modo, já temos as estruturas necessárias para guardar os dados e aceder futuramente. No entanto, decidimos criar uma estrutura *Aux*, muito parecida, aliás, à estrutura *Second*.

Aux

Esta estrutura é em tudo semelhante à estrutura anterior, uma vez que podemos guardar o nome da transformação geométrica, bem como os parâmetros necessários para a sua aplicação à figura. Surgiu da possibilidade da existência de grupos dentro de outros grupos, onde as transformações a um grupo principal também têm de ser aplicadas aos grupos subjacentes.

Cada posição desta estrutura contém um apontador para a próxima transformação presente na estrutura.

Este é o exemplo do procedimento aquando o algoritmo encontra a tag "*rotate*". Guardamos o valor presente no XML numa variável como o nome correto e, posto isto, enviamos toda esta informação para a função *addAux()*, que é responsável por inserir a mesma na estrutura de dados.

```
if (strcmp((char*)group->Value(), "rotate") == 0) {

    char transf[10];
    strcpy(transf, (char*)group->Value());

    x = 0;
    y = 0;
    z = 0;
    angle = 0;

    char* rotateX = (char*)group->Attribute("X");
    if (rotateX != NULL)
        x = atof(rotateX);

    char* rotateY = (char*)group->Attribute("Y");
    if (rotateY != NULL)
        y = atof(rotateY);

    char* rotateZ = (char*)group->Attribute("Z");
    if (rotateZ != NULL)
        z = atof(rotateZ);

    char* rotateAngle = (char*)group->Attribute("angle");
    if (rotateAngle != NULL)
        angle = atof(rotateAngle);

    addAux(g, i, transf, x, y, z, angle);

}
```

Figura 15 - Algoritmo do Rotate

Também é na função *addAux()* que verificamos se existe mais transformações de grupos "pai" a ter em conta.

Quando é encontrada a tag "*file*", copiamos toda a informação presente na estrutura *Aux* para a estrutura *Second*, uma vez que, a partir deste ponto, não haverá mais transformações ao modelo.

```
<group>
    <translate X="110" />
    <scale X="2" Y="2" Z="2" />
    <color R="1" G="0.9" B="0.9" />
    <models>
        <model file="sphere.3d" /> -- Saturn
    </models>
    <group>
        <rotate X="1" angle="-10" />
        <scale X="2" Y="0.02" Z="2" />
        <color R="0.9" G="0.9" B="1" />
        <models>
            <model file="sphere.3d" /> -- Saturn Rings
        </models>
    </group>
</group>
```

Figura 16 - XML de Saturno e seus anéis

No ficheiro XML, podemos tomar como exemplo o caso de Saturno. Saturno encontra-se a uma distância de 110 do centro do referencial 3D e tem o dobro do tamanho da esfera desenvolvida no gerador da primeira fase.

Assim, para testar esta partilha de transformações geométricas entre grupos, decidimos criar os Anéis de Saturno dentro do grupo do planeta, herdando as suas modificações. Podemos ver que os Anéis foram sujeitos a uma pequena rotação, tem o dobro do tamanho do planeta relativamente ao eixo dos XX e dos ZZ.

Realizados os testes, podemos ver que o resultado final foi bastante aceitável.

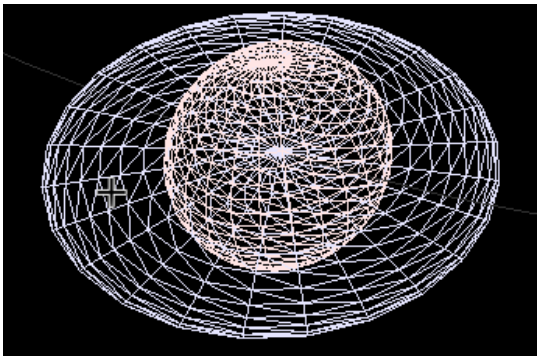

Figura 17 - Planeta Saturno

O mesmo procedimento foi efetuado para o planeta Terra e a Lua, onde a última é um grupo secundário. Podemos ver que o modelo final foi o esperado.

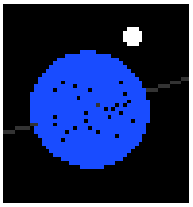

Figura 18 - Planeta Terra e Lua

Outro aspeto que podemos verificar é a existência da tag "*color*". Esta tag não foi pedida pelo docente da disciplina, mas o grupo decidiu implementá-la de modo a dar um pouco de cor aos modelos, tendo uma melhor ideia de que planeta se trata, e torna-lo visivelmente mais apetecível. Assim, é necessário fornecer o valor de *Blue*, *Green* e *Red* no ficheiro XML.

Resumidamente, é um processo bastante detalhado onde a informação é verificada e copiada de estrutura para estrutura, podendo afirmar que foi onde o grupo despendeu grande parte do tempo de implementação.

EXTRAÇÃO DAS COORDENADAS

A extração das coordenadas é efetuada na função *extractor()*. Esta permite extrair as coordenadas de um ficheiro .3d e armazenar em tuplos de três coordenadas.

O *extractor* desta segunda fase é muito semelhante ao da primeira fase.

Agora, temos de percorrer o vector *files*, descrito no tópico anterior, e retirar o nome de cada um dos ficheiros a aceder. No caso do Sistema Solar, todos os modelos irão ser esferas; no entanto, teremos de percorrer todas as posições do vector uma vez que não sabemos o número de esferas a desenhar.

Assim, abri-mos cada um dos ficheiros e guardamos as coordenadas num vetor de tuplos denominado *triangles*. Este vector vai ser acedido quando desenhar-mos os modelos na *renderScene()*.

```
coords point = make_tuple(triple[0], triple[1], triple[2]);
triangles.push_back(point);
```

Figura 19 - Coordenadas guardadas num tuplo

No entanto, não podemos simplesmente ir guardando todas as coordenadas de todos os modelos de uma só vez. Necessitamos de saber quando acaba um modelo e começa outro, de modo a aplicar as transformações corretamente.

Assim, o grupo pensou numa maneira de executar o que foi dito e chegou à conclusão que podemos acrescentar um triângulo "imaginário" ao vetor, que servisse de marca. Assim, na *renderScene()*, podemos verificar que um modelo acaba quando encontrar-mos este triângulo "imaginário", constituído por 3 tuplos de coordenadas (12345, 12345, 12345). Como é natural, não iremos desenhar este triângulo; apenas servirá de marca.

```
coords end = make_tuple(12345, 12345, 12345);
triangles.push_back(end);
triangles.push_back(end);
triangles.push_back(end);
```

Figura 20 - Triangulo "Imaginário"

Posto isto, temos já todas as coordenadas de todos os triângulos guardadas numa estrutura de dados. Assim, passamos para a última fase, o desenho dos modelos.

OPENGL

Como foi referenciado na fase anterior, utilizamos o *OpenGL,* uma API gratuita utilizada na computação gráfica, para o desenho dos modelos a 3 dimensões.

No tópico anterior da extração das coordenadas, confirmamos que temos um vetor preenchido com todas as coordenadas dos pontos a desenhar. Assim, é necessário agrupar os pontos em grupos de 3, de modo a criar triângulos. Todos estes triângulos ligam-se entre si, criando uma figura geométrica em 3 dimensões.

Mas antes de desenhar estes triângulos, temos de aplicar todas as transformações guardadas no Parser. Assim, percorremos o vector onde estão guardadas as transformações geométricas de um modelo e, consoante o nome da mesma, aplica-se a função correspondente predefinida do OpenGL. Como já foi dito, temos as 3 transformações geométricas requisitas no enunciado, mais a definição "color", resultado da ideia do grupo.

```
while (iSecond != first->transfs.end()) {

    Second* op = *iSecond;

    if (strcmp(op->transf, "scale") == 0)
        glScalef(op->params[0], op->params[1], op->params[2]);

    if (strcmp(op->transf, "rotate") == 0)
        glRotatef(op->params[3], op->params[0], op->params[1], op->params[2]);

    if (strcmp(op->transf, "translate") == 0)
        glTranslatef(op->params[0], op->params[1], op->params[2]);

    if (strcmp(op->transf, "color") == 0)
        glColor3f(op->params[0], op->params[1], op->params[2]);

    iSecond++;

}
```

Figura 21 - Aplicação das transformações

De referir que estas operações geométricas estão salvaguardadas pelas funções *glPushMatrix()* e *glPopMatrix()*. Isto traduz-se no facto de as transformações aplicadas a um modelo não serem aplicadas a todos os modelos seguintes, o que traria muitos problemas à implementação.

Posto isto, está na altura de desenhar as figuras, a partir do vector de tuplos *files*.

```
while (i != triangles.end() && flag != false) {

    coords one = *i;
    i++;
    coords two = *i;
    i++;
    coords three = *i;
    i++;

    if (!checkEnding(one, two, three)) {

        glBegin(GL_TRIANGLES);

        glVertex3d(get<0>(one), get<1>(one), get<2>(one));
        glVertex3d(get<0>(two), get<1>(two), get<2>(two));
        glVertex3d(get<0>(three), get<1>(three), get<2>(three));

        glEnd();

    }
    else
        flag = false;
}
```

Figura 22 - Algoritmo do desenho dos triângulos

Podemos constatar a presença da função *checkEnding()* e da variável booleana *flag*. Foi aqui documentada a existência da marca que separa um modelo do próximo modelo. Ora, aqui verificamos se o próximo triângulo a desenhar trata-se mesmo da marca implementada pelo grupo.

```
bool checkEnding(coords firstTriangle, coords secondTriangle, coords thirdTriangle)
{
    coords end = make_tuple(12345, 12345, 12345);

    return (firstTriangle == end && secondTriangle == end && thirdTriangle == end);
}
```

Figura 23 – Algoritmo da função *checkEnding()*

Se se tratar realmente do triângulo "imaginário" (a marca), então a *flag* passará a ter o valor *false*. Aqui, o algoritmo sabe que terá de passar para a próxima figura geométrica, repetindo o processo até toda a cena ficar completa.

Antes de mostrar o resultado final do Sistema Solar, o grupo decidiu incorporar mais uma adição à cena. Trata-se da linha de órbita dos planetas que constituem o Sistema.

Como sabemos, os planetas do Sistema Solar giram à volta do Sol numa órbita, convencionamos, circular. Deste modo, decidimos desenhar, para cada planeta, uma trajetória suave de cada um dos planetas.

Este é o exemplo da implementação da órbita para o planeta Terra.

```
glBegin(GL_LINE_LOOP);
for (a = 0; a < 360; a += 360 / sides) {
    heading = a * M_PI / 180;
    glVertex3f(cos(heading) * 130 - 210, 0, sin(heading) * 130);
}
glEnd();
```

Figura 24 - Algoritmo de desenho das órbitas

Uma das desvantagens desta implementação das órbitas é que se encontra *hard coded*. Por isto, queremos dizer que, para cada planeta, existe um *glBegin()*, um ciclo *for*, e um *glEnd()*, sendo que, também, o raio de cada órbita também foi calculada "à mão".

Esta é um dos pontos que tentaremos modificar numa próxima fase do projeto (calcular o desenho da órbita automaticamente a partir do XML do Sistema Solar).

SISTEMA SOLAR FINAL

Aqui apresentamos a imagem final do Sistema Solar implementado, contendo o Sol, os seus 9 planetas, a Lua, os Anéis de Saturno e as órbitas dos planetas.

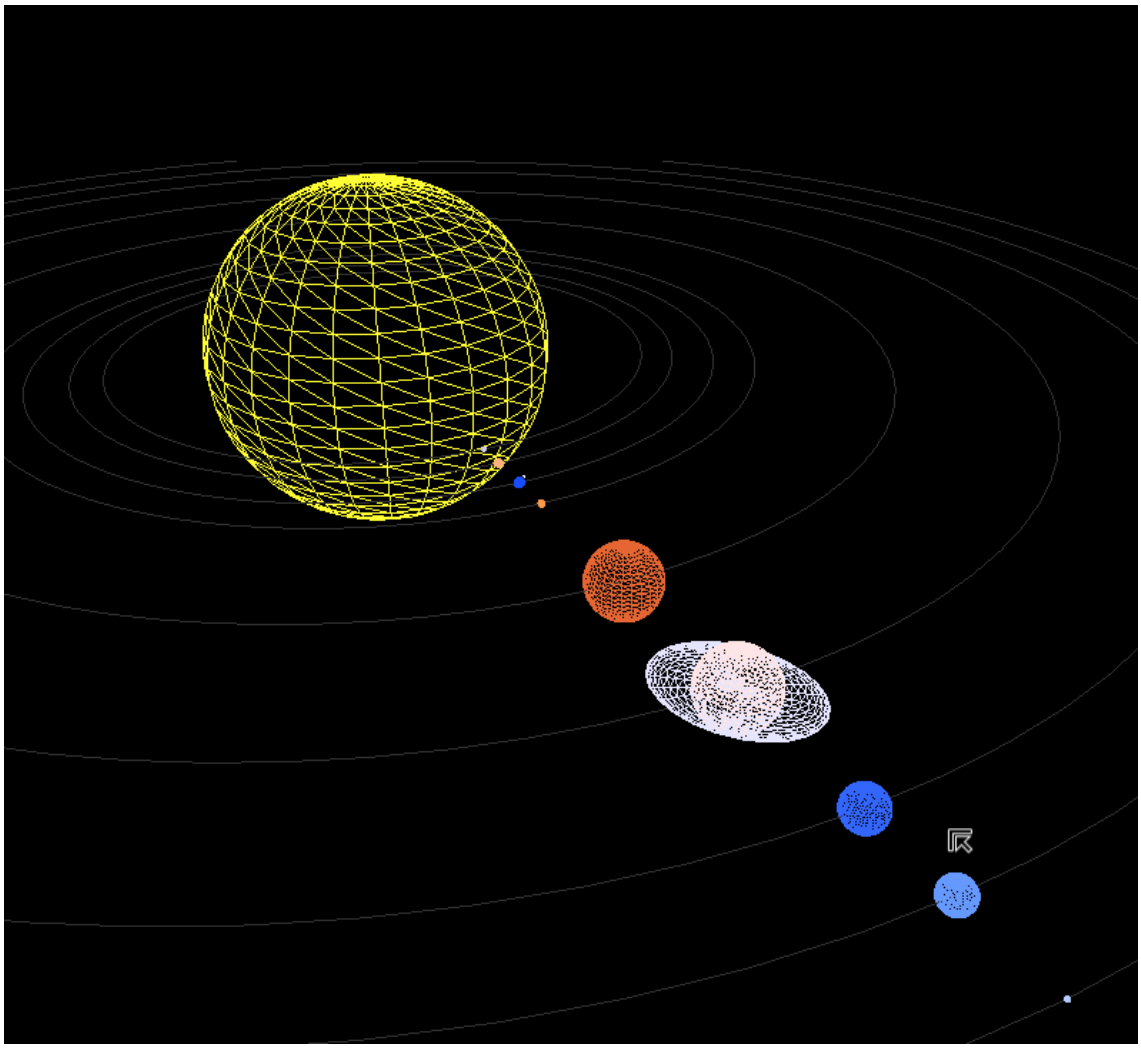

Figura 25 - Resultado final do Sistema Solar

Para o esquema final ser o mais parecido à realidade, o grupo pesquisou os raios dos planetas e as distâncias ao sol relativos, tentando também dar uma cor realista aos modelos criados.

CONCLUSÃO E TRABALHO FUTURO

Esta segunda fase do TP de Computação Gráfica foi bastante mais complexa do que a primeira, devido principalmente às enormes mudanças que tiveram que ser feitas ao Parser do XML. Guardar todos os dados nas estruturas corretas foi o verdadeiro desafio desta fase.

No entanto, podemos afirmar que o resultado final superou às expectativas criadas no início, uma vez que ainda conseguimos implementar, por exemplo, as cores e as órbitas dos planetas. Não foi uma tarefa difícil nem demorosa, mas achamos que torna o modelo bem mais completo, aumentando a vontade de continuar a trabalhar e melhorar o mesmo.

Por isto, podemos dizer que a nossa prática em C++ foi um pouco melhorada com este projeto até então, bem como os conhecimentos em OpenGL.

Para a próxima fase, iremos debruçar-nos sobre o uso de VBOs e codificação de curvas.

Com o trabalho realizado até aqui, sentimo-nos capazes de continuar na implementação do projeto, sendo que os objetivos propostos foram cumpridos e consolidados.

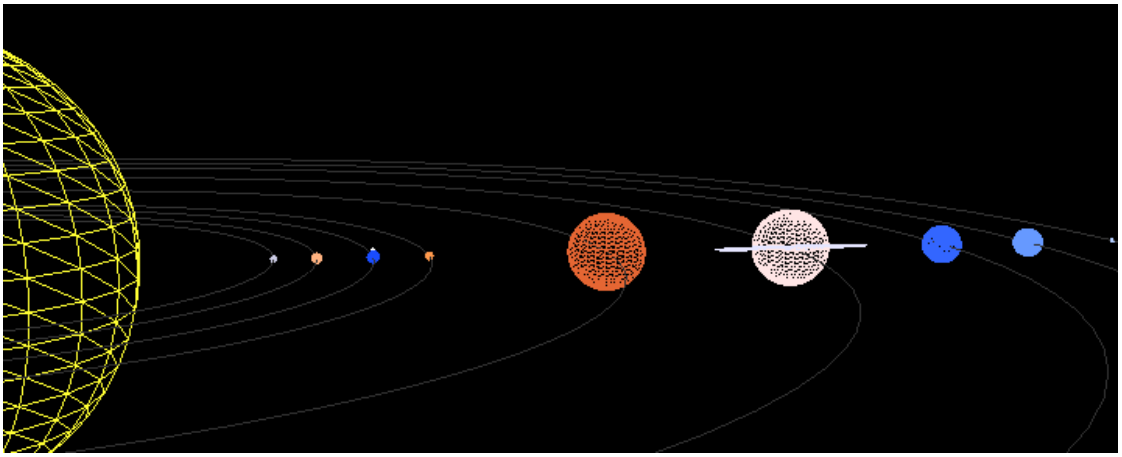

Figura 26 - Vista lateral do modelo