Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Computação Gráfica

Ano Letivo de 2023/2024

Trabalho Prático
Fase 3 - Curvas, Superfícies cúbicas e VBO's

António Filipe Castro Silva(a100533) Diogo Rafael dos Santos Barros(a100600)
Duarte Machado Leitão(a100550) Pedro Emanuel Organista Silva(a100745)

26 de abril de 2024

CG

Índice

Lista de Figuras

1 Introdução

Este relatório foi realizado no âmbito da 3.ª fase do Trabalho Prático da Unidade Curricular de Computação Gráfica que se continua a utilizar como base o trabalho desenvolvido nas fases anteriores criando nesta fase novas funcionalidades quer no *Generator*, quer na *Engine*, mais concretamente sobre curvas, superfícies cúbicas e VBO's.

Em relação ao *Generator*, as alterações principais foram feitas devido ao facto que agora o *Generator* tem de conseguir desenvolver ficheiros ".3d" a partir de ficheiros ".patch" utilizando as **superfícies de Bezier**, pois estes ficheiros são constituídos por **patches de Bezier**. Para tratarmos dessas superfícies criamos outro ficheiro para tratar da matemática delas.

Em termos de alterações na parte da *Engine*, tivemos de alterar o Leitor para este passar a ler o **Time** que pode aparecer ou não num rotate ou num translate de forma a dizer quando tempo demoram a ser feitos e os pontos de controlo nos Translates . Nos translates, os pontos de controlo que agora podem aparecer são, no mínimo, 4 pontos de controlo que servem para definir as **Curvas de Catmull-Rom**. Tivemos de definir um novo ficheiro para tratar da matemática por detrás destas novas curvas. Além disso, também tivemos de ter em conta se é pedido que a figura venha *aligned* e, se sim, fazer esse alinhamento de algum modo. Implementamos também os VBO's de forma que, no futuro, quando tiver um elevado número de pontos a serem renderizados ao mesmo tempo, continuarmos a ter bons *frames por segundo* (**FPS**), pois, atualmente, como ainda não são muitos, estes ainda não têm muito impacto.

Por fim, tal como na última fase, além de correrrmos os testes, fizemos um sistema solar, desta vez dinâmico, metendo todos os planetas a rodar com curvas de catmull-rom personalizadas demorando um certo tempo dependendo da sua localização e depois colocamos um cometa em órbita, sendo que este foi feito por nós a partir de um script python que cria um ".patch", também este estando em órbita utilizando uma curva de Catmull-Rom.

2 Generator

2.1 Geração de figuras a partir de ficheiros .patch

De forma a extrair a informação sobre os pontos de uma figura a partir de um ficheiro **".patch"**, começou-se por criar 4 funções que tratariam de extrair esses valores.

- **getPatchPontos** -> esta função lê o ficheiro patch, dá skip às linhas dos patches em si e guarda os pontos de controlo todos numa lista de pontos, retornando essa lista.
- **getPatches** -> parecida com a função anterior, mas esta em vez de ler os pontos de controlo, esta função extrai os índices desses pontos e organiza-os numa lista. Retorna então uma lista de listas de inteiros, onde cada lista interna representa os índices de controlo de uma patch.
- **getPontosFromPatches** -> esta função recebe como argumentos o que se consegue retirar a partir das últimas duas funções e utiliza essas informações para montar uma lista de listas de pontos, onde cada lista interna representa os pontos de controlo de uma patch em específico.
- **generateFromPatch** -> esta é a função que realmente trata da criação da Figura em si. Começa por correr as três funções anteriores obviamente para calcular a lista de listas de pontos de controlo de cada patch. Depois pegando em cada lista, de acordo com a *tesselation* escolhida pelo utilizador, divide cada patch em quadrados menores, calculando a *divisionSize*, o **u0** e o **v0**. Depois calcula os pontos intermédios dentro desses quadrados menores utilizando Bezier e essas três variáveis, que foram calculadas a partir da *tesselation* e em que parte do ciclo se encontra, utilizando a função **calculateBezierPoint** e adicionando cada ponto à Figura.

Estas 4 funções em conjuntos com o ficheiro "bezier.cpp"para fazer os cálculos de cada ponto, gera um novo ficheiro ".3d"com todos os pontos criados. Vamos explicar como funciona a matemática do ficheiro *bezier.cpp* na próxima secção.

Na main do Generator, de forma que esta agora conseguisse identificar o pedido para tratar de um ficheiro ".patch", tivemos de acrescentar mais um caso para isso. O modo de execução para a geração de uma figura a partir de ficheiros patch é da seguinte forma:

```
1    ./FASE3/build/Release/generator.exe patch
       path/ficheiro.patch tesselations path/out.3d
```

Para correr então a criação de um ficheiro *.3d* a partir de um ficheiro *.patch*, começa-se por executar o generator, dizer que é um patch que vai ser uysado, identificar o path do ficheiro *.patch*, o número de *tesselations* que queremos e, por fim, para onde o ficheiro ".3d"tem de ir. Para facilitação, colocamos a criação do teapot dos professores na nossa makefile:

```
1    ./FASE3/build/Release/generator.exe patch
        FASE3/patches/teapot.patch 10 FASE3/out/bezier_10.3d
2    @echo "Bezier done"
```

2.2 Bezier

Tal como já mencionamos, para tratamos do cálculo do Bezier criamos um ficheiro à parte chamado **bezier.cpp** que é constituído por 3 funções que tratam de fazer o cálculo que retorna no final um **"Ponto de Bezier"**.

As três funções são as seguintes:

- **calculateBezierPoint** -> inicializa um ponto nas coordenadas (0,0,0) e depois utilizando um loop duplo percorre todas as combinações de índices para os pontos de controle do patch Bezier (4x4 pontos). Para cada combinação calcula os coeficientes de Bernstein para as coordenadas (u,v) utilizando a função **coef**. Multiplica as coordenadas do ponto de controlo pelo coeficiente calculado e altera o ponto inicializado para essas novas coordenadas. Por fim, retorna o ponto calculado depois de passar pelo patch Beizer todo.

- **coef** -> função que calcula os coeficientes de Bernstein para uma determinada ordem e posição utilizando a função **binomial** e mais dois parâmetros: **umMenosT** e **tElevadoI**.

- **binomial** -> função recursiva que calcula os coeficientes binomiais que a função **coef** necessita para o cálculo dos coeficientes de Bernstein

Com estas três funções conseguimos com sucesso programar a fórmula de interpolação de Bezier de forma que conseguissemos calcular um ponto dentro de um patch Bezier, recebendo só um conjunto de pontos de controlo e duas coordenadas paramétricas.

3 Engine

A primeira alteração que tivemos de fazer na Engine foi ter de adicionar o *glutIdleFunc(renderScene)* na main, fazendo com que a cada ciclo do glut, a função de renderização fosse corrida, renderizando todas as alterações que são feitas ao longo do tempo. Para isso agora mantemos na renderScene uma variável *elapsedTime* que utiliza a função *glutGet(GLUT_ELAPSED_TIME)* para se saber quanto tempo se passou desde o início do programa, pois agora nesta fase a *Scene* deixa de ser algo estático, mas sim algo dinâmico fazendo com que certas figuras se possam alterar devido ao tempo passado.

Para verificarmos a quantos FPS a *Scene* se encontra, fizemos uma pequena função chamada **fpsCounter** que calcula os FPS atuais e depois fizemos outras duas funções chamadas **drawTexts** e **drawStats** que juntas escrevem no canto inferior esquerdo da Scene esse valor bem como outras informações de estado do sistema.

3.1 Alterações no Leitor e nas Transformações

Visto que agora nos XML podem aparecer novos dados, sendo eles: **time, align e pontos de controlo** nos translates, antes de mexermos na função **extrair_transform** do Leitor, tivemos de alterar a estrutura Transform do ficheiro *groups* que existia previamente para puder armazenas os novos dados.

```
struct transform{
    TransformType type;
    float time;
    bool align;
    std::list<Ponto> controlPoints;
    std::list<Ponto> points;
    float angle;
    float x;
    float y;
    float z;
};
```

Figura 3.1: Struct Transform alterada

Como podemos ver, o **time** é guardado num float, representando o número de segundos que são escritos lá, o **align** é guardado num bool, dizendo se é verdade ou não que o translate tem de ser alinhado na curva e os pontos de controlo são guardados numa lista de Pontos chamada **controlPoints**. Além disso, também vamos guardar no transform os pontos da curva

de Catmull-Rom (lista de pontos chamada **points**), sendo essa uma das formas que guardamos a curva, mais tarde explicaremos a outra.

Depois de fazermos estas alterações na estrutura, realmente pudemos começar a fazer as alterações na função **extrair_transform** que extrai as transformações dos *groups*. Vamos explicar a extração das transformações por tópicos:

- Scale -> Manteve-se inalterado em relação à fase anterior
- Translate sem time -> Manteve-se inalterado em relação à fase anterior, pois é uma translação estática, é algo instantâneo e fica logo feito.
- Translate com time -> Quando se repara que um translate utiliza um time, passamos a saber que representa uma curva de Catmull-Rom, logo este necessita de um tratamento diferente. Começamos por pegar no valor do align para sabermos se o model vai ter de ir alinhado com a curva ou não. Depois colocamos todos os pontos de controlo que aparecem no translate numa lista de pontos, calculamos os pontos de Catmull-Rom e colocamos na outra lista. Por fim, guardamos esta curva de duas formas: como uma Figura e como parte do Transform como já demonstramos. Na secção de Catmull-Rom explicaremos com mais detalhe como isso funciona.
- Rotate sem time -> Manteve-se inalterado em relação à fase anterior, pois é uma rotação estática, é algo instantâneo e fica logo feito.
- Rotate com time -> Se é uma rotação com time, em vez de fazer uma rotação de certo ângulo, o time representa quanto tempo o rotate demora a fazer 360º ao torno das coordenadas dadas, ou seja, se o time for 5 e as coordenadas forem (0,1,0), significa que ele demora 5 segundo a dar uma volta completa à volta do eixo y.

Depois de termos alterado a extração do transform, tivemos de tratar da aplicação deles na função **applyTransforms**. Tal como na explicação anterior, vamos explicar por tópicos, mas desta vez só os que sabemos que foram alterados.

- Translate com time -> começamos por pegar nos pontos de Catmull-Rom do Transform, calculamos um valor normalizado da Transformação com base no tempo que já se passou (sendo este o valor t regularmente usado no cálculo de pontos Catmull-Rom) e calculamos um vetor de pontos com base no valor normalizado, o primeiro ponto é um ponto de Catmull-Rom e o outro é um ponto derivado que será usado para calcular o align. Usamos o primeiro ponto da lista de pontos da figura como referência da direção da translação, calcula-se as diferenças nas coordenadas entre o ponto de Catmull atual e o ponto de referência para obter o vetor de translação, e aplica-se essa transformação a todos os pontos da figura. Se o align for true, utiliza-se três novas funções do ficheiro Ponto(**normalize, innerProduct e cross**) para calcular o ângulo de rotação e o eixo de rotação e, para finalizar, para cada ponto na lista original de pontos, a translação é aplicada adicionando as diferenças que foram calculadas e, se o alinhamento for verdadeiro, também se aplica uma rotação.

- Rotate com time -> começa por calcular o ângulo de rotação tendo em conta o tempo que decorreu desde o último frame até ao tempo atual (instantBefore e elapsedTime) em relação ao tempo total para a rotação, sendo que esta é calculada em graus completos divididos pelo tempo total. Existem dois tratamentos para a rotação da figura: ser uma curva ou não. Se não for, cada ponto da figura é rodado, mas se for são os pontos de controlo da curva que são rodados.

Conseguimos ver então que tratamos de todas as novas transformações pedidas. Temos em conta os efeitos que o tempo tem, vemos se é pedido que a figura vá alinhada e utilizamos os pontos de controlo para calcular as curvas de Catmull-Rom quando estas aparecem num translate com time.

3.2 Curvas de Catmull-Rom

Como já mencionamos, estas curvas são guardadas de duas formas simultaneamente: dentro do Transform onde se guarda os seus controlPoints e os points que calculamos depois delas e também como uma Figura apenas para efeitos de visualização. Para fazermos essa diferenciação numa Figura se é uma curva ou uma Figura normal como uma Plane ou um Teapot, tivemos de lhe alterar a estrutura.

```
struct figura
{
    bool curva;
    std::list<Ponto> pontosControl;
    std::list<Ponto> pontos;
};
```

Figura 3.2: Struct Figura alterada

Aquele bool serve para dizer se é True ou False que aquela figura é uma curva. A lista de pontos manteve-se inalterada, pois quer uma Figura normal, quer uma Curva tem a sua lista de pontos. Tivemos foi de acrescentar a lista de pontos **pontosControl** para guardarmos a lista de pontos de controlo para facilitar as rotações gerais e a rotação da curva em si mesma. De forma a que não houvesse problemas de desincronização entre estas listas de pontos na Figura e as mesma num Transform, ambas estas listas utilizam o mesmo pointer, logo ao alterarmos numa dela, automaticamente estamos a alterar na outra ao mesmo tempo. Desta maneira conseguimos aplicar transforms à curva e ao mesmo tempo essas alterações serão aplicadas na curva como transform.

De forma a facilitar contas, a curva é criada deste modo: criada como Figura quando estamos a extrair o transform, ou seja, ambas são criadas ao mesmo tempo, a do Transform e a da Figura. Esta é transformada no apply_Transforms onde rodamos os pontos de controlo, mas como o apontador é o mesmo, além de alterarmos na Figura, vamos estar a alterar no

Transform e, por fim, é atualizada na Engine utlizando a função **pontosCatmullParaDesenho**. Houve umas situações que tivemos de alterar de Figura, para void* por erros de includes, mas visto que sabemos que estamos a tratar de uma Figura, não houve problema nenhum.

O nosso Align formulado para as curvas de Catmull-Rom utiliza o primeiro ponto da lista original de pontos da figura como referência da direção da translação, por exemplo, encontra-se alinhado com o centro da Figura, ou seja, de certo modo o nosso Align faz um alinhamento, mas não com a frente da Figura.

Antes de explicar a matemática que foi utilizada para as curvas de Catmull-Rom, vale a pena mencionar que adicionamos a opção de ativar e desativar o desenho das curvas na Scene utilizando a tecla C no teclado. Além disso, acrescentamos no drawStats para mostrar no canto inferior esquerdo um ON/OFF caso as curvas estejam ligadas/desligadas.

Como já dissemos, para tratar dos cálculos das curvas de Catmull-Rom criamos um novo ficheiro *catmull.cpp* onde se encontram as duas funções que tratam de fazer os cálculos destas curvas:

- reorganizeControlPoints -> recebe um float t, que representa a posição na curva de Catmull-Rom, e uma lista de pontos de controlo. Esta função vai reorganizar os pontos de controlo de acordo com cálculos utilizando esse t, posteriormente calcula o índice dos pontos de controlo adjacentes necessários para fazer a interpolação e no fim retorna um vetor contendo os pontos de controlo reorganizados

- getCatmullRomPoint -> recebe o mesmo que a última função, converte a lista num vetor e usa a função anterior para reorganizar os pontos de controlo mandando o valor t recebido. Calcula os coeficientes do polinómio cúbico de Catmull-Rom utilizando a variável **newT** que é calculado na função anterior. Depois faz-se a interpolação dos pontos de controlo utilizando estes coeficientes. Para tratarmos do align tivemos de calcular as derivadas dos coeficientes, calcular a tangente (derivada da posição) e normalizar essa tangente. No final, devolve o ponto de Catmull-Rom que foi calculado e a sua tangente para ajudar no align.

Foi com este ficheiro que conseguimos com sucesso processar toda a matemática que existe no tratamento das curvas de Catmull-Rom

3.3 VBOs

Um dos pedidos desta fase era que os modelos deixassem de ser desenhados em modo imediato como foi feito até agora, mas sim utilizando **Vertex Buffer Objects (VBOs)**, pois estes ajudam a ganhar FPS quando o número de pontos escala drasticamente. O programa começa com eles desativados, mas clicando no V, estes são ativados e aparece no canto inferior esquerdo, utilizando novamente o **drawStats**, **VBO: ON**.

Começamos por criar 3 variáveis globais:

- **vector<GLuint> buffers** -> é um vetor de GLuint que armazena os identificadores dos VBOs criados, em que cada elemento desse vetor vai corresponder a um VBO que vai ser associado a uma Figura.

- **vector<vector<float*>> figurasVertices** -> é um vetor de vetores de pointers para floats. Este vetor armazena os ponteiros para os vértices das Figuras. Cada elemento do vetor externo corresponde a uma Figura e contém os ponteiros para os vértices dessa Figura. A utilização de pointers aqui melhora performance do update dos buffers visto que não há necessidade de aceder as figuras.

- **bool VBOstate** -> é um boolean que serve para indicar se os VBOs estão usados para renderização ou não.

Depois criamos a função **loadBuffersData** que é responsável por carregar os dados das Figuras nos buffers dos VBOs. Ela realiza a geração dos identificadores dos VBOs, associa os dados das Figuras aos VBOs e armazena os ponteiros para os vértices das Figuras no vetor *figurasVertices*. Ela funciona de duas formas: na primeira chamada quando *elapsedTime* ainda é igual a 0, ela gera os buffers dos VBOs, associa os dados das Figuras a esses VBOs e armazena os ponteiros para os vértices no vetor *figurasVertices*; nas chamadas seguintes simplesmente atualiza os ponteiros e os dados dos VBOs com os novos dados das figuras.

Após criarmos esta função, tivemos de alterar a função **drawFiguras** de modo a que ela desenhasse utilizando os VBOs. Continua a ter parte do código das fases anteriores, quando os VBOs estão desativados, simplesmente agora dentro dessa parte tem um if para o caso de uma das Figuras ser uma curva e o desenho das curvas estar ativado para utilizar a função **pontosCatmullParaDesenho** para calcular os pontos para depois fazer o desenho dessa curva utilizando GL_LINE_LOOP. No caso dos VBOs estarem ativados, também vai ter esse if, mas vai utilizar funções diferentes para fazer os desenhos sendo elas **glVertexPointer** e **glDrawArrays**, sendo que a segunda função, tal como no modo imediato, recebe GL_TRIANGLES ou GL_LINE_LOOP dependendo se é para desenhar uma Figura ou uma curva.

Por fim, na renderScene, tivemos de acrescentar que, se os VBOs estiverem ativados, tem de utilizar a função loadBuffersData para atualizar os dados dos buffers. Na main, acrescentamos o **glewInit** que é necessário para usar os VBOs e logo de início faz-se a primeira chamada do loadBuffersData para guardar as informações das Figuras nos buffers.

Conseguimos então implementar com sucesso os VBOs que nos vão ajudar na próxima fase quando for necessário desenhar uma quantidade absurda de pontos simultaneamente.

3.4 Saída limpa

Resolvemos tratar nesta fase de alguma forma de memory leaks e deixar o fecho do programa um pouco mais limpo. Para fazermos isso criamos uma função onExit que é chamada quando se quer fechar o programa utilizando outra função chamada **atexit** standard do C++ que é executada quando o programa termina.

```
void onExit(){
    for(GLuint buffer: buffers){
        glDeleteBuffers(1, &buffer);
    }
    for(vector<float*> vetor: figurasVertices){
        vetor.clear();
    }
    figurasVertices.clear();
    figuras.clear();
    deleteLeitor(leitor);
}
```

Figura 3.3: Função de saída do programa

Conseguimos ver que esta função trata de dar delete aos buffers que são usados para guardar os VBOs, limpa todos os vetores que contêm vértices dentro do vetor figurasVertices e, no final, dá clear a esse vetor, à lista de figuras e ao leitor.

4 Testes e Sistema Solar Dinâmico

4.1 Testes

Tal como temos feito em todas as fases até agora, antes de iniciarmos a formulação dos nossos próprios testes, começamos por utilizar os dos professores para sabermos em que estado estamos. Deste modo, ao corrermos o teste do teapot conseguimos ver que já tinhamos implementado com sucesso as curvas de Bezier e a leitura de patches.

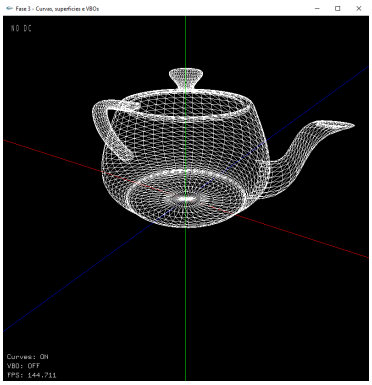

Figura 4.1: Teste do teapot

Depois do teste do teapot procedemos para correr o teste em que se testa se temos as curvas de Catmull-Rom a funcionar, tal como o align.

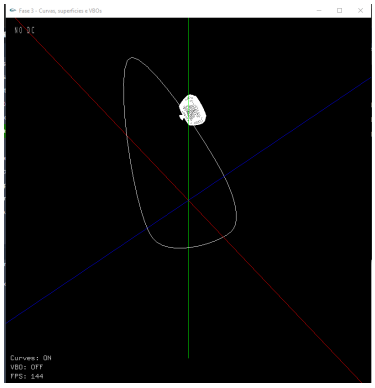

Figura 4.2: Teste do Catmull-Rom e Align

Como este teste já é algo mais dinâmico, não dá para demonstrar da melhor forma num relatório o que acontece, por isso procederemos a uma descrição de como este funciona: consegue-se ver com sucesso a curva de Catmull-Rom e a mesma a rodar; consegue-se ver também claramente que o Align não está 100% correto pois não é a frente do teapot que está a seguir a linha, mas consegue-se ver que de alguma forma o centro da figura está alinhada com a curva.

4.2 Sistema Solar Dinâmico

Após verificarmos que os testes dos professores estavam corretos, procedemos à formulação do nosso sistema solar dinâmico, que é basicamente uma evolução do sistema solar que tinhamos na fase anterior. Simplesmente agora cada planeta tem um tempo de translação que utiliza pontos de controlo para formular a curva de Catmull-Rom para fazer a sua órbita. O tempo de translação é proporcional à realidade, tendo, por referência, o período de translação da Terra que foi definido para 5 segundos, sendo esse o caso, Mercúrio demora 1.205 segundos e Neptuno demora 823.945 segundos, perto de 14 minutos.

```
<group>
    <transform>
        <translate time = "5" align = "false">
            <point x="0.1" y="0.0" z="2.5" />
            <point x="1.7069690242163482" y="0.0"
               z="1.915111107797445" />
            <point x="2.56201938253052" y="0.0"
               z="0.43412044416732604" />
            <point x="2.265063509461097" y="0.0"
               z="-1.2499999999999996" />
            <point x="0.9550503583141722" y="0.0"
               z="-2.349231551964771" />
            <point x="-0.7550503583141717" y="0.0"
               z="-2.3492315519647713" />
            <point x="-2.0650635094610963" y="0.0"
               z="-1.2500000000000001" />
            <point x="-2.3620193825305202" y="0.0"
               z="0.4341204441673249" />
            <point x="-1.506969024216349" y="0.0"
               z="1.9151111077974445" />
        </translate>
        <scale x="0.018" y="0.018" z="0.018" />
        <translate x="2.5" y="0" z="0" />
        <rotate angle="320" x="0" y="1" z="0" />
    </transform>

    <models>
        <model file="../out/planet.3d" /> <!-- generator
           sphere 1 10 10 planet.3d -->
    </models>
</group>
```

Figura 4.3: Group da Terra

Decidimos fazer uma redução de stacks e divisions nos planetas, visto que são pequenos e não se vê grande diferença e com essa redução há uma melhoria de performance, mas mantivemos as mesmas no Sol devido à sua dimensão. Também removemos a Lua pois não conseguimos pô-la a rodar à volta da Terra e achamos que tirava uma certa vericidade do sistema solar.

Também era nos pedido que fizessemos um cometa que tinha como trajetória uma curva de Catmull-Rom, tal como fizemos para os planetas. Além disso, o cometa tinha de ser feito utilizando Bezier patches, por exemplo, os pontos de controlo do teapot. Nós decidimos ir um passo à frente e criar um programa python que gera pontos para criar uma representação geométrica de uma esfera oval e no final guarda-os organizados com o formato preciso para ficarem num ".patch". Com esse **comet.patch**, utilizando o generator e 2 de tesselation criamos o **comet_2.3d** para desenharmos o cometa no sistema solar. Criamos um pequeno XML para o vermos de perto antes:

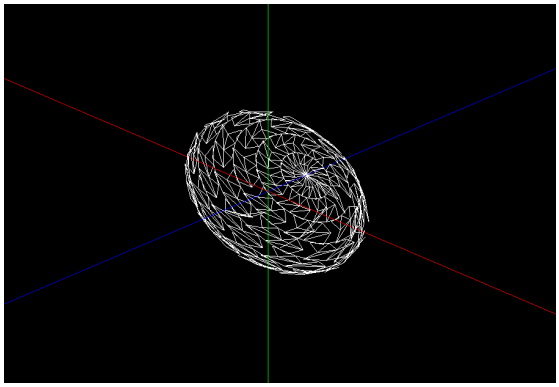

Figura 4.4: Desenho do cometa visto de perto

Após termos feito o cometa, finalmente temos o sistema solar dinâmico completo. Claramente no relatório não dá para ver, mas todos os planetas seguem aquelas curvas cada um com o seu tempo diferente. Há algumas curvas que parecem estar umas em cima das outras mas na verdade estão em cima ou em baixo, devido à inclinação natural que estas têm na vida real.

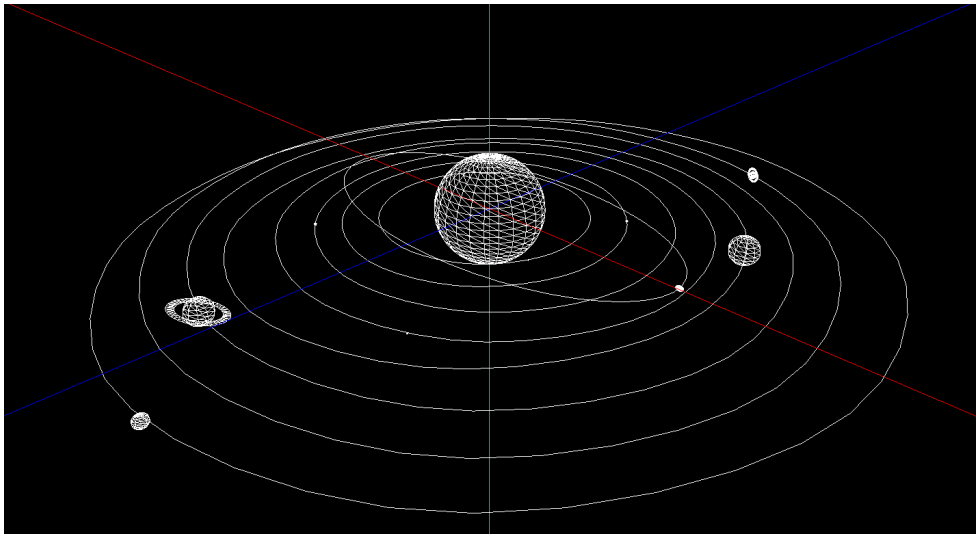

Figura 4.5: Sistema solar dinâmico completo

5 Conclusão

Com a resolução desta terceira fase do trabalho prático conseguimos consolidar de que forma funcionam as superfícies de Bezier, curvas de Catmull-Ron, utilização de tempo em computação gráfica e como funcionam os VBO's.

Em termos de satisfação, encontramo-nos satisfeitos, pois conseguimos implementar o Bezier para tratar das patches, implementamos com sucesso as curvas de Catmull-Ron, conseguimos fazer um bom sistema solar dinâmico com um cometa feito por nós, tratamos do *time* com sucesso e temos um *align* dentro dos possíveis e conseguimos implementar também com sucesso os VBOs, conseguindo trocar o método a qualquer momento. Além de termos conseguido implementar tudo o que era pedido, conseguimos fazer os testes com sucesso e fazer um bom sistema solar dinâmico, criando nós mesmos um *".patch"* para o cometa. Também fizemos o contador de FPS aparecer na Scene, tal como o estado das curvas e dos VBOs, e melhorar o fecho do programa para ser mais lempo

Concluindo, tendo em conta tudo o que produzido nesta fase, consideramos que nos encontramos com tudo o que é necessário para progredir para a última fase do trabalho prático.