Universidade do Minho

Licenciatura em Engenharia Informática

Computação Gráfica

Grupo 30

Trabalho Prático - **Fase 4**

Normais e Coordenadas de Textura

Joana Alves (A93290)

Maria Cunha (A93264)

Vicente Moreira (A93296)

Junho 2023

Conteúdo

1 Introdução

Este relatório foi desenvolvido no âmbito da quarta e última fase do trabalho prático da unidade curricular de Computação Gráfica da Licenciatura em Engenharia Informática, tendo como objetivo a evolução das aplicações criadas anteriormente, a partir da integração de materiais e texturas nos modelos previamente construídos, assim como, a implementação de luz especular, difusa, ambiente e brilho.

Este documento detalha as várias modificações efetuadas para alcançar os objetivos, assim como as decisões tomadas pelo grupo. Por fim, foram efetuados vários testes para verificar o bom funcionamento destas transformações.

2 Generator

A aplicação *generator* é responsável por gerar os ficheiros *.3d* com a informação acerca das primitivas. Visto que nesta fase o objetivo inclui a adição de suporte de iluminação assim como a aplicação de texturas nestas primitivas, foi necessário efetuar modificações nesta, de forma a gerar esta nova informação.

2.1 PointHolder e Formato *.3d*

Como referido anteriormente, foi necessário acrescentar o suporte não só de vetores normais aos pontos das primitivas como também coordenadas de duas dimensões para a aplicação de texturas, logo, foi adicionado à classe *PointHolder* dois novos *arrays* de *floats* com esta informação, assim como a mudança do método ***addPoint***, de forma a que este receba o vetor normal e as coordenadas de textura do ponto.

Depois desta informação armazenada, esta é escrita de forma semelhante ao formato anterior, começando por ser escrito um número inteiro que indica o número de *floats* de coordenadas presentes na primitiva. De seguida são armazenados todos os *floats* correspondentes aos pontos da primitiva, seguidos dos seus vetores normais e por último as coordenadas das texturas, tal como representado no seguinte diagrama:

numFloats : Int	points : float * numFloats	normals : float * numFloats	textCoords : float * (numFloats * 2 / 3)

2.2 Cálculos das Normais

Nesta secção vamos apresentar o raciocínio por detrás dos cálculos das normais das várias primitivas, notando que algumas primitivas não foram evoluídas.

2.2.1 Plano

Um plano é uma superfície plana e, por isso, as normais são constantes em todos os pontos. A normal de um plano é perpendicular à superfície e aponta na mesma direção em todos os pontos. Sendo assim, não há propriamente uma necessidade de calcular as normais, uma vez que os seus valores são estáticos em todo o plano, sendo adicionados sempre os mesmos valores em todos os pontos.

Assim, apresentamos os exemplos dos três planos, que, dado o plano de eixos em que se encontram e a seu sentido (positiva ou negativa), introduzimos os valores das normais correspondentes:

- **Plano XZ** (direção eixo **Y**): normal = facing ? (0,1,0) : (0,-1,0)
- **Plano XY** (direção eixo **Z**): normal = facing ? (0,0,1) : (0,0,-1)
- **Plano YZ** (direção eixo **X**): normal = facing ? (1,0,0) : (-1,0,0)

2.2.2 Box

A primitiva Box não é nada mais nada menos do que a aplicação sucessiva da primitiva **Plano** nos vários eixos correspondentes, assim, o cálculo das normais é feito por essa mesma primitiva, sendo que já foram explicados na secção anterior.

2.2.3 Cilindro

O cálculo dos pontos da primitiva Cilindro, tal como referido nas fases anteriores, é feito por "fatias"do mesmo, incluindo um triângulo do topo, dois triângulos do lado e, por fim, o triângulo da base. Sendo assim, para cada uma destas secções apresentamos o raciocínio efetuado:

- **Triângulos Topo/Base:** Para os triângulos de ambas as bases do cilindro, o raciocínio foi parecido com o da primitiva Plano, uma vez que a direção das bases do cilindro são sempre perpendiculares ao plano XZ, apenas alterando o seu "sentido", isto é, no sentido positivo ou negativo do eixo Y. Desta forma, para o **topo** a normal calculada é **(0,1,0)** e para a **base** é **(0,-1,0)**
- **Triângulos Lado:** Para as normais dos dois triângulos do lado da "fatia", sabíamos, que, por exemplo, a coordenada Y de todos as normais seria 0, uma vez que a direção é paralela ao eixo XZ. Assim, apenas era necessário calcular as coordenadas X e Z dos vetores. Estas coordenadas, dependendo do ponto e coordenada, são calculadas usando o ângulo de abertura ($360^{\underline{o}}$ / *slices*). Por exemplo, apresentamos o cálculo das coordenadas do vetor normal do ponto "atual"de uma dada iteração:

```
float actualNormalX = sin(angle * (float)i);
float actualNormalY = 0.0f;
float actualNormalZ = cos(angle * (float)i);
```

2.2.4 Esfera

Na primitiva Esfera, tal como referido nas fases anteriores, o cálculo dos pontos da mesma é feito por "secções"da esfera, isto é, a esfera é dividida em *slices* e cada *slice* é dividida em *stacks*. Estas divisões dão origem a secções constituídas por quatro pontos, onde, para cada um é calculado o vetor normal. Assim, apresentamos o cálculo das coordenadas do vetor normal para o primeiro ponto da secção:

```
float n1X = cos(subBeta * actStack - delta ) * sin(subAlpha * actSlice);
float n1Y = sin(subBeta * actStack - delta);
float n1Z = cos(subBeta * actStack - delta) * cos(subAlpha * actSlice);
```

2.2.5 Cone

No caso da primitiva Cone, para o cálculo das normais na base, tal como todos os cálculos explicados nas secções anteriores, como esta é paralela ao eixo XZ com orientação negativa, isto é, no sentido negativo do eixo Y, a normal é fixa em todos os pontos da mesma, em concreto **(0,-1,0)**.

Para o cálculo do vetor normal na restante superfície do cone, esta foi dividida nas suas componentes X, Y e Z, sendo que as componentes da normal X e Z seguem o mesmo cálculo das primitivas do cilindro. Já a componente Y da normal do cone foi calculada através do ângulo de "inclinação"do cone, tal como a figura demonstra:

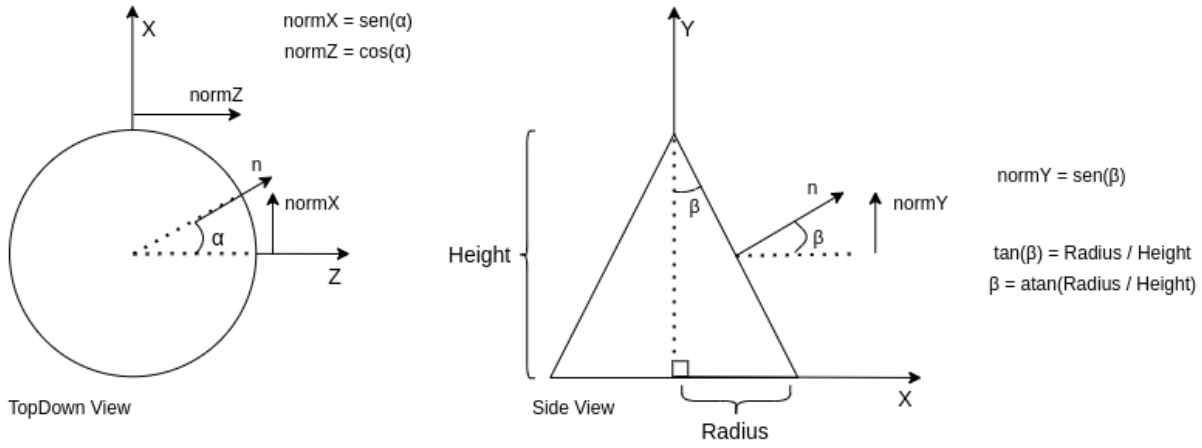


Figura 1: Cálculo da normal do cone

2.2.6 Cintura de Asteroides

Esta primitiva utiliza como primitivas auxiliares o ***Torus*** e a **Esfera**, fazendo uso do *Torus* para atingir a forma de "anel"e a da Esfera para construção de asteroides (esferas deformadas) nos diversos pontos do *Torus*. Desta forma, as coordenadas dos vetores normais são da responsabilidade da primitiva Esfera, tendo sido o seu raciocínio explicado na secção anterior respetiva.

2.3 Cálculo de Coordenadas de Textura

2.3.1 Plano

A implementação de texturas no modelo de um plano é a mais simples. Para este modelo, a estratégia escolhida é o mapeamento da totalidade da textura no plano, sendo que estes possuem a mesma forma. Logo, o único desafio encontrado é o cálculo das coordenadas das texturas nas divisões deste, sendo que este é efetuado através do cálculo "1 / nº Divisões", assim como a próxima figura demonstra

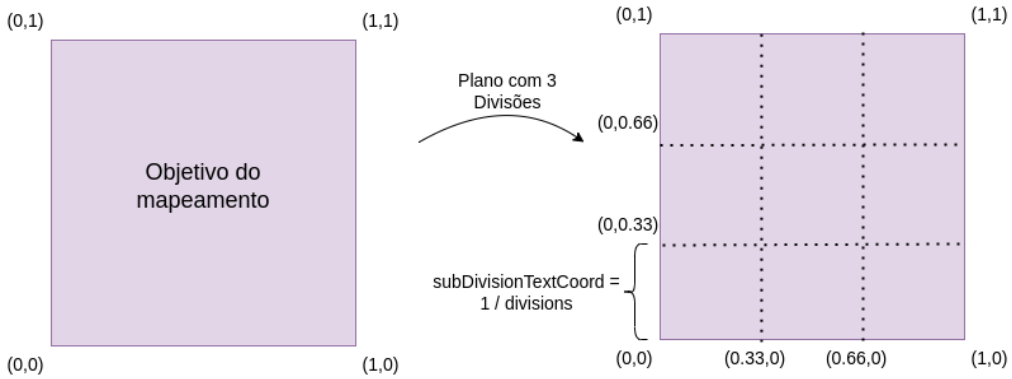


Figura 2: Estratégia de Cálculo de Texturas no Plano

2.3.2 Box

Para a primitiva Box, tal como no cálculo das normais, este cálculo é feito para cada plano correspondente pela primitiva **Plano**, tendo o seu raciocínio sido explicado na secção anterior.

2.3.3 Cilindro

No Cilindro, visto que há imensos desafios no mapeamento contínuo de uma textura retangular num círculo, decidimos que para a base do topo teríamos (0.5, 1) como coordenadas de textura e na base seriam (0.5,0), de forma a ter uma textura uniforme. No que toca aos dois triângulos do lado da "fatia", começamos por calcular o número de divisões a implementar para, de seguida, em cada iteração calcular as coordenadas de textura correspondentes. Desta forma, apresentamos a figura ilustrativa que mostra, para uma dada fatia, que coordenadas de textura são calculadas:

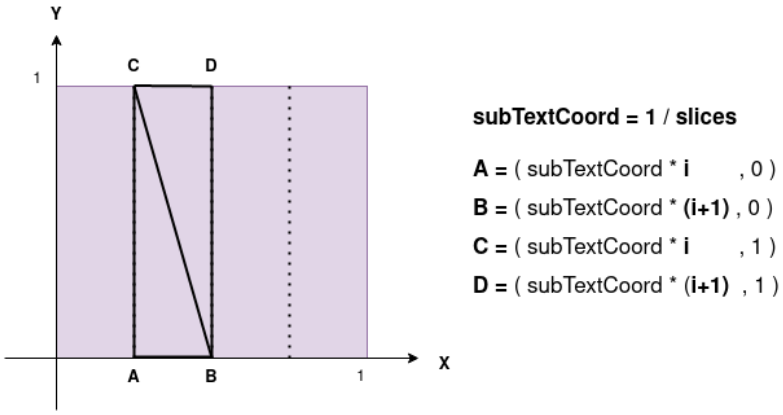


Figura 3: Estratégia de Cálculo de Texturas no Cilindro

2.3.4 Esfera

Na primitiva Esfera, tal como referido nas fases e secções anteriores, o cálculos dos pontos é feito através da divisão sucessiva da esfera em *slices* e *stacks*, obtendo, por fim, uma secção constituída por quatro pontos. Desta forma, temos de começar por calcular as divisões da textura, dividindo o eixo X pelo número de *slices* e o eixo Y pelo número de *stacks*. Assim, apresentamos uma figura dos cálculo das coordenadas de textura, para uma dada iteração:

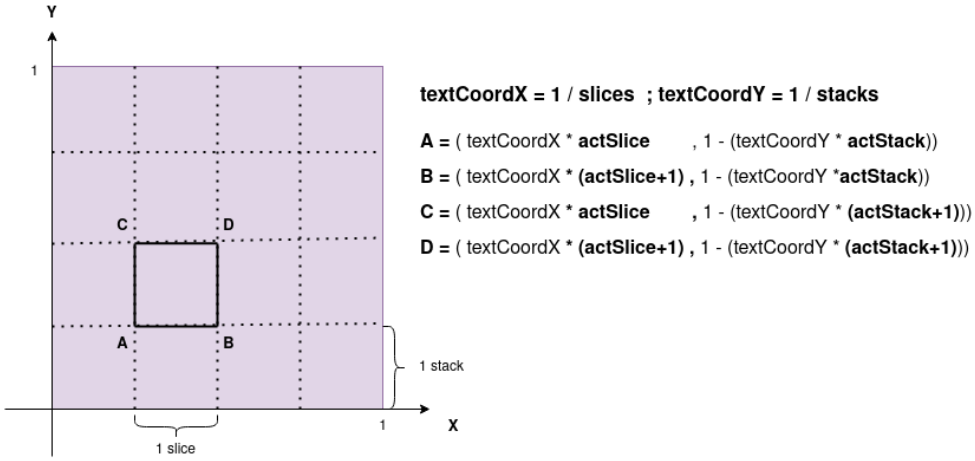


Figura 4: Estratégia de Cálculo de Texturas na Esfera

2.3.5 Cone

Para a primitiva Cone, tal como no Cilindro, como existem desafios no mapeamento de uma textura retangular num círculo, decidimos que para a base do cone as coordenadas de textura seriam (0.5,0), de forma a ter uma textura uniforme que corresponde à base da textura.

Para a restante superfície do cone, seguimos uma lógica semelhante utilizada na primitiva Esfera, visto que ambos utilizam o mesmo mecanismo de divisão da superfície por *slices* e *stacks*.

2.3.6 Cintura de Asteroides

Esta primitiva, tal como referido na secção do cálculo dos vetores normais, recorre a duas primitivas auxiliares, sendo que apenas a **Esfera** é responsável por efetivamente calcular tanto vetores normais como coordenadas de textura. Assim, não vamos apresentar o seu cálculo uma vez que foi explicado o raciocínio da mesma na secção anterior.

2.3.7 *Bezier Patches*

Para a aplicação de texturas em modelos que recorrem a *Bezier patches*, visto que é não é prático aplicar uma textura de forma uniforme no modelo todo, decidimos fazer a aplicação da textura em cada *patch* individual, havendo assim repetições da textura ao longo do modelo. Para isto, foi utilizada uma lógica semelhante à primitiva do Plano, onde a variável *tesselation* define o tamanho da subdivisões do *patch*, podendo até ser utilizado o valor de *step* diretamente para o cálculo das coordenadas da textura.

2.4 Primitivas Não Implementadas

No desenvolvimento desta última fase do projeto, acabamos por não implementar o cálculos dos vetores normais e das coordenadas de textura na primitiva *Torus*, sendo que também não foi possível calcular os vetores normais para os *Bezier Patches*.

No caso da primitiva *Torus*, reparamos que os cálculos dos pontos estariam mal feitos, uma vez que quando tentamos implementar tanto os vetores normais como as coordenadas de textura na mesma, obtivemos um resultado errado e distorcido. Devido à falta de tempo para averiguar em que parte dos cálculos nos teríamos equivocado, decidimos não evoluir a primitiva *Torus*.

No caso dos *Bezier Patches*, o grupo não implementou o seu cálculo dos vetores normais, maioritariamente por falta de tempo para desenvolver os mesmos.

3 Engine

Nesta secção vamos apresentar as diversas evoluções da aplicação *Engine*, responsável pela leitura de um ficheiro de configuração *XML* que descreve os vários aspetos de uma cena, montar esta e apresentá-la ao utilizador através das diretivas do *openGL*.

Desta forma, nesta última fase do trabalho prático, tínhamos como objetivo expandir mais uma vez as funcionalidades do mesmo ao permitir a implementação de luzes e aplicação de materiais e texturas nos modelos. Estas adições provocaram alterações na estrutura da aplicação, tal como vamos detalhar nas seguintes secções.

3.1 Class Light

Para a implementação de luzes no *engine*, foi necessário a criação de uma nova classe ***Light*** responsável por armazenar toda a informação relativa a uma fonte de luz, assim como tratar da sua renderização.

Esta começa por ler as suas propriedades a partir do nodo XML fornecido, como o seu tipo e as definições. Depois de carregada, esta é desenhada a partir da primitiva **glLightfv**, assim como a possibilidade de desenho da sua posição, caso o *debug mode* (a ser explicado no capítulo de novas *features*) esteja acionado.

3.2 Class Color

Outra classe desenvolvida para a extensão e implementação dos materiais nos modelos é a classe **Color**, sendo que esta lê o ficheiro de configuração da cena e armazena todas as variáveis relativas ao material do modelo, assim como define os materiais *default* caso estes não sejam explícitos.

3.3 Models

Sendo esta umas das classes com maiores alterações no contexto do *Engine*, foi necessário adicionar 2 novos *buffers* para os vetores normais e as coordenadas das texturas, assim como implementar a sua leitura a partir dos novos ficheiros *.3d*. Para além destes dois novos *buffers* e seu *buffering* para VBO's, também foi necessário adicionar a leitura do ficheiro de textura a ser utilizado.

Para a adição do material a ser utilizado, foi adicionada a classe Color ao modelo, assim como a correta aplicação deste material e informação para o desenho do modelo.

3.4 World

Para a implementação de luzes, foi necessário adicionar uma nova variável que armazena todas as luzes presentes na cena. Esta foi denominada **lights** e, na inicialização da cena, conforme os vários objetos da classe Light são criados, esta tem como responsabilidade manter o número/id das luzes lidas, através do uso do cálculo com o valor estático **GL_LIGHT0** + numLight. Assim, graças a este Id, quando as luzes forem desenhadas, estas poderão se identificar na primitiva **glLightfv**.

3.5 Adição de *Features*

3.5.1 *Debug Mode*

Com o objetivo de estender a funcionalidade de *debugging* inserida na fase anterior, decidimos adicionar um modo de "Debug", que pode ser ligada/desligada através da tecla F12. Quando ligado, a aplicação de *Engine* irá apresentar informação como os eixos de origem, as várias curvas de *CatmullRom* a serem aplicadas e a posição das luzes no cenário, assim como a figura demonstra:

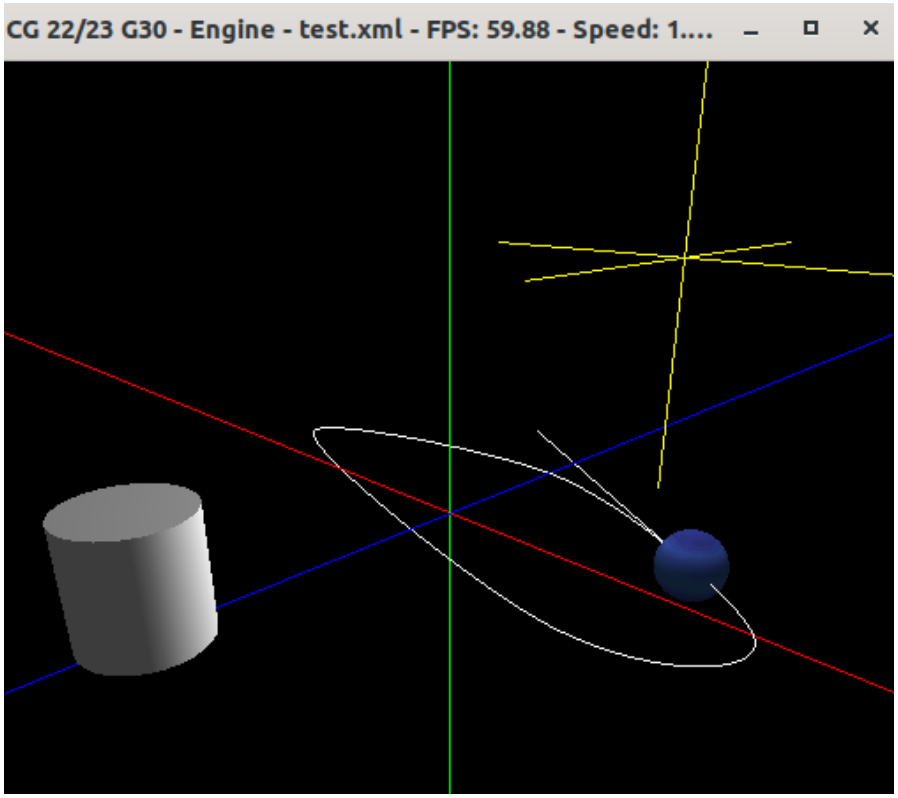


Figura 5: Modo Debug

3.5.2 Controlo de Tempo

Com a introdução de animações na fase anterior, foi introduzida uma variável **elapsedTime**, usada para calcular o tempo entre cada *frame* e assim processar o "avanço"das animações nas suas respetivas transformações. Através da manipulação desta variável com base num "fator", é possível introduzir um mecanismo de "reprodução"acelerada ou lenta, conforme o utilizador desejar. Este cálculo é feito a partir da fórmula:

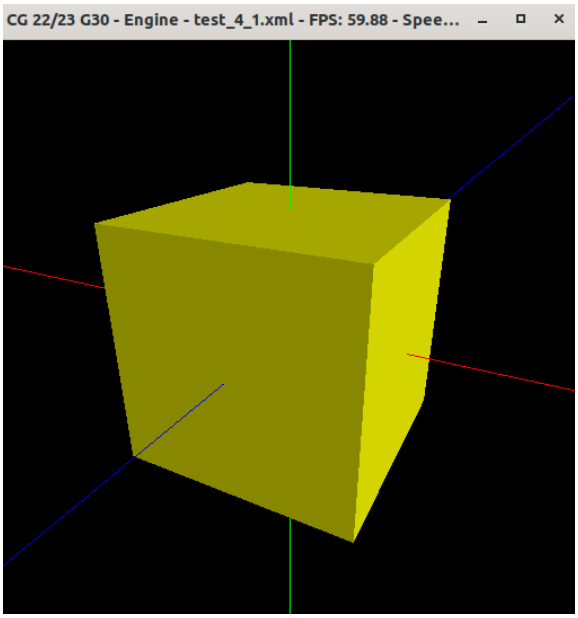
$$\textbf{elapsedTime} = elapsedTime * timeFact^2, (-10 <= timeFact <= 6)$$

4 Testes e Resultados

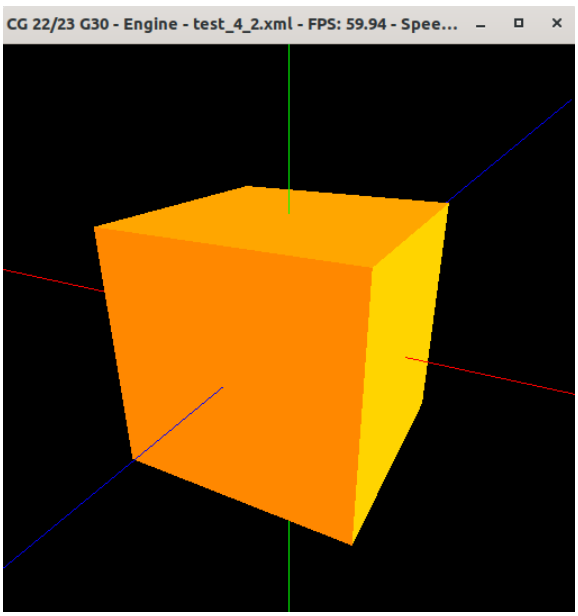
Nesta secção vamos apresentar os testes executados pelo grupo de modo a conseguir avaliar o funcionamento de ambas as aplicações quanto à sua correção e exatidão de resultados.

4.1 Testes Docentes

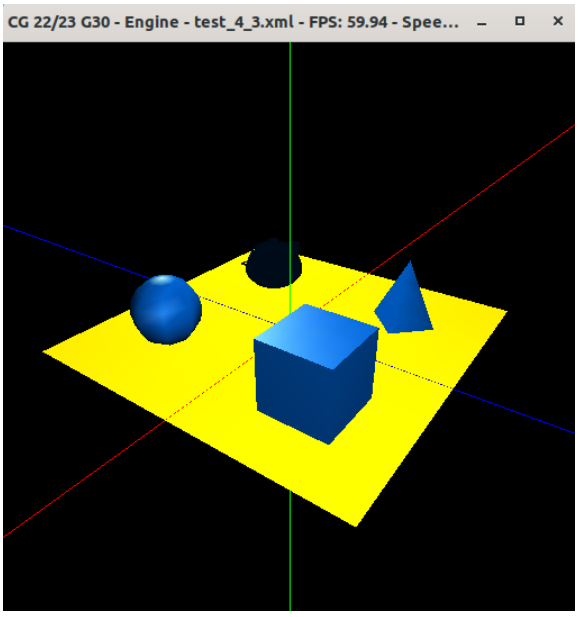
Alguns dos testes implementados foram cedidos pelos docentes de modo a verificar o comportamento correto das aplicações. Assim, apresentamos os resultados obtidos em cada um dos testes, sendo de notar que parte destes foram de encontro ao esperado, tirando à exceção o modelo do *Teapot*, que não possuí uma iluminação correta, devido à falta dos vetores normais.



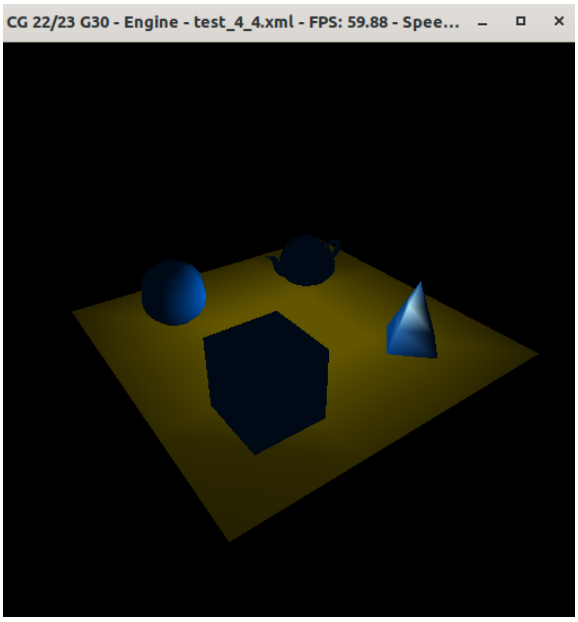
test_4_1



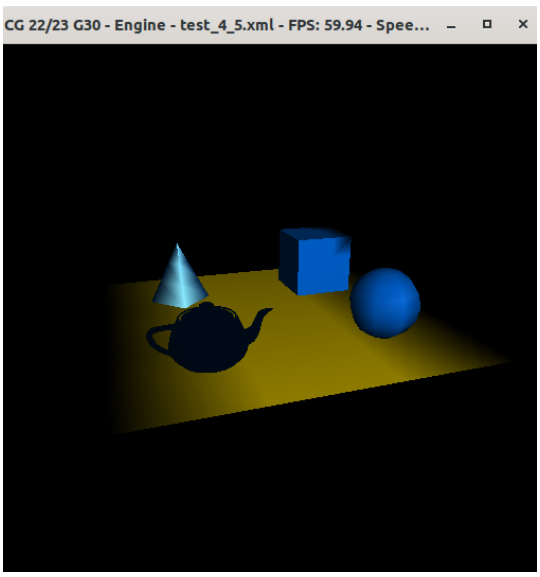
test_4_2



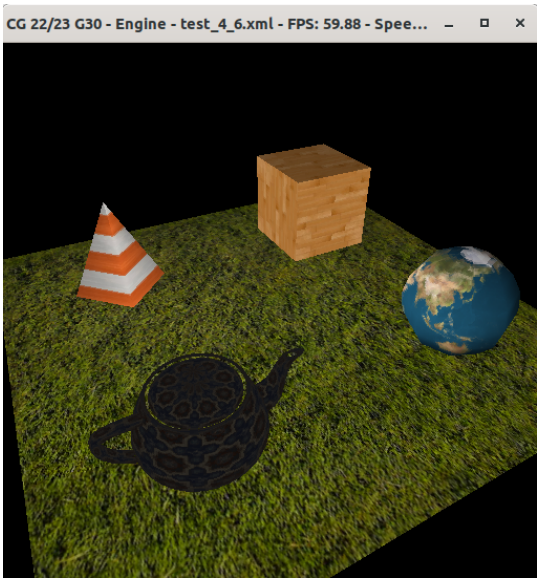
test_4_3



test_4_4



test_4_5



test_4_6

4.2 Demo Sistema Solar

Para a demonstração final do sistema Solar, aplicamos texturas a todos os planetas, assim como ajustar as rotações dos planetas e, por fim adicionar uma fonte de luz "point"no centro do sol. Apresentamos 3 imagens com a demonstração final:

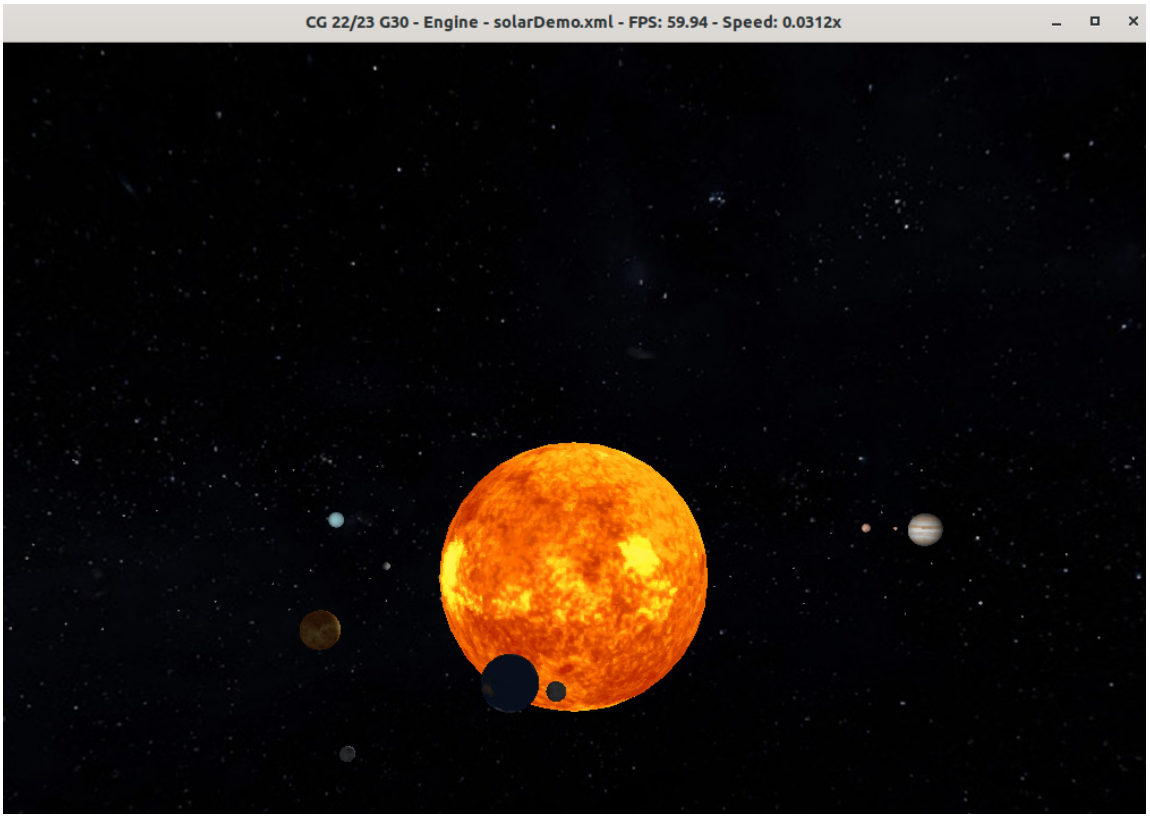


Figura 6: *Demo* Sistema Solar 1

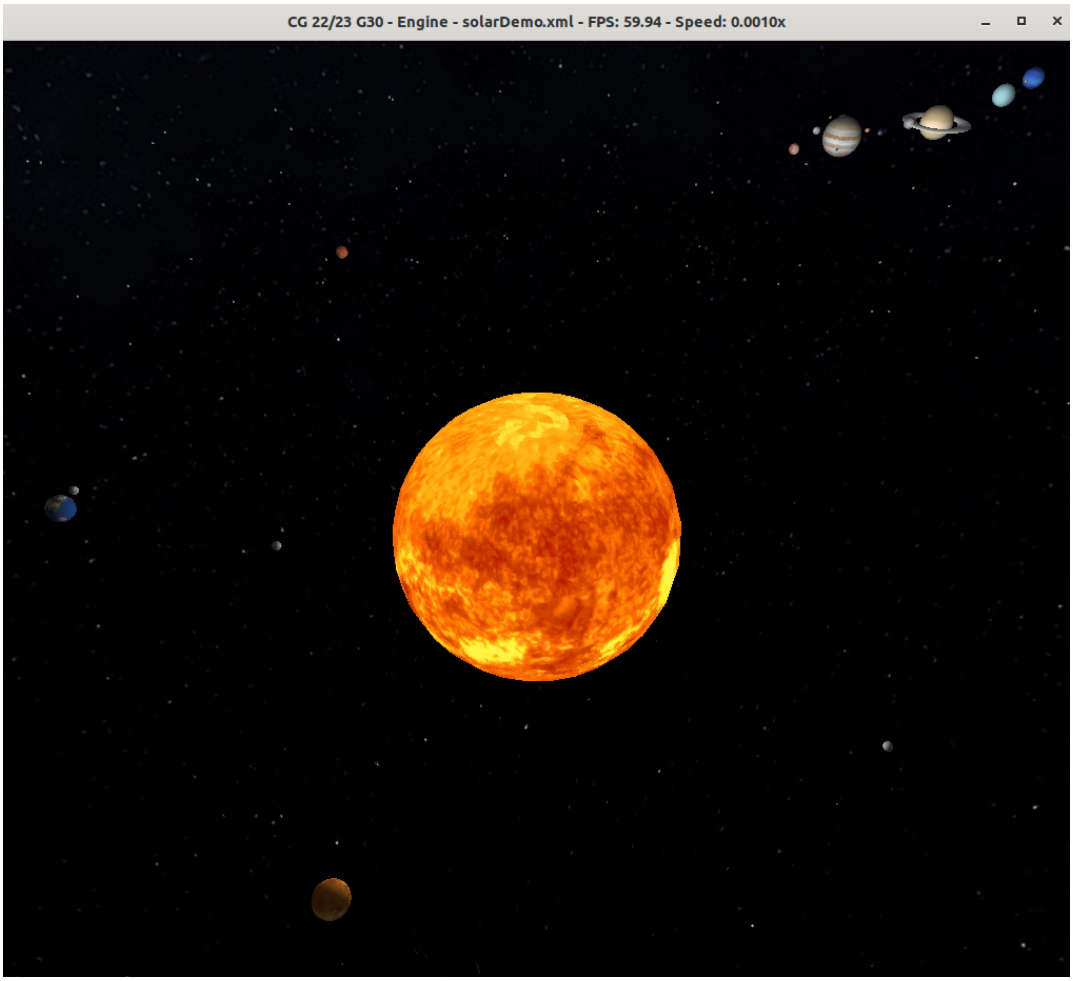


Figura 7: *Demo* Sistema Solar 2

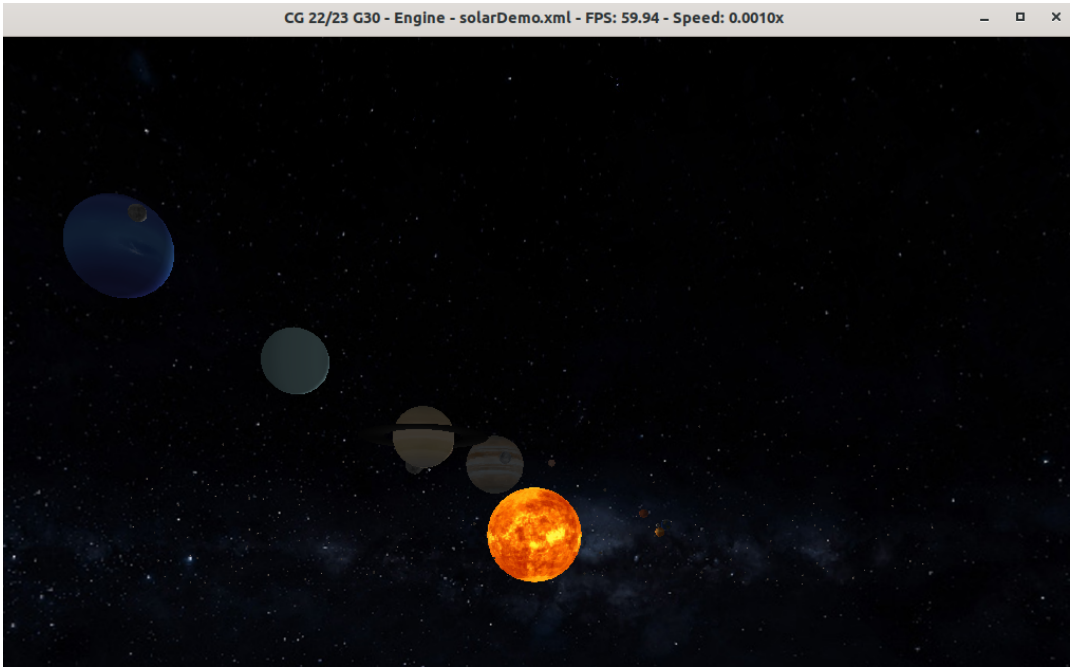


Figura 8: *Demo* Sistema Solar 3

5 Conclusão

Com a conclusão desta quarta e última fase do projeto, o grupo encontra-se razoavelmente satisfeito com o trabalho desenvolvido, tendo sido alcançados a maioria dos objetivos estabelecidos quer pelos docentes, como pelo próprio grupo.

Em suma, aprofundamos o conhecimento adicional lecionado nas aulas quer teóricas quer práticas após a segunda terceira do projeto, que nos ajudaram a perceber e a implementar variados tipos de luzes, (ambiente, especular, difusa e brilho), bem como a adição de texturas nos modelos. Em suma, aprofundamos, também, o conhecimento em relação à linguagem de programação *C++*, às várias primitivas gráficas a implementar, assim como a utilização da biblioteca *Glut* e da API *openGL*.