Computação Gráfica
Fase 1 - Primitivas Gráficas

Grupo 42
Ângelo Sousa[A80813], Diogo Ribeiro[A84442], Rui Mendes[A83712], and Rui Reis[A84930]

Universidade do Minho, Departamento de Informática, 4710-057 Braga, Portugal
e-mail: {a80813,a84442,a83712,a84930}@alunos.uminho.pt

Resumo Para a primeira fase do trabalho prático de Computação Gráfica vamos desenvolver dois módulos: um módulo para gerar ficheiros com vértices de modelos, e um engine capaz de ler ficheiros de configuração escritos em XML que reproduz os modelos.

1 Introdução e Contextualização

Para a cadeira de Computação Gráfica foi-nos proposto desenvolver na primeira fase do projeto 2 módulos. O primeiro módulo, ao qual chamamos generator, tem como principal funcionalidade, dados determinados parâmetros, gerar os vértices correspondentes ao modelo em questão e colocar este output num ficheiro com a extensão .3d. Os parâmetros que o generator aceita estão definidos no enunciado do projeto.

Além da listagem dos vértices implementamos uma funcionalidade extra que nos permite obter ganhos de performance, tal como abordamos nas aulas práticas. Essa funcionalidade consiste em remover os vértices duplicados no módulo generator. Essa funcionalidade foi implementada usando 2 estruturas de dados distintas, sendo que na primeira guardamos todos os vértices distintos, e na segunda guardamos uma lista de índices que apontam para a primeira estrutura e representam a ordem pela qual devemos desenhar os vértices.

O segundo módulo, ao qual chamamos parser, tem como principal funcionalidade interpretar os ficheiros XML fornecidos e popular estruturas de dados com a informação dos ficheiros .3d mencionados no ficheiro de configuração XML. Para lidar com os ficheiros XML recorremos ao tinyXML, sendo que foi recomendado e achamos que era a melhor opção.

2 Primitivas Gráficas

2.1 Plano

A definição de um plano é imediata, tendo em conta que para esta primitiva, o número de divisões não é contemplado, obtemos simplesmente uma figura que

pode ser totalmente definida por um conjunto de quatro vértices. O plano assenta sobre o plano xz, o qual apelidamos de P.

Assumindo o conjunto de vértices como $V = \{V_0, V_1, V_2, V_3\}$ de um plano de comprimento L, então é possível verificar que $v_y = 0$, $\forall v \in V$, isto porque $v \in P$, $\forall v \in V$.

Pelo que é possível definir todos os vértices da figura da seguinte forma:

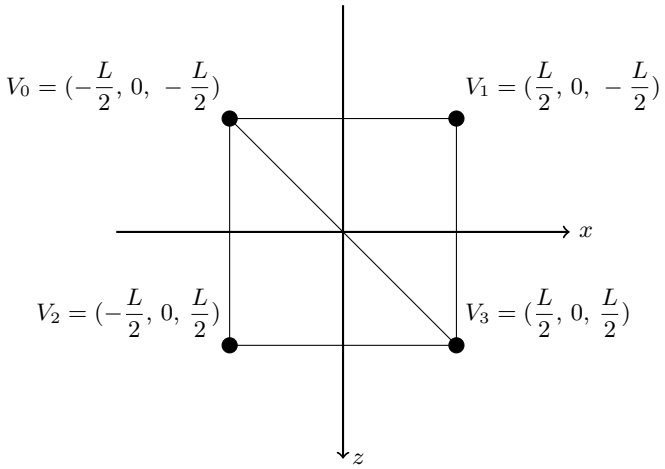

Figura 1. lorem ipsum

Tendo em conta que um plano é formado por dois triângulos, só falta então definir a ordem de indexação de cada vértice. De forma a seguir a *right hand rule*, a ordem para formar cada triângulo deve ser:

- **Triângulo 1**: $V_0 \longrightarrow V_2 \longrightarrow V_3$
- **Triângulo 2**: $V_0 \longrightarrow V_3 \longrightarrow V_1$

2.2 Caixa

A caixa apresentou-se como a primitiva gráfica mais desafiante, isto porque ao utilizarmos indexação, desenhando cada vértice uma e uma só vez, é nos apresentado um desafio superior. O de relacionar os vértices e os seus índices com todas as outras faces.

Tendo em conta que a nossa caixa tem em conta o conceito de divisões, o nível de conceito é incrementado, pois é necessário ter em atenção um número muito grande vértices, e a automatização de todas as indexações, utilizando ciclos.

A nossa caixa possui as seguintes medidas:

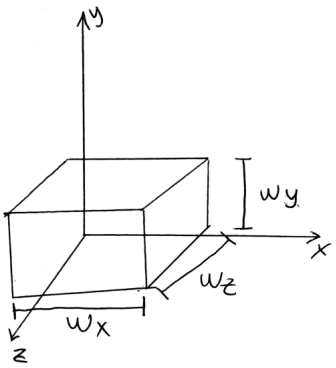

Figura 2. Representação da caixa e medidas.

Com a origem do referencial no ponto do meio da caixa.

Porém, surge o problema de definir os vértices. Inicialmente, consideramos que um cubo é constituído por uma banda e por duas fases laterais.

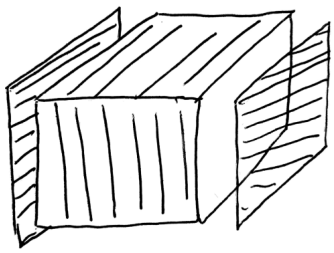

Figura 3. Representação da caixa e medidas.

Assumindo que estamos a utilizar N divisões, então cada face lateral vai possuir um total de $N \times N$ vértices. De forma semelhante, a banda à volta da caixa possui $4 \times (N-1) \times N$ vértices. Olhemos para uma das faces da caixa que contempla a banda, no caso de $N = 3$:

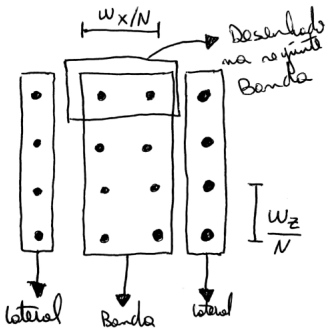

Figura 4. Representação da caixa e medidas.

Todos os pontos estão separados por $\frac{w_i}{N}$ unidades, com $i \in \{x, y, z\}$. Os vértices mais à esquerda e direita são definidos na construção das laterais, enquanto que os interiores são definidos aquando da criação da banda.

Após criados os vértices é necessário fazer a sua indexação de forma a ser possível formar triângulos. Para tal, a ordem de indexação é a seguinte:

1. Indexar os triângulos da lateral mais à esquerda, que contempla N^2 vértices.
2. Indexar a banda à volta da caixa, que contempla $4 \times (N-1) \times N$.
3. Indexar os triângulos da lateral mais à direita, que contempla N^2 vértices.

Por exemplo, na face mais à esquerda, a indexação é feita da seguinte forma:

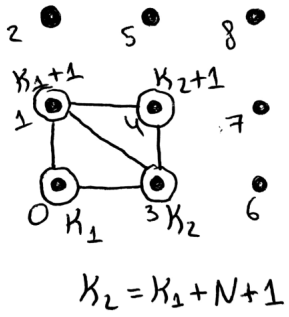

Figura 5. Representação da caixa e medidas.

Levando a que a ordem de definição dos quadrados seja a seguinte, assumindo que k_1 é o vértice a ser processado e $k_2 = k_1 + N + 1$.

- **Triângulo 1**: $k_1 \longrightarrow k_2 \longrightarrow k_1 + 1$

- **Triângulo 2**: $k_2 \longrightarrow k_2 + 1 \longrightarrow k_1 + 1$

Para a fase mais à esquerda o processo é semelhante com algumas diferenças na banda, mas equivalente em termos matemáticos.

O problema que surge agora é tentar conectar a banda a ambas as fases laterais. O que se mostrou ser um dos maiores desafios de obter um cubo com indexação. Apresenta-se a seguir um exemplo com $N = 2$, de forma a atestar a dificuldade deste processo.

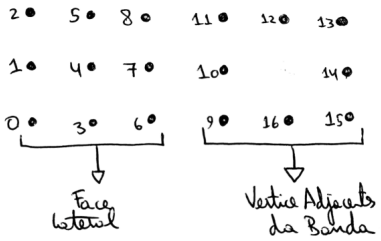

Figura 6. Representação da caixa e medidas.

Pelo que é preciso conectar os vértices mais externos, originando as seguintes ligações:

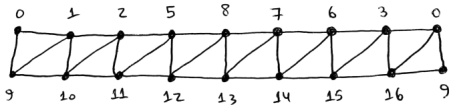

Figura 7. Representação da caixa e medidas.

O que é conseguindo na prática, utilizando um *array* auxiliar, que permite armazenar os *indexes* do vértices mais exteriores e de seguida formar os triângulos entre estes.

Pelo que, de forma simplificada, o algoritmo de geração da caixa surge como:

1. Gerar vértices da face mais à esquerda.
2. Indexar vértices da face mais à esquerda.
3. Gerar vértices da banda.
4. Indexar vértices da banda.
5. Gerar vértices da face mais à direita.
6. Indexar vértices da face mais à direita.
7. Indexar ligações entre face mais à esquerda e banda.
8. Indexar ligações entre face mais à direita e banda.

2.3 Cone

De forma a conseguir definir os vértices do cone temos primeiro de perceber quais são os seus input, a nossa definição de vértices recebe a seguinte informação:

- Raio, definido como r.
- Altura, definida como h.
- *Stacks*, definida como s
- *Slices*, definida como f

Para a definição de cone, assumimos que um cone é formado por vários círculos, onde usamos coordenadas cilíndricas. Sendo que cada círculo corresponde a um plano de corte em termos de altura.

Assumindo α como sendo a variação em termos de ângulo de forma a contemplar todas as *slices*, ou seja, $\alpha = \frac{2\pi}{f}$. E que i representa a i-ésima *slice*, $i \in [0, s-1]$. Para as coordenadas cilíndricas obtemos que:

$$z = r \cdot cos(i\alpha)$$
$$x = r \cdot sin(i\alpha)$$

Em termos de y, o que é feito é dividir a altura, em passos incrementais, obtendo portanto:

$$y = j\frac{h}{\#\text{ Stacks}}, \text{ com } j \in [0, s]$$

Portanto, definir cada vértice, aplicamos um simples algoritmo que para todas *stacks*, preenchem os vértices com cada uma das *slices* individualmente, conectando-as à posteriori.

O ponto *(0,0,0)* é o primeiro vértice a ser adicionado, pelo que ocupa o índice 0.

Os vértices das laterais, devem ser então conectados da seguinte forma: Assumindo k_1 como sendo o vértice corrente e k_2 como o vértice da stack imediatamente acima, tal que $k_2 = k_1 + f + 1$ (o +1 deve-se ao facto de a origem do referencial ocupar o índice 0.

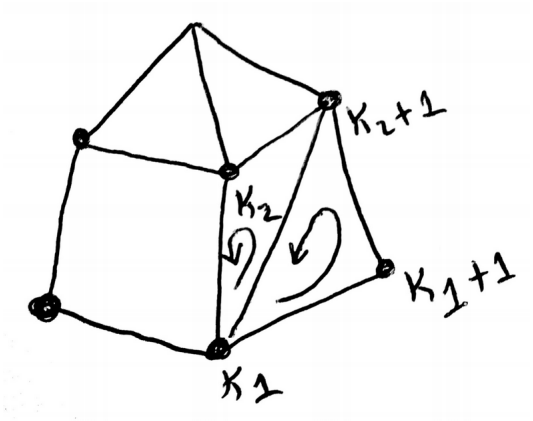

Figura 8. Representação da caixa e medidas.

Os triângulos da base e do topo são indexados num processo semelhante.

2.4 Esfera

O processo de criação da esfera foi em muito semelhante ao processo de criação do cilindro, exceto que utilizamos cordenadas polares, porém de igual forma continua a haver o conceito de *stacks* e *slices*.

stacks cortam a esfera transversalmente, enquanto que *slices* cortam a esfera longitudinalmente. Para definirmos as seguinte coordenadas, assumimos j como sendo a j-ésima *stack*, e i como sendo a i-ésima *slice*. Assumindo também que $\beta = \frac{\pi}{\#\text{ Stacks}}$ e $\alpha = \frac{2\pi}{\#\text{ Slices}}$. Obtendo então:

$$x = r \cdot sin(i\alpha) \cdot (-\frac{\pi}{2} + j\beta)$$
$$y = r \cdot cos(i\alpha) \cdot (-\frac{\pi}{2} + j\beta)$$
$$z = r \cdot sin(-\frac{\pi}{2} + j\beta)$$

O que permite originar todos os vértices, porém os vértices dos polos da esfera são adicionados manualmente. De forma semelhante ao cilindro, para todas as *stacks* originamos todos os vértices das respectivas *slices*, obtendo então a seguinte configuração das laterais:

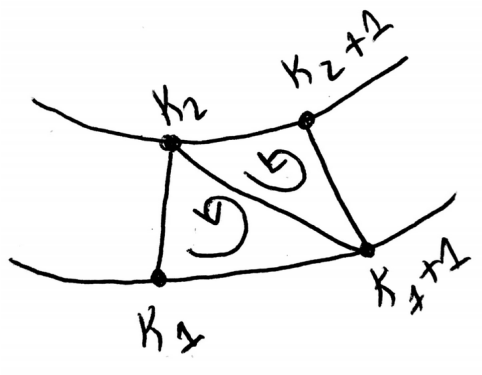

Figura 9. Representação de um quadrante da esfera.

Com k_1 a ser o vértice atual, e $k_2 = k_1 + \#$ Stacks como segundo vértice da seguinte *stack*. Permite então definir a seguinte ordem de indexação:

- **Triângulo 1**: $k_1 \longrightarrow k_1 + 1 \longrightarrow k_2$
- **Triângulo 2**: $k_2 \longrightarrow k_1 + 1 \longrightarrow k_2 + 1$

Para adicionar os triângulos correspondente ao polo norte e sul da esfera, é efetuado um ciclo que rodeia o vértice no polo dos vértices imediatamente adjacentes, formando assim triângulos.

3 Parser XML

De modo a manter a informação organizada foi necessário criar a representação de um objeto no parser. Para esse efeito criámos a classe Object3d que contém o total de vértices da figura, todos os pontos que a constituem e a ordem pelo qual os pontos devem ser desenhados. Esta classe fornece funções para carregar dados para a estrutura (loadObject) e para limpar a estrutura (destroyObject).

A função loadObject está encarregue de ler a informação presente nos ficheiros .3d descritos no ficheiro xml,passado ao programa como argumento, e construir o objeto com essa mesma informação. Para fazer isso foi necessário decidir o formato dos ficheiros .3d de modo a que o parser fosse capaz de montar os objetos. Optou-se então por iniciar o documento com o número distinto de vértices no objeto, seguido das coordenadas desses mesmos vértices e por fim a ordem pela qual os vértices seriam desenhados, sendo que o índice de cada vértice corresponderia à sua posição relativa aos outros vértices no inicio do ficheiro (ou seja o primeiro vértice para o qual se definisse as coordenadas seria o vértice 0).

Sabendo então esta informação o parser ia montando os tais Objetos3d e guardando a informação respetiva nos vectores pontos e índices para que mais tarde estes pudessem ser associados aos buffers necessários.

Para desenhar estes objetos foram escolhidos VBO's, desta forma tornando mais eficiente o processo de renderização, os buffers necessários eram criados depois do programa ler todo o ficheiro xml e dessa forma saber quantos objetos iriam ser desenhados para alocar o espaço necessário correto para os buffers (um por objeto). Estes buffers seriam depois então preenchidos pela informação presente em cada um dos Objetos3d já criados anteriormente.

Assim para a renderização destes objetos foi criada a função renderObject que percorria estes buffers e fazendo uso de funções próprias para lidar com VBO's desenhava os objetos fazendo uso da indexação dos pontos.

A função destroyObject é uma função simples que coloca o número de vértices de um objeto a 0 e limpa os vectores que contêm os pontos e os índices necessários para a renderização dos mesmos.

4 Conclusão

Após a conclusão do generator e do parser fomos capazes de abstrair algumas ideias essenciais que também nos têm sido transmitidas nas aulas práticas, bem como melhorar a nossa capacidade de representar figuras e manipular vértices.
Ao procurar uma solução que eliminasse os vértices repetidos das figuras e apenas incluísse a informação essencial no ficheiro .3d criado pelo generator fomos capazes de desenvolver um método mais eficaz de representar figuras, sendo que também ficámos com melhores noções dos ganhos de performance que podemos ter quando se tratam de figuras com um número extremamente elevado de vértices.
Com o desenvolvimento do parser fomos capazes de pôr em prática os conhecimentos que adquirimos na UC, especialmente com a utilização de VBO's, sendo que em conjunto com a tipologia dos ficheiros gerados pelo generator fomos capazes de obter um mecanismo relativamente eficiente. Essas eficiências são especialmente notórias com um grande fluxo de vértices onde o potencial ganho na performance é bastante elevado.