



Universidade do Minho
Escola de Engenharia

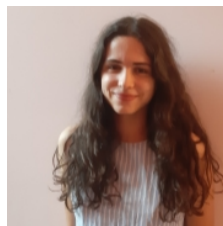
Computação Gráfica

Trabalho Prático - Fase 3

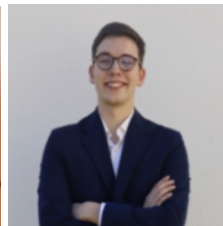
Grupo 02



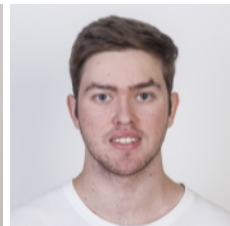
André Campos
a104618



Beatriz Peixoto
a104170



Diogo Neto
a98197



Luís Freitas
a104000

Índice

1. Introdução.....	2
2. Patches.....	2
2.1. Leitura de ficheiros .patch.....	2
3. Curvas Catmull-Rom.....	3
3.1. Representação da Curva.....	3
3.2. Movimento do objeto em função do tempo.....	4
3.3. Aplicação de Transformações.....	4
3.4. Alinhamento do objeto com a curva (opcional).....	4
3.5. Matriz de Rotação.....	5
3.6. Funções Adicionais.....	5
4. Cenário dinâmico.....	5
4.1. Translate.....	5
4.2. Rotate.....	6
5. VBOs.....	6
6. Extras.....	6
6.1. Torus.....	7
6.2. Câmera Livre.....	9
6.2.1. Vetor da câmara.....	9
6.2.2. Inicialização do Vetor da Câmera.....	11
6.2.3. Movimentação da Câmera.....	11
6.2.4. Definições da Câmera Livre.....	12
7. Demos.....	12
7.1. Demos do Enunciado.....	12
7.1.1. Ficheiro test_3_1.xml.....	12
7.1.2. Ficheiro test_3_2.xml.....	13
7.2. As nossas Demos.....	14
7.2.1. Ficheiro torus.xml.....	14
7.2.2. Ficheiro demo_1.xml.....	15
7.2.3. Ficheiro sistema_solar.xml.....	16
8. Conclusão.....	17

1. Introdução

Esta fase consistiu em criar novos modelos baseados em patches das curvas de Bezier, ter translações dinâmicas definidas por curvas de Catmull-Rom, rotações dinâmicas e desenhar os modelos usando VBOs. Nesta fase usaremos curvas para desenhar e deslocar modelos e teremos cenários dinâmicos, ou seja, cenários que mudam com base no tempo.

Neste sentido, nos tópicos abaixo explicamos o nosso raciocínio para a implementação desta fase.

2. Patches

De modo a conseguirmos cumprir o objetivo do enunciado, nesta fase, adicionamos ao gerados a capacidade de criar superfícies usando patches de Bézier, onde o objetivo é gerar malhas triangulares corretas para a renderização do conteúdo na engine.

2.1. Leitura de ficheiros .patch

O gerador lê ficheiros .patch com o seguinte formato:

- número total de patches
- 16 índices de pontos de controlo por cada patch
- número total de pontos de controlo
- pontos de controlo [coordenadas]

2.2. Matriz base de bézier

Para gerar superfícies a partir de patches de bézier, precisamos de calcular as coordenadas dos pontos na superfície a partir dos pontos de controlo. usamos então uma transformação matricial que acelera os cálculos, esta é feita usando a matriz base de Bézier :

$\{-1, 3, -3, 1\},$
 $\{3, -6, 3, 0\},$
 $\{-3, 3, 0, 0\},$
 $\{1, 0, 0, 0\}$

Para cada patch criamos 3 matrizes para as coordenadas de cada eixo: M_x , M_y e M_z , todas de dimensão 4×4 , onde cada índice é o valor da coordenada de um ponto de controlo.

De seguida, aplicamos a transformação: $\text{Matriz Base} * M_x * \text{Matriz Base}^T$ (análogo par M_y e M_z), de modo a obtermos os coeficientes $[C_x, C_y, C_z]$ corretos para calcular a posição da superfície em qualquer ponto (u, v) .

Para cada par de vetores u e v , construímos:

$$U = (u^3, u^2, u, 1)$$
$$V = (v^3, v^2, v, 1)$$

A posição do ponto na superfície é então calculada com: $x(u,v) = U * C_x * V^T$ (análogo para C_y e C_z) e conseguimos obter assim as coordenadas desejadas que, por fim, são juntadas aos pontos vizinhos formando triângulos e vamos construindo assim a malha desejada.

2.3. Tessellation dos patches

Para gerar a malha de um patch de Bézier, começamos por dividir o espaço dos parâmetros (u,v) em pequenas divisões iguais, onde o número de divisões depende do valor de tessellation level escolhido.

De seguida, para cada combinação de valores u e v , calculámos o ponto correspondente na superfície, este ponto foi obtido usando produtos matriciais, como explicado anteriormente, que nos dão as coordenadas do ponto, a seguir, com todos os pontos já calculados, ligámos quatro pontos vizinhos para formar um quadrado, onde cada um foi dividido em dois triângulos, que juntos cobrem toda a superfície do patch.

No final, cada patch gera: $2 * \{\text{tessellation level}\}^2$

2.4. Ficheiro .3d

Após gerar a malha, guardamos os triângulos em ficheiros .3d que, posteriormente, são carregados na engine, analogamente ao que já existia.

3. Curvas Catmull-Rom

Para esta fase foi necessário modificar a função `processaXML_grupo_getTransformacao`, dado que os arquivos XML receberam uma modificação em relação ao `translate`. Anteriormente, essa transformação apenas trasladava o eixo principal para as coordenadas especificadas na instrução, mas agora também pode fornecer novas informações que vamos explicar noutro ponto do relatório.

A nova versão da `processaXML_grupo_getTransformacao` funciona com ambas as versões do `translate` e, caso a lista de pontos tenha um tamanho inferior a 4, ela será ignorada. Caso contrário, será possível desenhar a curva de Catmull-Rom (quanto maior o número de pontos, melhor será a curva gerada).

Em seguida, foi necessário modificar a função `efetuaTransformacoes` do engine para que conseguisse interpretar os dados mencionados anteriormente e, assim, realizar o seguinte:

3.1. Representação da Curva

A trajetória é definida a partir do conjunto de pontos de controle. Para visualizar essa trajetória, interpola-se uma série de posições intermediárias ao longo da curva usando a função `renderCatmullRomCurve`. Essa função é responsável por desenhar a curva,

gerando uma quantidade fixa de segmentos (por exemplo, 100) que simulam a continuidade da curva. Cada ponto é obtido avaliando a curva de Catmull-Rom em diferentes valores de tempo.

3.2. Movimento do objeto em função do tempo

Para animar o objeto, utiliza-se o tempo de execução do programa (obtido continuamente) e normaliza-se esse tempo em relação à duração total do ciclo de animação. Isso permite determinar a posição exata do objeto ao longo da curva a qualquer momento, como uma porcentagem do trajeto completo.

Essa interpolação global é feita através da função `getGlobalCatmullRomPoint`, que converte o tempo global em um conjunto de quatro pontos consecutivos da curva. Em seguida, ela utiliza `getCatmullRomPoint` para calcular tanto a posição quanto a derivada naquele instante.

- A **posição** indica onde o objeto deve estar.
- A **derivada** indica a direção para a qual o objeto está se movendo.

Essa derivada é essencial para calcular a orientação do objeto mais adiante.

3.3. Aplicação de Transformações

Uma vez obtida a posição do objeto na curva, aplica-se uma translação para posicioná-lo exatamente nesse ponto.

3.4. Alinhamento do objeto com a curva (opcional)

Se a opção de alinhamento estiver ativada, o objeto será orientado automaticamente na direção do movimento:

- Usa-se a derivada como vetor tangente (eixo X local).
- Calcula-se um vetor perpendicular a esse usando o produto vetorial entre o tangente e um vetor vertical fixo (eixo Y global). Isso gera o eixo Z local.
- Depois, recalcula-se o eixo Y local como o produto vetorial entre os eixos Z e X.
- Os três vetores resultantes (X, Y, Z) formam um sistema ortonormal que é usado para construir uma matriz de rotação.
- Finalmente, aplica-se essa rotação ao objeto, garantindo que ele esteja sempre apontando na direção do movimento sobre a curva.

3.5. Matriz de Rotação

A função `buildRotMatrix` constroi essa matriz a partir dos três vetores mencionados. Essa matriz é usada para transformar o objeto de modo que ele fique corretamente orientado no espaço.

3.6. Funções Adicionais

- `normalize`: converte um vetor em um vetor unitário (de comprimento 1), o que é essencial para evitar erros nos cálculos de direção.
- `cross`: calcula o produto vetorial entre dois vetores, útil para obter um eixo perpendicular ao plano definido por outros dois vetores.

4. Cenário dinâmico

Nesta fase o cenário altera-se com base no tempo, em vez de alguma alteração feita pelo utilizador. Por isso, vamos ter que usar a função `glutIdleFunc` do OpenGL para atualizar o cenário automaticamente ao longo do tempo.

Durante o processamento do XML que descreverá o cenário a ser apresentado, verificamos se existe alguma parte do mesmo que faça com que o cenário seja dinâmico. Se for, então o `glutIdleFunc` terá que ser usado.

Para termos noção da passagem do tempo, usaremos o método do OpenGL chamado `glutGet(GLUT_ELAPSED_TIME)`.

4.1. Translate

A translação agora possui duas versões: uma apenas contém as coordenadas para transladar os eixos principais (como já vimos nas fases anteriores), e a nova versão inclui:

- **Points**: uma lista de pontos para criar uma curva de Catmull-Rom.
- **Time**: um valor de tempo medido em segundos que representa quanto tempo o modelo leva para dar uma volta completa na curva.
- **Align**: um valor booleano que, se verdadeiro, faz com que o modelo fique alinhado com a curva.

Esses valores passam a ter grande importância quando se trata de desenhar a curva de Catmull-Rom, como explicamos nos pontos anteriores.

4.2. Rotate

A rotação pode ser efetuada indicando um ângulo de rotação ou o tempo, em segundos, que demora para que a figura realize uma rotação de 360° .

Caso a rotação seja definida por tempo, devemos aplicar uma rotação cujo ângulo é calculado com base no tempo usando uma simples regra de três simples:

$$\frac{\text{Tempo para efetuar rotação de } 360^\circ}{\text{Tempo atual} - x} = \frac{360^\circ}{x}$$

As variáveis são:

- **Tempo para efetuar rotação de 360°** : O tempo especificado no ficheiro XML
- **Tempo atual** : tempo obtido com o `glutGet(GLUT_ELAPSED_TIME)`
- **X** : Ângulo de rotação a aplicar

Nota: Não é necessário fazer com que o ângulo não ultrapasse 360° pois

$$\text{ângulo} = x + 360^\circ \times i = x \quad (i \in \mathbb{Z})$$

5. VBOs

Nesta fase, foi solicitado o uso de VBOs (Vertex Buffer Objects). Em vez de desenhar cada ponto diretamente, como ocorria nas fases anteriores, todos os vértices são agora armazenados num VBO. Esta abordagem permite que o envio dos dados para a placa gráfica seja feito de forma mais eficiente, melhorando o desempenho da aplicação no momento do desenho.

Foi visto também que usar VBOs com índices permite os modelos se gerem de forma mais eficiente e rápida, uma vez que deixa de haver a repetição de vértices. Deste modo, modificamos a função que gera os ficheiros .3d para fazer uma análise sobre os pontos gerados e fornecer um índice de pontos que será usado pela engine. Graças a isto, permitimos à engine usar VBOs com índices.

Ao usar VBOs com índices, o desenho dos pontos é mais eficiente do que usar puramente VBOs. Também o tamanho dos ficheiros .3d reduziu aproximadamente em 75%.

*Ficheiro sphere_3_20_20.3d antes dos VBOs : **57,6 kB***

*Ficheiro sphere_3_20_20.3d depois dos VBOs com índices : **19,7 kB***

Além disso, é importante referir que as curvas de *Catmull-Rom* por serem geradas diretamente na engine, não utilizam VBOs com índices, pois calcular índices para estes pontos durante a execução da engine comprometeria o desempenho da aplicação.

6. Extras

6.1. Torus

O gerador gera um tórus centrado na origem. A função que constroi este tórus é a “geraTorus”. Esta função recebe como argumentos o raio maior, o raio menor, o número de slices (fatias ao longo do tubo do tórus) e o número de stacks (divisões ao longo da circunferência do tubo).

O tórus é uma figura que se assemelha a um anel. Contém um tubo de raio r (raio menor) cujo centro está a uma distância R (raio maior) do centro do tórus. Na imagem abaixo é apresentada a estrutura de um tórus que contém o raio menor, r , e o raio maior, R .

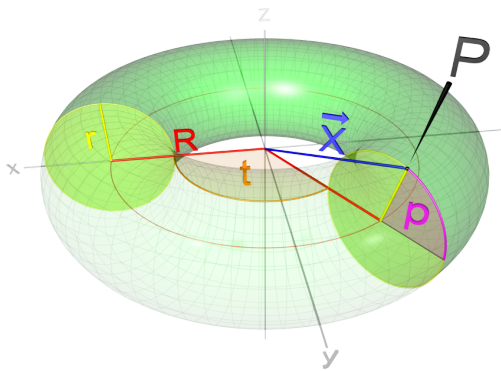


Figura 1. Representação do Torus

Para desenhar o tórus, utilizamos as suas coordenadas paramétricas que descrevem a sua superfície, utilizando dois ângulos, u e v . O ângulo u corresponde ao ângulo formado pelas slices e o ângulo v corresponde ao ângulo formado pelas stacks.

A imagem abaixo ilustra um troço de um tórus, onde são apresentadas duas slices (fatias) e o ângulo u formado pelas duas slices. Além disso, são assinaladas duas stacks, que formam o ângulo v .

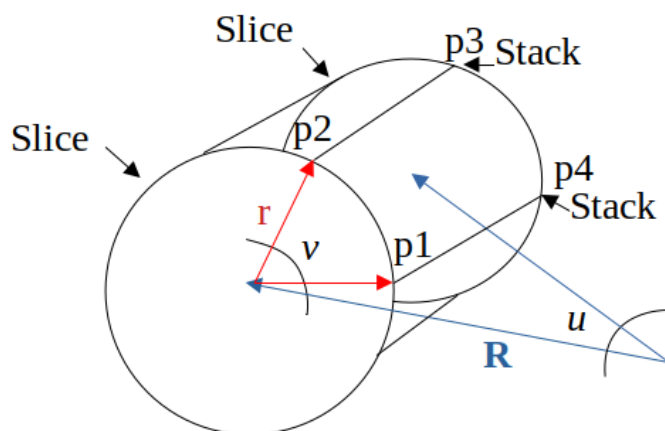


Figura 2. Esboço do Torus que será usado para a geração deste modelo

Desta forma, para construir esta figura, num ciclo externo são percorridas as stacks, onde são calculados os valores dos ângulos $v1$ e $v2$, cuja diferença forma o ângulo v , correspondentes ao início e fim de cada stack. Dentro de cada stack, o ciclo interior percorre as slices, onde são calculados os valores dos ângulos $u1$ e $u2$, cuja diferença forma o ângulo u , correspondentes ao início e fim de cada slice.

De seguida, são determinadas as coordenadas de cada ponto ($p1$, $p2$, $p3$ e $p4$), através das coordenadas paramétricas do tórus, calculadas a partir dos ângulos mencionados acima.

As equações abaixo correspondem às coordenadas paramétricas do tórus.

$$x(u, v) = (R + r \cos(v)) \cos(u)$$

$$y(u, v) = r \sin(v)$$

$$z(u, v) = (R + r \cos(v)) \sin(u)$$

Tórus 1

Raio maior: 5

Raio menor: 2

Slices: 30

Stack: 20

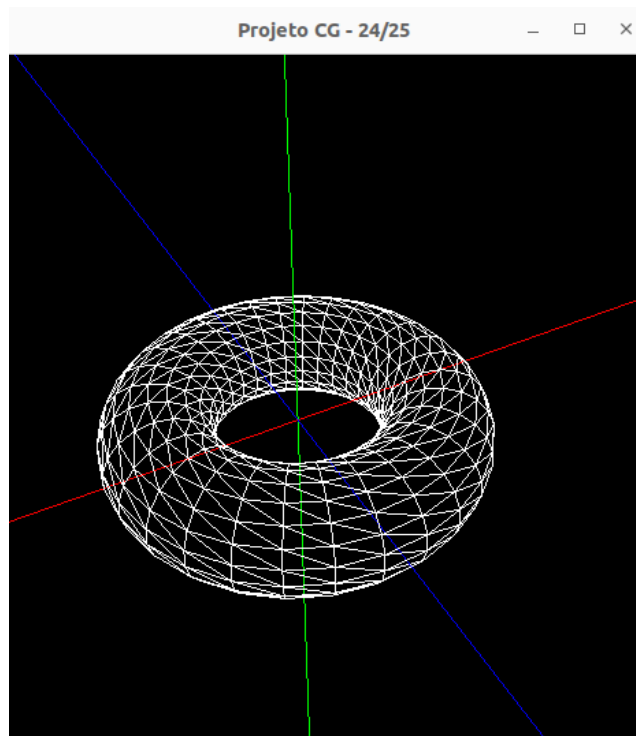


Figura 3. Exemplo 1 de um Tórus na Engine

Tórus 2

Raio maior: 8
Raio menor: 2
Slices: 8
Stack: 20

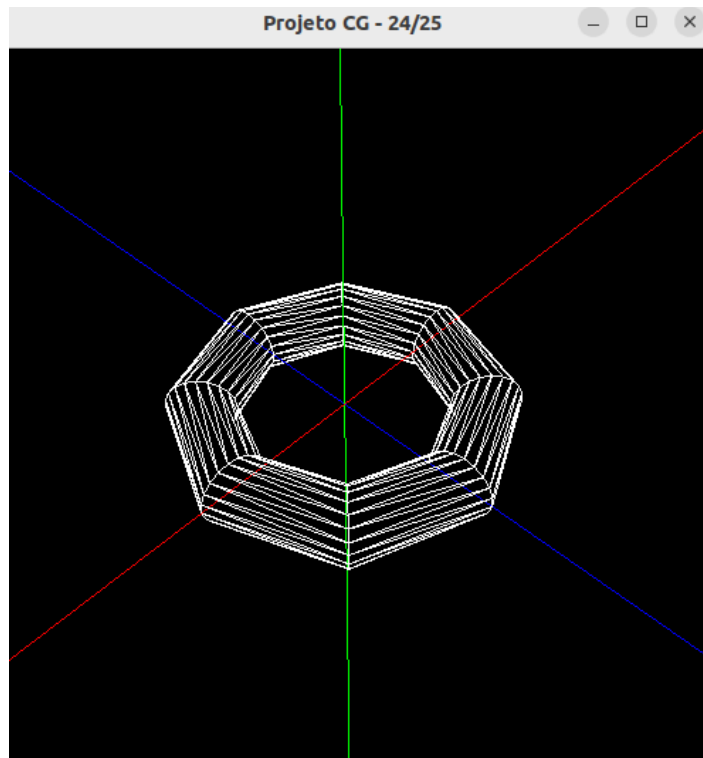


Figura 4. Exemplo 2 de um Tórus na Engine

6.2. Câmera Livre

Para uma melhor visualização do cenário, foi desenvolvida uma câmera livre, ou seja, o utilizador consegue se movimentar na cena e observá-la em ângulos e posições diferentes.

Para atingir este efeito, precisamos de duas componentes principais:

- **Posição atual** : esta posição é obtida através dos parâmetros nos ficheiros XML
- **Vetor da câmera** : Vetor que vai desde a posição da câmera até o objeto que está a ver

Nos cenários da engine, nem nos ficheiros XML, não há nenhuma menção deste vetor da câmera, e sim as coordenadas de onde ela deverá estar a olhar. Porém, utilizar um vetor da câmera é uma maneira bastante interessante e fácil de implementar esta câmera livre.

6.2.1. Vetor da câmera

Para definir os vetores da câmera, vamos usar dois ângulos:

- **Alfa** : Movimento horizontal
- **Beta** : Movimento vertical

Usamos umas funções que capturam o movimento do rato, considerando quais as alterações de movimentos com base na posição onde o rato clica pela primeira vez e na posição onde o rato ficou após o utilizador deixar de pressionar o rato. Com base nestas alterações, os ângulos alfa e beta são alterados. O ângulo beta, como é o ângulo do movimento vertical, impede que a câmera vire mais do que é suposto, limitando os seus valores a serem:

$$-90^{\circ} < \beta < 90^{\circ}$$

Após obter os novos valores dos ângulos, devemos obter o novo vetor da câmera. Para isso, convertemos os ângulos, que estão em graus, para radianos.

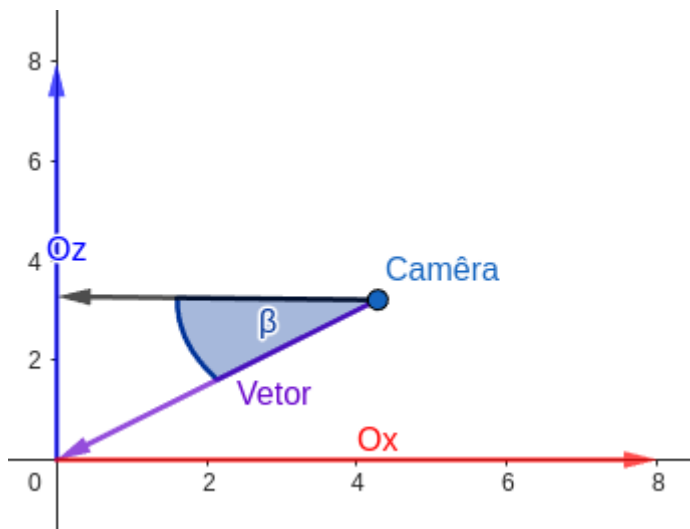


Figura 5. Cálculo do Vetor da Câmera usando o ângulo beta

Para obter a componente Y do vetor, usamos o valor de seno do ângulo beta. O ângulo alfa, responsável pelo movimento horizontal, é irrelevante para o cálculo da componente Y do vetor da câmera.

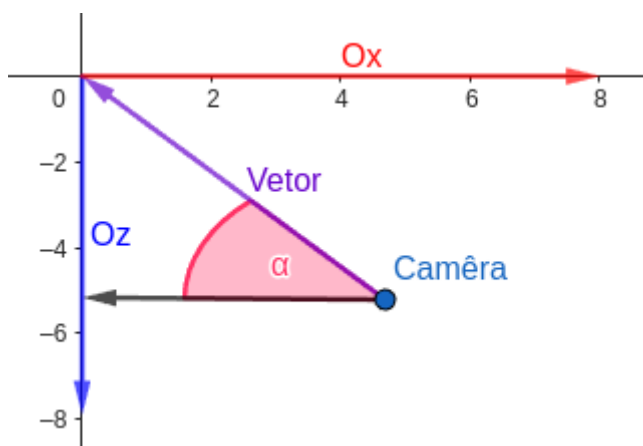


Figura 6. Cálculo do Vetor da Câmera usando o ângulo alfa

Para obtermos as componentes X e Z do vetor da câmera, vamos transformar o vetor 3D num vetor 2D para o cálculo. Para isso, multiplicamos pelo valor do cosseno do ângulo beta.

Para obter a componente X do vetor, usamos o valor de cosseno do ângulo alfa e na componente Z usamos o valor de seno do ângulo alfa

Em resumo:

- **Valor Y do vetor** : $\sin(\beta)$
- **Valor X do vetor** : $\cos(\beta) * \cos(\alpha)$
- **Valor Z do vetor** : $\cos(\beta) * \sin(\alpha)$

Após o cálculo, normalizamos o vetor resultante e atualizamos o vetor de câmera usado pela engine.

Por fim, a engine requer um ponto onde a câmera deverá estar a olhar. Para obtermos esse ponto, simplesmente fazemos uma soma da posição atual da câmera com o vetor da câmera.

6.2.2. Inicialização do Vetor da Câmera

Os ficheiros XML dão a posição da câmera e o ponto onde ela deverá estar a olhar. Porém, o vetor da câmera, que indica onde a câmera deve estar a olhar, utiliza dois ângulos, e não um ponto. Por isso, realizamos os seguintes procedimentos:

1. Usando a posição da câmera (ponto A) e o ponto para onde deverá olhar (ponto B), obtemos um vetor que inicia no ponto A e vai para o ponto B
2. Normalizamos o vetor resultante
3. Usamos a inversa do seno para obter o ângulo β
4. Usamos a inversa da tangente para obter o ângulo α . A inversa da tangente já nos dará o ângulo correto, considerando o quadrante onde está

Nota: para este cálculo não é preciso dividir pelo comprimento do vetor pois encontra-se normalizado

Com este procedimento, obtemos os ângulos alfa e beta e, desta forma, conseguimos obter o vetor da câmera.

6.2.3. Movimentação da Câmera

Para movimentarmos a câmera, o vetor da câmera vai ter um papel crucial para determinar a nova posição da mesma.

Para determinarmos essa nova posição devemos aplicar uma translação da posição da câmera com um vetor que indica qual a sua movimentação. Para a câmera andar para a frente (tecla 'W') ou para trás (tecla 'S'), apenas somamos ou subtraímos o vetor da câmera à posição da mesma, respetivamente.

Em relação às movimentações laterais, temos que obter um vetor que seja perpendicular ao vetor normal da câmera e ao vetor da câmera. Para isso, realizamos um produto vetorial sobre estes dois vetores e depois normalizamos o vetor perpendicular.

Para a câmera andar para a direita (tecla 'D') ou para a esquerda (tecla 'A'), somamos ou subtraímos o vetor perpendicular à posição da câmera, respetivamente

A movimentação é determinada por um fator de velocidade que pode ser definido no XML

6.2.4. Definições da Câmera Livre

A câmera livre sempre estará disponível. É possível deslocar a câmera com as teclas AWSD e é possível girá-la pressionando o botão esquerdo do rato e movendo-o com o botão esquerdo pressionado.

Todavia, decidimos adicionar opções para o XML para customizar a experiência com a câmera livre. Opções estas que são:

- **Disable** : indica se pretende desativar a câmera livre
- **Sensitivity** : indica o quanto a câmera gira com a deslocação do rato
- **NoNeedClick** : não é necessário pressionar o botão esquerdo do rato para movimentar a câmera pois ela sempre estará a seguir o movimento do rato (*Estilo jogos FPS*)
- **LimitedAngle** : indica se deverá ser possível a câmera girar mais do que deveria, fazendo com que o ângulo beta tenha qualquer valor

$$-360^{\circ} \leq \beta \leq 360^{\circ}$$

- **Speed** : indica o quanto a câmera se desloca

```
<camera>
  <free disable="True" sensitivity="1.0" noNeedClick="True" limitedAngle="False" speed="1.0"/>
</camera>
```

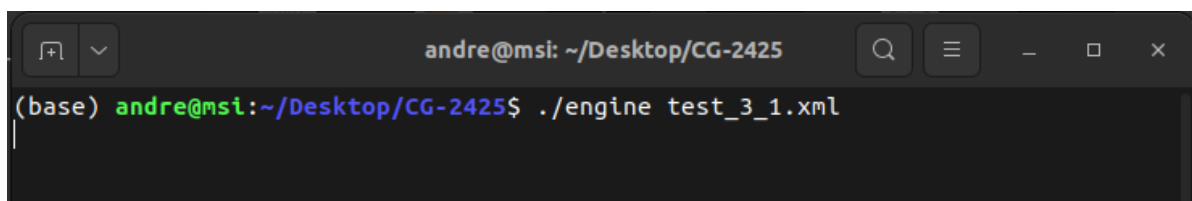
Figura 7. Exemplo de definições da câmera livre nos ficheiros XML

7. Demos

Aqui estão representadas as demos que fizemos (tanto as que foram fornecidas pelo enunciado, bem como as nossas) para esta fase do trabalho prático da UC Computação Gráfica.

7.1. Demos do Enunciado

7.1.1. Ficheiro test_3_1.xml



```
andre@msi: ~/Desktop/CG-2425
(base) andre@msi:~/Desktop/CG-2425$ ./engine test_3_1.xml
```

Figura 8. Execução da Engine do Teste 1 da Fase 3

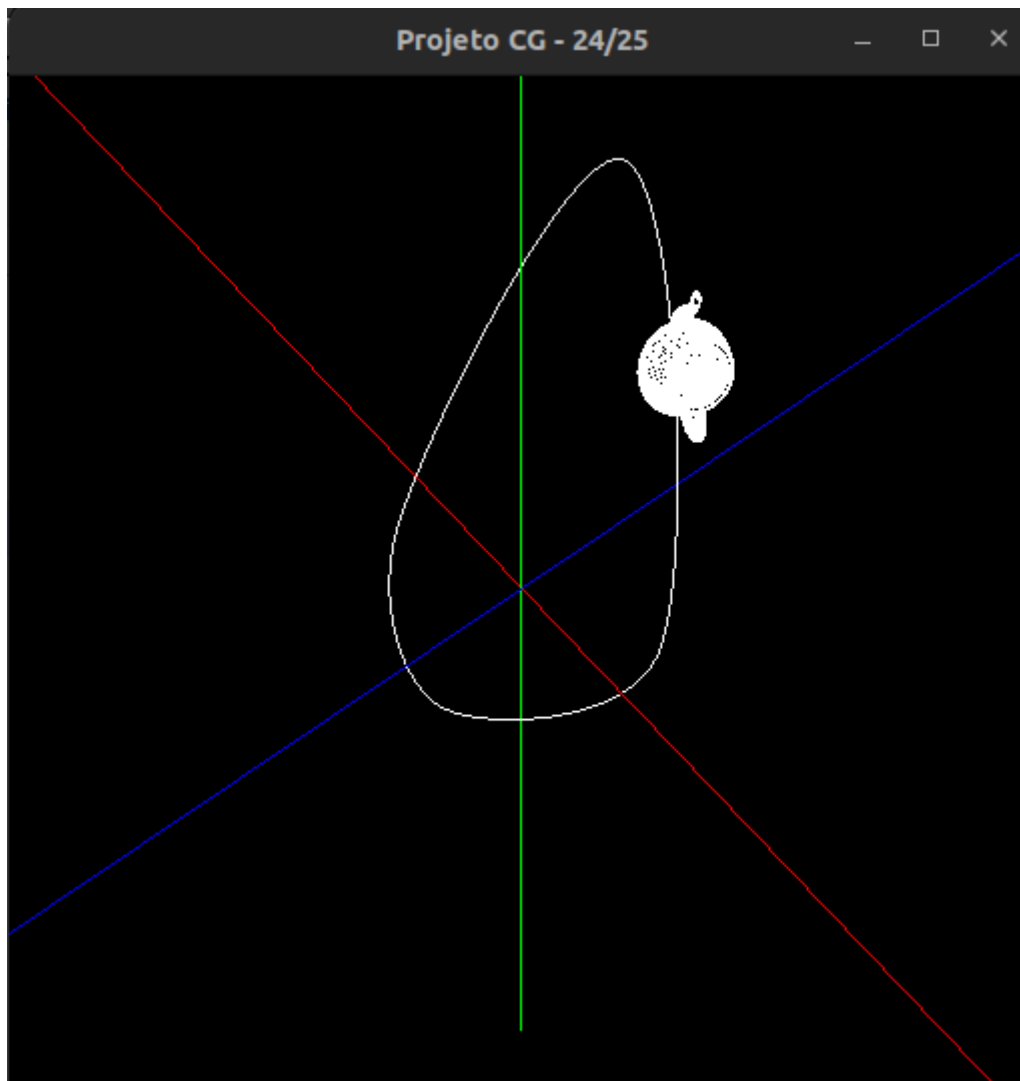


Figura 9. Cenário desenhado pela Engine descrito pelo ficheiro XML Teste 1 da Fase 3

7.1.2. Ficheiro test_3_2.xml

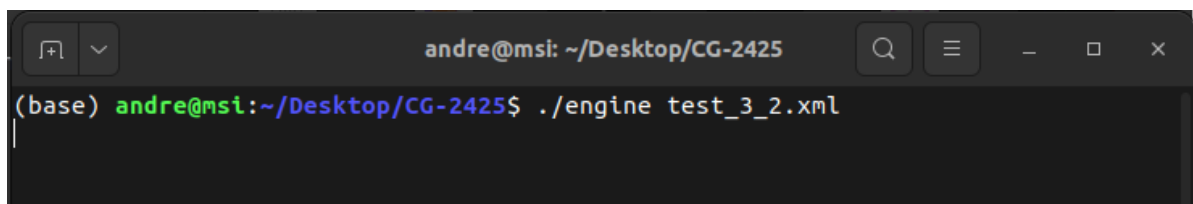


Figura 10. Execução da Engine do Teste 2 da Fase 3

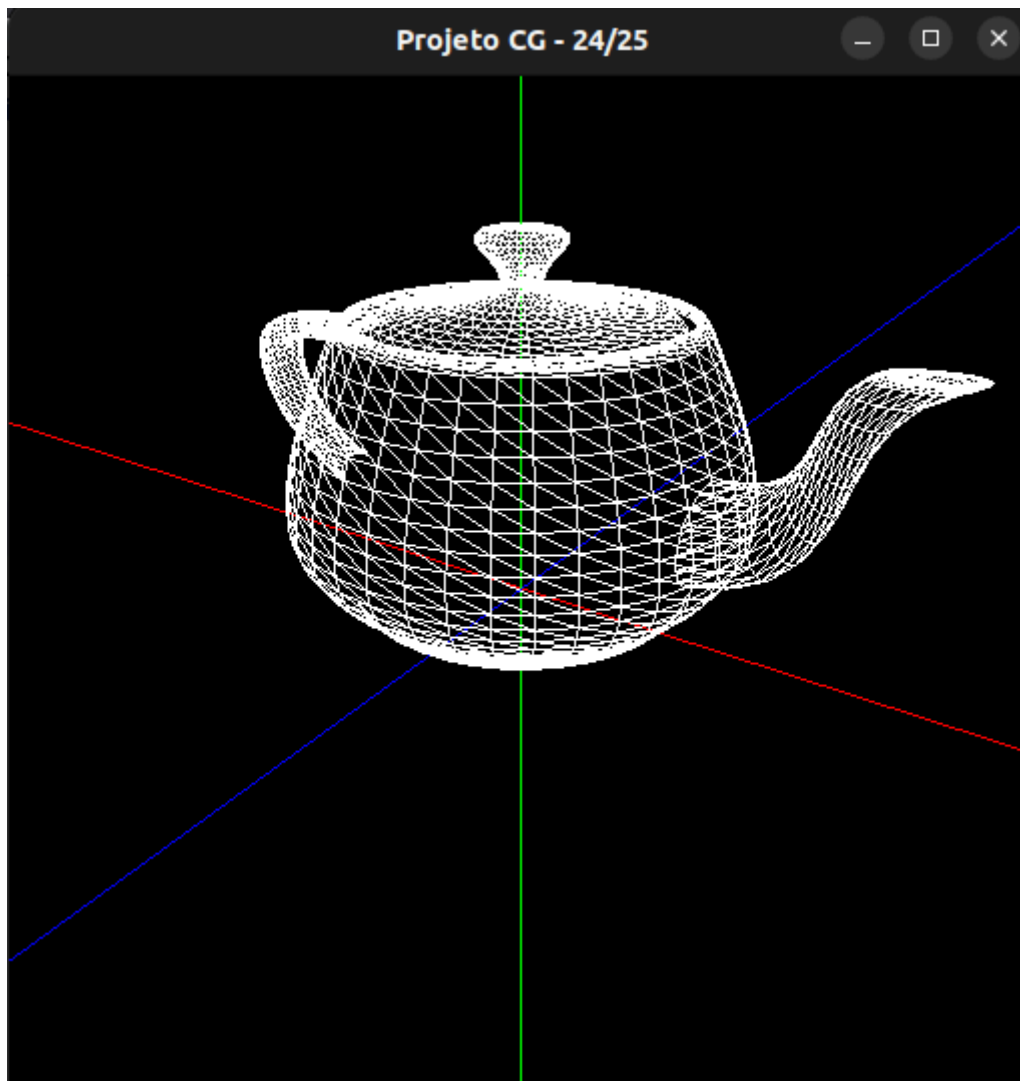


Figura 11. Cenário desenhado pela Engine descrito pelo ficheiro XML Teste 2 da Fase 3

7.2. As nossas Demos

7.2.1. Ficheiro torus.xml

```
andre@msi: ~/Desktop/CG-2425/files
(base) andre@msi:~/Desktop/CG-2425/files$ ./engine torus.xml
```

Figura 12. Execução da Engine do Torus da Fase 3

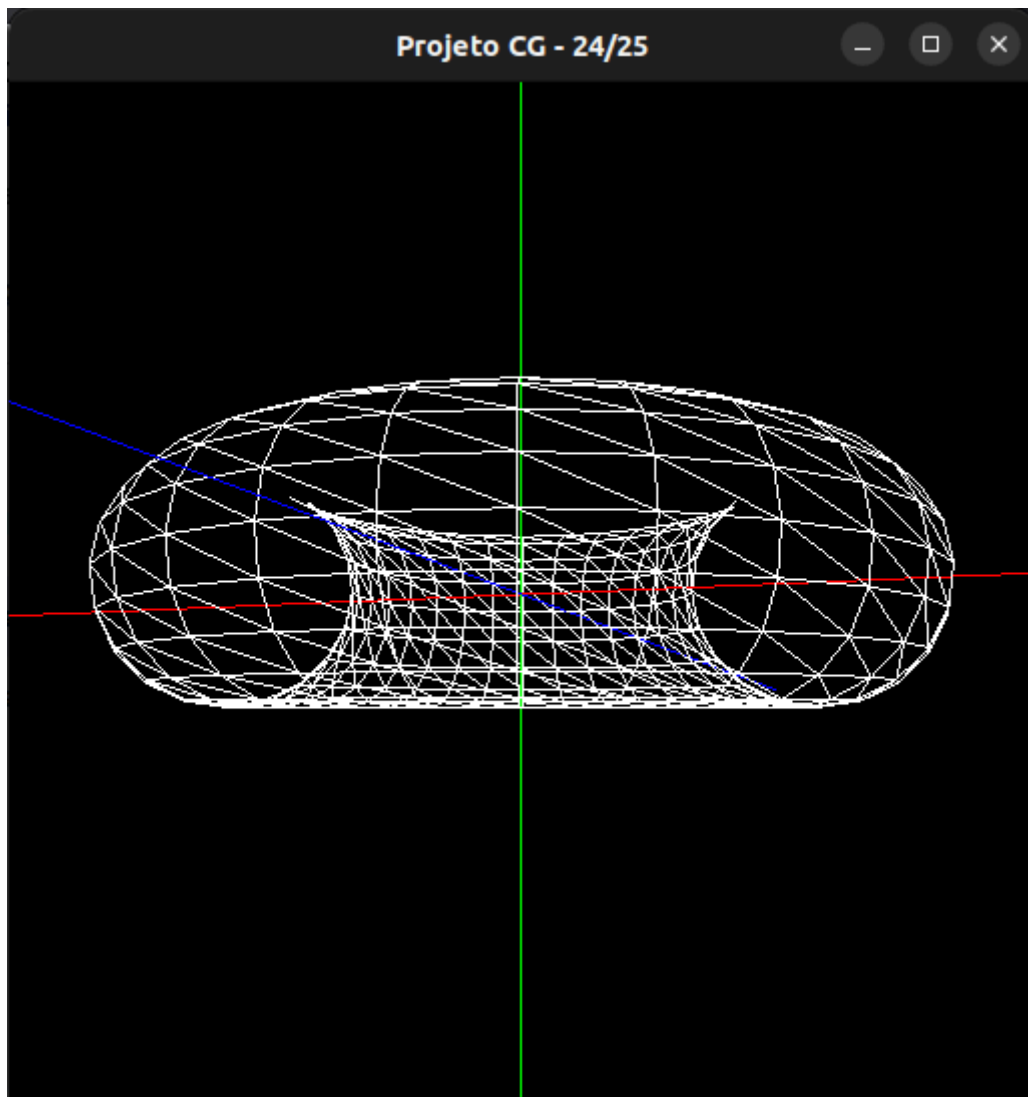


Figura 13. Cenário desenhado pela Engine descrito pelo ficheiro XML Torus da Fase 3

7.2.2. Ficheiro demo_1.xml

```
andre@msi: ~/Desktop/CG-2425/files
(base) andre@msi:~/Desktop/CG-2425/files$ ./engine demo_1.xml
```

Figura 14. Execução da Engine do Demo 1 da Fase 3

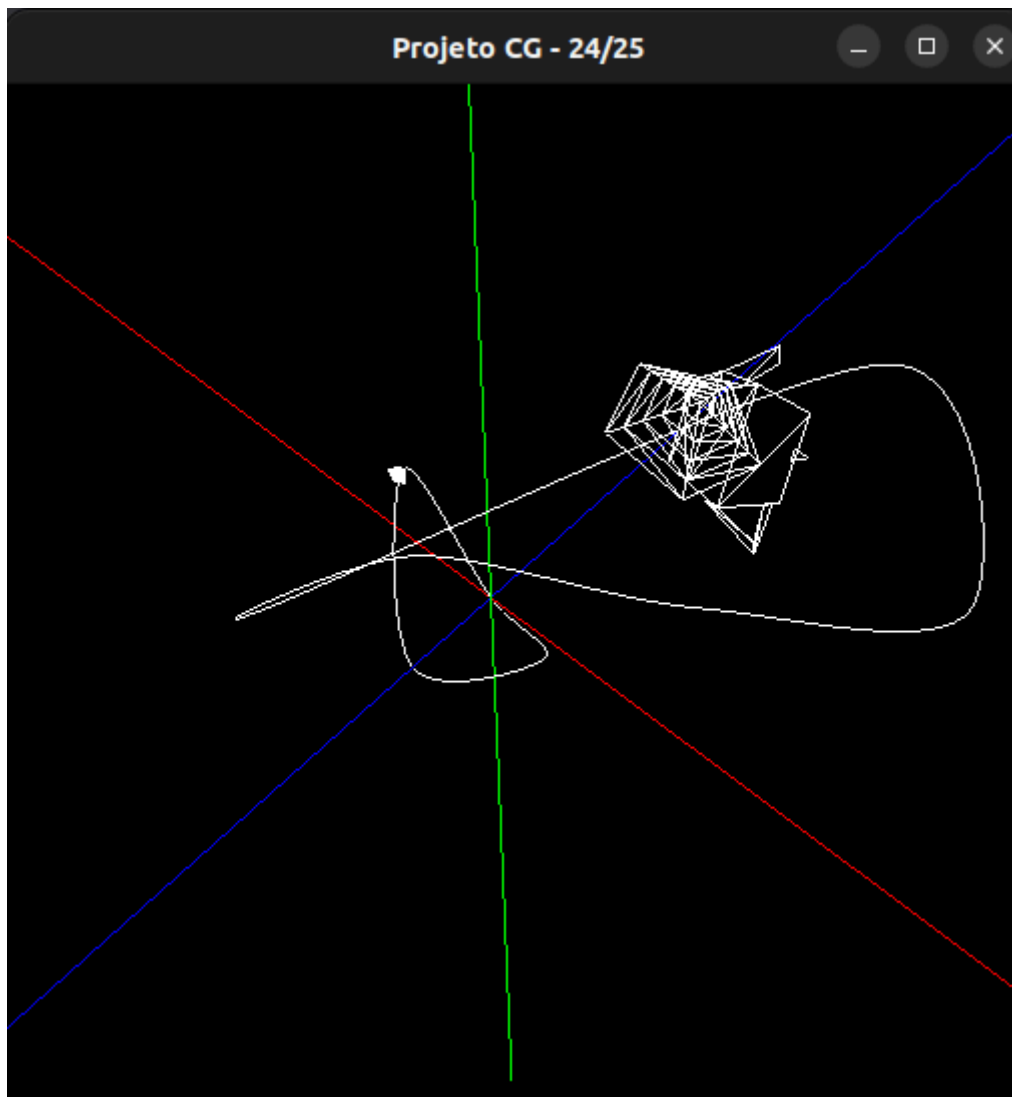


Figura 15. Cenário desenhado pela Engine descrito pelo ficheiro XML Demo 1 da Fase 3

7.2.3. Ficheiro sistema_solar.xml

Nesta fase foi pedido que o sistema solar da Fase 2 fosse dinâmico, ou seja, os planetas movimentam-se ao longo do tempo, usando as translações e rotações dinâmicas vistas acima. Foi pedido também que fosse adicionado um cometa, feito com base nas curvas de Bezier, e a sua trajetória usando curvas de Catmull-Rom.

Para esse efeito, as rotações e translações foram substituídas, usando tempo em vez de ângulos de rotação e vetores de deslocação, respetivamente. Nesta fase foi adicionado o **tórus** para simular o anel de Saturno.

Para gerar a curva das trajetórias dos planetas, usamos as curvas de Catmull-Rom na translação. Para obter os pontos necessários para definir essas rotas, usamos um script chamado **calcular_pontos_catmulrom_sistema_solar.py** que, dado um raio como argumento, dá 16 pontos que definem a curva.

Para gerar o cometa, usamos um script chamado **bezier_cometa.py** que cria um patch necessário para a criação do ficheiro .3d do cometa. Este script junta 8 patches, onde cada patch é um “gomo” de uma esfera em cada um dos octantes.

```
andre@msi: ~/Desktop/CG-2425/files
(base) andre@msi:~/Desktop/CG-2425/files$ ./engine sistema_solar.xml
```

Figura 16. Execução da Engine do Sistema Solar da Fase 3

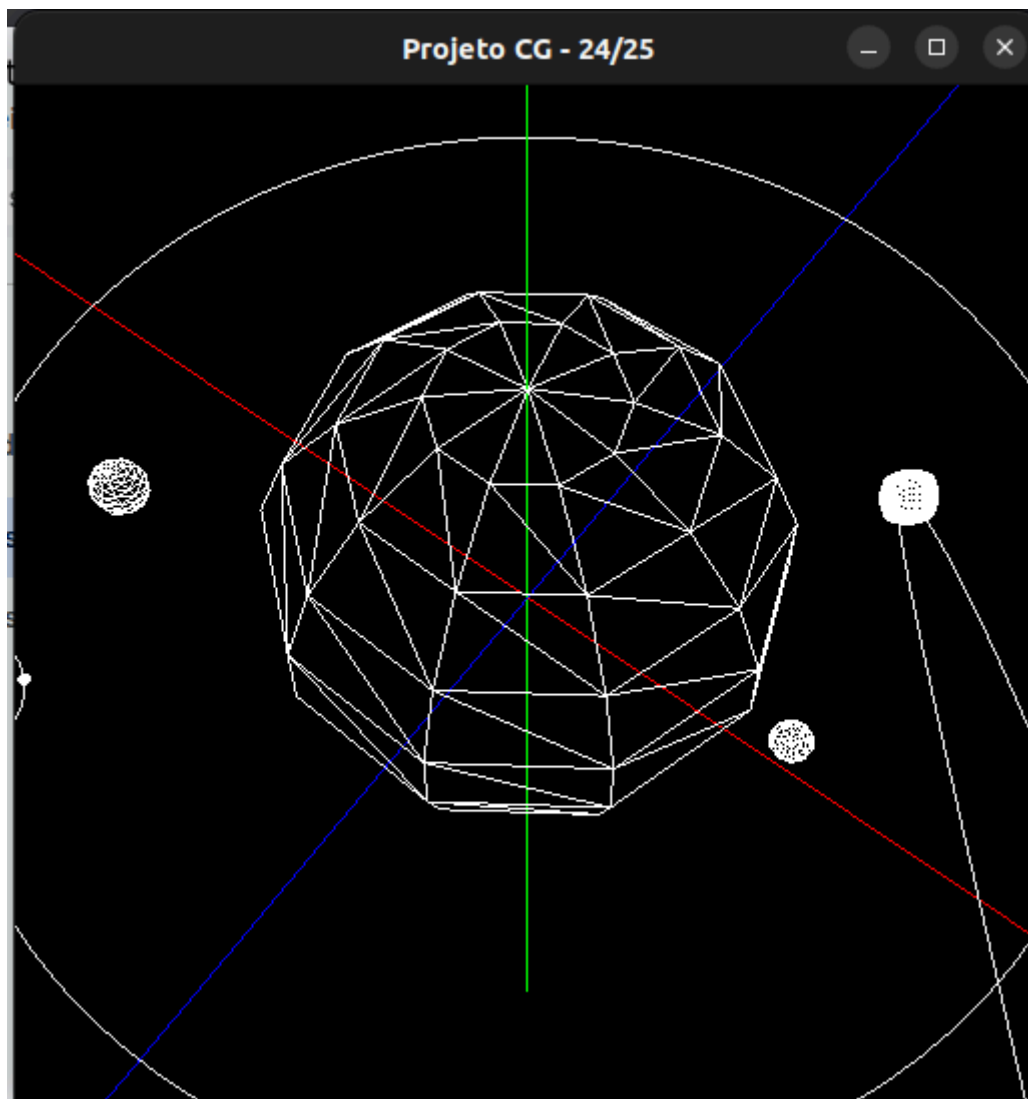


Figura 17. Cenário desenhado pela Engine descrito pelo ficheiro XML Sistema Solar da Fase 3

8. Conclusão

Com este projeto, conseguimos obter cenários dinâmicos que alteram conforme a passagem do tempo, algo pedido nesta fase.

O gerador consegue criar novos modelos com base em curvas de Bezier e os ficheiros .3d não contêm apenas pontos, mas também índices para serem usados pelos VBOs da engine.

A engine agora lida com o tempo, fazendo com que não haja apenas cenários estáticos. Os modelos podem se deslocar (translação) por um movimento descrito por uma curva, definida pelas curvas de Catmull-Rom, ao longo do tempo, indicando se alinham com a curva ou não. Para além dessa deslocação, os modelos também podem rodar ao longo do tempo.

Nesta fase conseguimos aprender como trabalhar e definir curvas, mais precisamente curvas de Catmull-Rom e curvas de Bezier, como usar o tempo em cenários e como usar os VBOs para desenhar os modelos de forma mais eficiente e rápida.

Por fim, os extras mencionados nas fases anteriores, nomeadamente novos modelos e uma câmera livre, foram implementados nesta fase.