Licenciatura em Engenharia e Desenvolvimento de Jogos Digitais

Introdução à Programação 3D

Trabalho Prático

1ª fase

Aluno: 10242
Aluno: 13682

SAD ZOMBIES
IPCA - Introdução à Programação 3D [ED.3D] - 2º ano

IPCA INSTITUTO POLITÉCNICO DO CÁVADO E DO AVE

Indice

Introdução

De modo a demonstrarmos as nossas capacidades e o conhecimento que vamos adquirindo ao longo das aulas, foi-nos proposto um trabalho prático que consiste na criação de um jogo 3D de tanques, utilizando a ferramenta Monogame do Microsoft Visual Studio.

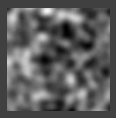
O monogame é uma implementação open-source do Microsoft XNA 4 Framework. Este tenta implementar a totalidade da API XNA 4, atraves de plataformas Microsoft usando DirectX e SharpDX. Esta permite fácil implementação de jogos para Windows, Windows Phone, Xbox 360,iOS, Android, Mac OS X, Linux e Windows 8 Apps.Os seus recursos gráficos proveem de OpenGL, OpenGL ES ou DirectX.

O Xna contém uma vasta livraria que será bastante útil ao longo do desenvolvimento deste projeto, o que facilitará todo o processo de implementação.

Para a primeira fase de entrega do projecto iremos criar um programa com as funcionalidades de render do terreno com textura e controlo da posição e orientação de uma câmara com surface follow.Para tal iremos criar duas classes(uma para o terreno e outra para a câmara).

Terreno

Para a criação e modelação do terreno, foram necessários serem importados para o content pipeline, dois assets, nomeadamente, um mapa de altura e uma textura.




Através de um mapa de alturas fornecido pelo professor, determinamos as posições de cada um dos vértices do terreno a serem criados:

```
...
terrainWidth = heightMap.Width;
terrainHeight = heightMap.Height;
...
```

Criamos um método **LoadHeightData()** que processa as cores do height map para um array de floats com os valores RGB de cada pixel.

```
...
heightMapColors = new Color[terrainWidth * terrainHeight];
heightMap.GetData(heightMapColors);
...
```

Multiplicamos a largura com a altura do terreno para definir o tamanho do array necessário para armazenar todos os valores. Esses valores são utilizados para preencher o array bidimensional **heightData[]** com os valores das alturas (eixo y) do terreno. Para cada posição vai buscar o valor R da cor e multiplica-a por um escalar, permitindo assim uma facilidade na manipulação das alturas.

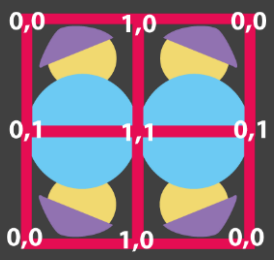
```
...
heightData = new float[terrainWidth, terrainHeight];
for (int x = 0; x < terrainWidth; x++)
    for (int z = 0; z < terrainHeight; z++)
        heightData[x, z] = heightMapColors[x + (z * terrainWidth)].R * scale;
...
```

O método **CreateVertices()** é usado para definir os vértices que constituem o terreno,assim como as coordenadas de textura, para isso o array de vértices é inicializado com multiplicação entre a altura e largura.

Criamos cada vértice com uma altura diferente acedendo ao **heightData[]** para cada (x,z).

```
...
for (int z = 0; z < terrainWidth; z++)
{
    for (int x = 0; x < terrainHeight; x++)
    {
        vertex[x + (z * terrainHeight)] = new VertexPositionNormalTexture(
            new Vector3(x, heightData[x, z], z),
            new Vector3(0, 1, 0),
            new Vector2((x % 2), (z % 2)));
    }
}
..
```

As coordenadas de texturas formam um espelho à volta dos dois triângulos iniciais replicando-se ao longo das strips.



Preenchido a array de vértices defenimos o vertexbuffer com o array usado, para que passemos a informação dos vértices apenas uma vez para a gráfica, reduzindo o tráfego CPU-GPU.

Posteriormente a criarmos os vértices precisamos de definir como é que eles se vão ligar por forma a criar os triangulos, para tal criamos o método **CreateIndices()**.

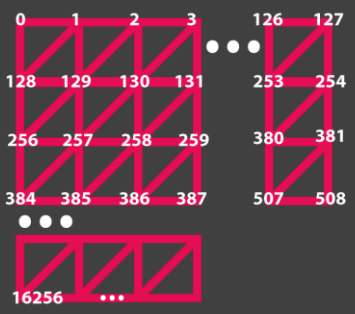
Definimos um array de shorts que é inicializado a, com o intuito de criar triangle strips verticais. Subtraimos 1 à largura para não preenchermos fora do array de vértices.

```
...
indexCount = 2 * terrainHeight * (terrainWidth - 1);
...
```

Assim preenchemos esse array com a ordem que os vértices devem ser desenhados na **Draw().**

```
...
for (int xi = 0; xi < terrainWidth - 1; xi++)
{
  for (int zi = 0; zi < terrainHeight; zi++)
  {
     index[(2 * zi) + 0 + (2 * xi * terrainHeight)] = (short)(zi * terrainWidth + xi);
    index[(2 * zi) + 1 + (2 * xi * terrainWidth)] = (short)(zi * terrainHeight + 1 + xi);
  }
}
    ...
```

Após temos o array de shorts com a ordem que os vértices se vão ligar em triangleStrips, podemos definir um indexBuffer que será responsável por mandar o array de indices apenas uma vez, passando-lhe o array recentemente criado e reduzindo o tráfego de informação CPU-GPU.



Para podermos ver o resultado de todo o mundo precisamos de o desenhar, para tal defimos a função **Draw()** para o terreno.

Embora cada objeto tenha a sua world Matrix, a view e projection matrix têm de ser actualizadas para cada movimento que a camara possa sofrer e para que conseguimos ver o movimento da camara temos de actualizar a view Matrix, acedendo ao valor respectivo da camara.

```
...
effect.View = cam.viewMatrix;
...
```

Defimos os buffers a usar pelo device com o buffers previamente inicializados.

```
...
device.SetVertexBuffer(vertexBuffer);
device.Indices = indexBuffer;
...
```

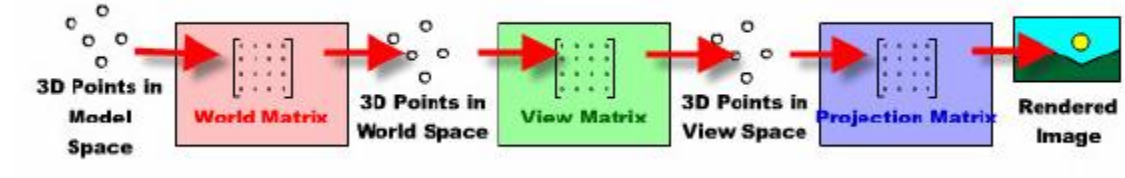
Por fim confirmamos os setting até agora defenidos e criamos um ciclo de draw com o objetivo de desenhar 127 triangle strips através do vertex e index buffers passados, para tal usamos um offset com incrementos de 256, para coincidir com os indices criados.

```
...
for (int i = 0; i < terrainWidth - 1; i++)
   {
    device.DrawIndexedPrimitives(PrimitiveType.TriangleStrip,
    0, i * 2 * terrainWidth,     (terrainWidth * 2) - 2);
    }
...
```

Esta função será então chamada na draw da game por forma a acutalizar o mundo em relação à camera e redesenhar a view matrix.

Camara

Quando falamos em renderizar uma cena em monogame estamos a falar das coordenadas do World Space, que são especificadas em coordenadas tridimensionais. Para tal precisaremos de usar 3 matrizes fundamentais para a conversão das coordenadas espaciais em coordenadas 2D(ecrã).



//World

- A world matrix é única para cada objeto dentro do seu mundo e é responsável por transformar os vértices de um objeto de seu próprio espaço local, em um sistema de coordenadas comum chamado world space.

//View

- A matriz de visão fornece o conceito de uma câmera móvel, quando é realidade, a câmera é o único ponto de referência constante no mundo. A view matrix é uma transformação que é aplicada a cada objeto na cena (mas não é exclusiva para cada objeto) e fornece a ilusão de uma câmera.
- A ilusão é basicamente o inverso do que poderia ser considerado uma world matrix para a câmera. No entanto, em vez de mover a própria câmera, ela fornece movimentos opostos ao resto da cena..
- A view Matrix é usada para transformar os vértices do world-space para a view-space. Esta matriz pode concatenar com a world matrix e com a projection matrix para transformar os vértices do espaço para o ecrã.
- Se W representar a world matrix(ou matriz de modelo), V representar a View matrix e P a projection matrix então podemos concatenar as matrizes simplesmente multiplicando as três juntas.

WVP = W*V*P

```
//Projection
```

- A matriz de projeção é responsável pela conversão de um mundo 3D no espaço de tela homogêneo que você vê na tela. Esta é a matriz usada para representar a sua visão frustum, e geralmente é representada como uma projeção ortográfica ou de perspectiva.

Surface Follow

Para definirmos uma câmera capaz de se adaptar às diferentes alturas de um terreno começamos pelo princípio, definir a world matrix, view matrix e a projection matrix, fundamentais para a renderização de qualquer cena.

```
...
cameraPosition = new Vector3(60f, 30.0f, 60.0f);
directionBase = new Vector3(1.0f, 0.0f, 1.0f);
cameraLookAt = cameraPosition + directionBase;
...
```

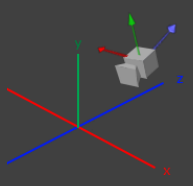
Em que cameraPosition será um vetor tridimensional que representa a posição da câmera no world space.

Para calcularmos a direção que a câmara está a apontar precisamos de um vetor direcional (directionBase), neste caso (1,0,1), para somar à posição da câmara na inicialização. Posteriormente será usado para calcular a direção da câmara na update.

As matrizes foram definidas da seguinte forma usando a Matrix.CreateLookAt e Matrix.CreatePerspectiveFieldOfView.

```
...
worldMatrix = Matrix.Identity;
viewMatrix = Matrix.CreateLookAt(cameraPosition, cameraLookAt, Vector3.Up);
projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.ToRadians(45.0f),
aspectRatio, 0.0001f, 1000.0f);
...
```

Posto isto temos a câmera inicializada e a apontar para o terreno, mas precisamos de criar movimento. Para tal criamos o método Update() que será responsável pelo movimento assim como actualizar as world matrix e view matrix.



Para calcular a rotação precisamos de criar uma matriz de rotação, para tal:

- Criamos dois floats**(yaw e pitch)** que representam a rotação e inclinação e inicializamos a 45º.

```
...
this.yaw -= MathHelper.ToRadians(1.0f);
...
this.pitch -= MathHelper.ToRadians(1.0f);
...
```

- Através do input de comandos do teclado ou rato, somamos e subtraímos ao valor do yaw/pitch, seguidamente usamos o valor das variáveis para calcular a matriz de rotação.

```
...
Matrix rotation = Matrix.CreateFromYawPitchRoll(yaw, pitch, 0);
...
```

- Tendo a rotação calculada podemos transformar o vetor 3D director(directionBase)(1,0,1) multiplicando o vetor com a matriz de rotação.

```
...
Vector3 direction = Vector3.Transform(this.directionBase, rotation);
...
```

- Seguidamente processamos o input do movimento dependendo do input (frente,trás). Para tal somamos ou subtraímos à posição da câmera o vetor de direção calculado ao multiplicar pela velocidade de movimento.

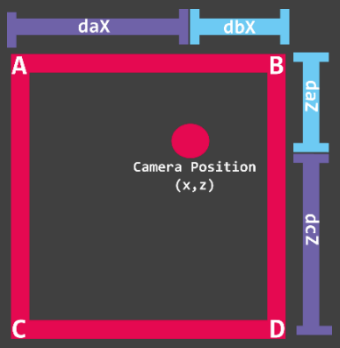
```
...
if (kb.IsKeyDown(Keys.W))
{
  this.cameraPosition += this.cameraVelocity * direction;
}
...
```

Após todos os testes de input procedemos a calcular a posição que a câmera fica apontar, independentemente de ter havido input, para re-calcularmos a world matrix da câmara e a view matrix que será geral a todos os objetos da cena.

Posto isto a câmera está apta a receber input e rodar/mover através do espaço, mas para tornar a câmera Surface Follow precisamos de actualizar a altura da camera(y) de acordo com a sua posição(x,z). Para tal acedemos ao array de alturas(HeightData).

```
...
cameraLookAt = cameraPosition + direction;
this.worldMatrix = rotation * Matrix.CreateTranslation(cameraPosition);
this.viewMatrix = Matrix.CreateLookAt(cameraPosition, cameraLookAt, Vector3.Up);
...
```

Para optimizar o movimento da câmara e a tornar mais fluida, recorremos a métodos de interpolação linear para calcular o valor de y médio entre os 4 vetores vizinhos onde a câmera se encontra a cada update.



Criamos um método dedicado a calcular os valores de y acedendo ao array de alturas para comparar os 4 vértices vizinhos em que a posição da câmera está em qualquer momento. Começando por calcular as coordenadas do vértice “a”.

```
...
int x = (int)cam.cameraPosition.X;
int z = (int)cam.cameraPosition.Z;
...
```

Determinamos as alturas para os diferentes vértices.

```
...
float ya = heightData[x,z];
float yb = heightData[x+1,z];
float yc = heightData[x,z+1];
float yd = heightData[x+1, z-1];
...
```

Determinamos as distâncias da posição da câmera aos 4 vértices por X e por Z.

```
...
float daX = cam.cameraPosition.X - x;
float dbX = 1 - daX;
float dcX = daX;
float ddX = 1 - dcX;
float daZ = cam.cameraPosition.Z - z;
float dcZ = 1 - daZ;
...
```

Determinamos o valor das alturas entre ab e cd multiplicando os pesos, calculados pelas distâncias aos vértices pela coordenada X.

Finalmente podemos multiplicar estas alturas pelos pesos, calculados pela distância aos vértices pela coordenada Z.

```
...
float yab = (dbX * ya) + (daX * yb);
float ycd = (ddX * yc) + (dcX * yd);
float yfinal = yab * dcZ + ycd * daZ;
...
```

Por fim fazemos um update à posição da câmara na coordenada Y(altura) com um offset +5 para que a câmara se mantenha acima do nível do terreno.

```
...
cam.cameraPosition.Y = yfinal + 5;
...
```

Esta função é então chamada na update da game. Atualizando a cada update/movimento.

```
...
terrain.Update(cam, GameTime);
...
```

Game1

Tendo as classes criadas e estruturadas, partimos para a implementação das mesmas no jogo. Para tal, fomos à class Game1 e inicializamos a nossa câmara e terreno.

```
...
cam = new ClsCamara (this,GraphicsDevice);
terrain = new GameTerreno (GraphicsDevice, Content, cam);
...
```

No metodo **Update()** vamos atualizando a posição da câmara , com a assistência dos valores.

```
...
input = Keyboard.GetState();
msInput = Mouse.GetState();
cam.Update(input,msInput);
...
```

No metodo **Draw()** o terreno é desenhado com a devida perspetiva de câmara surface follow.

```
...
terrain.Draw(GraphicsDevice, cam);
...
```

Conclusão

Através desta primeira fase do projecto, aproveitamos muitos conhecimentos da aula que nos ajudou a acimentar a materia lecionada.

A feramenta MonoGame é bastante apelativa e permitiu-nos explorar as frameworks mais detalhadamente.

O desenvolvimento do terreno obrigou-nos a explorar a lógica posicional dos diferentes vértices assim como a melhor forma de criar os indices, que usando triangle strips para o rendeer nos permitiu reduzir o tamanho do index buffer. O posicionamento da textura também levou-nos a perceber melhor como criar repetições e espelhar a textura.

Este trabalho foi bastante útil para perceber como a camara interage com os diferentes objetos no espaço, sendo que a view matrix passa a ser geral a todos os objetos renderizados pela camara, assim como as diferentes rotações, inclinalões e translações a criar para que a camara se movimente com o efeito pretendido.

Ao desenvolver este projecto muitas barreiras foram ultrapassadas, umas com mais dificuldades que outras, mas no final conseguimos cumprir os requezitos pedidos. Agora esperanos a segunda fase para podermos enriquecer este projecto.