INSTITUTO POLITÉCNICO DO CÁVADO E AVE

TRABALHO PRÁTICO – FASE 1

RELATÓRIO

JORGE MIGUEL AREZES NORO | 15705

ENGENHARIA E DESENVOLVIMENTO DE JOGOS DIGITAIS

INTRODUÇÃO À PROGRAMAÇÃO 3D

DOCENTE: JOSÉ HENRIQUE BRITO

2º ANO - 1º SEMESTRE

OBJETIVOS

Este relatório serve de apoio ao código e programa entregue. Os objetivos definidos pelo professor para a primeira fase do trabalho prático da unidade curricular são criar uma superfície texturada e apresentar a mesma no ecrã, de forma interativa, através de uma ou mais câmaras. Estes objetivos podem ser observados no programa entregue, dentro da pasta "programa". Todo o código fonte pode ser consultado na pasta "codigo".

Durante a elaboração deste trabalho foi tida em consideração a portabilidade e modularidade do código, de forma a poder ser reaproveitado ao máximo para outros projetos e facilitar futuras consultas.

Para realizar este trabalho, foi utilizado o software de edição e compilação de código Microsoft Visual Studio[1], a *framework* Monogame[2] 3.6 e GitHub[3] para *source control*.

[1] https://visualstudio.microsoft.com
[2] http://www.monogame.net
[3] https://github.com

CRIAÇÃO DE UM PLANO

A criação da superfície que toma o papel de terreno revolve sobre a ideia de um plano bidimensional. Foi auto proposto o desafio de primeiro criar um plano orientado em XZ no sistema de eixos absoluto, especificando a sua dimensão em comprimento de profundidade, o número de subdivisões que o plano irá conter e a escala a aplicar às coordenadas da textura, UV. Após este objetivo ser cumprido, iremos ter muita mais flexibilidade sobre a geometria e facilmente podemos criar funções para modificar os vértices e índices deste plano, que é o segundo passo: desenvolver uma função que através de uma textura, modifica a altura dos vértices (coordenada Y).

O plano fará parte do sistema interno de componentes da *framework*, herdando assim da classe *GameComponent*.

Com estes objetivos em mente, rapidamente e sem nenhuma dificuldade que mereça ser descrita, foi construída a classe "Plane". Podemos ver no *Code Snippet 1* os parâmetros presentes no construtor:

```
public Plane(Game game, string textureKey, float width = 10f, float depth = 10f,
       int xSubs = 1, int zSubs = 1, float uvscale = 1f)
```

Code Snippet 1

A criação da geometria é algo relativamente simples. É calculada a dimensão das subdivisões em cada eixo (X e Z) e de seguida são feitos dois ciclos (*nested*) que percorrem X e Z até estes serem iguais ao número de subdivisões fornecidas. É importante que o valor chegue a ser igual, pois o número de vértices em determinado lado será sempre $nVertices = nSubs + 1$. Dentro dos ciclos, é calculada a coordenada no espaço em X e Z através da seguinte fórmula: $x = i * subWidth - \frac{PlaneWidth}{2}$. Subtrair a coordenada pelo comprimento (ou profundidade) dividido por dois é o que irá fazer com que a origem do plano fique na coordenada (0, 0, 0) absoluta. São também calculadas as coordenadas U e V, para mais tarde os *shaders* aplicarem as texturas (*mapping)*. O cálculo das coordenadas U e V é feita pela fórmula $u = \frac{i}{nWidthSubs * UVScale}$. Os vértices são depois guardados num *array* e é alocado o *buffer* na *GPU* com a informação dos vértices. Nesta fase, está a ser usado um *VertexPositionTexture*, visto que para já não precisamos de normais para iluminação nem navegação.

Como irá ser utilizada renderização indexada, garantido assim uma melhor performance, ao custo de alguma memória *RAM*, é necessário criar os índices respetivos à triangulação das subdivisões. A tarefa é trivial, por isso não se irá entrar em muito detalhe. São feitos novamente dois ciclos (*nested*) e são depois calculados dois triângulos por cada subdivisão. O único cuidado a ter, será o *winding* do triângulo. Temos de garantir que desenhamos o triangulo no sentido dos ponteiros do relógio, pois caso contrário, a *GPU* irá proceder ao *culling* do triângulo, o que significa que não será renderizado.

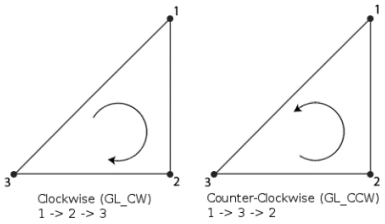

Figura 1- Winding Order[4]

Na Figura-1 podemos observar como funciona o *winding*.

Após ter a classe construída, podemos observar um exemplo renderizado, em que podemos ver as subdivisões. O *Code Snippet 2,* cria um plano de dimensão 10x10 (X e Z) e subdivide-o em 2x2:

```
Plane plane = new Plane(this, "pink", 10, 10, 2, 2);
```

Code Snippet 2

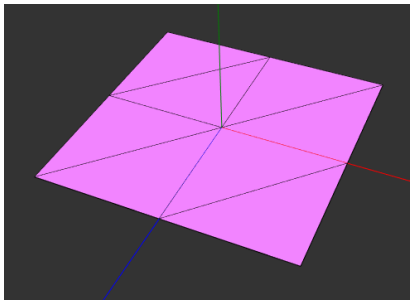

Figura 2- Resultado da execução do Code Snippet 2

O plano resultante, estará centrado na origem absoluta, desta forma, quando necessário rodar ou escalar o objeto 3D, podemos esperar resultados mais homogéneos e evitar tempo e recursos a centrar antes de realizar estas operações.

De seguida, foi implementada a função que modifica os vértices do plano em Y. Esta função recebe como parâmetros a textura a interpretar e um multiplicador. O primeiro passo da função é extrair o valor das cores da textura para um *array*, para assim podermos obter a cor correspondente em determinada coordenada. De seguida, é feito um ciclo por cada um dos vértices, atualizando o seu valor em Y conforme o valor da cor da textura (neste momento apenas o canal da cor vermelha está a ser tido em conta) e multiplicado pelo multiplicador fornecido. Após atualizar todos os vértices, é necessário colocar de novo a informação atualizada no *buffer* alocado na *GPU*. Para isso executamos o código no *Code Snippet 3*:

```
// unbound the buffer, so we can change it
Game.GraphicsDevice.SetVertexBuffer(null);

// set the new data in
VertexBuffer.SetData<VertexPositionTexture>(VertexList);
```

Code Snippet 3- Buffer Update

[4] https://www.khronos.org/opengl/wiki/Face_Culling

Precisamos primeiro de fazer um *unbound* do *buffer* para nos ser possível modica-lo, para isso fazemos o *bound* de um valor *null* ao dispositivo. Abaixo o resultado após executar o seguinte código:

```
// initialize the plane with the prefered settings
plane = new Plane(this, "grey", 128 * 2, 128 * 2,
       terrainHeightMap.Width - 1, terrainHeightMap.Height - 1, 0.5f);

// initialize the plane with the prefered settings
plane.SetHeightFromTexture(terrainHeightMap, 0.09f);
```

Code Snippet 4 - Modificar a altura dos vértices com recurso a heightmap

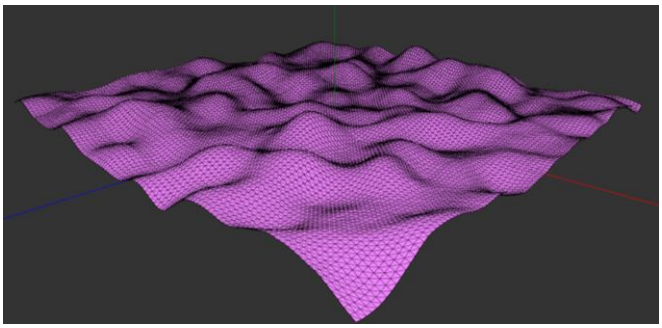

Figura 3 - Resultado da execução do Code Snippet 4

Para mostrar o plano no ecrã, é chamado o método *Draw(GameTime)*. Este método irá ser responsável por realizar todas as operações de *rendering*. Em primeiro lugar, fazemos *bound* na GPU dos *buffers* de vértices e índices que queremos utilizar. É aplicado o *shader* (*Effect* em Monogame) e chamada a função de *DrawIndexedPrimitives*:

```
// send a draw call to the gpu, using a triangle list as a primitive.
// I could be using a triangle strip, but nowadays computers have tons of memory
// and I'd rather keep the draw calls at a minimum and save the cpu the struggle
Game.GraphicsDevice.DrawIndexedPrimitives(PrimitiveType.TriangleList, 0, 0,
IndicesList.Length / 3);
```

Code Snippet 5 - DrawIndexedPrimitives

Após isto, nesta fase, foi incluída a funcionalidade de fazer *render* em *wireframe*. Para dar uso a essa funcionalidade foi feita alguma pesquisa em busca da melhor opção. Inicialmente o programa usava uma *LineStrip*, que conectava todos os vértices, mas causava uma linha ser desenhada a mais de coluna para coluna. Foi pensado usar um ciclo por coluna, mas isso iria tonar necessário várias *draw calls*. Como o meu objetivo é sempre performance, este método não era opção[5]. Após a pesquisa, a solução passou por ser bastante simples, usar um *Rasterizer* com a propriedade *FillMode* em *FillMode*.*Wireframe*.

[5] http://hacksoflife.blogspot.com/2010/01/to-strip-or-not-to-strip.html

CÂMARAS

Todas as câmaras partem de uma classe base *Camera*. Esta classe base já contém todas as propriedades básicas que uma câmara necessita: *Position*, *Target*, *ViewTransform* (*Matrix*), *ProjectionTransform* e *FieldOfView*.

Nesta fase, o programa entregue, contém três câmaras:

BASIC CAMERA:

Esta câmara é algo simples, usada para começar a visualizar os objetos construídos rapidamente. A câmara orbita sobre o target (*default* no (0,0,0)) e oscila a coordenada Y. Todos os parâmetros são convenientemente ajustáveis pelo programador.

FREE CAMERA:

Uma câmara livre que possibilita o utilizador navegar livremente pelo cenário. A câmara é controlada pelo utilizador, usando o teclado (WASD) e o cursor.

Para a construção desta câmara foi usado como recurso um artigo disponível online[6].

A câmara usa o movimento *delta* do cursor (diferença entre a posição atual e a última posição) para calcular os ângulos *Pitch* e *Yaw*, usando as funções base da trigonometria, seno e cosseno. Um aspeto importante que foi retido do artigo, é a necessidade de bloquear o ângulo *Pitch*. Isto acontece quando o utilizador "olha" para cima e passa os 90 graus. Além de resultados inesperados, pode acontecer de a câmara bloquear um ângulo de liberdade, chamado de *Gimbal Lock*.

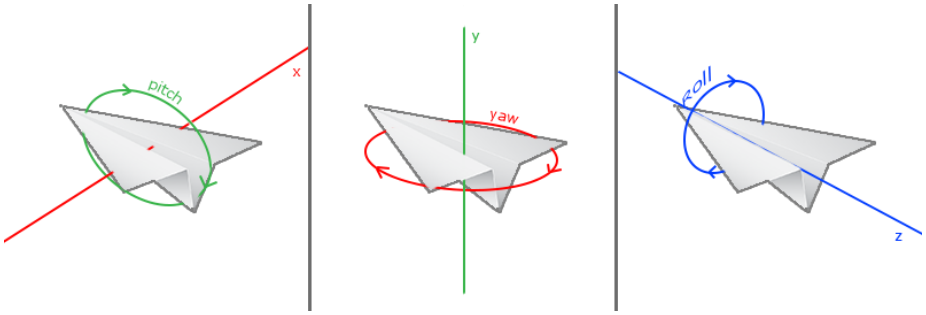

Figura 4 - Ilustração dos 3 ângulos de liberdade[7]

É também possível fazer *zoom in* e *zoom out* usando a terceiro botão do rato, se existente.

A câmara usa o conceito de aceleração, velocidade máxima e atrito para a deslocação no espaço.

O funcionamento da câmara pode ser lido na integra no website referido e um artigo sobre o *Gimbal Lock* pode ser lido no website W*ikipedia*[8].

[6] https://learnopengl.com/Getting-started/Camera
[7] https://learnopengl.com/img/getting-started/camera_pitch_yaw_roll.png
[8] https://en.wikipedia.org/wiki/Gimbal_lock

Surface Follow Camera:

Esta câmara apenas funciona quando fornecida uma superfície.

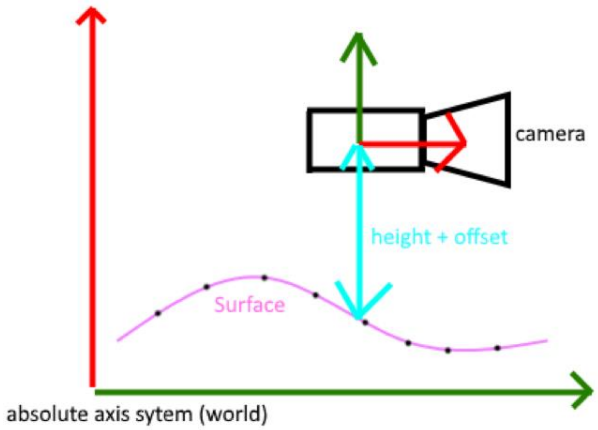

Figura 5 - Ilustração do funcionamento

A câmara herda da *FreeCamera* obtendo todas as funcionalidades de livre navegação. A única *feature* a adicionar, serão algumas restrições no movimento, nomeadamente os limites onde se pode deslocar (limites da superfície) e a altura em Y, que será igual à altura da superfície nessa coordenada, mais um *offset* definido.

A restrição dos limites é trivial. Verificar se a posição é anormal, e corrigir mediante a situação.

A restrição em Y, é algo mais interessante. Para conseguir obter a altura da superfície na coordenada em que a câmara se encontra, temos de aceder à informação dos vértices. A forma como a tarefa é realizada, é fazer um arredondamento da posição, retirando a parte decimal. Após obter um X e Z, obtemos o vértice na coordenada, num *array* unidimensional da seguinte forma:

$i = (SurfaceXSubs + 1) * Z + X$, em que *i* é o índice do vértice pretendido.

Obtivemos o índice do topo esquerdo da subdivisão em que a câmara se encontra, mas só essa informação não nos é suficiente. Foi feito um teste usando diretamente este valor e o resultado foi algo que já se previa. A câmara "salta" de vértice em vértice, um efeito que deixa algo a desejar.

Para obter um movimento suave, precisamos de calcular uma interpolação entre os quatro vértices vizinhos da posição da câmara. Como já temos o índice do primeiro, calcular os restantes é fácil, por isso podemos seguir para o próximo passo. Vejamos na *Figura - 6* abaixo o resultado que pretendemos.

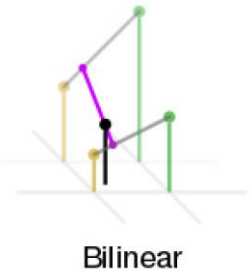

Figura 6 - Resultado pretendido[9]

Com os 4 vértices em mão, realizamos então uma interpolação bilinear com a posição da câmara, em que a função de cada componente, é o retorno da posição Y[10]. Observemos abaixo a fórmula para o cálculo entre vértices, implementada no código:

```
float x0 = vertex0.X;
float x1 = vertex3.X;
float z0 = vertex0.Z;
float z1 = vertex3.Z;

// interpolate the x's
float x0Lerp = (x1 - position.X) / (x1 - x0) * vertex0.Y + (position.X - x0) / (x1 - x0) * vertex1.Y;
float x1Lerp = (x1 - position.X) / (x1 - x0) * vertex2.Y + (position.X - x0) / (x1 - x0) * vertex3.Y;

// interpolate in z
float zLerp = (z1 - position.Z) / (z1 - z0) * x0Lerp + (position.Z - z0) / (z1 - z0) * x1Lerp;

return zLerp;
```

Code Snippet 6 - Interpolação bilinear

É realizada a interpolação nos dois segmentos em X (vertex0 -> vertex1 e vertex2 -> vertex3) e de seguida é realizada a interpolação em Z desses dois resultados (x0Lerp -> x1Lerp). O resultado, será então a altura média da nossa posição. A esta altura, será ainda adicionado um *offset* definido pelo programador, se não for feito, metade da câmara andará "por baixo" da superfície devido à restrição do *near plane*.

[9] https://upload.wikimedia.org/wikipedia/commons/9/90/Comparison_of_1D_and_2D_interpolation.svg
[10] https://en.wikipedia.org/wiki/Bilinear_interpolation

CONTROLOS

Foi criada uma classe estática para centralizar os controlos a usar no programa. A classe possui propriedades de conveniência como por exemplo:

```
public static Keys Forward     = Keys.W;
public static Keys Backward    = Keys.S;
public static Keys StrafeLeft  = Keys.A;
public static Keys StrafeRight = Keys.D;
```

Code Snippet 7 - Propriedades de controlos

Desta forma, torna-se mais flexível modificar os comandos do programa. A classe também mantém o registo do estado do cursor e teclado atual e anterior ao *tick* atual. Temos então em posse, uma classe que podemos consultar desde qualquer ponto no nosso programa, e fazer "perguntas" tais como:

```
// update the camera position, based on the updated vectors
if (Controls.IsKeyDown(Controls.Forward))
{
      Velocity += Front * AccelerationValue;

}
```

Code Snippet 8 - Exemplo de utilização da classe Controls

O aspeto mais importante desta classe, é que deixamos de fazer chamadas desnecessárias aos métodos de *input* da *framework* em cada objeto que necessite de ser controlado pelas interfaces com o utilizador.

EXTRAS

À medida que o projeto foi avançando e se iam experimentando várias funcionalidades da *framework* e dos algoritmos a utilizar, foram adicionados alguns extras tais como:

- *Anti-Aliasing*:
  - O perfil dos gráficos está definido como "*HiDef*"[11]. Este perfil permite aplicar propriedades e métodos avançados na gráfica, se o computador em que o programa corre assim o suportar.
    Uma dessas propriedades é o *MultiSampling Anti-Aliasing*. A gráfica está configurada com uma contagem de *MSAA* de 8.
- *GUI Text:*
  - Foi adicionado texto de suporte para que o utilizador saiba o que está a acontecer e como pode utilizar todas as funções disponibilizadas pelo programa.
- *Wireframe Rendering*:
  - Como já foi referido, é possível ligar e desligar o modo *wireframe*.
- *World Absolute Axis System:*
  - É feito o *render* de um sistema de eixos absoluto para servir de referência no espaço global.

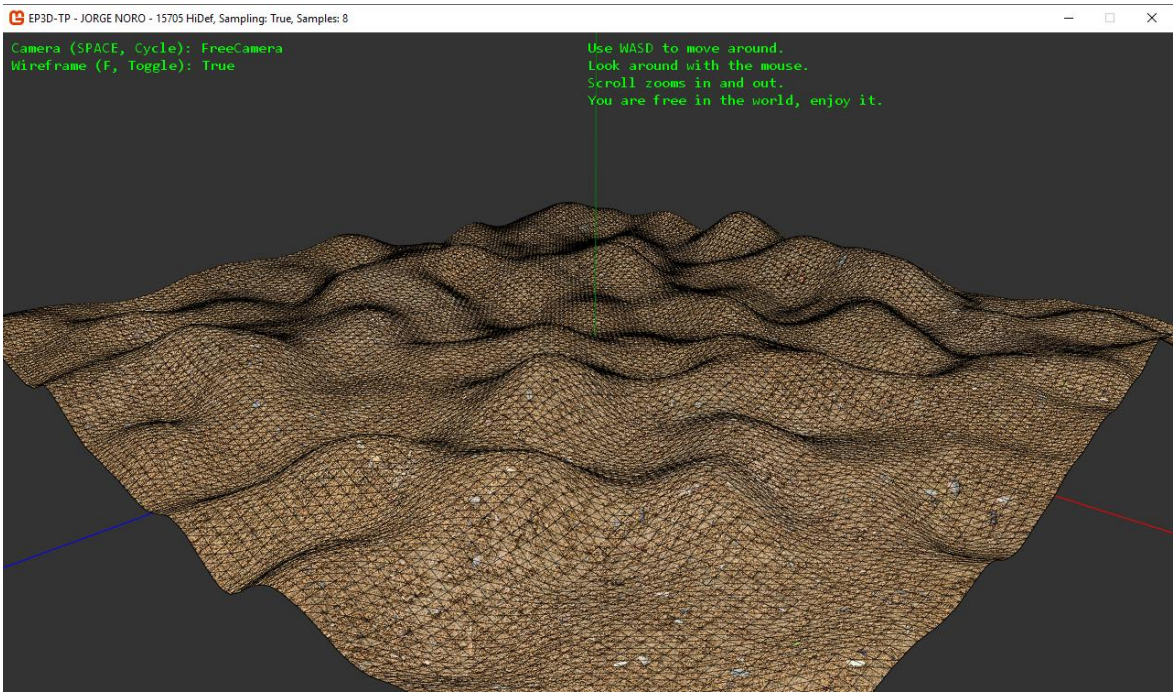

Figura 7 - Programa final

[11] https://blogs.msdn.microsoft.com/shawnhar/2010/03/12/reach-vs-hidef/

CONCLUSÕES

A realização desta fase trouxe grande satisfação à medida em que os resultados se formavam no ecrã. A exploração da parte 3D da *framework* era algo que desde o ano passado já tinha questionado.

A maior dificuldade deste trabalho foi perceber a trigonometria associada aos cálculos da câmara, mas após pesquisar extensivamente e aplicar os conceitos, a parte teórica começou a fazer sentido.

Um aspeto que ainda fica na dúvida, é a performance de *render* em *TriangleStrips*. Pela minha pesquisa, a conclusão é de que cada modelo (*model mesh*) terá um método mais apropriado para ser triangulado. O que não percebo, é como é que os vários *engines* lidam com estas situações. Como detectar qual o melhor método em *runtime*? Em alguns artigos, aprendi que para objetos como terrenos (um simples plano deformado) e objetos primitivos do género, *TriangleStrips* e *TriangleFans* seria sempre a melhor opção, por outro lado, podem existir *meshes* que não permitam fazer um bom *stripping*. Nestes casos, o que pudesse ser renderizado com *stripping*, assim o era, e o resto dos triângulos, eram preenchidos com *TriangleList*. Na minha opinião pessoal, seria muito mais simples renderizar tudo com uma lista de triângulos indexados e diminuir drasticamente o número de *draw calls*. Ainda depois de estar confuso com este tema, aprendi sobre *degenerate triangles*[12], triângulos de área 0 que não são *rasterizados* e servem para resolver problemas como o que tinha inicialmente, ao desenhar em modo *wireframe* (a linha que passava de coluna para coluna). Fica a porta aberta para mais testes e pesquisa.

Neste momento, estou bastante satisfeito com o resultado, e estou ansioso para começar a próxima fase.

[12] https://docs.microsoft.com/en-us/windows/desktop/direct3d9/triangle-strips