Universidade do Minho

Departamento de Informática
Mestrado Integrado em Engenharia Informática
3º Ano, 2º Semestre

# Computação Gráfica

## Trabalho Prático - Fase III

Grupo 21

Ana Pereira
A81712

Francisco Freitas
A81580

Maria Dias
A81611

Pedro Freitas
A80975

April 20, 2019

# Índice

# 1 Introdução

Partindo do trabalho já desenvolvido nas duas fases que se sucederam, passamos agora à terceira fase de construção de uma aplicação que nos permitirá representar o Sistema Solar.

Nesta fase do projeto, pretende-se a inclusão de curvas e superfícies cúbicas no modelo já desenvolvido, com o objetivo de criar um modelo dinâmico, no qual se inclui um cometa a girar em torno do Sistema Solar.

Assim, a aplicação *generator* passará nesta fase a conseguir traduzir *Bezier patches* em modelos que o *engine* é capaz de desenhar. Com esta nova funcionalidade, o *generator* passa então a poder receber como argumentos o nome de um ficheiro, contendo vários pontos de controlo relativos a vários patches, um nível de tecelagem e, como resultado, guarda num ficheiro de saída uma lista de pontos que definem os triângulos dessa superfície.

A aplicação *engine* também sofrerá alterações, sendo que aos elementos `translate` e `rotate` foi acrescentado um novo atributo, `time`, que é aplicado a cada um destes elementos de forma a definir curvas Catmull-Rom, que definirão o movimento dos modelos aos quais são aplicados. Para além disso, os modelos passarão agora a ser desenhados com o auxílio de VBOs, ao invés de os desenhar imediatamente, melhorando assim a performance do programa.

Após aplicar no nosso modelo as alterações referidas, ficaremos com um modelo do Sistema Solar ainda mais realista do que o da fase anterior, pois passa de um modelo estático a dinâmico, aproximando-se do verdadeiro sistema que pretendemos modelar.

# 2 Arquitetura do Código

Para esta fase foram mantidas as classes definidas nas etapas anteriores: *Point*, *Shape*, *Group* e *Action*. Sendo que esta última classe sofreu alterações de modo a introduzir curvas Catmull-Rom na nossa aplicação. Foi ainda necessário a criação de uma nova classe, *Bezier*, de forma a adicionar a funcionalidade de reproduzir figuras descritas num ficheiro de formato .patch.

## 2.1 Classes inalteradas

### 2.1.1 Point.cpp

Classe que guarda as coordenadas x, y e z de um ponto, necessário para o desenho de cada vértice de um triângulo, uma vez que a base de todas as figuras desenhadas é o triângulo.

### 2.1.2 Shape.cpp

Classe que armazena num vector o conjunto de todos os pontos pertences a um determinado modelo.

### 2.1.3 Group.cpp

Classe que guarda num vetor todas as ações presentes num determinado group do ficheiro XML e noutro vetor o conjunto das figuras necessárias para o desenho desse mesmo group.

## 2.2 Classes novas/alteradas

### 2.2.1 Action.cpp

Classe que representa as ações que podemos aplicar aos modelos. Para a realização desta etapa foi necessário criar a subclasse *Translate* que para além da tag, x, y e z herdados acrescenta ainda a variável **time**, que define o número de segundos que demora a percorrer a curva Catmull-Rom e ainda vetores que guardam os pontos dessa mesma curva.

A variável **time** também foi adicionada à subclasse *Rotate*.

### 2.2.2 Bezier.cpp

Classe encarregue de efetuar o parse do Bezier patch e conversão do mesmo num ficheiro de formato .3d, contendo os vértices da figura a desenhar.

# 3 Generator

Tal como já acontecia na fase anterior, o generator é responsável por gerar ficheiros contendo o conjunto de vértices que formam as diferentes primitivas geométricas que se pretende gerar, conforme alguns parâmetros. Nesta fase, foi incluída uma nova funcionalidade a esta aplicação, sendo agora possível gerar modelos a partir de *Bezier patches*.

## 3.1 Bezier Patches - Implementação

### 3.1.1 Leitura de Ficheiros .patch

Em primeiro lugar, fez-se uma análise ao ficheiro do formato patch que nos foi fornecido, de forma a definir uma estratégia para a leitura e interpretação do mesmo. A primeira linha do ficheiro define o **número de patches** contidos no mesmo. Este número foi lido e, com este, tínhamos o número de iterações necessárias a fazer para capturar os índices dos pontos de controlo relativos a cada um dos patches. Cada uma dessas linhas foi lida de forma a guardar individualmente cada um dos índices num array. De seguida, surge uma linha que contém o **número de pontos de controlo**, ou seja, o número de linhas que teríamos de ler após esta. Então, para cada uma dessas, lemos as três coordenadas que definem cada ponto.

Posto isto, podemos aplicar aos pontos capturados os cálculos de Bezier, de forma a obter as coordenadas reais dos vértices do modelo definido no ficheiro patch.

### 3.1.2 Curvas e Superfícies Bezier

Para criar uma curva Bézier precisamos de quatro pontos, designados por pontos de controlo. Uma curva não existe até se combinar estes 4 pontos, pesando-os com alguns coeficientes. Uma curva paramétrica é uma curva definida por uma equação. Como tal, tem uma variável, que se designa por parâmetro e normalmente é identificado pela letra t, que representa a tesselagem e varia entre 0 e 1.

A equação geral para uma destas curvas é dada por:

$$P_{curva}(t) = p0 \times k_0 + p1 \times k_1 + p2 \times k_2 + p3 \times k_3,$$

onde $p0$, $p1$, $p2$ e $p3$ são os pontos de controlo, e $k_0$, $k_1$, $k_2$ e $k_3$ são os coeficientes que calculam o peso de cada ponto de controlo. Os coeficientes $k_i$ são calculados a partir do valor de t, conforme as expressões de polinómios de *Bernstein*:

$$\begin{aligned}
k_0(t) &= B_{0,3}(t) = (1-t) \times (1-t) \times (1-t) \\
k_1(t) &= B_{1,3}(t) = 3(1-t)^2 \times t \\
k_2(t) &= B_{2,3}(t) = 3(1-t) \times t^2 \\
k_3(t) &= B_{3,3}(t) = t^3
\end{aligned}$$

Com a expressão anterior conseguimos calcular as coordenadas de qualquer ponto na curva. Assim, para a visualizar, é necessário calcular o resultado da equação da

curva para diferentes valores de t, incrementados num intervalo certo, obtendo assim vários pontos da curva. Como seria de esperar, quanto menor for o intervalo entre os valores de t, mais preciso será o resultado final.

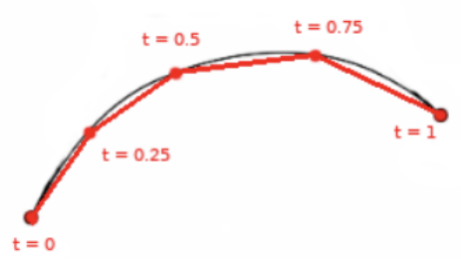
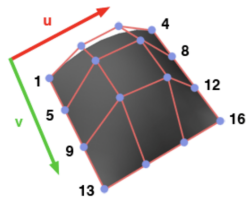



Figura 1: Pontos numa curva calculados para diferentes valores de t

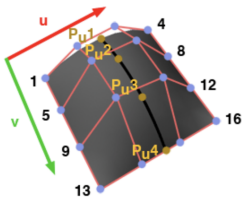
Tendo estudado o conceito de curva de Bezier, passamos agora a explicar a noção de superfície de Bezier, que segue um raciocínio semelhante. Uma superfície cúbica de Bezier pode ser definida por 16 pontos de controlo, que formam uma grelha 4x4. Na curva Bezier, tínhamos um parâmetro t a mover-se ao longo da curva. Neste caso temos dois, u e v, sendo que ambos variam entre 0 e 1 e representam também a tesselagem. Um ponto numa superfície cúbica Bezier pode ser descrito pela dupla soma de curvas cúbica bezier, ou seja:

$$P_{superficie}(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} B_{i,3}(u)B_{j,3}(v)p_{ij}$$

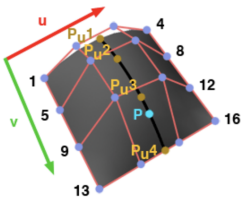
Para traduzir um Bezier patch numa rede poligonal (neste caso, de triângulos), avaliamos a posição de um ponto na superfície para um dado valor de u e v. Cada fila da rede 4x4 deve ser tratada como uma curva bezier individual. Podemos pegar no parâmetro $u$ para avaliar a posição do ponto ao longo de uma das curvas, usando o algoritmo referido anteriormente para as curvas de Bezier (u passa agora a ser tratado como o parâmetro t deste algoritmo), e usando os 4 pontos resultantes disso, que podem ser vistos como pontos de controlo de uma outra curva, podemos calcular a posição final do ponto, agora para o parâmetro v. O ponto que resulta (ponto P na figura que se segue) destes cálculos corresponde à posição da superfície Bezier para o par de valores (u,v).




(a) Superfície Bezier e os 16 pontos de controlo




(b) Pontos de Controlo de uma Curva




(c) Ponto P do Patch Bezier

Figura 2: Patch Bezier e pontos de controlo

Dependendo do nível de tesselagem, são calculados para cada par de valores (u,v) os vértices que lhes correspondem, e são guardados no ficheiro segundo uma ordem que permita definir o conjunto de triângulos que modelam a figura pretendida. Quanto maior for este nível, maior será o número de divisões de u e v, o que leva a que se calcule um maior número de pontos e, por consequência, se obtenha um resultado final mais preciso.

Como forma de exemplo, apresentamos a próxima figura que ilustra a ordem do desenho dos triângulos que compõem a figura a desenhar, para cada par de valores (u,v). Para cada valor inicial de u e v, aos quais chamamos, respetivamente, u1 e v1, existe um sucessor, u2 e v2, que distam do anterior em uma tesselagem. Na figura demonstramos a ordem de desenho utilizada que, seguindo a regra da mão direita, é 0→1→3 para o primeiro triângulo e 0→3→2 para o segundo.

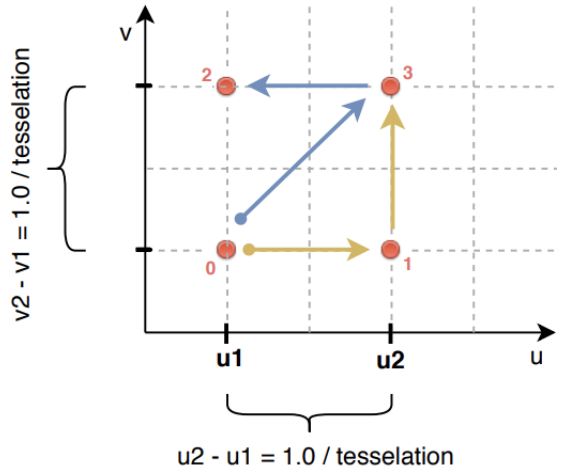



Figura 3: Ilustração da ordem de desenho dos vértices

# 4 Engine

## 4.1 VBOs

Tal como referido anteriormente esta fase do projeto conta com auxílio de VBOs (Vertex Buffer Object) para desenhar os modelos, os quais anteriormente eram desenhados imediatamente. VBO é uma funcionalidade oferecida pelo OpenGL, que oferece métodos capazes de inserir informação sobre os vértices diretamente na placa de vídeo do dispositivo utilizado. O principal objetivo com a implementação de VBOs é conseguir uma perfomance melhor, visto que a informação passa a residir na placa de vídeo em vez da memória do sistema, sendo esta diretamente renderizada pela placa de vídeo, diminuindo a sobrecarga do sistema visto que já não é necessário armazenar a informação em memória. Consequentemente, os fps (frames per second) observados serão mais do que os observados na fase anterior.

Para implementar os VBOs foi necessário alterar a classe *Shape* e o ficheiro **main.cpp** do *Engine*. Na classe *Shape* foi introduzido um vertex buffer, que são arrays onde serão inseridos todos os vértices que pertencem ao modelo que se pretende desenhar. Também foram criadas novas funções, como por exemplo **vbo**, responsável por preencher o array, e **draw**, responsável por desenhar o modelo, usando a função **glDrawArrays**. Já no ficheiro **main.cpp** do *Engine* foi adicionada a função **passTovbo** que chama a função **vbo** para todas as shapes, e desta forma a função **renderGroup**, que é invocada dentro da função **renderScene**, desenha os modelos através da chamada da função **draw**.

## 4.2 Curva Catmull-Rom

### 4.2.1 Translate

A esta ação foram adicionadas as seguintes variáveis:

- **time**, número de segundos necessários para efetuar uma volta completa.
- **points** e **curvePts**, vetores que armazenam os pontos lidos do xml e os pontos da trajetória a desenhar, respetivamente.
- **yzero**, array que serve para alinhar o objeto ao longo da curva.

De modo a efetuar a translação, recorremos a dois arrays auxiliares, pos[3] e deriv[3], que guardam o ponto para a próxima translação da curva e a derivada do ponto anterior, respetivamente. De seguida, aplicamos a função **getGlobalCatmullRomPoint**, que recebe os pontos lidos do xml, os dois arrays e um valor **gt**, global t. Com o valor de gt e o número total de pontos obtemos o coeficiente correspondente à porção da translação na curva. Assim, a função preenche os arrays *pos* e *deriv* com os valores pretendidos, chamando a função **getCatmullRomPoint**.

Tendo o array *pos* preenchido conseguimos então efetuar a translação em si:

```
glTranslatef(pos[0],pos[1],pos[2]);
```

Por sua vez o array *deriv*, em conjunto com o array *yzero*, é utilizado pela função **rotateCurve** para que o objeto siga a orientação da curva.

### 4.2.2 Desenho de curvas Catmull-Rom

Para efetuar o desenho da curva, seja esta a órbita do cometa ou dos planetas do sistema solar, aplicamos inicialmente a função **createCurvePoints** que gera os pontos da curva através dos pontos contidos no ficheiro xml. Aí é variado o valor de $t$ entre 0 e 1 com incrementos de 0.01, chegando assim a 100 pontos da curva. Tais pontos são desenhados posteriormente na função **renderCatmullRomCurve**.

### 4.2.3 Rotate

Como referido anteriormente foi introduzida a variável **time** à rotação. Através desse valor obtemos o ângulo da rotação, **ang**, a cada momento.

```
aux = glutGet(GLUT_ELAPSED_TIME)%(int)(time*1000);
ang = ((aux/(time*1000))*360);
glRotatef(ang,x,y,z);
```

Inicialmente, obtemos o tempo da rotação atual recorrendo à função **glutGet()**. De seguida, com esse valor e o tempo de cada rotação em milisegundos, chegamos ao coeficiente que multiplicado por 360º nos dá o ângulo correspondente.

# 5 Resultados Obtidos

## 5.1 Tea Time

Para além do modelo do Bule previamente fornecido para processamento de ficheiros do tipo .patch, o grupo preocupou-se em extender a gama de exemplos deste tipo, de forma a testar mais extensivamente esta funcionalidade do programa. Assim sendo, apresentamos de seguida o resultado que obtemos correndo o ficheiro de teste "teatime.xml", composto por três modelos diferentes, e as diferentes formas de o visualizar.

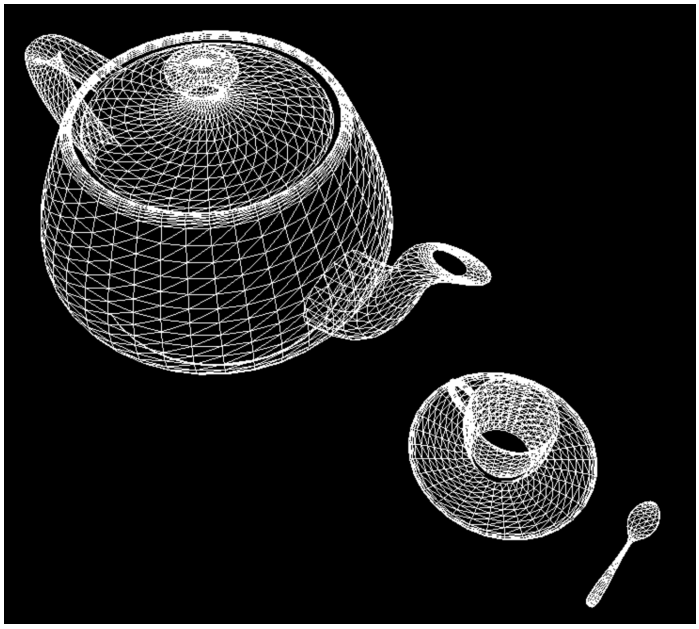


Figura 4: Visualização por linhas (Comando L)

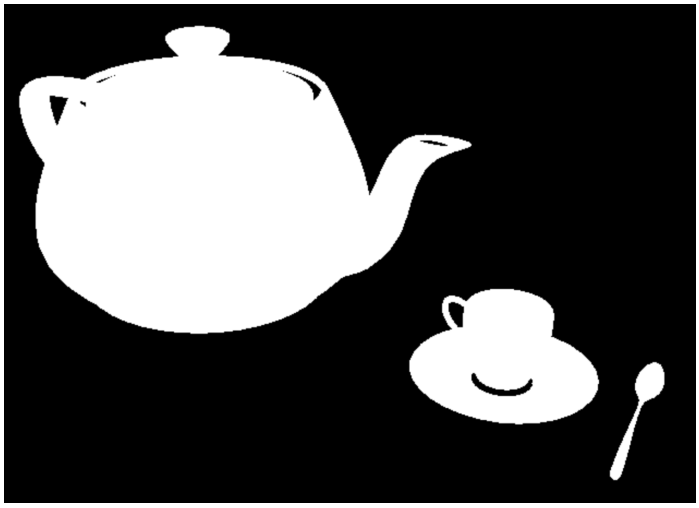


Figura 5: Visualização das figuras preenchidas (Comando F)

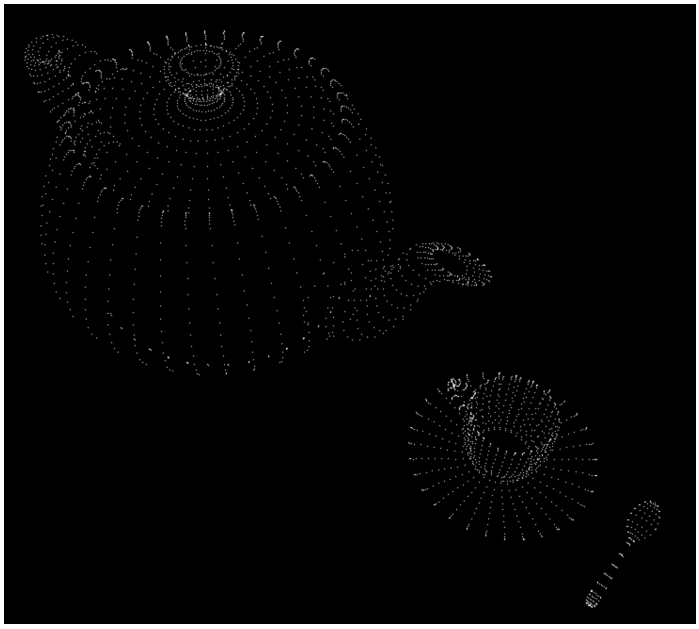


Figura 6: Visualização por pontos (Comando P)

## 5.2 OVNI

Nesta fase foi criado um ficheiro xml capaz de gerar um objeto voador não identificado (OVNI), com o intuito de representar um visitante do nosso Sistema Solar.

Assim sendo, apresentamos de seguida o resultado que obtemos correndo o ficheiro de teste "nave.xml", e as diferentes formas de o visualizar.

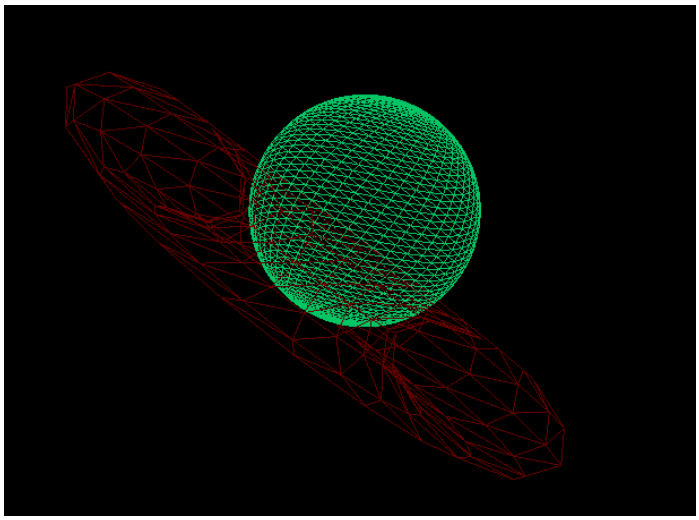


Figura 7: Visualização por linhas (Comando L)

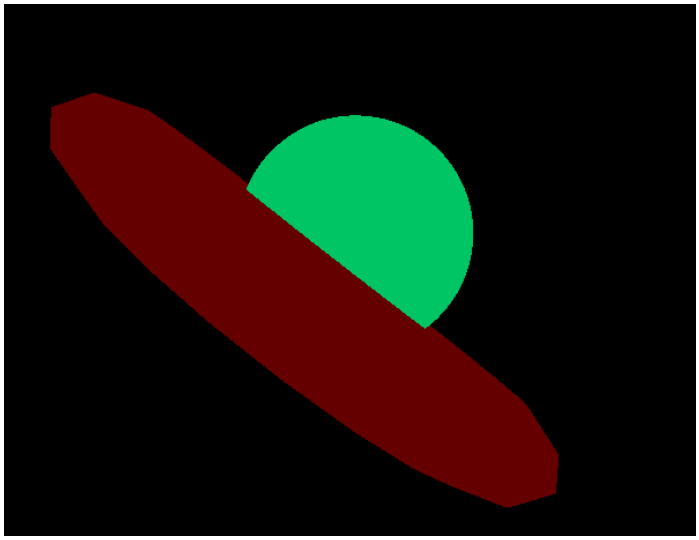


Figura 8: Visualização por preenchimento (Comando F)

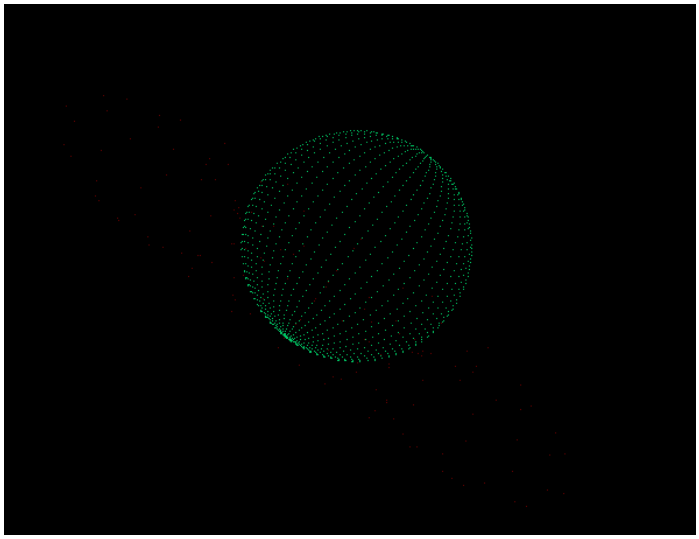


Figura 9: Visualização por pontos (Comando P)

## 5.3 Sistema Solar

Depois de implementadas as funcionalidades foi hora de pormos em prática de forma a ir de encontro com o pedido: uma maqueta dinâmica do sistema solar. Segue nas próximas figuras o resultado obtido através do ficheiro "solarsystem.xml".

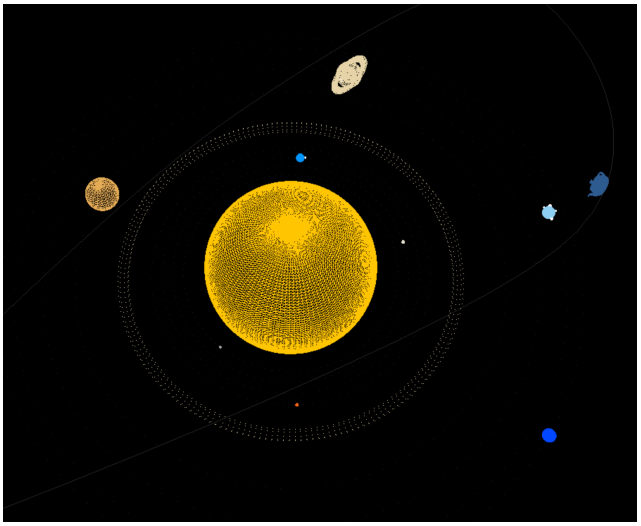


Figura 10: Sistema Solar visto de cima

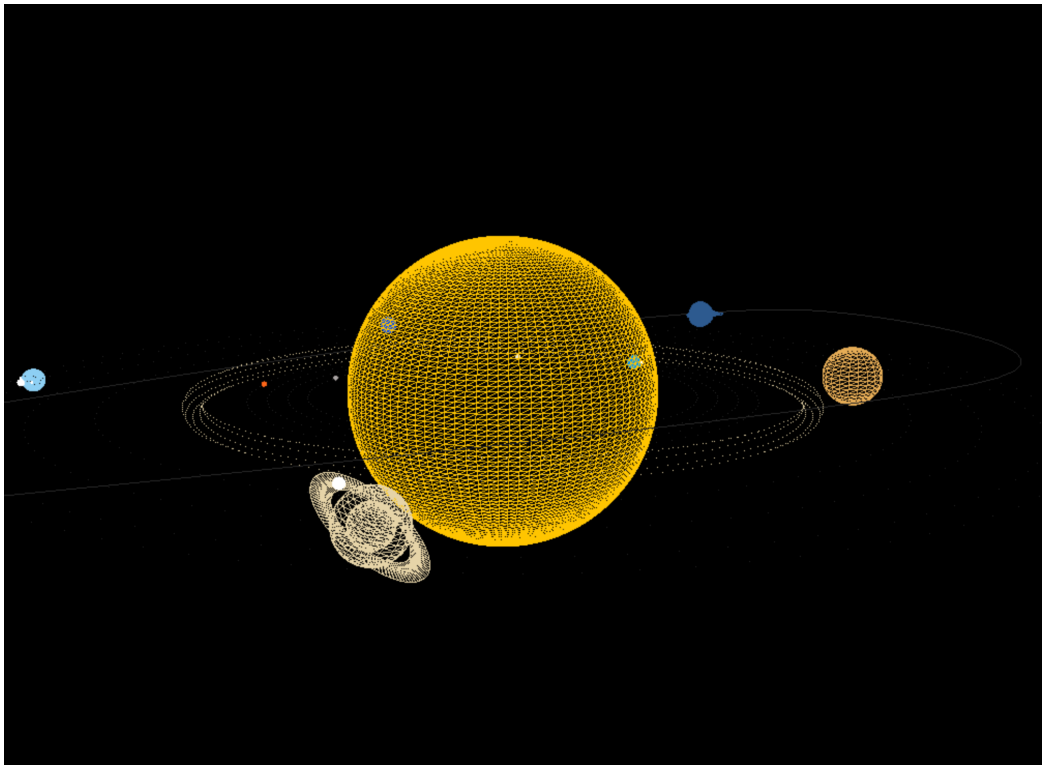


Figura 11: Sistema Solar visto de outra perspetiva

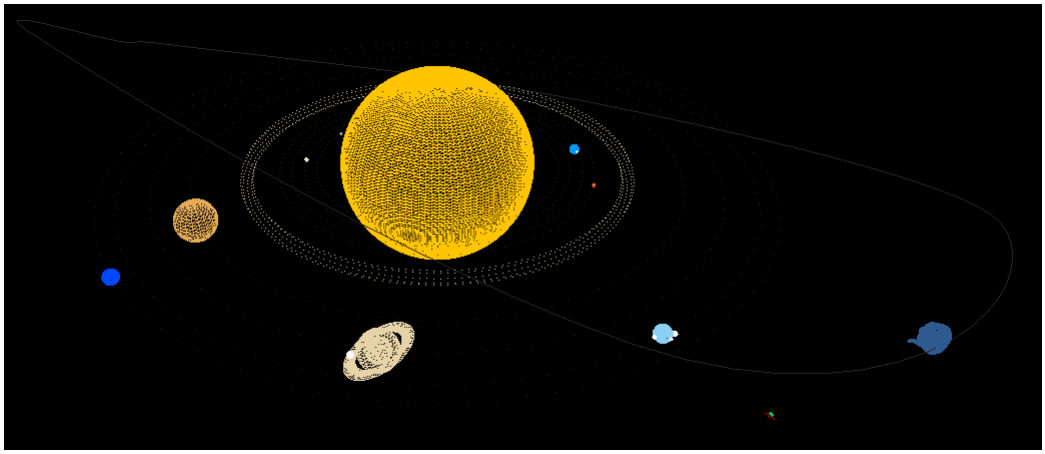


Figura 12: Sistema Solar com OVNI(1)

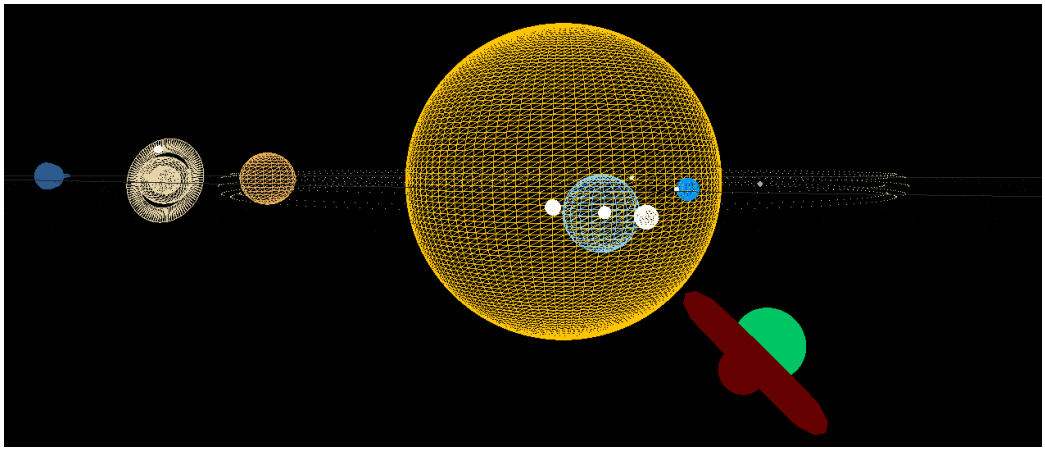


Figura 13: Sistema Solar com OVNI(2)

# 6 Conclusão

A elaboração desta etapa do projeto veio trazer um novo nível de complexidade para o mesmo. Nesta fase, foram implementados algoritmos complexos, sobre os quais tivemos de efetuar uma alargada pesquisa para perceber o seu funcionamento e, assim, conseguir implementá-los corretamente no projeto.

Fazendo uma análise crítica a todo o trabalho até agora desenvolvido, consideramos que esta etapa foi aquela que exigiu maior esforço e tempo por parte do grupo de trabalho, não só pela quantidade de funcionalidades a implementar, mas sim pela complexidade das mesmas.

Não obstante, terminamos esta fase do projeto confiantes do resultado obtido, e consideramos que foram atingidos todos os objetivos propostos até ao momento, tanto pelo professor, como pelo próprio grupo, no sentido de desenvolver e demonstrar ao máximo todos os conhecimentos adquiridos nesta unidade curricular.