Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Computação Gráfica

Ano Letivo de 2023/2024

Trabalho Prático
Fase 4 - Normais e Coordenadas de Textura

António Filipe Castro Silva(a100533) Diogo Rafael dos Santos Barros(a100600)
Duarte Machado Leitão(a100550) Pedro Emanuel Organista Silva(a100745)

26 de maio de 2024

CG

Índice

Lista de Figuras

1 Introdução

Este relatório foi realizado no âmbito da 4.ª fase do Trabalho Prático da Unidade Curricular de Computação Gráfica que representa a última fase deste Trabalho Prático onde nos era pedido o tratamento de normais e coordenadas de textura.

Em relação à fase anterior, tivemos de alterar o *Parsing* dos XML uma última vez de forma a ler a informação sobre as cores e as luzes, criando novas estruturas, e fazer uma pequena alteração nos ficheiros ".3d" por causa do cálculo das normais. Também reparamos num problema dos *VBOs* que alteramos e agora eles realmente funcionam de forma perfeita.

Conseguimos calcular as normais para todas as diferentes figuras, mas infelizmente não conseguimos implementar texturas, logo não calculamos as suas coordenadas. Conseguimos contudo meter as luzes e as cores a funcionar com boa qualidade.

Por fim, realizamos os testes dos professores, exceto o das texturas e fizemos um sistema solar com cores, pois não conseguíamos utilizar as texturas neles.

2 Alterações relativamente à fase anterior

2.1 Estruturas de Dados

Em primeiro lugar, depois de analisarmos os XML desta fase, reparamos logo que teríamos de manusear luzes e cores, sendo que as luzes são aplicadas à cena toda e as cores a uma figura em específico, logo vamos guardar as luzes na estrutura do Leitor, enquanto que as cores vamos guardar nas figuras, como vamos mostrar depois.

```
struct leitor {
    float posicao[3];
    float lookAt[3];
    float up[3];
    float projection[3];
    std::vector<Lights> lights;
    Group group;
};
```

Figura 2.1: Struct Leitor alterada

Esta foi a estrutura final do Leitor que ficamos adicionando um novo vetor de Lights. Para extrair os dados tivemos de fazer duas novas funções: **extrair_lights** e **extrair_colors**. O **extrair_lights** trata de extrair os dados das luzes sendo chamado no **extrair_XML** pois as luzes são configuradas antes dos **groups**. O **extrair_colors** trata de extrair as cores de uma figura e é chamada no **extrair_grupo** para cada model existente naquele grupo, pois cada model representa uma figura e cada figura pode ter uma cor diferente.

2.1.1 Luzes

Para as luzes fizemos um Enum e uma estrutura. O Enum **LightType** guarda o tipo de luz: **Directional**, **Point** ou **Spot**. A estrutura **lights** guarda este tipo de luz, as coordenadas da posição da luz, as coordenadas da direção para onde a luz vai e o cutoff.

```
enum class LightType{
    DIRECTIONAL,
    POINT,
    SPOT
};

struct lights{
    LightType type;
    float posx, posy, posz, dirx, diry, dirz, cutoff;
};
```

Figura 2.2: Structs das Luzes

2.1.2 Cores

Para as cores já criamos duas estruturas. A estrutura **rgb** que vai guardar os valores de **Red(R)**, **Green(G)** e **Blue(B)** de uma cor. A estrutura **colors** vai guardar uma estrutura **rgb** para cada tipo diferente de forma que esta cor em específico vai tratar da luz: **diffuse, ambient, specular** e **emissive**. Além disso também vai guardar um inteiro que representa a **shininess** da cor.

```
struct rgb{
    int R;
    int G;
    int B;
};

struct colors{
    RGB diffuse;
    RGB ambient;
    RGB specular;
    RGB emissive;
    int shininess;
};
```

Figura 2.3: Structs das Cores

2.2 Mudança nas figuras e nos ficheiros .3d

Em termos da estrutura das figuras adicionamos-lhe o seu **tipo** (plano, esfera, cone, etc), o seu centro absoluto em forma de Ponto, uma estrutura **Color** e até chegamos a guardar o *path* da textura (apesar de este acabar por nunca ser utilizado, mas significa que fizemos parse de tudo o que vinha no XML).

Este centro absoluto foi calculado e adicionado à figura de forma a otimizar a forma como as transformações funcionavam nesta fase final do trabalho.

Passamos a ter o tipo da figura na estrutura em si, para ajudar no cálculo posterior das normais. Além disso, passamos a colocar nos ficheiros .3d o tipo da figura no início do ficheiro para facilitar a definição do tipo da figura na estrutura quando fazemos a leitura destes ficheiros.

```
struct figura
{
    std::string type;
    bool curva;
    std::list<Ponto> pontosControl;
    std::list<Ponto> pontos;
    Ponto centroAbs;
    Color color;
    const char* texture;
};
```

Figura 2.4: Structs das Figuras alterada

2.3 VBOs melhorados

Ao tratarmos dos VBOs nesta fase para tratar das luzes e das cores reparamos que afinal não os estávamos a fazer completamente correto a sua implementação. Após fazermos uma pequena correção, realmente conseguimos reparar na sua potencialidade principalmente nesta fase.

Nos testes das primeiras fases não há quase alterações de FPS, tal como era suposto acontecer teoricamente, devido ao pequeno número de pontos. Já nos testes desta fase, principalmente nos testes 3,4 e 5, que vamos mostrar posteriormente, reparamos numa diferença drástica de FPS entre com e sem VBOs. Sem VBOs dava por volta de 210 fps, ativando os VBOs chegavam a 440. Já no nosso sistema solar dinâmico com as curvas dava por volta de 240 fps e ao ativar os VBOs passou para 360. Desativando as curvas, vai de 280 fps sem VBOs para 430 com VBOs.

3 Cálculo das normais

O cálculo das normais foram feitos para otimizar a forma como a luz reflete nas figuras conseguindo assim apresentar as curvas das figuras mais suavemente, principalmente em casos como o *teapot* e na esfera. Para cada figura diferente, existe a sua própria função.

3.1 Plano

Para o plano fizemos a função **calculatePlaneNormals**, que recebe a lista de vértices do plano para o qual queremos calcular a Normal. Começa por calcular duas arestas do plano a partir de três vértices iniciais. O produto vetorial dessas duas arestas é calculado para obter um vetor perpendicular ao plano, que é a sua normal. Depois a normal é normalizada para ter comprimento 1, dividindo cada componente pelo comprimento do vetor. Após essa normalização de comprimento, a normal calculada é replicada para todos os vértices da lista, garantindo que todos os vértices compartilham a mesma normal. Finalmente, a lista de normais é convertida para um vetor e retornada terminando assim a função.

Vai ser este procedimento em todos os diferentes tipos de figuras que vai assegurar que elas tenham uma normal consistente para cálculos de iluminação e renderização.

3.2 Cubo

Para o cubo fizemos a função **calculateCubeNormals**, que recebe a lista de vértices do cubo para o qual queremos calcular a Normal. Começa dividindo cada face do cubo em dois triângulos, e, para cada triângulo, a normal é calculada da seguinte forma:

- Duas arestas do triângulo são determinadas subtraindo as coordenadas dos vértices.
- O produto vetorial das duas arestas é calculado para obter um vetor perpendicular ao plano do triângulo, resultando na normal do triângulo.
- A normal é normalizada para ter comprimento 1, dividindo cada componente pelo seu comprimento.

- A normal calculada é adicionada a uma lista várias vezes para representar todos os vértices dos dois triângulos que formam a face do cubo.

Finalmente, a lista de normais é convertida para um vetor e retornada.

3.3 Esfera

Para a esfera fizemos a função **calculateSphereNormals**, que funciona de forma diferente que as últimas duas. Esta depois de receber a lista de vértices da esfera para o qual queremos calcular a Normal começa por fazer o cálculo do centro da Esfera. As coordenadas médias de todos os vértices são calculadas para determinar o centro da esfera. As somas das coordenadas x, y e z de todos os pontos são divididas pelo número de vértices.

Depois mais concretamente para o cálculo das normais começa-se por calcular um vetor do centro da esfera até o vértice para cada vértice. Esse vetor é então normalizado (dividido pelo seu comprimento) para obter a normal. As normais normalizadas são armazenadas em um vetor. Sendo este o vetor que é retornado no final.

Este método garante que as normais apontem radialmente para fora da superfície da esfera.

3.4 Cone

Para o cone fizemos a função **calculateConeNormals** que volta à mesma estrutura do Plano e do Cubo.

O código percorre a lista de vértices que recebeu, processando três vértices por vez para formar um triângulo. São calculadas duas arestas do triângulo subtraindo as coordenadas dos vértices. O produto vetorial das duas arestas é calculado para obter um vetor perpendicular ao plano do triângulo, resultando na normal do triângulo. A normal é normalizada para ter comprimento 1, dividindo cada componente pelo comprimento do vetor. A normal calculada é adicionada três vezes à lista de normais para representar os três vértices do triângulo.

No final, a lista de normais é convertida para um vetor e retornada.

3.5 Superfícies de Bezier

Para as superfícies de *bezier* fizemos a função **calculateBezierPatchNormals** que funciona mais ou menos como a da Esfera.

Com a lista de vértices começa por calcular o centro da *Patch*. As coordenadas médias de todos os vértices são calculadas para determinar o centro do *Patch*. As somas das coordenadas x, y e z de todos os pontos são divididas pelo número de vértices.

Com estes valores faz-se o cálculo das normais. Para cada vértice, é calculado um vetor que vai do centro do *Patch* até o vértice. Esse vetor é normalizado (dividido pelo seu comprimento) para obter a normal. As normais normalizadas são armazenadas em um vetor.

No final esse vetor é retornado. Tal como na esfera, este método assegura que as normais apontem radialmente para fora a partir do centro do *Patch* de *Bezier*, o que é fundamental para cálculos precisos de iluminação e sombreamento.

3.6 Anel

Para o anel fizemos a função **calculateRingNormals** que apesar do que se espera, não funciona como a esfera e as superfícies de bezier, mas como as outras figuras.

Com a lista de vértices processa três vértices por vez para formar um triângulo. São calculadas duas arestas do triângulo subtraindo as coordenadas dos vértices. O produto vetorial das duas arestas é calculado para obter um vetor perpendicular ao plano do triângulo, resultando na normal do triângulo. A normal é normalizada para ter comprimento 1, dividindo cada componente pelo comprimento do vetor. A normal calculada é adicionada três vezes à lista de normais para representar os três vértices do triângulo.

Esta lista de normais é convertida para um vetor e retornada.

Foi com todas estas funções que conseguimos fazer com sucesso o tratamento de todas as Normais existentes para as luzes refletirem corretamente da melhor forma possível.

4 Engine

Depois de termos feito a extração de tudo o que é de novo (Luzes e Cores) dos *XMLs* com sucesso e feito as funções para calcular as normais das figuras caso existam luzes, procedemos a fazer as funções que tratariam de aplicar as cores e as luzes na **Engine**.

Essas funções foram as seguintes:

- **applyColors** -> esta função vai receber a cor da figura a qual vai aplicar, caso esta tenha cor, senão usa a cor *default* do *openGL*. Depois temos diferentes chamadas de função que extraem as diferentes componentes de cor e a propriedade de brilho da cor recebida (Color c). Cada cor no formato RGB (0-255) é convertida para o formato *GLfloat* (0.0-1.0), que é o formato esperado pela *OpenGL*. Cada *array* **vals** contém os valores RGBA para as diferentes propriedades de cor do material. As chamadas **glMaterialfv** e **glMaterialf** aplicam as propriedades de material à superfície frontal do objeto. Os parâmetros usados são:
  - **GL_FRONT**: Aplica as propriedades à face frontal dos polígonos.
  - **GL_DIFFUSE**: Determina a cor do material sob iluminação difusa.
  - **GL_AMBIENT**: Determina a cor do material sob iluminação ambiente.
  - **GL_SPECULAR**: Determina a cor dos reflexos especulares (reflexos brilhantes).
  - **GL_EMISSION**: Determina a cor do material emitida como se fosse uma fonte de luz.
  - **GL_SHININESS**: Determina o tamanho do ponto brilhante de especularidade. Valores mais altos resultam em pontos mais brilhantes e concentrados.
- **applyLights** -> esta função configura as várias fontes de luz. A função recebe um vetor de objetos **Lights**, onde cada objeto representa uma fonte de luz com suas propriedades. Inicializa a variável **light** com a primeira luz disponível em *OpenGL* (GL_LIGHT0). A função percorre o vetor de luzes e configura cada uma delas. Define a iluminação ambiente global para a cena, aplicando uma luz branca. Define as componentes difusa e especular da luz como branca. Os *arrays* **position** e **direction** são preenchidos com as coordenadas da posição e direção da luz, respetivamente. A função **normalizeVector** é chamada para normalizar a direção da luz. A função configura diferentes tipos de luzes com base no tipo especificado no objeto **Lights**:

- Para luzes direcionais, a direção da luz é definida usando **GL_POSITION** e usando a **direction**.

- Para luzes pontuais, a posição da luz é definida usando **GL_POSITION** e usando a **position**.

- Para luzes *spot*, tanto a posição quanto a direção são definidas, junto com o ângulo de corte **(GL_SPOT_CUTOFF)** e o expoente do foco **(GL_SPOT_EXPONENT)**.

Após configurar cada luz, a variável **light** é incrementada para passar para a próxima luz disponível (GL_LIGHT1, GL_LIGHT2, etc).

- **resetColor** -> esta função restaura as propriedades de material padrão para a superfície frontal de objetos. Define os valores padrão para as propriedades difusa, ambiente, especular, emissiva e de brilho. Utiliza novamente chamadas *OpenGL* (*glMaterialfv* e *glMaterialf*) para aplicar os valores padrão às propriedades de material da superfície frontal dos objetos.

A **applyColors** é chamada em cada figura pois cada figura tem a sua própria cor, enquanto que a **applyLights** é somente chamada uma vez na inicialização, pois as luzes são sempre iguais. A **resetColor** é chamada no final de desenhar cada figura, para o caso de a próxima figura não ter cor, mas tentar utilizar a cor da última figura, pois o **applyColor** altera as cores *default* do *OpenGL*, e utilizar a outra cor não estaria correto.

5 Testes e Sistema Solar

5.1 Testes

Tal como fizemos em todas as fases até agora, após desenvolvermos todo o código, corremos os testes dos professores de forma a ver o quão correta a nossa implementação se encontra. Infelizmente, nesta fase, não vamos conseguir corre-los todos, mais em específico, o teste 6 desta fase, pois este utiliza texturas. De resto conseguimos correr todos os outros testes.

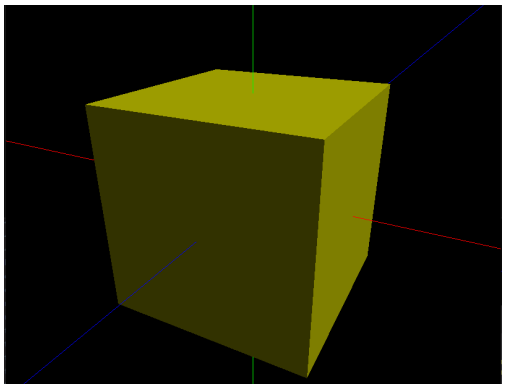

Figura 5.1: Teste 1 da Fase 4

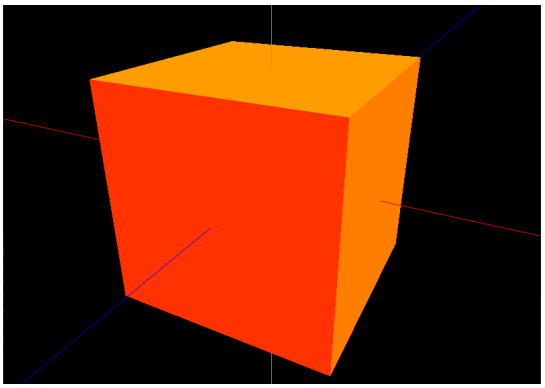

Figura 5.2: Teste 2 da Fase 4

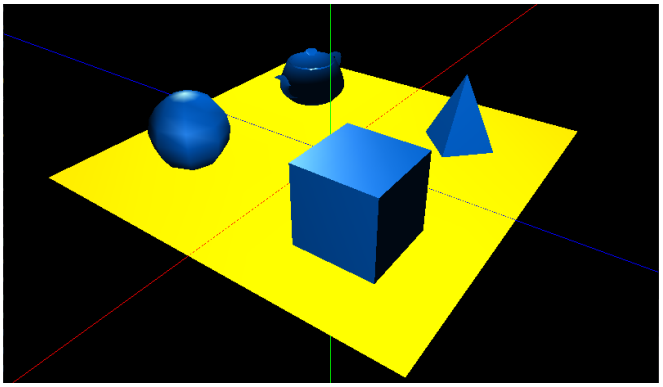

Figura 5.3: Teste 3 da Fase 4

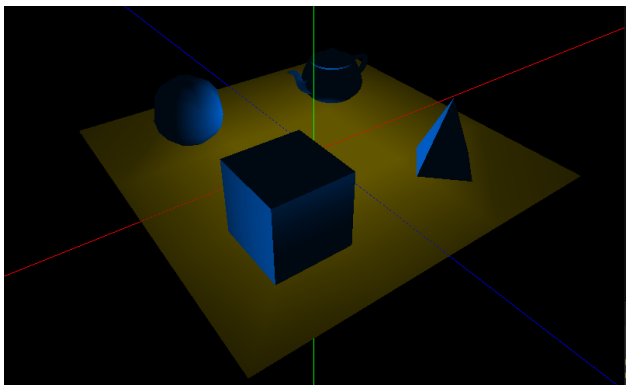

Figura 5.4: Teste 4 da Fase 4

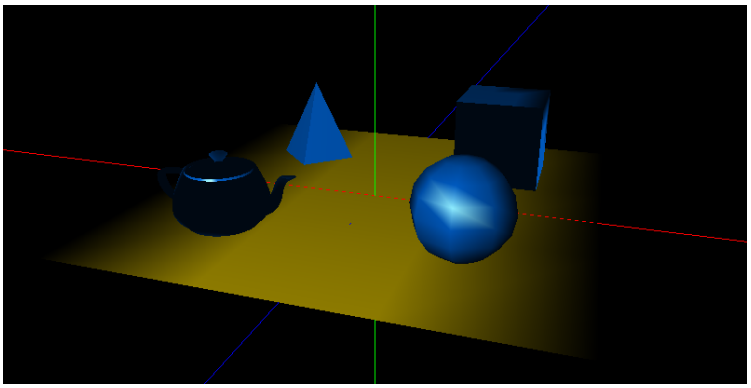

Figura 5.5: Teste 5 da Fase 4

Conseguimos ver que as normais estão bem implementadas em todos os casos, somente existe uma pequena diferença entre a iluminação do resultado dos testes dos professores e dos nosso testes. Mesmo assim, conseguimos ver de alguma forma que a iluminação até se importa bem implementada.

5.2 Sistema Solar

O nosso sistema solar para esta fase consiste em pegar no sistema solar da fase anterior, transformar o sol na fonte de luz da demonstração e por cada planeta com a sua cor correspondente. Caso tivéssemos implementado as texturas, pegaríamos em diversas texturas e colocaríamos a cada planeta com a sua textura específica.

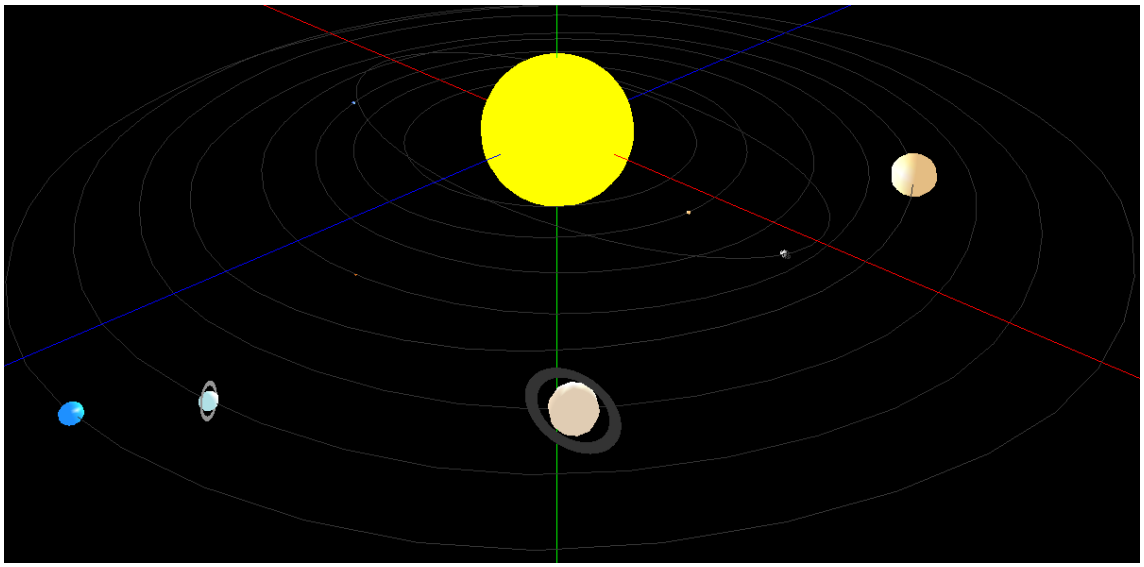

Figura 5.6: Sistema Solar com cores

6 Conclusão

Com a resolução desta quarta e última fase do trabalho prático conseguimos consolidar os conceitos de textura e iluminação que foram dados durante as aulas. Conseguimos fazer isso com o cálculo das normais e a manipulação das luzes em termos práticos e em termos teóricos ainda tentamos compreender as texturas, mas não conseguimos complementar.

Em termos de satisfação, estamos contentes com o que conseguimos implementar nesta fase (as normais, as luzes e as cores), pois encontram-se bem formuladas, mas encontramos-nos desapontados por não ter conseguido fazer as texturas.

Visto que esta é a última fase, conseguimos olhar para o trabalho de uma forma mais geral e ver que temos todos os elementos necessários para ter um bom trabalho tendo conseguido consolidar todos os ensinamentos adquiridos nas aulas teóricas e práticas da cadeira de Computação Gráfica.