Universidade do Minho

Licenciatura em Engenharia Informática

Computação Gráfica

Grupo 30

Trabalho Prático - **Fase 3**

Curvas, Superfícies Cúbicas e VBO's

Joana Alves (A93290)

Maria Cunha (A93264)

Vicente Moreira (A93296)

Maio 2023

Conteúdo

1 Introdução

Este relatório foi desenvolvido no âmbito da terceira fase do trabalho da unidade curricular de Computação Gráfica da Licenciatura em Engenharia Informática, tendo como objetivo a evolução das aplicações criadas anteriormente, a partir da integração e suporte de novas primitivas utilizando leitura de ficheiros *patch* no *generator* e modificação no modo de desenho dos modelos utilizando VBO's no *engine*, assim como permitir animações rudimentares.

Este documento detalha as várias modificações efetuadas para alcançar os objetivos, assim como as decisões tomadas pelo grupo. Por fim, foram efetuados vários testes para verificar o bom funcionamento destas transformações.

2 Generator

Tal como referido nas fases anteriores, esta aplicação é responsável pela geração de coordenadas de certas figuras geométricas. Nesta fase em particular, tínhamos como objetivo evoluir a mesma ao criar um novo tipo de modelo baseado em ***Bezier patches***. Adicionalmente, o grupo desenvolveu uma nova primitiva para a figura ***Torus*** de modo a diversificar a oferta de modelos e, posteriormente, ser aplicado na *demo* do sistema solar desenvolvido pelo grupo.

2.1 Bezier Patches

Relativamente aos *bezier patches*, o *generator* irá receber como parâmetros o nome do ficheiro de *input* contendo todos os pontos de controlo *Bezier* e um nível de *tesselation* necessário. O ficheiro de *output* irá conter, tal como nas outras primitivas, uma listagem de todos os pontos de todos os triângulos necessários para desenhar a superfície.

2.1.1 Leitura do Ficheiro de Configuração

A primeira etapa da construção da superfície é, naturalmente, ler os parâmetros contidos no ficheiro de *input*, tendo este uma formatação fixa, sendo, por isso, mais fácil a sua leitura e correta interpretação. Desta forma, para armazenarmos de maneira correta a informação presente no ficheiro criamos uma estrutura de dados para armazenar o número de *patches* e os respetivos *arrays* de índices e, por fim, o número de pontos de controlo e os respetivos *arrays* com as suas coordenadas, sendo estes parâmetros lidos na função **readBezierPatch**:

```
typedef struct bezierHolder{
    int numPatches;
    int numControlPoints;
    int** patchIndexes;
    float** controlPoints;
} *BezierHolder;
```

2.1.2 Cálculos dos Pontos

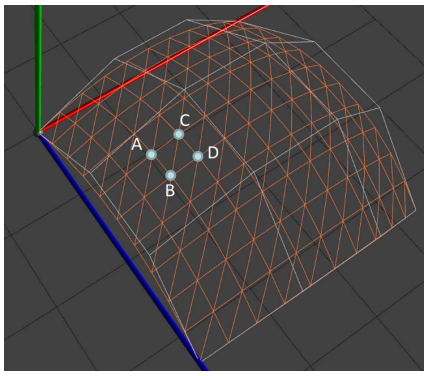
Criamos uma função designada **bezier** que, dado quatro pontos e uma parâmetro t, retorna o ponto calculado através da fórmula de *Bezier*. É de notar que utiliza uma função auxiliar para o cálculo da multiplicação de matrizes com vetores (*multMatrixVector*).

```
float* bezier(float t, float* p0, float* p1, float* p2, float* p3) {
    float* points[4] = { p0,p1,p2,p3 };
    float vectorT[4] = { (pow((1 - t),3)       , 3 * t * pow((1 - t),2),
                         3 * (1 - t) * pow(t,2), pow(t,3) };

    auto* res = (float*)malloc(sizeof(float*) * 3);
    multMatrixVector(points, vectorT, res);
    return res;
}
```

De seguida, desenvolvemos a função **bezierPatch** que, para cada *patch* e de quatro em quatro pontos, calcula o respetivo ponto resultante:

```
float* bezierPatch(float u, float t, float** points, int* index) {
    float* controlPoints[4];
    for (int i = 0;i < 4;i++) {    // for every 4 points
        float* p0 = points[index[4 * i]];
        float* p1 = points[index[4 * i + 1]];
        float* p2 = points[index[4 * i + 2]];
        float* p3 = points[index[4 * i + 3]];

        float* point = bezier(u, p0, p1, p2, p3);
        controlPoints[i] = point;
    }
    return bezier(t, controlPoints[0],
                     controlPoints[1],
                     controlPoints[2],
                     controlPoints[3]);
}
```

Por fim, todas estas funções são combinadas na função principal **generateBezier** que trata de calcular os pontos *Bezier* e escreve os mesmos para o ficheiro de *output*. Para tal, definimos um **step** com valor igual a 1/tessellation (sendo este dado como argumento), onde, para cada iteração, deparamo-nos com quatro pontos essenciais como demonstrado na figura seguinte:



De modo a invocarmos a função **bezierPatch**, recorremos à criação de quatro parâmetros com os seguintes valores (notando que j e k são valores de iterações):

- **u** = step * j
- **v** = step*k
- **u1** = step*(j+1)
- **v1** = step*(k+1)

Por fim, relativamente aos pontos presentes na figura acima, temos de invocar a função *bezierPatch* com os parâmetros corretos, assim, os pontos possuem os seguintes parâmetros:

- **A:** u e v
- **B:** u e v1
- **C:** u1 e v
- **D:** u1 e v1

2.2 Torus

Esta primitiva foi adicionada ao *generator* com o principal objetivo de ser incluída na *demo* do sistema solar para simular, por exemplo, os anéis de Saturno, de modo a obtermos um sistema solar o mais realista possível.

Desta forma, a geração da primitiva *Torus* foi particularmente semelhante à primitiva **esfera**, fazendo uso do raciocínio dos ângulos de divisão dos vários eixos. Assim, é necessário definir dois ângulos, **theta** (θ) e **phi** (ϕ), para efetuar os cálculos dos pontos. *Theta* representa o ângulo de abertura de um círculo imaginário no plano XZ, já o ângulo *phi* está situado no plano XY (ou ZY), sendo que ambos poderão efetuar um completo 360^{o}. Com estes ângulos, um raio total da figura (rt) e um raio interno dos círculos do *torus* (ri), é possível calcular as coordenadas de um ponto do mesmo através das seguintes fórmulas:

$$\mathbf{X} = (rt + ri * cos(\phi)) * cos(\theta)$$
$$\mathbf{Z} = (rt + ri * cos(\phi)) * sin(\theta)$$
$$\mathbf{Y} = ri * sin(\phi)$$

Assim, na função de geração, começamos por calcular os valores dos sub-ângulos *theta* e *phi*, provenientes do número de *slices* e *stacks* fornecidas, respetivamente. Depois as várias *slices* e *stacks* são iteradas, e, em cada iteração, são calculados 4 pontos correspondentes a uma "secção", como ilustrado na figura:

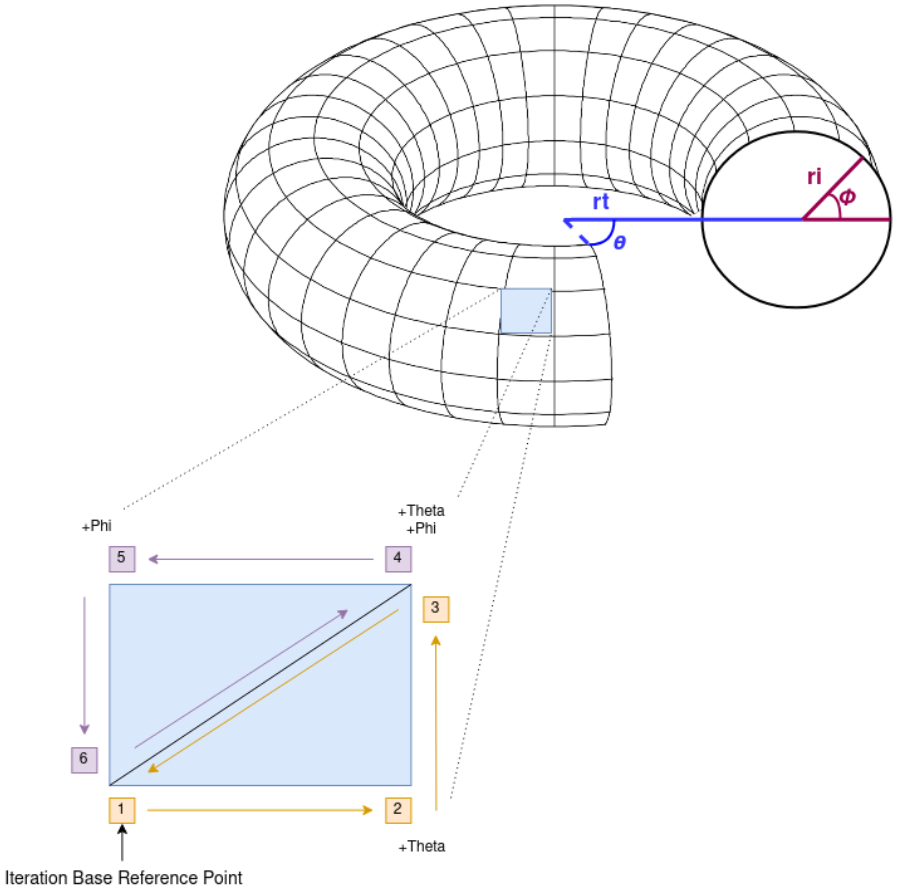


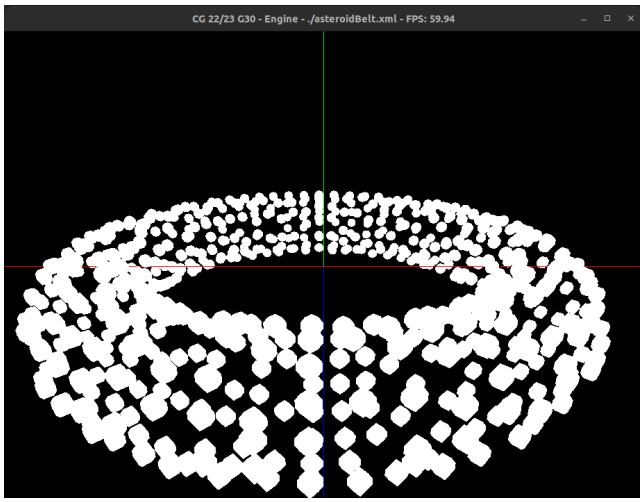
Figura 1: Construção da primitiva *Torus*

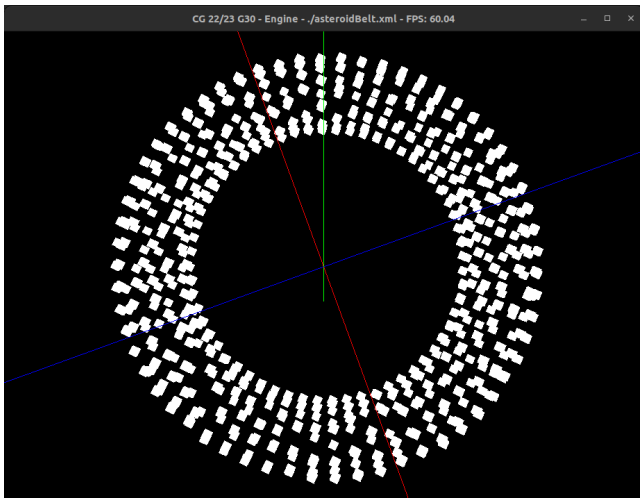
2.3 Cintura de Asteroides

Após o desenvolvimento da primitiva acima referida, o grupo ponderou a sua utilidade para a evolução da *demo* do sistema solar para além dos anéis de Saturno. Assim, chegamos à conclusão que, com algumas adaptações à primitiva original, conseguiríamos criar uma primitiva para simular a **cintura de asteroides** presentes no sistema solar (entre as órbitas de Marte e Júpiter) ao adicionar pequenas **esferas** em alguns pontos do ***torus***.

Desta forma, ao contrário da função original de cálculo dos pontos do *torus*, apenas calculamos um ponto por secção (ao invés de quatro), tendo sido o cálculo desta "secção"explicado acima (Figura 1). Para além disto, para conseguirmos obter posições aparentemente aleatórias para as esferas, implementamos iterações não uniformes sobre as *stacks*, isto é, a incrementação não é sequencial mas sim aleatória, podendo saltar no máximo 3 *stacks* à frente.

Por último, para facilitar a construção das esferas, foi também necessário adaptar a primitiva esfera para receber as coordenadas de um ponto, visto que a primitiva original constrói a esfera tendo como base a origem dos eixos (0,0,0). Desta forma, a cada ponto calculado em cada secção, é construída uma esfera com centro nesse mesmo ponto. Após a sua implementação, testamos a mesma obtendo os seguintes resultados:





3 Engine

3.1 Implementação de Animações

Uma das novas funcionalidades a implementar nesta fase inclui o suporte a 'animações' rudimentares dentro do **engine**, através do ficheiro de configuração inicial do mundo. Estas animações incluem a rotação de um modelo, dado o tempo para completar uma rotação de 360^{o} e a translação de modelos baseada numa curva cúbica de **Catmull-Rom**. Esta deverá ter os seus pontos de controlo definidos e o período no qual esta curva é terminada, sendo o objetivo que o modelo deverá seguir esta curva. O modelo também poderá ser sujeito a um "alinhamento"aquando segue esta curva, ou seja, poderá ser orientado de forma a seguir a tangente da curva.

3.1.1 Modificações Iniciais

Visto que nesta fase do projeto será necessário evoluir e adicionar novas funcionalidades às transformações geométricas a serem aplicadas aos modelos, sendo necessária a adição de novas variáveis e funções, decidimos dividir cada transformação geométrica numa classe independente, de forma a modularizar o projeto e evitar o desenvolvimento de uma classe 'monolítica' (**Grupo**).

As três novas classes criadas **rotate**, **scale** e **translate** são responsáveis pela aplicação da transformações geométricas de rotação, escala e translação, respetivamente, e todas estas possuem uma função de inicialização de leitura do seu nodo e atributos, assim como a aplicação da transformação. Para esta fase do projeto, as classes **rotate** e **translate** serão o principal alvo para a adição das novas funcionalidades de animação do **engine**.

Outra ligeira modificação efetuada para permitir a implementação das animações foi a criação de uma variável '**elapsedTime**', sendo esta responsável por indicar o tempo decorrido entre cada frame.

3.1.2 Rotação

Para a animação de rotação, foi apenas necessário calcular uma nova variável '**angPerTime**', calculada através da fórmula '**360 / time**', sendo de notar que este cálculo apenas ocorre caso um valor de tempo seja lido.

Após a leitura do nodo, basta apenas fornecer o '**elapsed time**' à função de aplicação da transformação, sendo apenas necessário calcular e aplicar a rotação a ser efetuada, como se apresenta na seguinte secção de código:

```
void rotate::calculateAndApplyRotate(float timeElapsed){
    this->rotAngle += timeElapsed * this->angPerTime;

    if (this->rotAngle >= 360) this->rotAngle -= 360;
    if (this->rotAngle < 0) this->rotAngle += 360;

    glRotatef(this->rotAngle,this->rotX,this->rotY,this->rotZ);
}
```

3.1.3 Curvas Cúbicas Catmull-Rom

Passando para a animação das translações baseada em curvas cúbicas de Catmull-Rom, o primeiro passo que a equipa tomou foi a implementação de funções auxiliares para efetuar os cálculos necessários, sendo que estas funções foram desenvolvidas num ficheiro auxiliar independente '**catmullRom.cpp**'. Este ficheiro contém funções de normalização, multiplicação de matrizes com vetores, construção da matriz de rotação e, principalmente, o cálculo de pontos numa curva, assim como a sua derivada, sendo a sua função principal descrita da seguinte forma:

```
void getCatmullRomPoint(float t, float *p0, float *p1, float *p2, float *p3,
                        float *pos, float *deriv) {
    // catmull-rom matrix
    float m[4][4] = {    {-0.5f,  1.5f, -1.5f,  0.5f},
                         { 1.0f, -2.5f,  2.0f, -0.5f},
                         {-0.5f,  0.0f,  0.5f,  0.0f},
                         { 0.0f,  1.0f,  0.0f,  0.0f}};

    for (int i = 0; i < 3; i++) {
        float vectorA[4] = {0,0,0,0};
        float vectorP[4] = { p0[i],p1[i],p2[i],p3[i] };
        multMatrixVector(*m,vectorP,vectorA);

        float vectorT[4] = { float(pow(t,3)),float(pow(t,2)),float(pow(t,1)),1 };
        float vectordT[4] = { float(3 * pow(t,2)),2 * t,1,0 };

        pos[i] = (vectorT[0] * vectorA[0]) + (vectorT[1] * vectorA[1]) +
        (vectorT[2] * vectorA[2]) + (vectorT[3] * vectorA[3]);
        deriv[i] = (vectordT[0] * vectorA[0]) + (vectordT[1] * vectorA[1]) +
        (vectordT[2] * vectorA[2]) + (vectordT[3] * vectorA[3]);
    }
}
```

Após o desenvolvimento destas funções auxiliares, adicionamos novas variáveis à classe **translate**, nomeadamente, a matriz bidimensional '**points**' que irá conter todos os pontos de controlo fornecidos no ficheiro de configuração inicial, e que, na invocação das funções auxiliares com o '**elapsedTime**' e estes pontos, irá obter o vetor de translação a ser aplicado.

Outro fator a ter em conta foi a necessidade de suportar o "alinhamento"do modelo à curva que efetua, sendo necessário aplicar uma rotação específica baseada no vetor tangente à curva no ponto em em que se encontra e o vetor '**Up**' do modelo. Para isso foi necessário criar uma variável '**previousY**' que grava o estado deste vetor '**Up**' conforme este modelo efetua rotações e, através do vetor da derivada obtido na função de catmullRom, podemos efetuar os seguintes cálculos para obter e aplicar a seguinte matriz de rotação:

```
float* li1M = deriv;
normalize(li1M);

float li3M[3] ;
cross(li1M, previousY, li3M);
normalize(li3M);

float li2M[3] = { 0,0,0 };
cross(li3M, li1M, li2M);
previousY[0] = li2M[0];
previousY[1] = li2M[1];
previousY[2] = li2M[2];
normalize(li2M);

float* m = (float* )malloc(sizeof(float) * 20);
buildRotMatrix(li1M, li2M, li3M, m);
glMultMatrixf(m);
```

3.2 Utilização de VBO's

Uma das modificações necessárias a esta fase a ser efetuada no engine é a utilização de **VBO's** (vertex buffer object), para o armazenamento e desenho de modelos, ao invés da utilização do modo imediato do OpenGL.

Visto que estas mudanças são modificações à lógica de desenho do modelo, foi apenas necessário alterar a classe '**model**' para adicionar o suporte à utilização de VBO's, começando pela adição da variável '**buffer**' na qual os pontos do modelos serão *buffered*. Para que isso aconteça foi necessário alterar a função de inicialização do modelo, em específico, após à leitura do ficheiro '**.3d**' estes pontos são *buffered* da seguinte forma:

```
glGenBuffers(1,buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer[0]);
glBufferData(GL_ARRAY_BUFFER, (long) (sizeof(float) * numFloats),
                points, GL_STATIC_DRAW);
```

Após o *buffering* feito, basta modificar a lógica de desenho do modelo para recorrer à utilização deste *buffer*, sendo apenas necessário definir o *buffer* a utilizar, o modo de desenho e o tamanho de pontos a desenhar, como descrito no seguinte segmento:

```
glBindBuffer(GL_ARRAY_BUFFER, buffer[0]);

glVertexPointer(3, GL_FLOAT, 0, nullptr);
glPolygonMode(GL_FRONT_AND_BACK, GL_POLYGON);

glColor3f(1.0f,1.0f,1.0f);
glDrawArrays(GL_TRIANGLES, 0, this->numPoints);
```

3.3 Evoluções Adicionais

Nesta secção o grupo irá apresentar novas funcionalidades desenvolvidas para além daquelas requeridas para esta fase do projeto. Estas novas funcionalidades são referentes ao ficheiro de configuração inicial e têm como objetivo principal auxiliar o utilizador através de opções de *debug* na definição de cenários.

3.3.1 Axis

Sendo este um novo atributo booleano do nodo '**world**', quando acionado (EX: '<**world axis="True"**/>), os 3 eixos X, Y, Z serão desenhados a partir da origem, com as respetivas cores: vermelho, verde e azul.

3.3.2 Line

Esta função foi desenvolvida em paralelo com a implementação da translação numa curva de Catmull-Rom e permite analisar em tempo real a totalidade da curva que o modelo irá seguir, assim como apresentar o vetor tangente ao ponto na curva. Para visualizar esta funcionalidade é necessário fornecer um valor booleano verdadeiro ao nodo **translate** o atributo '**line**'. (EX: '<**translate line="True"**/>)

3.4 Exemplo das funcionalidades adicionais

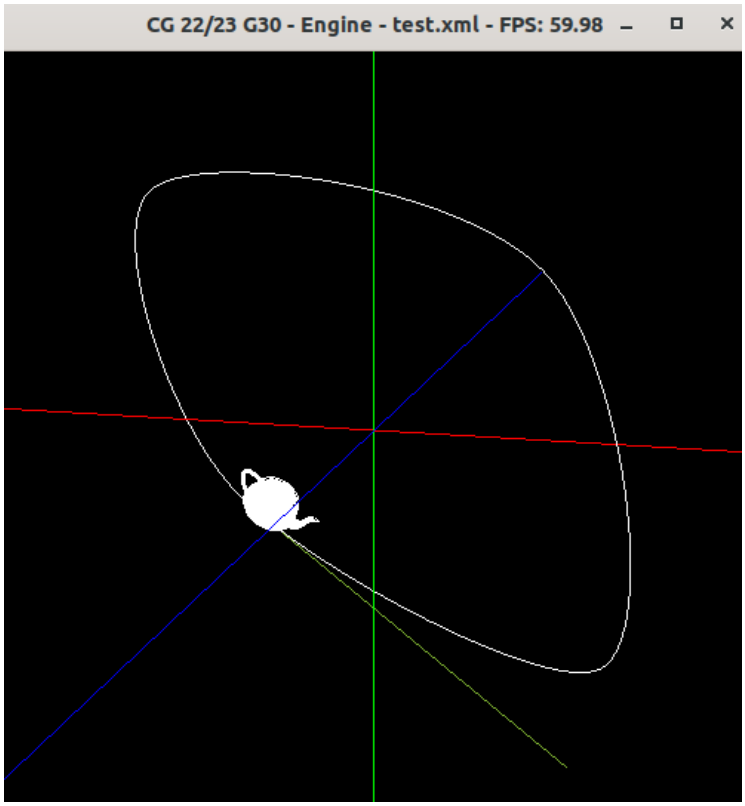


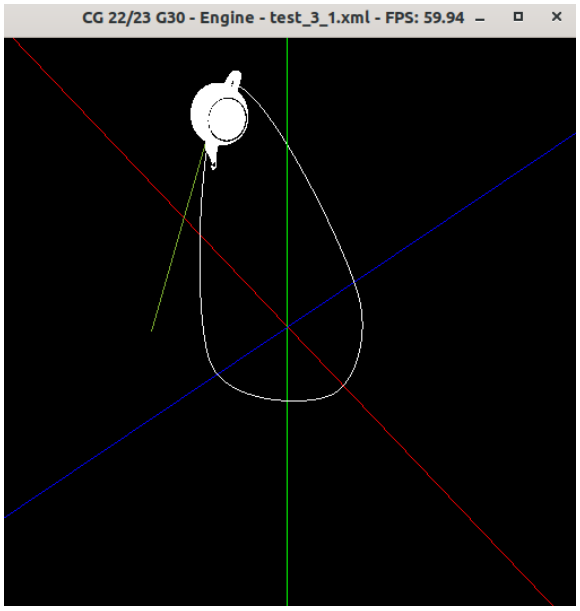
Figura 2: Teste das novas funcionalidades 'Axis' e 'Line'

4 Testes e Resultados

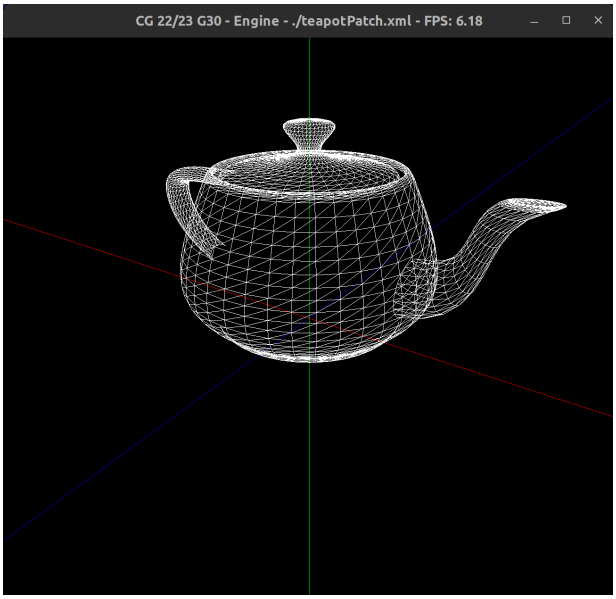
Nesta secção vamos apresentar os testes executados pelo grupo de modo a conseguir avaliar o funcionamento de ambas as aplicações quanto à sua correção e exatidão de resultados.

4.1 Testes Docentes

Alguns dos testes implementados foram cedidos pelos docentes de modo a verificar o comportamento correto das aplicações. Assim, apresentamos os resultados obtidos em cada um dos testes, sendo de notar que todos estes foram de encontro ao esperado.



test_3_1



test_3_2

4.2 Demo Sistema Solar

Para a demonstração desta fase, o grupo incorporou as novas funcionalidades, assim como algumas novas de forma a melhor representar o Sistema solar, sendo estas as seguintes:

- **Orbitas dos Planetas** - Os planetas possuem uma órbita em torno do sol, esta é representada numa escala onde cada segundo no *engine* equivale a cerca de 30 dias. (As luas à exceção da 'Lua' (Terra) não seguem esta escala, para evitar órbitas muito curtas.)
- **Rotação dos Planetas** - Os planetas possuem períodos de rotação. A escala desta é independente para evitar rotações elevadas (1s = 10h)
- **Cometas** - Foram criados dois cometas, um gerado aleatoriamente e outro baseado no cometa Halley, ambos representados pelo modelo de um **teapot** (de bezier patches).
- **Anél de Saturno** - Através da utilização da primitiva **torus** com um número de *stacks* igual a dois, foi possível criar um anel plano, utilizado para representar o anél de Saturno
- **Cintura de Asteroides** - Como dito anteriormente, através do algoritmo da primitiva **torus**, mas com a geração de esferas, foi possível criar uma cintura de asteroides.

Apresentamos de seguida a demonstração final do sistema solar obtida.

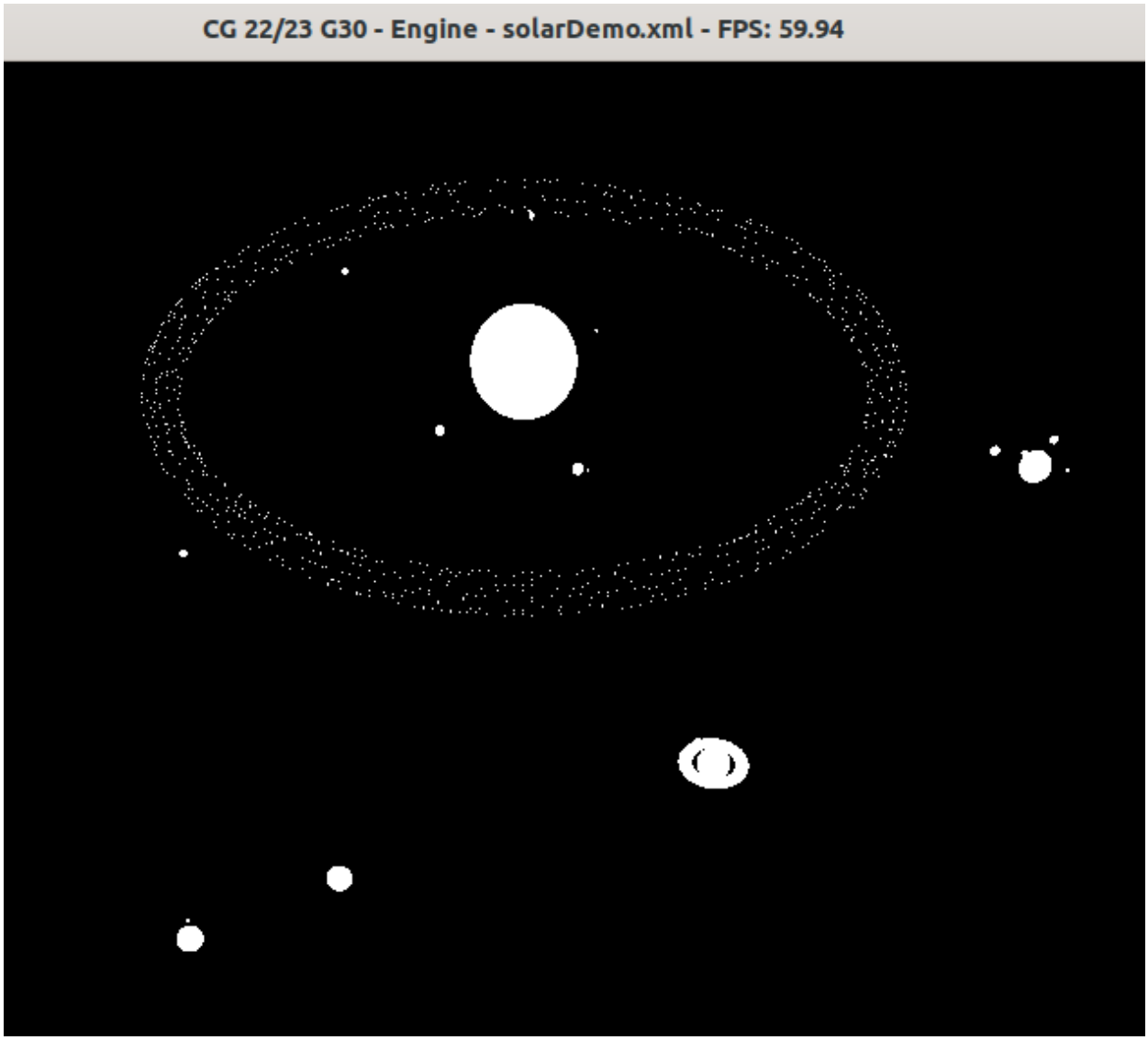


Figura 3: *Demo* Sistema Solar

5 Conclusão

Com a realização desta terceira fase do projeto, o grupo encontra-se satisfeito com o trabalho desenvolvido, tendo sido alcançados todos os objetivos estabelecidos quer pelos docentes, como pelo próprio grupo.

Em suma, aprofundamos o conhecimento adicional lecionado nas aulas quer teóricas quer práticas após a segunda fase do projeto, que nos ajudaram a perceber e a implementar técnicas de leitura de ficheiros *patch* e consequente manipulação para cálculo de pontos *Bezier*, utilização de VBO's para um *render* mais eficiente e aplicação de curvas cúbicas Catmull-Rom.