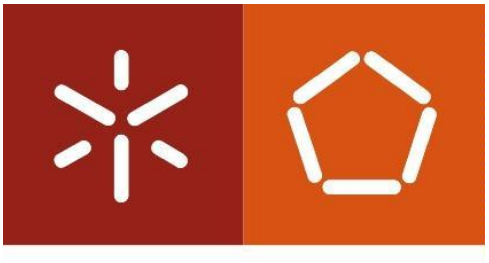


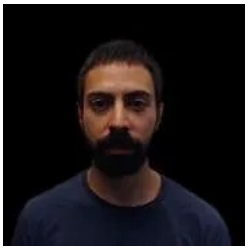
Computação Gráfica

Relatório Trabalho Prático - Fase 3

Grupo 33 LEI - 3º Ano - 2º Semestre

Ano Letivo 2024/2025

Tiago Guedes
A97369



Adélio Fernandes
A78778

Tiago Carneiro
A93207

Braga,
27 de abril de 2025

Conteúdo

Lista de Figuras

1 Introdução

Devido a diversos erros de lógica e à falta de estrutura no código das fases anteriores, foi necessária uma reestruturação completa de todo o código previamente desenvolvido, não tendo sido, portanto, aproveitado nenhum código das fases anteriores.

Numa primeira instância, serão explicadas todas as alterações realizadas em relação ao código entregue nas fases 1 e 2. Em seguida, será apresentada a fase atual, a fase 3.

2 Reestruturação das Fases anteriores

2.1 Fase 1

2.1.1 Modelos

Foram efetuadas alterações significativas na estrutura dos modelos. Atualmente, estes são calculados através da criação de um *vector* de *floats* em que cada conjunto de três valores representa um vértice. Posteriormente, é criado um *vector* de inteiros que define a ordem de desenho desses vértices. A principal vantagem desta nova abordagem é a facilidade de reutilizar vértices e evitar a criação de vértices duplicados.

Ao escrever estes modelos num ficheiro com extensão .3d, começamos por registar os vértices numa única linha, seguindo-se os respetivos índices na linha seguinte.

Nesta fase foram pedidas quatro primitivas para implementação (o plano, o cubo, o cone e a esfera), podendo em seguida visualizar uma representação gráfica de cada uma dessas primitivas:

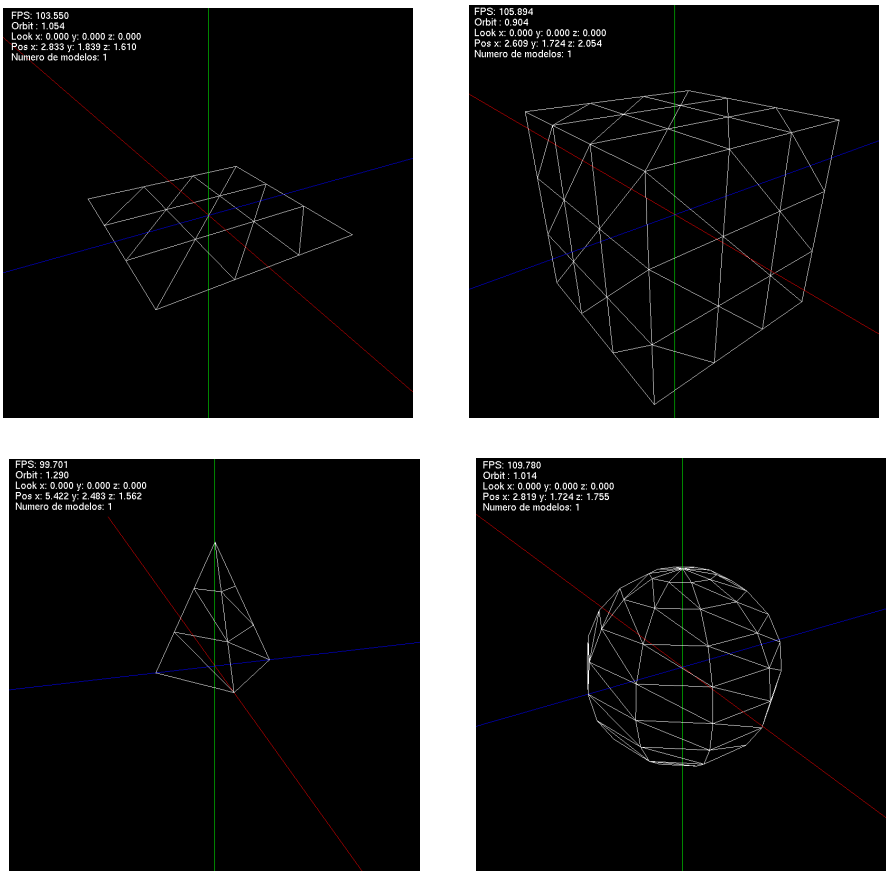


Figura 1: Todas as primitivas da Fase 1

2.1.2 XML Parser

A forma como é feito o *parse* dos ficheiros *XML* também foi alterada. A leitura dos ficheiros *XML* foi reestruturada e passou a utilizar as seguintes estruturas principais:

- **Parser_xml**: contém as estruturas Camera_xml, Window_xml e Groups_xml.
- **Camera_xml**: armazena dados relativos à câmara, como a posição, o *lookAt*, *up vector*, entre outros parâmetros.
- **Window_xml**: guarda a informação relativa à janela, nomeadamente a altura e a largura.
- **Groups_xml**: estrutura que armazena os modelos, transformações e subgrupos, permitindo a recolha recursiva dos grupos - algo fundamental para a organização hierárquica dos mesmos.
- **Model_xml**: guarda o caminho para o ficheiro do respetivo modelo.
- **Transformations_xml**: estrutura que guarda as sub-estruturas das transformações.
- **Scale_xml/Translation_xml/Rotation_xml**: cada uma destas transformações guarda a informação necessária para a sua execução. Todas partilham ainda uma variável comum, um inteiro que define a sua ordem de aplicação. Importa referir que, no caso da rotação ao longo do tempo, esse tempo é também armazenado nesta estrutura, sendo assumido o valor 0 caso não esteja definido. No caso das TimedTranslation_xml, estas são armazenadas na estrutura Translation_xml.
- **TimedTranslation_xml**: aqui são guardados os pontos da translação assim como a sua duração. É também guardado se o objeto será alinhado, e se será seguido (*tracking*, algo referido mais à frente nas extensões XML).

2.1.3 Câmara

A câmara consiste agora numa estrutura (diferente da estrutura onde é guardada a sua informação a partir do ficheiro *XML*) e dois modos de operação, o modo orbital e o modo *First Person*. Aqui entraremos em mais detalhe no modo *First Person*, sendo o modo orbital explicado em mais detalhe na secção dos Extras.

A câmara em primeira pessoa contém seis atributos que a definem:

- **FPS Position**: posição concreta em que a câmara se encontra.
- **FPS Right**: vetor localizado ao longo do eixo X local à câmara, que representa o movimento da câmara nesse eixo.
- **FPS Front**: vetor localizado ao longo do eixo Z local à câmara, que representa o movimento da câmara nesse eixo.
- **FPS Up**: vetor localizado ao longo do eixo Y local à câmara, que representa a orientação da câmara, definindo o eixo para o qual está orientada.
- **FPS Yaw**: rotação em torno do eixo Y local à câmara, que faz com que esta rode da esquerda para a direita ou vice-versa.
- **FPS Pitch**: rotação em torno do eixo X local à câmara, que faz com que esta rode de cima para baixo ou vice-versa.

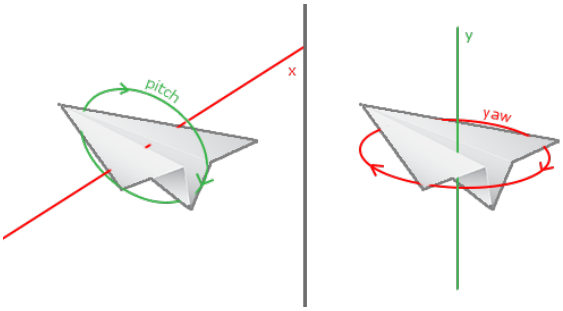


Figura 2: Lógica de Yaw e Pitch na câmara em primeira pessoa

Nas implementações de câmaras *First Person*, é normal ser implementado também um vetor *Roll*, capaz de fazer uma rotação em torno do eixo Z local. Embora essa prática seja comum, decidimos não implementar esse mesmo vetor, por não considerarmos pertinente no contexto do projeto.

A partir da implementação anteriormente mencionada, implementar o movimento pressionando teclas foi simples :

- Pressionando a tecla 'W' ou 'S', aumentamos ou diminuímos respetivamente o vetor *FPS Front* à posição da câmara.
- Pressionando a tecla 'D' ou 'A', aumentamos ou diminuímos respetivamente o vetor *FPS Right* à posição da câmara.
- Pressionando a tecla 'Space' ou 'C', aumentamos ou diminuímos diretamente o valor Y da posição da câmara, sendo possível subir ou descer a câmara desta forma.

Já para movimentar o *lookAt* da câmara, é possível arrastar o rato, sendo calculado quanto moveu ao longo do eixo X e do eixo Y respetivos à janela. Após esse cálculo, é aumentado/diminuido os valores do *FPS Yaw* e do *FPS Pitch*, sendo que o valor do cálculo relativo ao eixo X afeta o *FPS Yaw* e o eixo Y afeta o *FPS Pitch*.

2.1.4 *Demo files*

Para esta primeira fase, todos os ficheiros de teste foram corridos obtendo o resultado esperado. Para além disso foram criados dois novos ficheiros de demo, ambos para as novas primitivas que serão mencionadas na secção dos extras.

Podemos observar nas seguintes imagens as demos extra para esta fase :

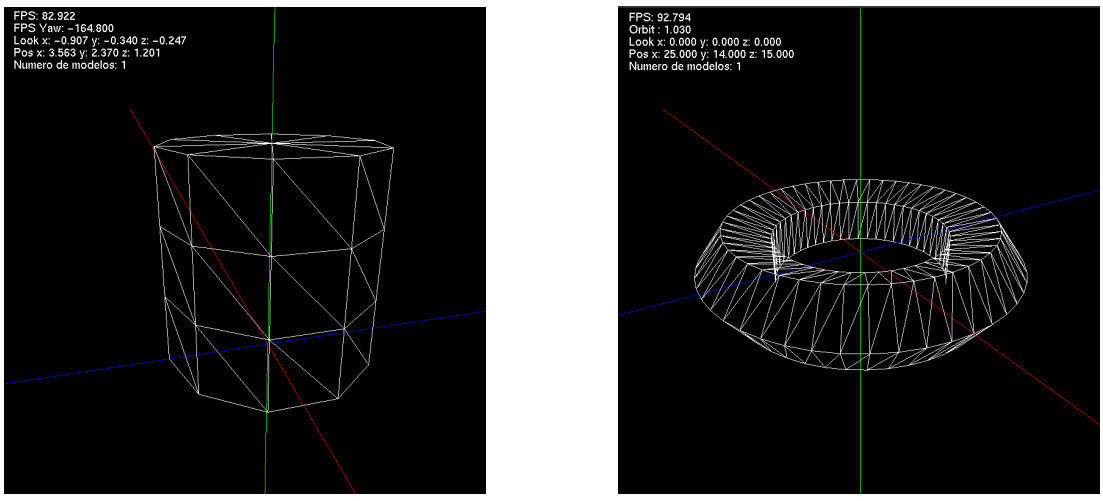


Figura 3: Ficheiros de Demo do Cilindro e Torus

2.2 Fase 2

Na segunda fase, dois conceitos foram especialmente importantes para percebermos como foram implementadas as translações e rotações :

- Primeiramente, a recolha de grupos passou a ser recursiva; portanto, os grupos contêm uma lista dos seus subgrupos.
- O segundo ponto importante é o facto de que agora o desenho dos modelos é ele também recursivo, permitindo assim que as transformações aplicadas a um grupo inicial sejam aplicadas recursivamente aos seus subgrupos.

2.2.1 Desenho Recursivo

Entrando em mais detalhe sobre o desenho recursivo dos modelos, aproveitamos a maneira como foram guardados anteriormente os grupos e subgrupos de cada grupo e, no momento do desenho de cada grupo, após o seu desenho com sucesso, são também desenhados os seus subgrupos.

2.2.2 Transformações Estáticas

Estas transformações são feitas antes do desenho dos objetos e, tal como referido anteriormente, devido à natureza da recursividade dos grupos, ao desenhar os mesmos também estas transformações são realizadas recursivamente.

2.2.3 *Demo files*

Tal como a fase anterior, ao correr os *test files* disponibilizados pelos professores, obtivemos os resultados pretendidos. Também para esta fase foram criados dois novos ficheiros de teste, um representando um Pinguim e um representando uma réplica do modelo de uma vaca do jogo *Minecraft*.

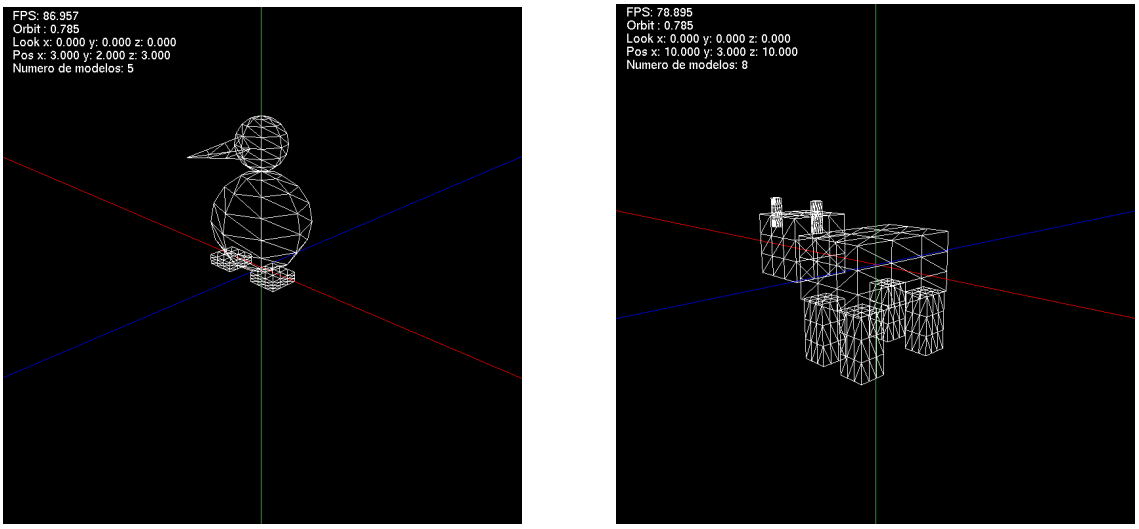


Figura 4: Pinguim e Vaca do Minecraft

3 Fase 3

Agora falaremos finalmente da fase atual, a fase 3. Iremos comentar como foram implementados os VBOs (*Vertex Buffer Objects*), as Superfícies de Bezier e as Translações/Rotações ao longo do tempo. Iremos também falar de ficheiros de demo adicionais criados.

3.1 VBO

Para a implementação dos VBOs, foi criada uma estrutura nova para guardar as seguintes variáveis :

- **Vertices**: identificador dado após a informação dos vértices ser enviada para a memória da gráfica.
- **Total vertices**: número total de vértices.
- **Indexes**: identificador dado após a informação dos índices ser enviada para a memória da gráfica.
- **Total indexes**: número total de índices.
- **Cube**: lista de valores xmin, xmax, ymin, ymax, zmin and zmax dos vértices do modelo. Utilizada na implementação do *View Frustum Culling*.

Foi também criado um dicionário que utiliza como chave o caminho para o ficheiro onde está contido um modelo, e como valor a instância da estrutura referida anteriormente relativa a esse mesmo modelo. Quando a *Engine* começa, este dicionário é populado com todos os modelos presentes no ficheiro *XML* aberto, de forma a que, no momento do desenho destes modelos, toda a informação necessária para tal já se encontre presente na gráfica.

Assim, no momento de desenho dos grupos, facilmente é pesquisado no dicionário pela informação relativa ao modelo a desenhar, tornando o desenho do mesmo rápido e eficiente.

3.2 Superfícies de Bezier

Para a criação de superfícies de Bezier, como é claro, foi necessário aplicar os conhecimentos das curvas de Bezier. Para uma curva de Bezier, dados 4 pontos P_0, P_1, P_2 e P_3, é possível definir a fórmula para representar a curva ao longo do tempo através da seguinte fórmula:

$$P(t) = (1-t)^3 P_0 + 3t(1-t^2)P_1 + 3t^2(1-t)P_2 + 3t^3 P3$$

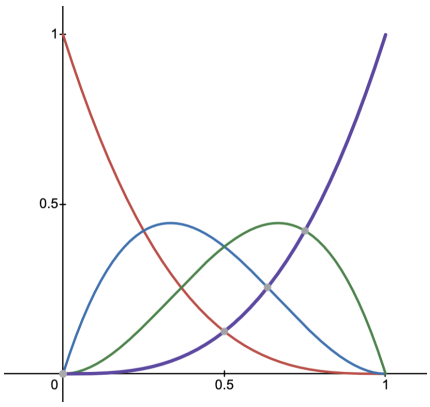


Figura 5: Polinómios de Bernestein das curvas de Bezier

Acima, observamos a representação gráfica dos polinómios de Bernstein, podendo associar a cada uma das curvas um dos polinómios coloridos na fórmula anterior.

Da mesma forma que se cria uma curva de Bezier, é também possível criar uma superfície, necessitando de 16 pontos, ao contrário das curvas que necessitam apenas de 4. Em seguida podemos observar uma superfície a ser criada, primeiro criando curvas u, e utilizando os pontos dessas curvas para criar uma nova curva, v, representando os pontos finais da superfície como sendo $p(u, v)$.

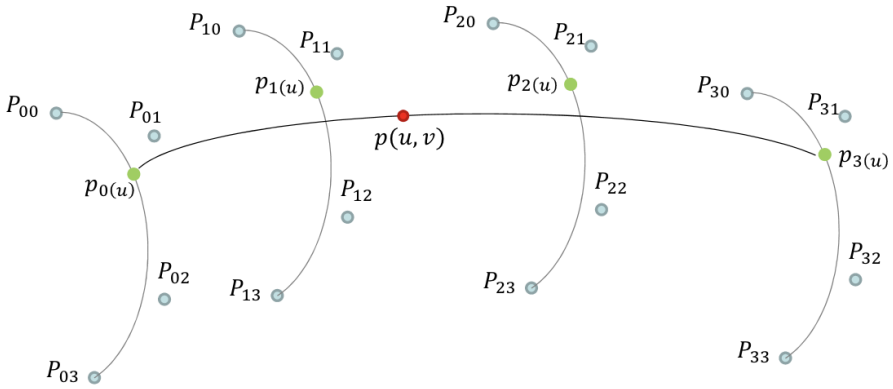


Figura 6: Superfície de Bezier

Este foi o pensamento utilizado para construir as superfícies de Bezier. No ficheiro *teapot.patch* fornecido, a informação pertinente a estes *patches* (ou seja, os 16 pontos de controlo necessários) é providenciada. Lendo este ficheiro, guardando as informações para cada *patch* e aplicando o algoritmo anteriormente demonstrado, é possível construir todos os conjuntos de *patches* e, por fim, obter a imagem do objecto final.

É possível criar estes mesmos *patches* (e subsequentemente o objecto final) com um nível maior ou menor de detalhe, delimitando a quantidade de pontos que serão criados em cada curva.

A seguir podemos observar como é utilizado este algoritmo para criar dois *Teapots*, um com um nível de detalhe de 1 e o outro com um nível de detalhe de 20.

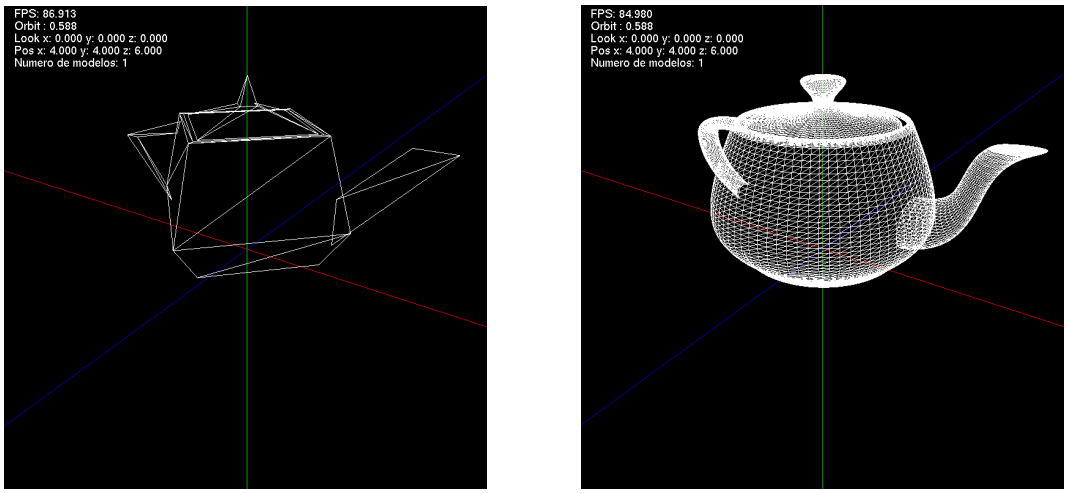


Figura 7: *Teapot* com diferentes níveis de detalhe

3.3 Translações Temporais

Nas translações temporais, é nos dado uma lista de pontos, juntamente com uma duração de tempo que o objecto deve demorar para se mover ao longo destes pontos. Para podermos explicar como foram calculadas as curvas que os objectos devem seguir, é necessário entender as curvas de Catmull-Rom.

As curvas de Catmull-Rom são curvas criadas a partir de 4 pontos de controlo, P_0, P_1, P_2 e P_3, que obrigatoriamente passam por P_1 e P_2.

Em seguida podemos observar como estas curvas podem ser usadas :

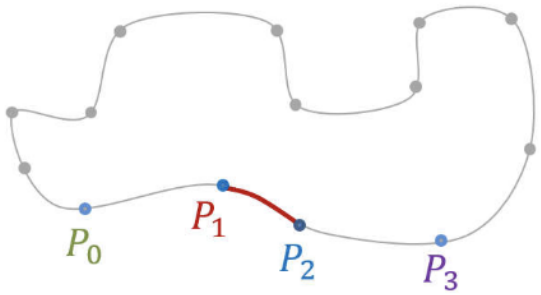


Figura 8: Curva de Catmull-Rom

A equação que representa a curva previamente demonstrada pode ser representada da seguinte forma:

$$P(t) = C_0(t)P_0 + C_1(t)P_1 + C_2(t)P_2 + C_3(t)P_3$$

- $C_0(t) = -0.5t^3 + t^2 - 0.5t$
- $C_1(t) = 1.5t^3 - 2.5t^2 + 1$
- $C_2(t) = -1.5t^3 + 2t^2 + 0.5t$
- $C_3(t) = 0.5t^3 - 0.5t^2$

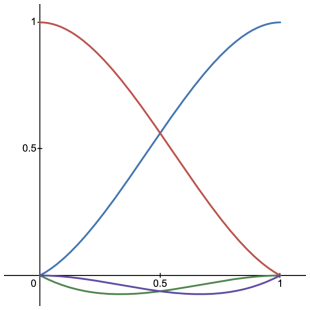


Figura 9: Polinómios De Bernestein das curvas de Catmull-Rom

Conhecendo estas curvas, podemos utilizar os pontos adquiridos num determinado ficheiro *XML*, criando curvas de Catmull-Rom ao longo destes pontos, e depois fazemos translações ao longo dos novos pontos dessas mesmas curvas.

Para podermos alinhar o objecto ao longo da curva, foi necessário fazer o seguinte :

- Começar por pegar no vetor tangente à curva no ponto, e normalizar o vetor.
- Fazer o produto vetorial do vetor tangente após a normalização com o vetor *up*, e normalizar o vetor resultante (chamado z neste caso).
- Após isso, fazer um novo vetor *up*, fazendo o produto vetorial entre o vetor tangente e o vetor z, e normalizar.
- Criar uma matriz de rotação com o vetor tangente, vetor z e vetor *up*, e aplicar essa matriz.

Com esta implementação, fomos capazes de garantir que os objectos se mantêm alinhados ao longo das curvas, à medida que se movimentam na curva.

3.4 Rotações Temporais

As rotações temporais foram trivais, pois apenas o tempo para a rotação de 360° graus foi necessário, calculando-se o ângulo de rotação em cada segundo a partir dele. Após isso, é apenas necessário executar a rotação.

3.5 *Demo files*

Para esta fase fomos capazes de recriar todos os *test files* providenciados pelos professores, obtendo o resultado pretendido.

Em conjunção com os ficheiros fornecidos, dois novos ficheiros de teste foram criados. O primeiro conjugando a superfície de Bezier criada para o *Teapot* com uma translação temporal da réplica da vaca do *Minecraft* mencionada anteriormente. O segundo ficheiro extra conjuga uma rotação temporal novamente com uma superfície de Bezier.

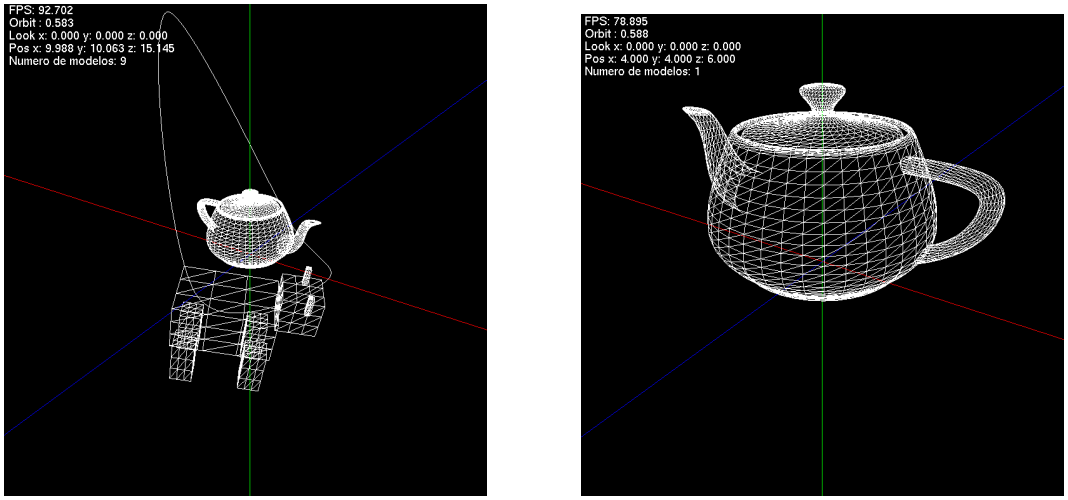


Figura 10: Ficheiros de Teste extra da Fase 3

Por fim, tal como pedido, foi criado um ficheiro para a representação de um sistema solar, com planetas e um cometa, utilizando um ficheiro .patch, no nosso caso sendo este um *teapot*.

De seguida podemos observar este mesmo ficheiro, assim como o cometa :

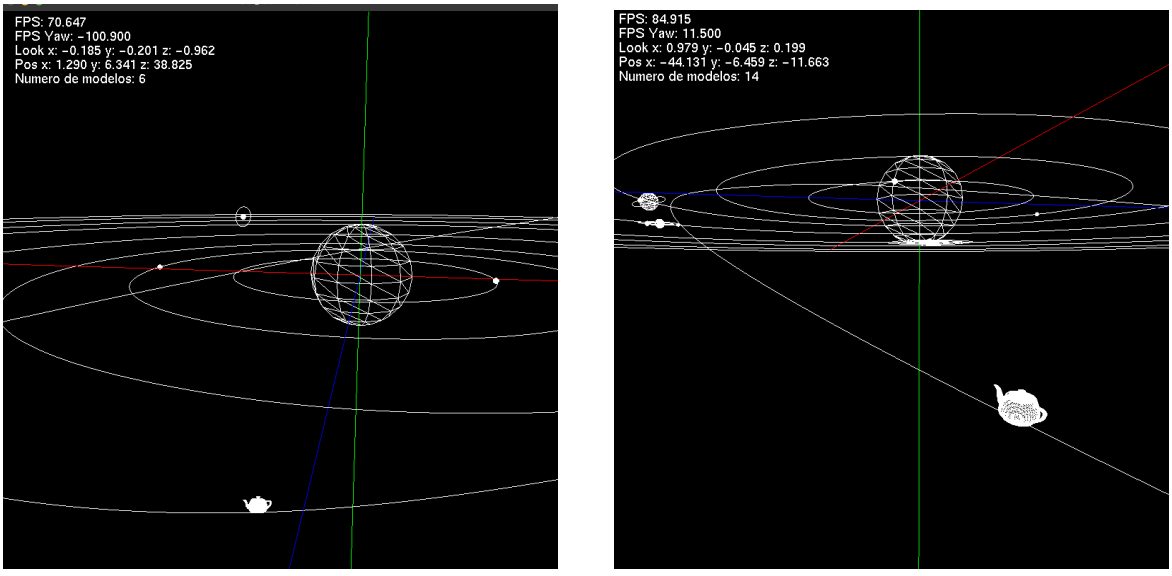


Figura 11: Sistema Solar e Cometa Teapot

4 Extras

4.1 Câmara orbital

Para a nossa câmara orbital, decidimos utilizar uma estrutura composta por uma posição *lookAt*, para onde a câmara aponta, bem como pelas variáveis *radius*, *polar* e *azimuth*. Em seguida explicamos o que cada uma significa, assim como uma representação visual destas variáveis em relação à câmara e em relação aos planos XYZ.

- **Radius**: distância ao ponto *lookAt*
- **Polar**: ângulo de elevação em relação ao plano XZ
- **Azimuth**: ângulo de rotação ao redor do eixo Y

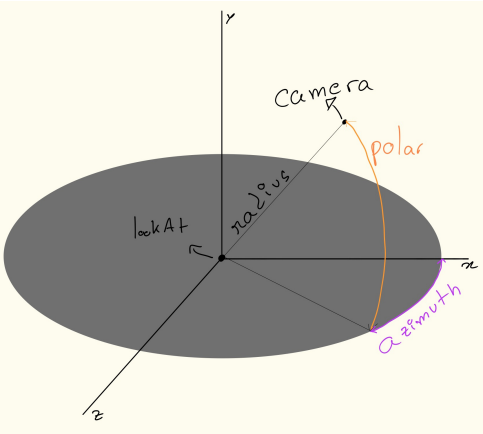


Figura 12: Representação da câmara orbital

Com todas estas variáveis, a posição da câmara para um determinado *lookAt* foi calculada da seguinte forma:

- $x = lookAt.x + radius * cos(polar) * sin(azimuth)$
- $y = lookAt.y + radius * sin(polar)$
- $z = lookAt.z + radius * cos(polar) * cos(azimuth)$

Como podemos observar, ao alterar o *lookAt* conseguimos facilmente fazer com que a câmara passe a orbitar um novo objecto, algo que será mais relevante na explicação de uma das novas formatações de *XML* numa secção mais à frente.

Para podermos movimentar a câmara orbital, é possível utilizar as teclas 'A', 'W', 'S' e 'D' assim como também é possível arrastar o rato para alterar os valores dos ângulo *polar* e *azimuth*. Para alterar o valor do *radius* é possível utilizar as setas para cima e baixo para alterar esse valor.

Em seguida explicamos como cada tecla afeta esses mesmos valores:

- As teclas 'A' e 'D' decrementam e incrementam o *azimuth* respetivamente.
- As teclas 'S' e 'W' decrementam e incrementam respetivamente o *polar*. De notar que existe um *clamp* para que a câmara não faça uma rotação indesejada.
- As setas para cima e para baixo decrementam e incrementam respetivamente o valor do *radius*.

Já para alterar estes valores arrastando o rato, tal como para a câmara em modo *First Person*, é calculado quanto o rato moveu no seu eixo local X e Y. Após esse cálculo ser feito, o movimento no eixo do X afeta o valor do ângulo *azimuth*, e o Y afeta o valor do ângulo *polar*.

4.2 Novas primitivas

Duas novas primitivas foram adicionadas: o cilindro e o *torus*. Falaremos agora de como ambas estas primitivas são construídas.

4.2.1 Cilindro

Para o cilindro, aplicámos conceitos semelhantes aos utilizados no cone, recebendo, assim, os mesmos argumentos que esta outra primitiva.

A diferença notável comparativamente com o cone é o facto de que o cilindro não tem um ponto focal no topo e, portanto, as *stacks* desta primitiva não diminuem em raio, o que facilita os cálculos da mesma.

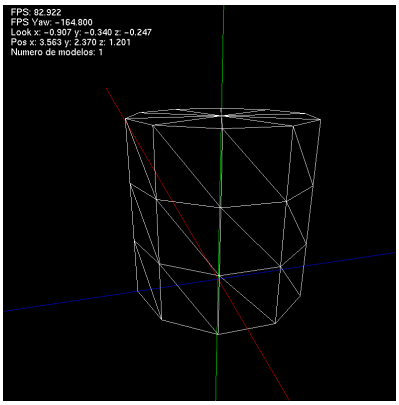


Figura 13: Ficheiro Demo do Cilindro

4.2.2 Torus

Já o *torus* recebe quatro argumentos (excluindo o ficheiro de saída), sendo estes: o *inner radius*, o *outer radius*, o número de *slices* e o número de *stacks*.

De seguida, apresentamos novamente a imagem do *torus*, juntamente com a explicação de como foi construído, para que o algoritmo possa ser mais facilmente compreendido.

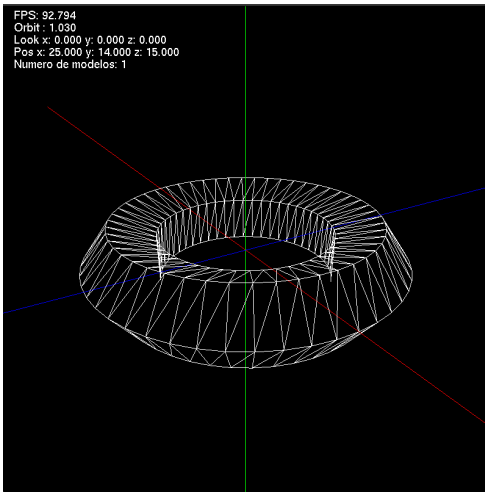


Figura 14: Ficheiro Demo do Torus

Começamos por percorrer cada *slice*, girando em torno do centro do *torus* e, para cada uma delas, imaginamos o círculo que forma o *torus* nessa posição.

A seguir, para cada *slice*, determina-se onde fica o centro desse pequeno círculo - ou seja, deslocamo-nos ao redor do anel maior e, em seguida, traçamos o contorno do *torus* à volta desse novo ponto, criando vários vértices igualmente espaçados ao longo dele, em número igual ao definido pelas *stacks*.

Repetindo esse processo para cada *slice* e *stack*, mapeamos toda a superfície, garantindo que cada ponto do tubo esteja corretamente posicionado em relação ao anel maior.

4.3 Extensão da formatação XML

Para possibilitarmos algumas funcionalidades extra que queríamos implementar, foi necessário expandir aquilo que era possível escrever e ler dos ficheiros *XML*. Para tal, duas novas secções foram adicionadas ao *XML*:

4.3.1 xmls

Ao criarmos o ficheiro para a representação do sistema solar, imaginamos uma funcionalidade excelente para o mesmo : importação de sub ficheiros *XML* para um novo ficheiro *XML*.

Isto seria extremamente útil para este mesmo ficheiro pois, assim, seria possível criar e alterar ficheiros individuais de planetas, e apenas chamá-los recursivamente no ficheiro maior (o ficheiro sistema solar).

Para podermos implementar tal funcionalidade, ficou claro que seria necessário criar novas *tags* para o ficheiro *XML*. Decidimos seguir o funcionamento das *tags* para os modelos, criadas pelos professores, tendo assim criado duas novas *tags* :

- xml - *tag* contendo o atributo *file*, que consiste no caminho para o ficheiro xml a importar
- xmls - *tag* utilizada para encapsular as *tags* xml, podendo assim suportar vários ficheiros *XML* e não apenas 1.

Após a criação destas novas *tags*, a implementação da leitura dos ficheiros xml recursivos foi trivial, pois consiste apenas na abertura e leitura da informação dos mesmos, descartando as informações relativas à câmara e à janela, e tratando o resto da informação deste ficheiro como pertencendo toda a um grupo.

De seguida demonstramos uma representação desta implementação:

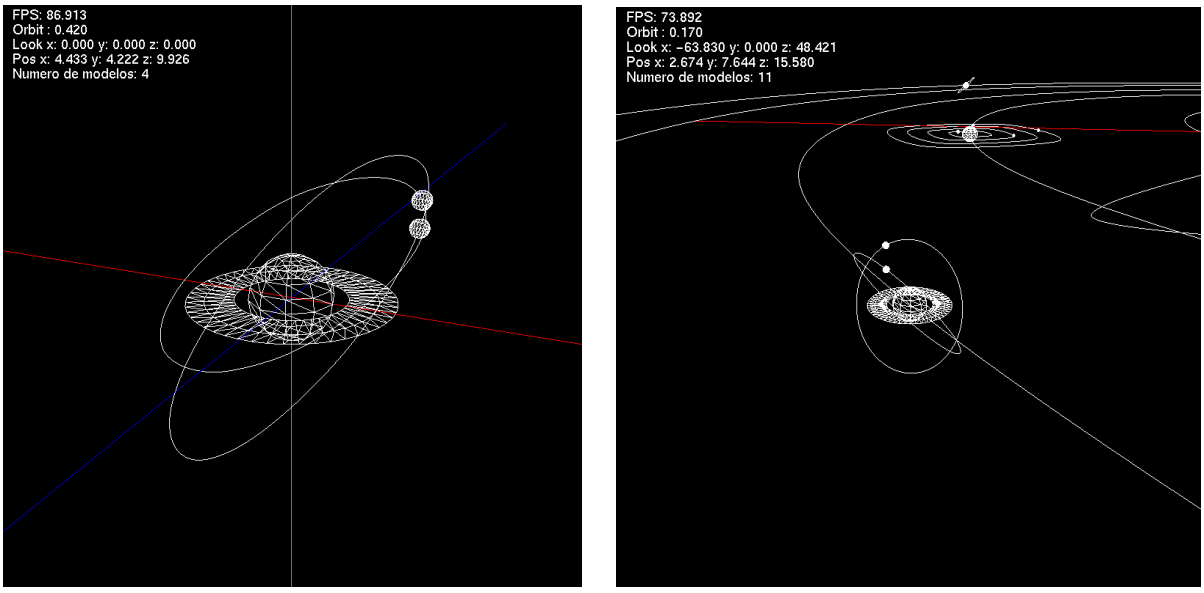


Figura 15: Representação de Saturno

Nas imagens anteriores começamos por observar o ficheiro saturn.xml, ficheiro este que contém a informação pertinente relativa ao planeta Saturno do nosso sistema solar. Em seguida observamos o ficheiro do sistema solar onde encontramos este mesmo ficheiro chamado de forma recursiva, observando que é representado exatamente igual à sua versão singular.

4.3.2 Tracking

Para além de terem sido adicionadas novas *tags*, foi também adicionado um novo atributo a uma *tag* já existente, o *tracking*.

Este atributo foi criado para permitir que a câmara orbital seja capaz de seguir um objecto que se encontre numa translação temporal, sendo assim, naturalmente, um atributo da *tag translate*.

Quando uma translação está a ser seguida, esta recebe um identificador único no momento da leitura do ficheiro xml, identificador este que é utilizado na fase de desenho para guardar a sua posição à medida que a translação vai sendo realizada, num dicionário que utiliza como chave o identificador e como valor um trio com o seu x, y e z atuais.

É, assim, guardado o identificador referente ao objecto atualmente observado pela câmara orbital, e sempre que é desenhado um *frame*, é atualizado o *lookAt* da câmara para a nova posição do objecto a ser observado (posição esta que está guardada no dicionário mencionado anteriormente). Devido à maneira como foi implementada a câmara orbital, mencionado anteriormente, a posição da câmara é automaticamente mudada ao mudar o *lookAt*.

Para alterar o objecto observado, é possível utilizar as setas da esquerda e direita do teclado para alternar entre os objectos que estão a ser seguidos.

4.4 View Frustum Culling

A nossa abordagem para a implementação do *View Frustum Culling* foi de começar por criar um *bounding body* para cada uma das nossas primitivas. Rapidamente ficou claro que, pelo menos para uma versão inicial desta optimização, seria ideal usarmos a metodologia AABB.

Assim, quando o objecto é inicialmente lido do ficheiro de *input*, são guardadas num *array cube* o valor mais baixo para o x, y, z e os valores máximos para estes eixos também, permitindo então construir a *bounding box*.

No momento de desenho recursivo mencionado anteriormente utilizamos esta *bounding box*, juntamente com a matriz MVP (*Model View Projection Matrix*), para percebermos se este objecto deve ser desenhado ou não comparando para cada um dos 6 planos - *Near, Far, Left, Right, Up, Down*, se o seu correspondente ponto mais longe está contido no *View Frustum*.

De seguida podemos observar a representação visual destas mesmas *bounding boxes*.

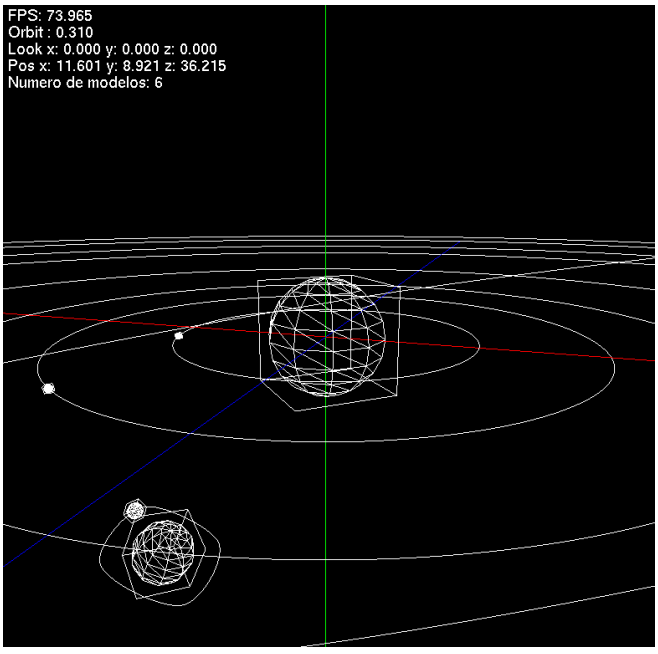


Figura 16: Representação visual das *Bounding Boxes*

4.5 *Menu*

Um pequeno extra adicionado foi um menu, que aparece ao utilizar a tecla do lado direito do rato. Este foi criado utilizando as funções disponíveis na biblioteca do *Glut*.

Em seguida podemos observar este mesmo *menu*:

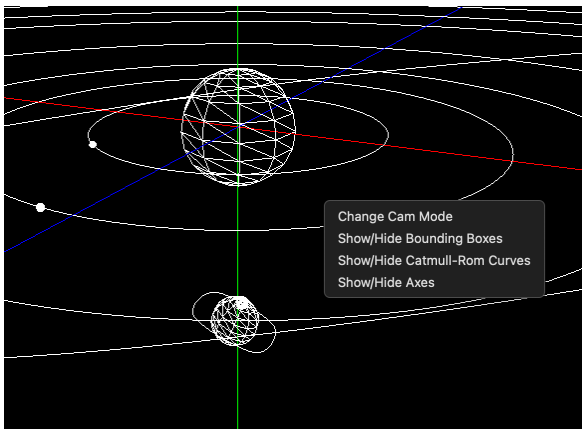


Figura 17: Menu

Embora seja um extra pequeno, achamos pertinente falar do mesmo, pois facilita o processo de *debugging* de certas implementações como, por exemplo, na implementação do *View Frustum Culling* anteriormente mencionado, em que se tornava fácil e intuitiva a representação das *bounding boxes*.

De notar, no entanto, que todas as funcionalidades disponíveis no *menu* se encontram também disponíveis através de teclas do teclado, sendo :

- 'M' : para alterar o modo da câmara.
- 'B' : para mostrar/esconder as *bounding boxes*.
- 'P' : para mostrar/esconder a representação visual das curvas de Catmull-Rom para as translações temporais.
- 'O' : para mostrar/esconder os eixos X, Y e Z globais.

A vantagem de ter um *menu* como o que implementamos é maioritariamente o facto de ser mais *user friendly* e também de mais fácil uso.

4.6 Texto

Outro pequeno extra implementado foi a representação de texto no ecrã, mostrando a seguinte informação :

- **FPS**: *Frames* por segundo.
- **Radius/Yaw**: Em modo orbita, apresenta o valor do *radius*, em modo *FPS*, apresenta o valor do *Yaw*.
- **Look**: Valores x, y e z para a posição do *lookAt*.
- **Pos**: Posição atual da câmara.
- **Número de modelos**: quantidade de modelos atualmente desenhados.

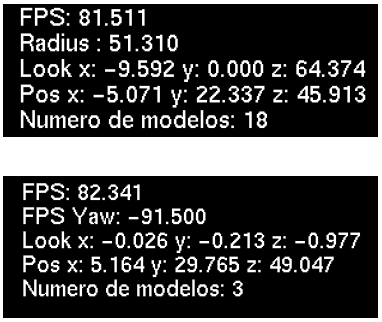


Figura 18: Texto representado

5 Trabalho Futuro

Para a fase final, pretendemos não só implementar os requisitos restantes, mas também focar-nos na melhoria do *tracking* de objectos, com o objetivo de o tornar o mais suave possível, encontrando uma forma mais eficaz de o representar.

Para novos extras a adicionar, pretendemos implementar algum tipo de *Spacial Partionning*, e tentar implementar uma versão simples de *Occlusion Culling*, começando possivelmente por alterar a ordem de desenho, fazendo com que os objectos perto da câmara sejam desenhados primeiro.