

Universidade do Minho
Escola de Engenharia

Fase IV Normais e Coordenadas de Textura

Trabalho Prático Computação Gráfica

Grupo 48

Gabriela Santos Ferreira da Cunha - a97393

João Gonçalo de Faria Melo - a95085

Nuno Guilherme Cruz Varela - a96455



a97393



a95085



a96455

junho, 2023

Conteúdo

1	Introdução	3
2	Normais	3
2.1	Plano	3
2.2	Caixa	4
2.3	Esfera	4
2.4	Cone	5
2.5	Superfícies de <i>Bézier</i>	6
2.6	<i>Torus</i> e Elipsoide	7
3	Coordenadas de Textura	7
3.1	Plano	7
3.2	Caixa	8
3.3	Esfera	9
3.4	Cone	9
3.5	Superfícies de <i>Bézier</i>	10
3.6	<i>Torus</i>	10
3.7	Elipsoide	10
4	Iluminação	11
4.1	Fontes de Luz	11
4.2	Materiais	12
5	Aplicação no Sistema Solar	13
6	<i>View Frustum Culling</i>	13
6.1	<i>Generator</i>	14
6.2	<i>Engine</i>	14
7	Resultados Finais	16
7.1	Sistema Solar	16
7.2	Testes	18
8	Conclusão	21

1 Introdução

No âmbito deste projeto, foi-nos proposto o desenvolvimento de um mecanismo 3D baseado num cenário gráfico, dividido em 4 fases.

O presente relatório é referente à última fase do trabalho, cujo objetivo consistia na adição das funcionalidades de textura e iluminação aos cenários. Desta forma, o *generator* deve gerar as coordenadas de textura e as normais para cada vértice do modelo e o *engine* deve ativar estas funcionalidades, lendo e aplicando as normais e as coordenadas de textura de cada ficheiro de pontos gerado.

2 Normais

Uma normal consiste no vetor unitário perpendicular à superfície num dado ponto.

O cálculo deste vetor é uma etapa importante no processo de renderização dos objetos, sendo o conjunto das normais usado para determinar como a luz interage com as superfícies destes objetos. Ao calcular as normais dos vértices de um objeto, o *OpenGL* pode determinar a direção em que a luz incide em cada ponto da superfície e, com base nisso, é possível calcular como a luz é refletida ou dispersada, resultando em diferentes efeitos de iluminação.

A escrita destas coordenadas no ficheiro ocorre depois da escrita dos índices dos triângulos e é similar à escrita dos vértices gerados, sendo cada uma um conjunto de 3 valores reais.

2.1 Plano

O plano é a primitiva gráfica mais elementar das desenvolvidas, sendo as suas normais obtidas de uma forma bastante simples, uma vez que são iguais para qualquer vértice que o constitui. Cada plano gerado é paralelo ao eixo $x0z$, sendo a normal de qualquer ponto igual a $(0, 1, 0)$.

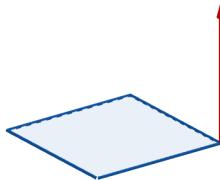


Figura 1: Normal do plano.

2.2 Caixa

A caixa é constituída por 6 planos, onde 2 são paralelos ao eixo $x0z$, outros 2 ao eixo $x0y$ e os restantes ao eixo $y0z$. Deste modo, para cada par de planos paralelos ao mesmo eixo, a normal irá ser praticamente idêntica, sendo a coordenada não nula a mesma. Contudo, os valores destas coordenadas não nulas irão ser simétricos uma vez que, por exemplo, para os planos paralelos ao eixo $x0z$, um estará virado para baixo e outro para cima, sendo as suas normais $(0, -1, 0)$ e $(0, 1, 0)$, respectivamente.

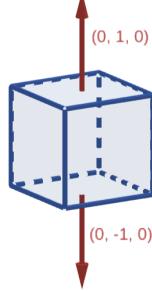


Figura 2: Normais dos planos paralelos ao eixo $x0z$.

De notar que cada ponto é definido pela sua posição, pela sua normal e pela sua coordenada de textura. Os pontos que, visualmente, pertencem a 2 ou 3 faces da caixa, como é o caso do ponto da figura 3, são considerados diferentes pontos, uma vez que possuem uma normal diferente para cada plano.

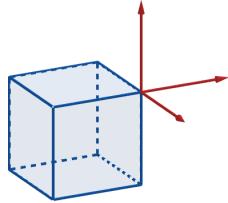


Figura 3: Normais da caixa.

2.3 Esfera

Relativamente à esfera, o cálculo das normais é feito com base nas coordenadas esféricas. Desta forma, para cada vértice calculamos o vetor com origem em $(0, 0, 0)$ e destino no ponto com coordenadas $(\cos \beta \times \sin \alpha, \sin \beta, \cos \beta \times \cos \alpha)$. Com isto, garantimos que o vetor está já normalizado, visto que as coordenadas esféricas utilizadas têm raio 1.

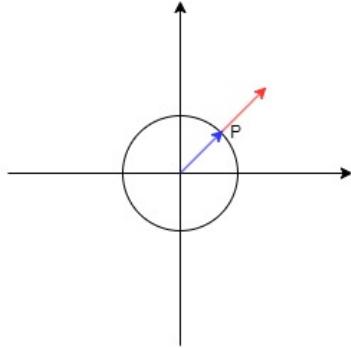


Figura 4: Normais da esfera.

De uma forma mais visual, a normal, assinalada a vermelho, do ponto P da figura acima corresponde ao vetor com cor azul, assumindo que o raio de circunferência é 1.

2.4 Cone

O cone foi a primitiva gráfica que trouxe mais dificuldades ao grupo. As normais dos vértices da base do cone correspondem ao vetor $(0, -1, 0)$. Relativamente às normais do resto do cone, observamos que para cada geratriz as normais dos vértices são todas iguais. Deste modo, para cada *slice* pré-computamos a normal e aplicámo-la a todos os vértices da geratriz.

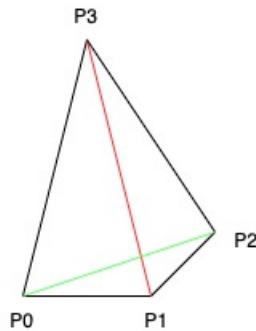


Figura 5: Normal a cada geratriz.

O cálculo da normal de cada geratriz é feito com base no produto vetorial de dois vetores. O primeiro corresponde ao vetor da geratriz, assinalado a vermelho na figura 5, que pode ser calculado através do topo do cone ($P3$) e do vértice correspondente à interseção da base com a geratriz ($P1$). O segundo vetor corresponde ao vetor formado entre os vértices da interseção da geratriz anterior

e da geratriz seguinte com a base do cone, ou seja, $P0$ e $P2$, respectivamente. Finalmente, realizamos o produto vetorial dos dois vetores, tendo em conta a regra da mão direita, e normalizamos o vetor resultante.

2.5 Superfícies de *Bezier*

Para o cálculo das normais para as superfícies de *Bezier*, realizamos o produto vetorial de dois vetores tangentes à superfície. Estes vetores correspondem às derivadas parciais em ordem a u e v , que podem ser calculadas da seguinte forma:

$$\frac{\partial p(u, v)}{\partial u} = [3u^2 \quad 2u \quad 1 \quad 0] M \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix} \quad \text{e} \quad \frac{\partial p(u, v)}{\partial v} = [u^3 \quad u^2 \quad u \quad 1] M \begin{bmatrix} 3v^2 \\ 2v \\ 1 \\ 0 \end{bmatrix},$$

onde M corresponde à matriz pré-computada inicialmente.

Com os dois vetores calculados, realizamos o produto vetorial entre ambos, tendo em consideração a regra da mão direita. Finalmente, o vetor resultante é normalizado.

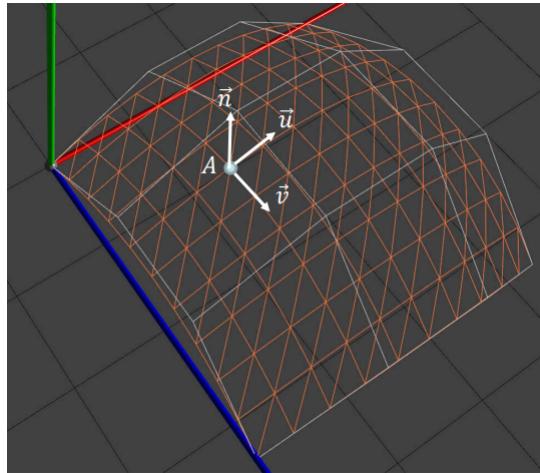


Figura 6: Normal a uma superfície de *Bezier*.

Na realização de testes para as normais das superfícies de *Bezier*, averiguamos que existiam casos em que as componentes da normal calculada eram “NaN“ (*Not a number*). Nesses casos optamos por atribuir a normal calculada imediatamente antes. Caso não haja uma normal calculada anteriormente, atribuímos o vetor $(0, 1, 0)$.

2.6 *Torus* e *Elipsoide*

O cálculo das normais dos vértices do *torus* e do elipsoide é feito de uma forma semelhante à estratégia utilizada no cálculo das normais da esfera. Cada normal é, portanto, caracterizada pela seguinte expressão $(\cos \beta \times \sin \alpha, \sin \beta, \cos \beta \times \cos \alpha)$, que corresponde às coordenadas esféricas de raio 1.

3 Coordenadas de Textura

Nesta fase os objetos podem ter texturas e, portanto, o *generator* deve ser capaz de gerar coordenadas de textura, de modo a conseguirmos mapear a textura nele. As coordenadas de textura são escritas a seguir às normais e consistem num conjunto de 2 valores reais.

Para algumas primitivas gráficas, como o plano e a caixa, realizamos diferentes mapeamentos das texturas. O mapeamento que o utilizador pretende, seja ele *tiled* ou *stretched*, terá de ser indicado nos argumentos do *generator*, sendo *stretched* por predefinição.

Através da biblioteca Devil, conseguimos carregar a imagem que contém a textura e a mesma é criada no *engine*, a partir dessa imagem. Assim como os modelos, que são diretamente lidos da pasta “models”, contida na pasta “demo-scenes”, a leitura da imagem foi configurada para se efetuar a partir da pasta “textures”, também contida na pasta “demo-scenes”.

3.1 Plano

O plano é construído através do seu ponto inferior esquerdo. Desta forma, conseguimos estabelecer uma correspondência direta entre os vértices do plano e a textura.

Como referido acima, realizamos diferentes mapeamentos para o plano, sendo eles o mapeamento da textura na totalidade da primitiva e o mapeamento da textura a cada dois triângulos, ou seja, por cada divisão do plano.

Relativamente ao mapeamento da textura na totalidade da primitiva, tivemos que dividir o intervalo $[0, 1]$ da textura pelo número de divisões do plano, de maneira a sabermos o “salto” a dar na textura para cada vértice. Utilizamos, desta forma, a seguinte expressão para mapear a textura em cada vértice: $(i \times texPart, j \times texPart)$, onde $texPart = \frac{1}{divisions}$, *divisions* é o número de divisões do plano e *i* e *j* correspondem aos iteradores dos ciclos que iteram pelo plano.

O mapeamento da textura em cada dois triângulos é feito de uma forma direta com base no mapeamento dos cantos da texturas nos cantos do quadrado formado pelos dois triângulos.

De seguida, podemos visualizar a diferença dos dois mapeamentos referidos acima.

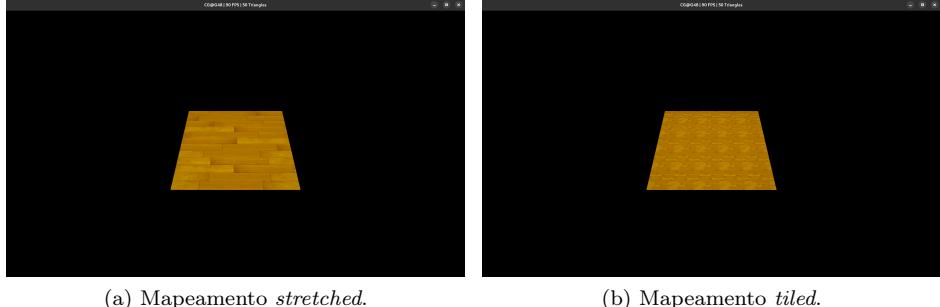


Figura 7: Mapeamentos do plano.

Para gerar o primeiro mapeamento devem ser utilizados os seguintes argumentos na execução do *generator*:

```
generator plane {length} {divisions} stretched {filename.3d}
```

Para o segundo, deve ser alterado o tipo de mapeamento da seguinte forma:

```
generator plane {length} {divisions} tiled {filename.3d}
```

Como já tínhamos referido, caso o tipo de mapeamento não seja indicado, o mapeamento obtido será o primeiro.

3.2 Caixa

O cálculo das coordenadas de textura da caixa é feito com uma estratégia igual à do plano. Desta vez, teremos de replicar 6 vezes, visto que uma caixa é constituída por 6 planos. À semelhança do plano, também possuímos diferentes mapeamentos para cada plano: *tiled* e *stretched*.

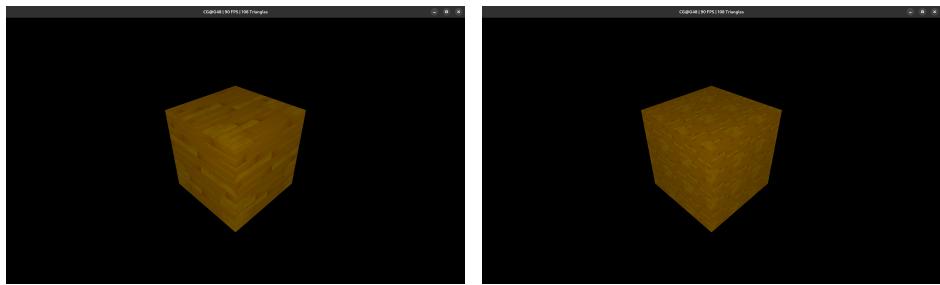


Figura 8: Mapeamentos da caixa.

3.3 Esfera

No mapeamento da textura na esfera, começamos por calcular os tamanhos dos “saltos“ que vamos tomar em ambos os eixos da textura, com base nas *slices* e *stacks*, através das seguinte expressões: $xTexPart = \frac{1}{slices}$ e $yTexPart = \frac{1}{stacks}$.

De seguida, calculamos as coordenadas de textura de uma forma muito semelhante à forma utilizada no mapeamento *stretched* do plano. Considerando que i e j são as variáveis que iteram pelas *slices* e *stacks* respetivamente, deduzimos a seguinte fórmula para as coordenadas de textura de cada vértice : $(i \times xTexPart, j \times yTexPart)$.

3.4 Cone

O cálculo das coordenadas de textura para o cone foram divididas em duas fases: cálculo das coordenadas para a base e para o corpo do cone.

Em relação à base do cone, mapeamos a textura através de coordenadas de uma circunferência com raio de 0.5. De modo a colocar a circunferência no espaço de textura no intervalo de $[0, 1]$ em ambos os eixos, realizamos um deslocamento de 0.5 em ambas as componentes.

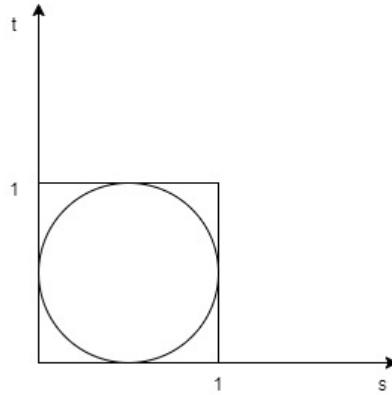


Figura 9: Mapeamento da base do cone.

O mapeamento feito para o corpo do cone é realizado seguiente a estratégia utilizada para a esfera, em que dividimos o espaço textura no intervalo $[0, 1]$ pelas *slices* e *stacks* do cone. De seguida, com base nestes valores e nas iterações dos ciclos, utilizamos a fórmula acima apresentada para calcular as coordenadas de textura de cada vértice.

Outro dos mapeamentos que podia ter sido feito era dedicar uma zona exclusiva do espaço textura para mapear diretamente na base do cone e outra

parte para mapear no restante cone. No entanto, acabamos por não realizar este último.

3.5 Superfícies de *Bezier*

O cálculo das coordenadas de textura para as superfícies de *Bezier* revelou-se bastante simples, visto que no cálculo dos vértices percorremos uma espécie de *grid* em que as variáveis u e v no intervalo de $[0, 1]$. Desta forma, conseguimos estabelecer uma correspondência direta entre o vértice e a sua coordenada da textura.

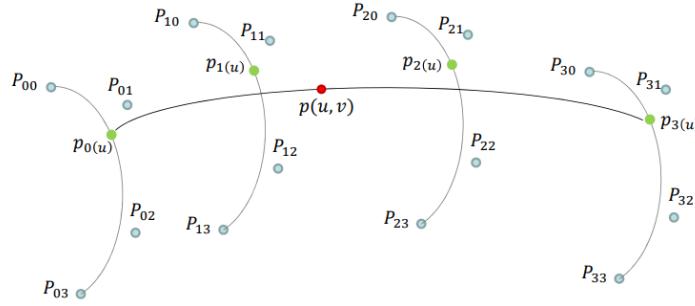


Figura 10: Mapeamento da textura nas superfícies de *Bezier*.

Neste caso, o vértice p da figura acima apresentada terá (u, v) como coordenadas de textura, sendo que u e v são as variáveis que iteram pela grelha.

3.6 Torus

Relativamente às coordenadas de textura da primitiva gráfica *torus*, utilizamos um mapeamento em que repetimos a textura por cada *slice* do torus. Desta maneira, por cada *slice* repetimos a textura e aplicámos-a na totalidade para todas as *stacks* dessa *slice*, dividindo o espaço de textura pelas n *stacks*. Em comparação com os mapeamentos já apresentados, não realizamos um mapeamento total do espaço textura na primitiva gráfica.

3.7 Elipsoide

As coordenadas de textura dos vértices do elipsoide são equivalentes às coordenadas calculadas para a esfera, ou seja, dividimos o espaço textura em ambos os eixos pelo número de *slices* e *stacks* para descobrir o “salto” no espaço textura. De seguida aplicamos a expressão apresentada na secção da esfera para cada vértice.

4 Iluminação

De modo a introduzir iluminação aos cenários, é necessário inicializar uma fonte de luz, isto é, ativar a iluminação e definir uma cor para a mesma e, em seguida, renderizá-la, definindo a posição da luz e um material para cada modelo.

4.1 Fontes de Luz

Para definirmos cada fonte de luz, recorremos à hierarquia definida pela seguinte figura.

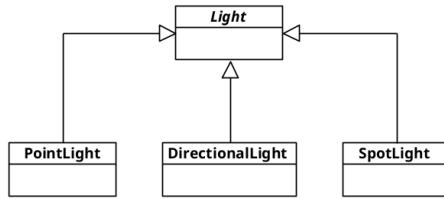


Figura 11: Hierarquia definida para os tipos de luz.

Assim, no momento em que é lido o ficheiro XML, são criadas instâncias para cada fonte de luz presente. A classe “world” vai conter um vetor com todas estas instâncias. Independentemente do tipo de iluminação, cada instância contém o valor da luz que deve ser utilizada que vai depender da posição que ocupa no vetor, sendo 8 as que o *OpenGL* suporta. A aplicação da luz é feita através do método abstrato “apply”, de acordo com as restantes variáveis de cada classe que diferem de tipo para tipo:

- No caso dos pontos de luz, é utilizado um vetor com 4 posições, onde as 3 primeiras correspondem às coordenadas da posição da fonte de luz e a última tem o valor 1, indicando que se trata de um ponto de luz;
- No caso das luzes direcionais, é utilizado o mesmo vetor mas a última posição tem o valor 0, indicando que se trata de uma luz direcional. Este vetor representa, assim, a direção da luz;
- No caso dos focos de luz, são utilizados 2 vetores: um com a posição, semelhante ao vetor da posição dos pontos de luz, e outro com a direção da luz, apenas com 3 posições que representam as respetivas coordenadas. Para além disto, é, ainda, guardado um valor de *cutoff*, que especifica o ângulo de corte do feixe de luz que varia entre 0º e 180º.

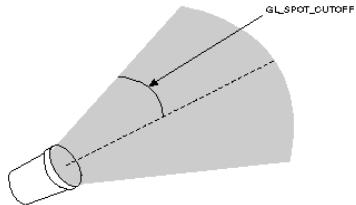


Figura 12: Foco de luz.

Para a inicialização das luzes, é utilizado o método “setup” da classe “Light”, herdado pelas subclasses - o modo de inicialização é igual para todos os tipos de luz. Antes de se efetuar o carregamento dos modelos, é percorrido o vetor das luzes e invocado esse método para cada uma. Portanto, este procedimento é efetuado apenas 1 vez, enquanto que a aplicação das luzes é efetuada a cada *frame*.

4.2 Materiais

A outra componente da iluminação diz respeito aos materiais de cada modelo importado. Os atributos de cada material são:

- **componente difusa** - cor que representa a reflexão da luz direta sobre a superfície do objeto;
- **componente ambiente** - cor geral da superfície quando iluminada indiretamente. É a cor que o objeto reflete quando não há uma fonte de luz direta nele;
- **componente especular** - cor que representa a reflexão da luz direta de uma fonte de luz especular, como uma luz brilhante. É responsável por criar os pontos de destaque brilhantes na superfície;
- **componente emissiva** - cor que representa a luz emitida pela superfície do objeto, ou seja, este irá emitir luz, independentemente de ser iluminado por uma fonte externa;
- **shininess** - valor que define o tamanho e a intensidade do ponto de destaque especular na superfície. Quanto maior o valor, mais brilhante e concentrado será o destaque.

Para representarmos os materiais, implementamos a classe “Color“. Esta classe guarda os *arrays* para as 4 componentes da luz e o valor de *shininess*. Cada modelo terá uma *color* a si associada, sendo que se não for explícita no ficheiro XML, os valores que deverão ser utilizados por *default*, de acordo com o enunciado, são:

```
componente difusa = (255, 255, 200)
componente ambiente = (50, 50, 50)
componente especular = (0, 0, 0)
componente emissiva = (0, 0, 0)
shininess = 0
```

A extensão “color“ previamente definida pelo grupo, poderia ser utilizada em caso de falta de indicação destas componentes no ficheiro XML de *input*. Contudo, uma vez que existem estes valores por predefinição, a anterior extensão foi descontinuada.

5 Aplicação no Sistema Solar

Para ilustrar o funcionamento correto das funcionalidades implementadas, foram adicionadas texturas e iluminação ao cenário principal do projeto, correspondente ao sistema solar.

Inicialmente, de modo a iluminar o sistema, foi adicionado um ponto de luz no Sol, ou seja, nas coordenadas (0, 0, 0).

Para adicionar dimensão, criou-se uma *skydome* que consiste num *background*, incluso numa esfera. A textura é projetada nas faces das esfera, criando assim a ilusão de um ambiente tridimensional distante. As texturas utilizadas para a *skydome*, o sol, a lua e todos os planetas foram transferidas dos *sites* Planetary Pixels Emporium e Solar System Scope, referenciados no enunciado do trabalho. Já para as texturas dos restantes satélites, dos asteroides e do cometa foram utilizadas algumas texturas dos recursos do *site* da NASA e outras texturas fictícias criadas e publicadas por utilizadores no DeviantArt.

6 *View Frustum Culling*

De modo a otimizar o desempenho da renderização, o grupo implementou o *view frustum culling*. O *view frustum* representa a região de espaço visível pela câmara a partir de uma determinada posição e orientação, ilustrado na seguinte figura.

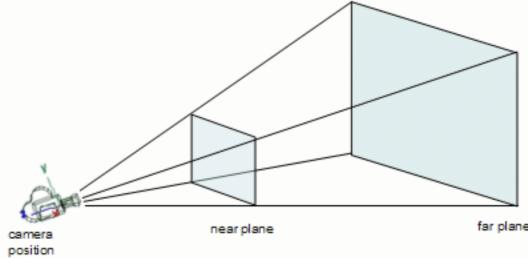


Figura 13: *View Frustum.*

Durante o processo, cada modelo é testado em relação ao *frustum* da câmara para determinar se está completamente dentro, fora ou parcialmente dentro do *frustum*. Aqueles que se encontram completamente fora são descartados, visto que não precisam de ser renderizados. Isto resulta num melhor desempenho geral do sistema, pois evita o desperdício de recursos na renderização de objetos que não serão visíveis no resultado final, sendo esta otimização mais notável quanto mais extenso e complexo for o cenário.

6.1 *Generator*

De modo a testar se os modelos viriam a ficar dentro ou fora do *frustum*, definimos *bounding volumes* para todos os modelos. Isto permite-nos realizar testes muito mais simples, uma vez que não necessitamos de verificar todos os pontos pertencentes ao modelo. De notar que por uma questão de simplificação, implementamos apenas o volume AABB, ou seja, caixa alinhada com os eixos.

Optamos por calcular estes volumes na aplicação *generator* aquando do cálculo dos vértices, triângulos, normais e coordenadas de textura. Desta forma, a estrutura do ficheiro de extensão “.3d” teve de ser alterada, possuindo, agora, uma primeira linha com o número de pontos do volume, seguida dos diversos pontos. Apesar de o volume AABB conter sempre 8 pontos, optamos por colocar uma primeira linha com o número de pontos, tendo em perspetiva, no futuro, utilizar volumes que não tenham um número fixo de pontos.

6.2 *Engine*

Relativamente ao *engine*, adicionamos a opção de ativar/desativar o *view frustum culling* no menu, com o objetivo de podermos comparar as 2 versões.

Inicialmente, a cada *frame*, calculamos as equações dos 6 planos que compõem o *frustum*. Para este cálculo optamos por recorrer à abordagem das matrizes em vez da geométrica, visto que simplifica todo o processo. Com isto, primeiramente, computamos a matriz que nos permite passar um ponto do “modelspace” para “clipspace”.

De seguida, extraímos todos os coeficientes dos 6 planos do *frustum* através da matriz calculada, que devem ser devidamente normalizados para posteriormente aplicarmos o método da distância de um ponto a um plano. Recorremos à classe “FrustumPlane” para guardar estes coeficientes. Nesta classe foi definido um método que nos permite calcular a distância de um ponto ao plano para, depois, averiguar se está ou não do lado da normal do plano.

Na renderização, já com os planos definidos, testamos para cada um destes se todos os pontos do volume estão no lado errado. Com base nesta avaliação, a renderização pode ou não ser feita. Um dos problemas levantados está relacionado com as transformações, uma vez que teremos de calcular os novos vértices do volume do modelo consoante estas transformações. Como na última fase já tinhamos definido a função que calcula a matriz resultante de todas as transformações, vimos a necessidade de apenas multiplicar esta matriz pelos vértices do volume.

Finalmente, para comparar a versão com e sem a utilização do *view frustum culling*, efetuamos uma contagem de triângulos para apresentar na barra da aplicação.



(a) Resultado obtido sem *view frustum culling*. (b) Resultado obtido com *view frustum culling*.

Figura 14: Comparação das versões com/sem *view frustum culling*.

Das figuras acima apresentadas, podemos observar que sem *view frustum culling* o *engine* renderizou 514300 triângulos. Pelo contrário, com o *view frustum culling* ativado desenhou 4900 triângulos. Conseguimos observar uma grande diferença entre o número de triângulos renderizados, como esperado.

7 Resultados Finais

Nesta secção encontram-se os resultados finais obtidos na *demo scene* correspondente ao sistema solar e, ainda, a comparação entre os resultados obtidos e os resultados esperados em cada teste disponibilizado no enunciado do projeto.

7.1 Sistema Solar

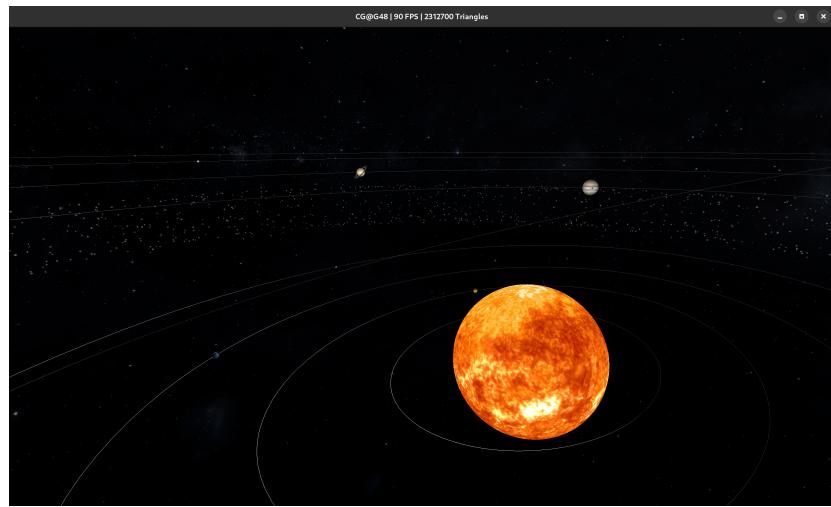


Figura 15: Visão geral do cenário.

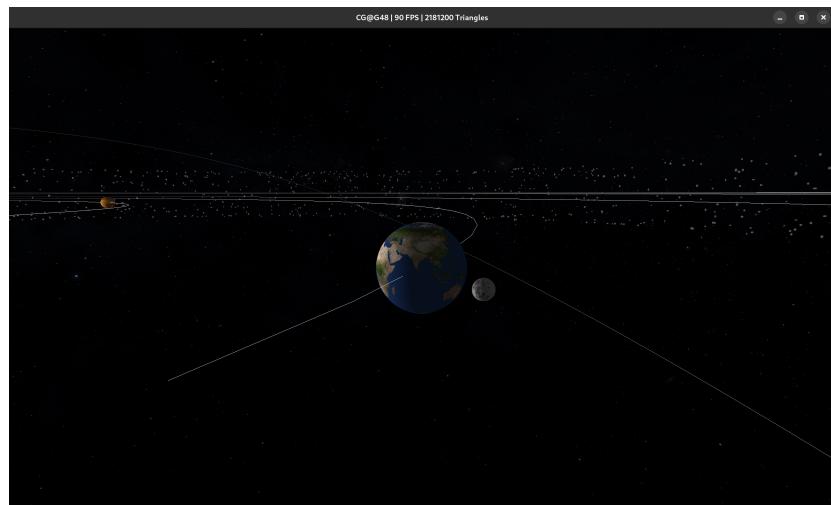


Figura 16: Terra e Lua.



Figura 17: Saturno e o seu anel.



Figura 18: Cometa.

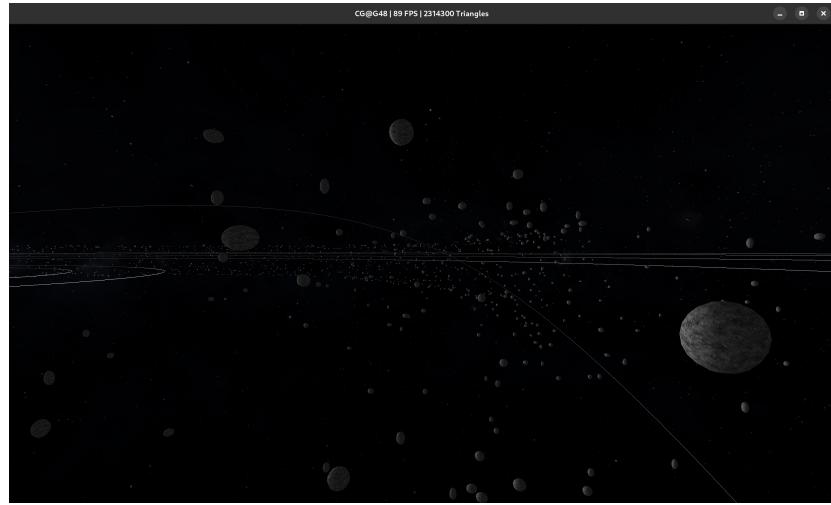


Figura 19: Cintura de asteroides.

7.2 Testes

Teste 1

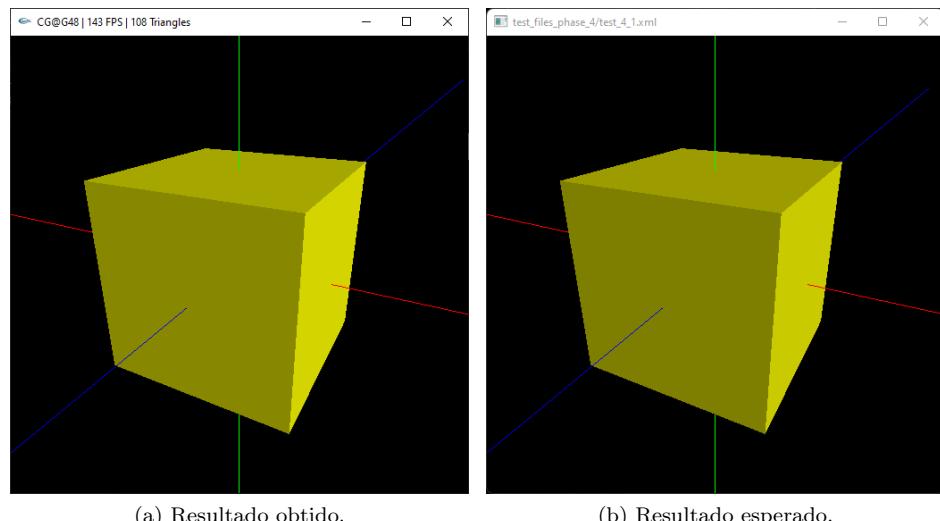


Figura 20: Resultados do Teste 1.

Teste 2

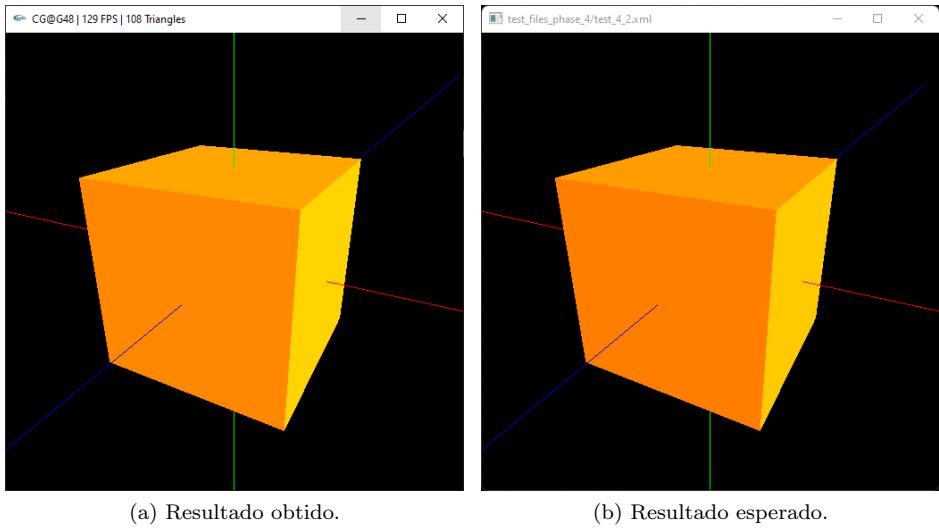


Figura 21: Resultados do Teste 2.

Teste 3

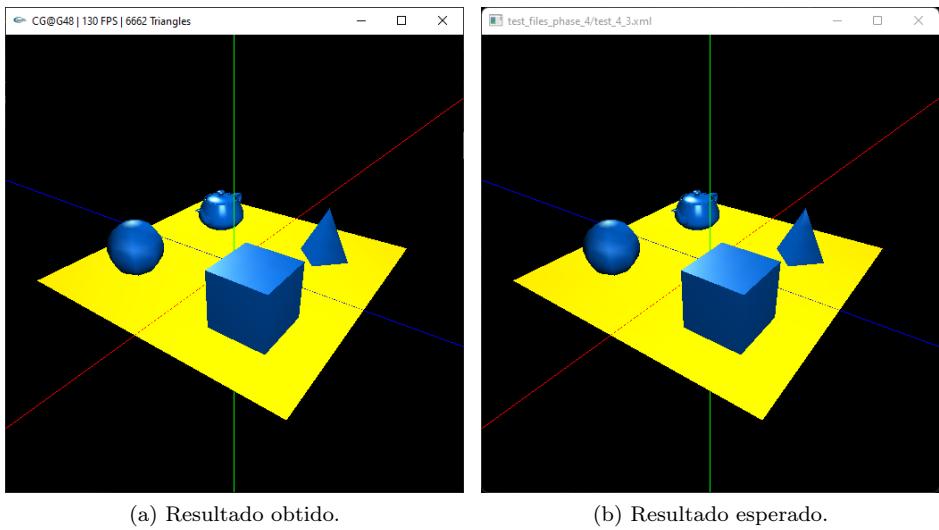


Figura 22: Resultados do Teste 3.

Teste 4

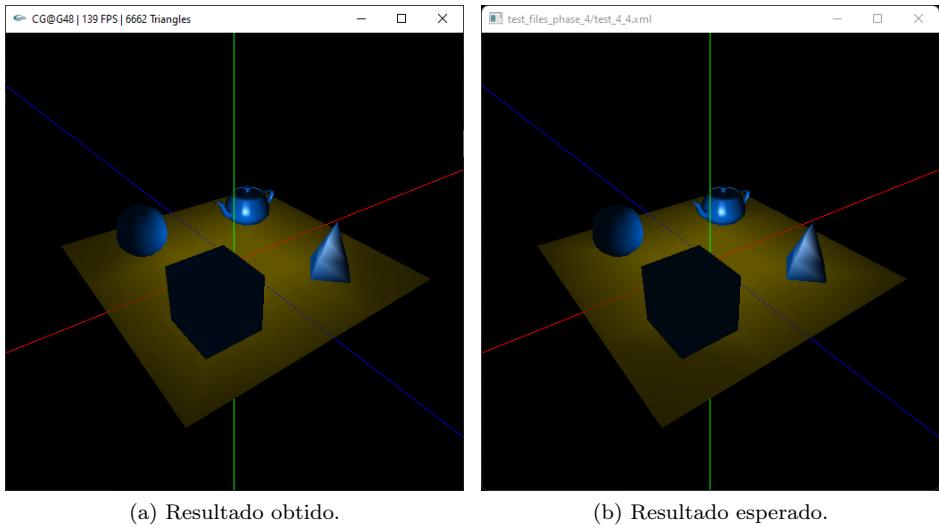


Figura 23: Resultados do Teste 4.

Teste 5

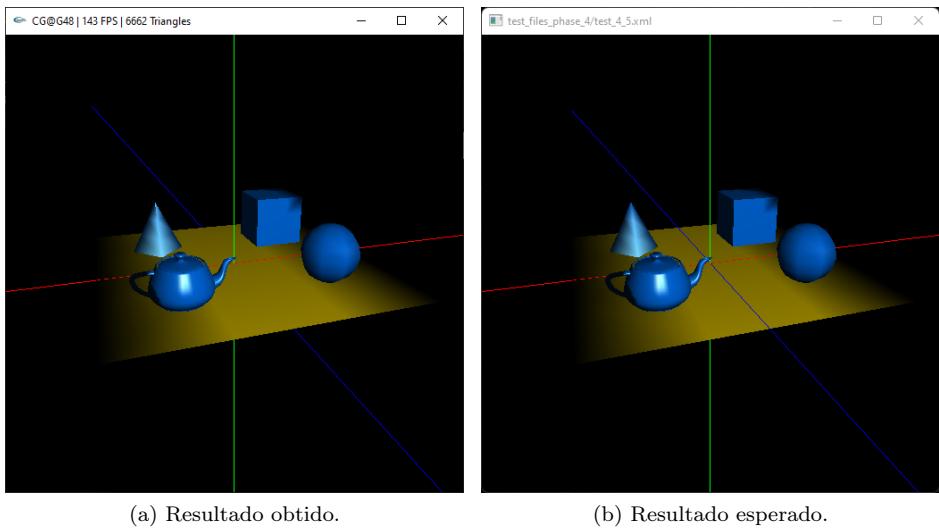


Figura 24: Resultados do Teste 5.

Teste 6

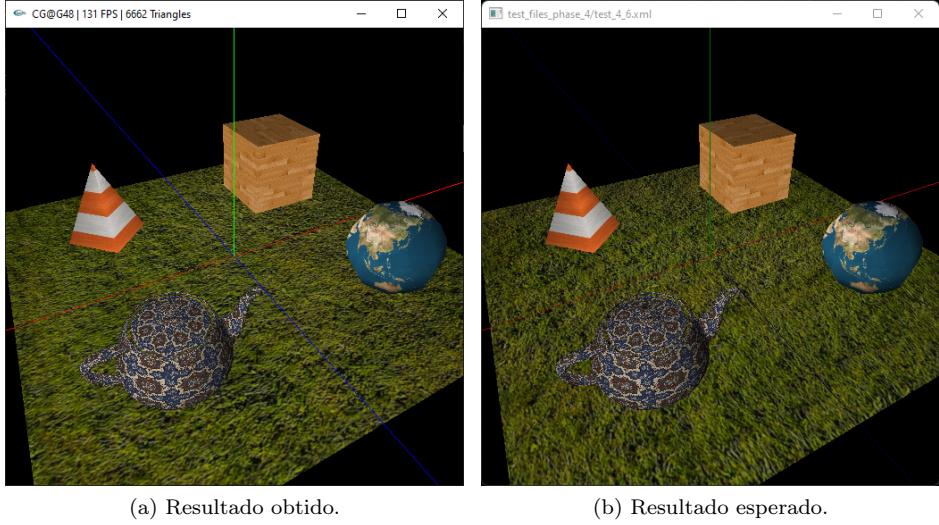


Figura 25: Resultados do Teste 6.

8 Conclusão

Esta quarta e última fase do projeto permitiu-nos concluir os últimos requisitos para a finalização do nosso projeto. Acreditamos que, com a adição de texturas e iluminação, o sistema solar desenvolvido tornou-se muito mais realista e apelativo ao olho humano, pelo que estamos bastante satisfeitos com o resultado final.

Fazendo uma retrospectiva do trabalho realizado em todas as fases, concluímos a implementação de todas as funcionalidades propostas no enunciado do trabalho prático. Ao longo de cada fase, preocupamo-nos sempre com pequenos detalhes e tivemos sempre o trabalho realizado anteriormente em conta, realizando algumas melhorias ao que foi desenvolvido, como foi o caso do modo explorador e também da cintura de asteroides. Para além das funcionalidades requeridas, procuramos ir mais além e desenvolvemos um conjunto de funcionalidades extra que ajudaram a impulsionar o nosso conhecimento e a tornar o nosso projeto mais apelativo.

Consideramos que algo que ficou por fazer neste projeto foi estender as funcionalidades implementadas nas 2 últimas fases a novos cenários. Na segunda fase, acabamos por criar uma *demo scene* extra que se manteve estática até a presente fase. Durante a realização do *engine*, preocupamo-nos sempre em mantê-lo numa aplicação abstrata, funcional para qualquer tipo de cenário e

não só para o cenário do sistema solar, pelo que esta tarefa que ficou por fazer poderia ser rapidamente efetuada, alterando apenas o ficheiro XML. De notar, também, que as funcionalidades extra, tais como os modos de câmara, o teletransporte associado ao modo explorador, assim como o *view frustum culling*, foram desenvolvidos para funcionar devidamente para qualquer cenário em questão. Ainda relacionado ao *frustum*, poderíamos também ter desenvolvido mais *bounding volumes*, como é o caso da esfera, e apresentar uma forma mais apelativa de visualizar o *frustum*.

Assim, consideramos que, após a realização, na integridade, das quatro fases deste trabalho prático, fomos capazes de obter e aprofundar vários conceitos relativos a Computação Gráfica, nomeadamente a criação, manipulação e visualização de figuras geométricas, os conceitos matemáticos associados e as técnicas que deveríamos utilizar. Desenvolvemos, ainda, bastantes competências práticas ao trabalhar com ferramentas como o OpenGL e GLUT, pelo que nos sentimos preparados para a sua utilização em projetos futuros.