Universidade Federal de Santa Maria
Departamento de Computação Aplicada
Prof. Cesar Tadeu Pozzer
Disciplina: Computação Gráfica Avançada
Autores: Flavio Franzin, Gabriel Backes
pozzer@inf.ufsm.br
19/11/2018

Unity Shader Tutorial

1. Motivação

A Unity é uma engine gráfica com muitas facilidades que não são encontradas em APIs mais básicas como OpenGL e Direct3D. Ela possui diversas ferramentas e praticidades que aumentam a produtividade. Ao trabalhar com engines gráficas, o usuário usufrui de uma interface amigável que facilita e agiliza diversas atividades.

Dentre todas as facilidades, a principal diferença refere-se ao processo de renderização de um modelo (*mesh*), que em OpenGL utiliza buffers como VAO e VBO. Em níveis superiores, esses buffers ainda existem, porém, são abstraídos por componentes primitivos da engine.

Em OpenGL 4.x, também é necessário a utilização de shaders escritos em GLSL para instruir como a GPU irá manipular os dados fornecidos pelos buffers. Na Unity, os shaders são escritos numa variante da linguagem HLSL. Internamente, são utilizados diferentes compiladores para a compilação dos shaders. Com isso, a Unity permite a compilação de shaders para diversas plataformas, sem a necessidade de reescrita dos shaders.

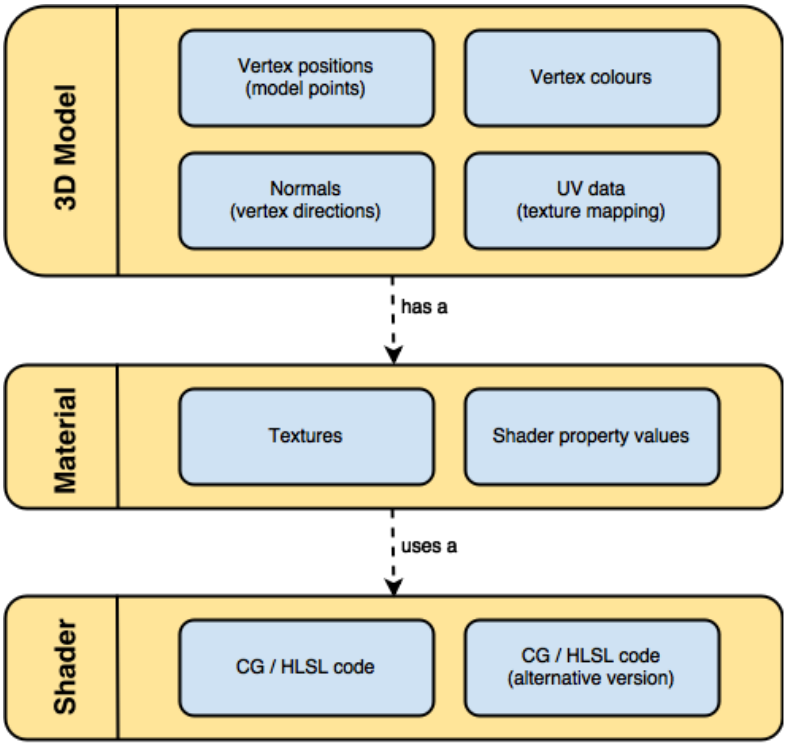
Estas são apenas algumas das funcionalidades que a Unity pode fornecer, mas agilizam drasticamente diversas tarefas que quando feito em APIs mais básicas podem demandar mais tempo do programador.

Este documento fornece o básico de programação de shader com Unity. Inicialmente é abordado conceitos sobre Shader, incluindo o Standard Shader (shader padrão da Unity). Na sequência é feito um estudo sobre Custom Shaders, com foco nos estágios de Vertex e Fragment Shader. O documento também demonstra como criar um VBO e vinculá-lo com os componentes da Unity.

Shader Unity

Uma forma de visualizar algo na Unity é através de materiais associados a Meshes 2D/3D. Estes materiais têm seu comportamento definido por um shader, e nos fornecem um controle que vai desde a manipulação de vértices do objeto até a etapa final de visualização.

A Unity, por Default, já possui alguns shaders incorporados que, na grande maioria das vezes, satisfazem as necessidades de seus usuários, pois são extremamente otimizados e flexíveis. No entanto, aplicações mais específicas exigem do programador a adaptação ou a criação de novos shaders.



Standard Shader

O Standard Shader é atualmente o principal shader da Unity, com diversas aplicações na texturização de objetos comuns. O principal fator que o torna altamente utilizável é a robustez, pois este suporta diversos fatores que melhoram o visual dos objetos durante a renderização, tornando-os mais realistas. Além disso, o shader não está preso a estas informações, ou seja, a não utilização de algumas destas não restringem o uso do mesmo. As informações que caracterizam o standard shader são:

- Albedo map
- Normal map
- Specular map
- Occlusion map
- Metallic map
- Emission map

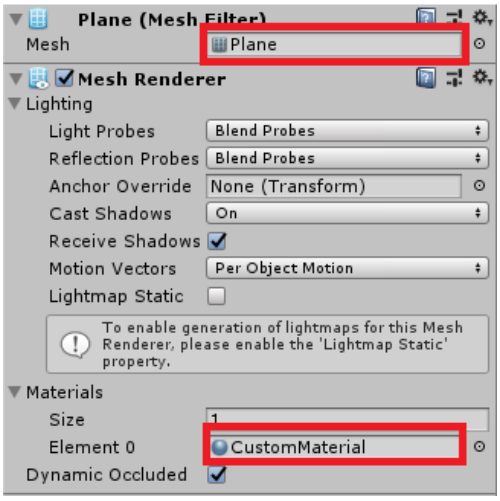
Os ótimos resultados obtidos pelo Standard Shader caracterizam-se pelo modelo de iluminação baseado em Physically Based Shading (PBS), que permite a geração de materiais mais condizentes com a realidade. Isso ocorre porque são consideradas todas as características listadas acima, e que impactam diretamente no resultado entre a interação da luz e o material do objeto que está sendo renderizado. Além disso, o Standard Shader permite renderizar

objetos de diferentes formas, sendo elas: *opaque, cutout, transparent e fade. (Standard Shader - Render Mode)*
Com isso fica evidente a gama de materiais que pode-se criar somente utilizando o standard shader da unity.

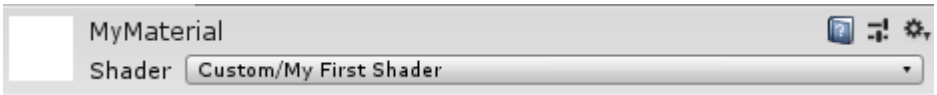
Customs Shaders

Por mais robusto que o Standard Shader seja, nem sempre ele cobre as necessidades de aplicações mais específicas. Dessa forma, a unity permite a criação de Customs Shaders, que nos possibilitam receber vários tipos de informações, e a utilização das mesmas da forma que desejarmos.

Inicialmente é necessário o VBO, que pode ser criado manualmente (apresentado na seção *Criando Uma Mesh*) ou através do uso de primitivas da Unity. Nesse tutorial usaremos a segunda opção, fazendo uso de um plano. Ao inserirmos o plano na cena é possível notar 2 componentes em especial: o *MeshFilter* e o *MeshRenderer*. O *MeshFilter* é quem possui a referência para mesh que será renderizada pelo *MeshRenderer* utilizando o material com o nosso shader.



Para a criação do shader é importante entendermos a anatomia dele, ou seja, compreender as etapas do pipeline gráfico na qual podemos manipular através do nosso shader. Para isso, acessamos a aba Asset/Create/Shader/UnlitShader e criamos um shader. É possível notar que ele já possui alguns trechos de código, mas para o nosso tutorial partiremos do zero para entendermos o porquê de cada linha de código. Na sequência devemos criar um material e associá-lo ao shader criado, no nosso caso ele se chama “My First Shader” e está dentro da aba “Custom”, como é definido no código abaixo.



No trecho de código abaixo também podemos notar a primeira divisão, que é entre as *Properties* e o *SubShader*. As *Properties* são como campos *public* dos arquivos *C#*, ou seja, todo campo definido dentro desse bloco será exposto no inspector do Material que estiver esse shader associado. No bloco *SubShader* é onde programamos as etapas do pipeline. Também é possível criar variações de um shader através dos SubShader, por exemplo, um SubShader com maior nível de detalhamento para desktops e outro mais simples e que necessite menos processamento para ser usado em smartphones.

```
Shader "Custom/My First Shader" {
        Properties{
                /*    */
        }
        SubShader {
                /*    */
        }
}
```

Dentro de um SubShader é necessário definirmos a *Pass*. Nesse bloco é onde definimos as funções que irão tratar de cada etapa dentro do pipeline gráfico. Por exemplo, a etapa de Vertex Shader e Fragment Shader. Logo, é esse bloco que fará com que nosso objeto seja renderizado.

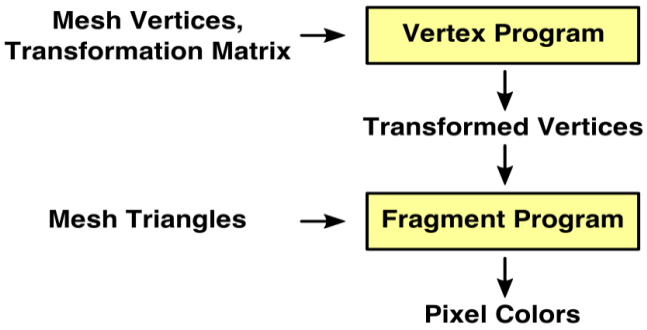
Podemos definir também várias Pass, o que significa que o objeto será renderizado mais de uma vez, o que é muitas vezes necessário para aplicar efeitos.

```
Shader "Custom/My First Shader" {
        Properties{/*    */}

        SubShader {

                Pass {

                }
        }
}
```

Para escrever o shader é utilizado a Unity's shading language, que é uma variante das linguagens HLSL e CG. Assim, utilizamos as keywords CGPROGRAM e ENDCG para definir o começo e fim do bloco de código.

```
Pass {
        CGPROGRAM

        ENDCG
}
```

Shaders são construídos sob sub-programas, cada um representando um estágio do pipeline gráfico. O *vertex program* é responsável por processar os dados dos vértices de uma mesh. O *fragment program* é responsável por colorir pixels individuais que se encontram dentro dos triângulos dessa mesh. Além destes, outros podem ser adicionados, por exemplo, o estágio de tessellation.



Para isso é feito a definição da função, com o respectivo nome escolhido e relacionado com o estágio que esta representa, por exemplo, o *MyVertexProgram* refere-se ao estágio de *vertex.* Para que a Unity localize essas funções elas devem ser precedidas da diretiva *#pragma.*

```
Pass {
        CGPROGRAM

        #pragma vertex MyVertexProgram
        #pragma fragment MyFragmentProgram

        void MyVertexProgram () {
        }

        void MyFragmentProgram () {
        }

        ENDCG
}
```

A ideia de pipeline é que cada estágio receba uma entrada e após ser processada seja produzido uma saída. Para isso, devemos indicar no escopo da função o retorno, que no caso de *Vertex Program* é a coordenada final de um vértice e no *Fragment Program* é uma cor de um pixel. Ambos são representados por um float4 (o mesmo que Vector4 do C#).

Também temos que compreender o conceito de semântica. A ideia por trás da semântica é explicitar/indicar uma determinada função ou valor. Para exemplificar podemos notar a keyword **SV_POSITION** (**S**ystem **V**alue **POSITION**) no estágio de vertex shader, que representa a posição final de um vértice. Também notamos a keyword **SV_TARGET (S**ystem **V**alue **TARGET)** no estágio de fragment que indica onde será escrito a cor de um determinado pixel**.** Shader Semantics

```
Pass {
        CGPROGRAM

        #pragma vertex MyVertexProgram
        #pragma fragment MyFragmentProgram

        float4 MyVertexProgram () : SV_POSITION{
        }

        float4 MyFragmentProgram (float4 position : POSITION) : SV_TARGET {
        }

        ENDCG
}
```

Como foi comentado anteriormente, o pipeline requer que um estágio receba uma entrada e retorne uma saída. Assim, a saída de um estágio deve ser associada a entrada de outro, por exemplo, a saída do estágio de *vertex program* e a entrada do *fragment program*. Nesse contexto, o *vertex program* prove informação para o *fragment program*. Logo, os parâmetros de saída de um função deve ser equivalente ao de entrada do outro. Além disso, é comum passarmos mais de uma informação entre estágios. Por isso, na grande maioria das vezes são utilizadas structs para troca de informações entre estágios. Atente-se novamente para o uso de semântica para definir a posição do vértice e da uv.

```
Pass {
        CGPROGRAM

        #pragma vertex MyVertexProgram
        #pragma fragment MyFragmentProgram
```

```
        struct appdata
        {
                float4 vertex : POSITION;
                float2 uv : TEXCOORD0;
        };

        struct v2f /*Vertex to Fragment*/
        {
                float2 uv : TEXCOORD0;
                float4 vertex : SV_POSITION;
        };

        v2f MyVertexProgram (appdata v) : SV_POSITION{
        }

        float4 MyFragmentProgram (v2f i) : SV_TARGET {
        }

        ENDCG
}
```

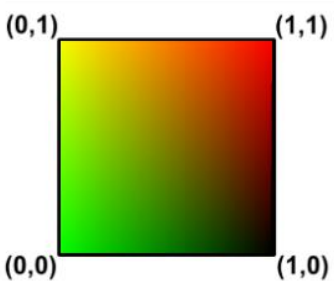
Customs Shaders - Transformação entre espaços

Um conceito fundamental que devemos ter ao trabalhar com shader é a definição de espaço. Em computação gráfica o espaço mais usado e intuitivo é o *World Space* que representa a posição de um vértice no mundo virtual. No entanto, além deste existem outros, cada qual com um propósito diferente. Toda coordenada contida em um determinado espaço pode, através de multiplicação de matrizes, ser convertida para outros espaços.
Dito isso, é importante compreendermos em qual espaço estamos trabalhando e se necessário convertê-lo para o espaço que necessitamos. No caso da Unity, o nosso VBO é fornecido pelo MeshRender e os vértices da mesh são definidos em *Object Space*. Porém, para que possamos visualizar algo é necessário que os vértices estejam em *Screen Space,* e com isso sendo necessário que façamos a conversão *Object Space -> Screen Space*. Para isso usamos a função *mul*, passando como parâmetro a matriz MVP (model, view e projection) e a coordenada do vértice. Como retorno obtemos uma coordenada em *scree space.*

```
v2f MyVertexProgram (appdata v) : SV_POSITION{
        v2f o;
        o.vertex = mul(UNITY_MATRIX_MVP, v.vertex);
        o.uv = TRANSFORM_TEX(v.uv, _MainTex);
        return o;
}
```

Customs Shaders - Texturização

Após a transformação de coordenadas entre espaços, o objeto está devidamente pronto para ser renderizado. Para iniciarmos a texturização e já verificarmos se está tudo correto com o que definimos anteriormente podemos fazer um teste de texturização apenas utilizando a UV (coordenadas utilizadas no mapeamento de textura) do objeto. O teste consiste em definimos uma cor com base na posição da UV, e considerando que o retorno do MyFragmentProgram como um fixed4(Red, Green, Blue, Alpha) a interpolação que iremos obter será entre as cores Red e Green.




```
v2f MyVertexProgram (appdata v) : SV_POSITION{
	/*	*/
}

float4 MyFragmentProgram (v2f i) : SV_TARGET {
	return fixed4(i.uv.x, i.uv.y, 0, 1); /* ou simplismente fixed4(i.uv,0,1);*/
}
```

Após essa etapa é possível começar a trabalhar com o shader do material, adicionando informações aos vértices e à textura. Inicialmente trabalharemos com o estágio de fragment. Para isso, a primeira etapa é a definição da textura dentro do bloco *Properties.* Uma vez que foi definido os recursos desejados dentro do *Properties,* é necessário definir estes mesmos recursos dentro do bloco *Pass*. Esta redefinição deve ser feita com os mesmos nomes de variáveis.

Após definida, a textura pode ser associada ao material através do inspector. O acesso à mesma é feito através da UV do vértice e da função *tex2D*, passando a textura na qual desejamos fazer uma amostra.

```
Shader "Custom/My First Shader" {
	Properties{
		_Tint ("Tint", Color) = (1, 1, 1, 1)
		_MainTex ("Texture", 2D) = "white" {}
	}
```

```
        SubShader {

            Pass {
                sampler2D _MainTex;

                v2f MyVertexProgram (appdata v) : SV_POSITION{
                /*    */
                }

                float4 MyFragmentProgram (v2f i) : SV_TARGET {
                        float4 color = tex2D(_MainTex, i.uv);
                        return color;
                }
            }
        }
}
```

A etapa de Fragment é uma mescla entre programação e trabalho artístico, muitas vezes sendo gasto bastante tempo sobre essa etapa. Dito isso, abordaremos um exemplo simples, sem nos aprofundar muito nessa etapa uma vez que ela é bem repetitiva e varia conforme a aplicação.

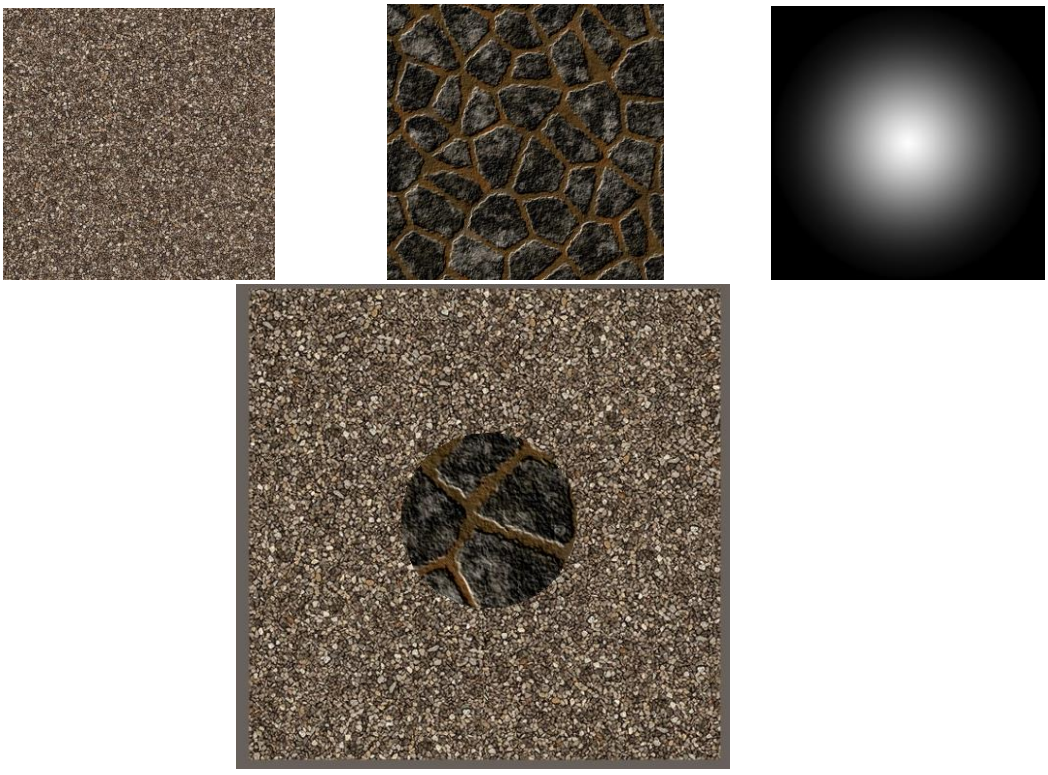
Para o exemplo que iremos trabalhar será necessário o uso de 2 texturas e uma máscara (qualquer textura representada em tons de cinza). Uma vez que temos essas três texturas devemos defini-las em nosso shader (no bloco properties e dentro do Pass, como visto anteriormente). No estágio de fragment iremos fazer *samples* nas duas texturas e na máscara. Dependendo do valor da máscara iremos colocar uma texturaA ou a texturaB.

```
float4 MyFragmentProgram (v2f i) : SV_TARGET {
        fixed4 colorA = tex2D(_TexA, i.uv);
        fixed4 colorB = tex2D(_TexB, i.uv);
        float mask = tex2D(_Mask, i.uv).r;

        fixed4 finalColor = fixed4(1,1,1,1);

        if(mask < 0.5)
                finalColor = colorA;
        else
                finalColor = colorB;

        return finalColor;
}
```

Também podemos utilizar um valor definido no bloco properties, e modificá-lo através do inspector do material.

```
Properties{
        _AlphaInfluence ("AlphaInfluence", float) = 1
}

float4 MyFragmentProgram (v2f i) : SV_TARGET {
        ……
        if(mask < 0.5)
        if(mask < _AlphaInfluence)
        ….
}
```

Como resultado devemos obter uma mescla da textura A (a esquerda) e B (o centro) baseado em nossa máscara (a direita).



Um aspecto que devemos considerar é a transição entre as texturas, pois muitas vezes buscamos algo suave, diferentemente do que ocorre no resultado obtido acima. Por isso, é comum o uso de interpolações, como definido no trecho de código abaixo juntamente com o respectivo resultado obtido.

Abaixo vemos a interpolação linear entre duas texturas baseado no valor da máscara. Atente-se também ao uso da função *saturate*, que faz o clamp de valores para 0 e 1. Ela pode ser removida, mas resultará cores diferentes das texturas A e B, dependendo do valor de *_AlphaInfluence*.

```
float4 MyFragmentProgram (v2f i) : SV_TARGET {
	fixed4 colorA = tex2D(_TexA, i.uv);
	fixed4 colorB = tex2D(_TexB, i.uv);
	float mask = tex2D(_Mask, i.uv).r;

	return  lerp(colorA, colorB, saturate(mask * _AlphaInfluence));
}
```

Customs Shaders - Vertex

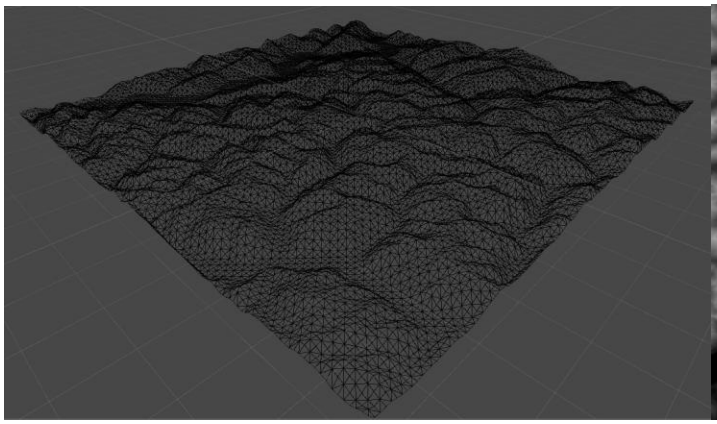
É comum termos que fazer a edição dos vértices dentro do shader. Essa etapa é comumente utilizada para a criação de terrenos, onde a altura do vértice é definida por uma textura, nesse caso chamada de heightmap. A criação de um terreno via shader requer o conhecimento de fatores como a resolução (escala de um pixel em escala de mundo, por exemplo, 1px = 1m) e recursos como tessellation e geometry shader, para a criação de novos vértices. No entanto, para o nosso tutorial será feito o displacement dos vértices baseando-se em um heightmap, sem nos aprofundarmos em questões mais específicas de geração de terrenos.

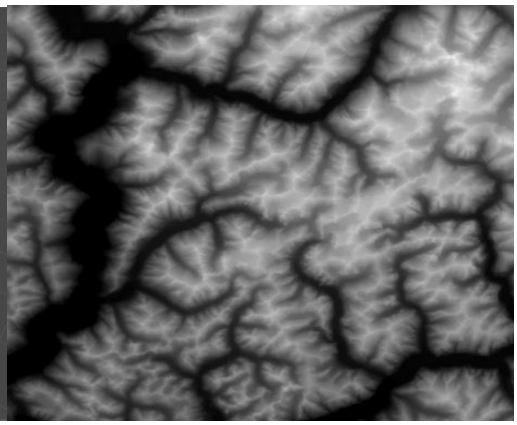
Para fazermos o displacement iremos usar uma mesh em formato de grid e dentro do shader iremos aplicar um offset em y. A mesh pode ser gerada via código (como mostrado na seção Criando uma Mesh) ou através de um modelo 3D.

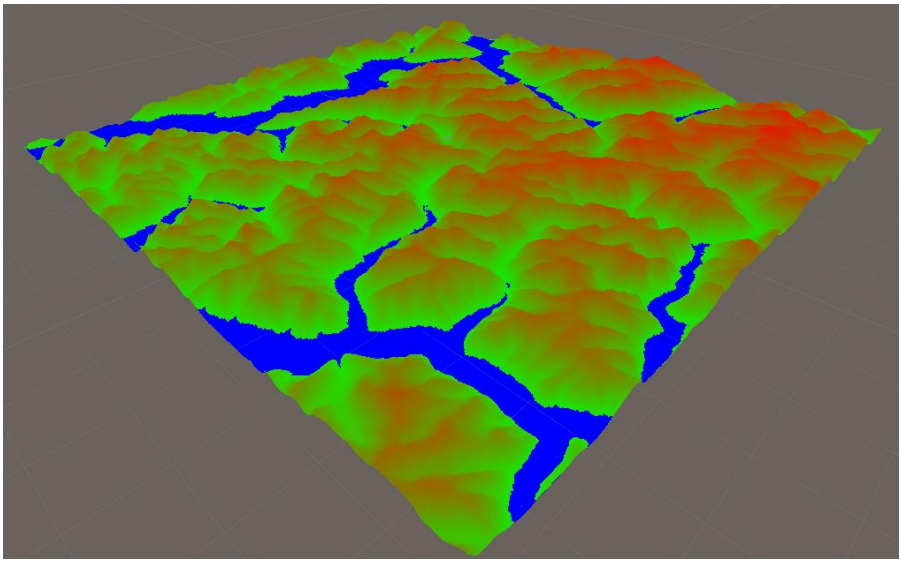
Assumindo que a mesh já está pronta, será feito o displacement dos vértices no estágio de vertex shader. Para isso, será necessária uma textura representando nosso heightmap (definido no bloco properties, como visto anteriormente). Após a definição da textura é feita uma amostragem, dentro do estágio de vertex, para cada vértice no heightmap e definido a altura deste com base nesse valor amostrado juntamente com um multiplicador (considerando que amostra está definida entre 0-1).

A texturização também pode ser feita com base no heightmap, apenas interpolando cores entre tons de Red e Green ou totalmente azul para alturas inferiores a um threshold.

```
v2f MyVertexProgram (appdata v) : SV_POSITION{
	v2f o;

	float h = tex2Dlod(_HeightMap, float4(v.uv, 0, 0)).r * K;
	v.vertex.y += h;

	o.vertex = mul(UNITY_MATRIX_MVP, v.vertex);
	o.uv = TRANSFORM_TEX(v.uv, _MainTex);
	return o;
}

float4 MyFragmentProgram (v2f i) : SV_TARGET {
	float h = tex2D(_HeightMap, i.uv).r;

	if(h < 0.1)
		return float4(0,0,1,1);
	return lerp(float4(0,1,0,1), float4(1,0,0,1), h);
}
```







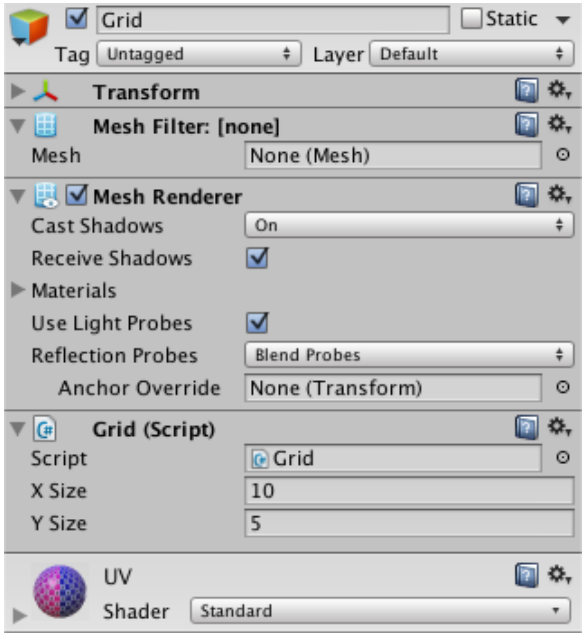
Criando uma *mesh*

É comum em determinadas ocasiões não usarmos modelos 3D, mas sim criamos as nossas próprias meshes 3D. Um caso muito comum onde é necessário essa criação é na geração de terrenos. Dito isso, iniciamos o estudo de um estágio muito importante que é a criação de VBOs, que na grande maioria das vezes fica abstraído pela Engine.

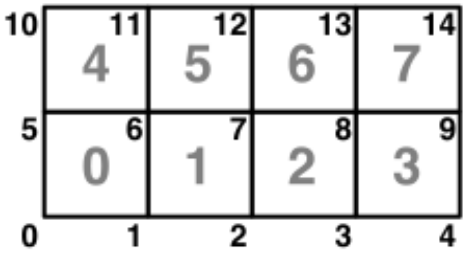
Para criar uma *mesh*, podemos representar primeiramente uma *grid* retangular. Utilizando um script C#, podemos definir o tamanho horizontal e vertical da *grid*. Como visto anteriormente, é necessário que o objeto tenha os componentes *mesh filter* e um *mesh renderer*. Na Unity, podemos utilizar o atributo *RequireComponent*. Este faz com que os componentes sejam adicionados automaticamente a um objeto (como visto no código abaixo).

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(MeshFilter), typeof(MeshRenderer))]
public class Grid : MonoBehaviour {

        public int xSize, ySize;
}
```

Criando um novo objeto e adicionando o script a este, veremos que todos os componentes vão estar criados.



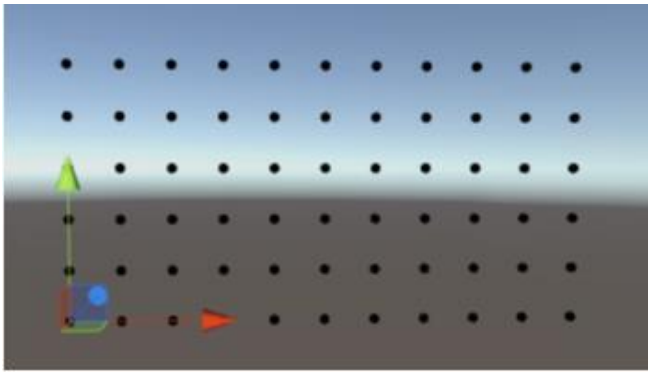
A *grid* pode ser representada por uma série de Vector3 para armazenar seus pontos. A quantidade de vértices dependerá do tamanho da *grid*. Cada *quad* da *grid* é representado por 4 vértices, entretanto os *quads* adjacentes compartilham o mesmo vértice. Sendo assim, o tamanho final do vetor é de um vértice a mais do que o número de *quad* em cada dimensão, ou seja, (#x+1)(#y+1). A criação do array de vértices pode ser observada no código abaixo.




```
	private Vector3[] vertices;

	private void Generate () {
		vertices = new Vector3[(xSize + 1) * (ySize + 1)];
		for (int i = 0, y = 0; y <= ySize; y++) {
			for (int x = 0; x <= xSize; x++, i++) {
				vertices[i] = new Vector3(x, y);
			}
		}
	}
```

Para visualizar estes vértices é possível utilizar o método *OnDrawGizmos* para desenhar uma pequena esfera preta na visão de cena para cada vértice.

```
	private void OnDrawGizmos () {
		if (vertices == null) {
			return;
		}
		Gizmos.color = Color.black;
		for (int i = 0; i < vertices.Length; i++) {
			Gizmos.DrawSphere(vertices[i], 0.1f);
		}
	}
```



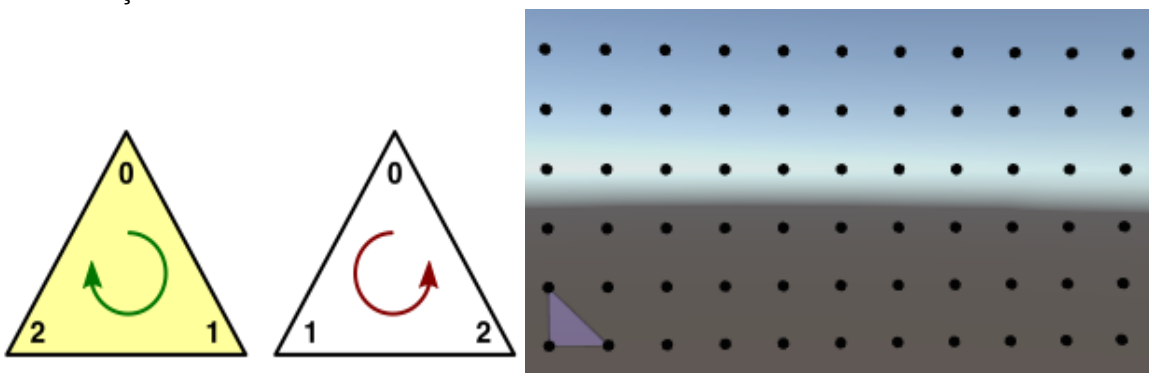
Com o posicionamento correto da *grid*, podemos lidar com a *mesh* real. Para isso, criamos e associamos os vértices criados a uma nova mesh. Além disso, para que a mesh seja renderizada também é necessário definirmos a mesh dentro do component MeshFilter.

```
	private Mesh mesh;

	private void Generate () {
	…
		GetComponent<MeshFilter>().mesh = mesh = new Mesh();

		mesh.vertices = vertices;
	}
```

Para que possamos visualizar a *mesh* é necessário configurar primeiramente os seus triângulos. Os triângulos são definidos via um array de vértices. Cada triângulo tem três pontos, assim, três consecutivos índices representam um triângulo.

```
	private void Generate () {
		…

		int[] triangles = new int[3];
		triangles[0] = 0;
		triangles[1] = xSize + 1;
		triangles[2] = 1;
		mesh.triangles = triangles;
	}
```

Até o momento nossa mesh já está praticamente toda definida, faltando apenas um ponto a ser configurado: a orientação de seus índices dos vértices. Por padrão, se eles estiverem dispostos no sentido horário, o triângulo é considerado virado para a frente e visível. Triângulos anti-horários são descartados, eliminando a parte interna dos objetos e otimizado o tempo de renderização.



Pode ser utilizado um loop duplo para criar toda a *grid*, iterando os índices de vértices e triângulos. O código abaixo representa a configuração da orientação de cada triângulos da *grid*.

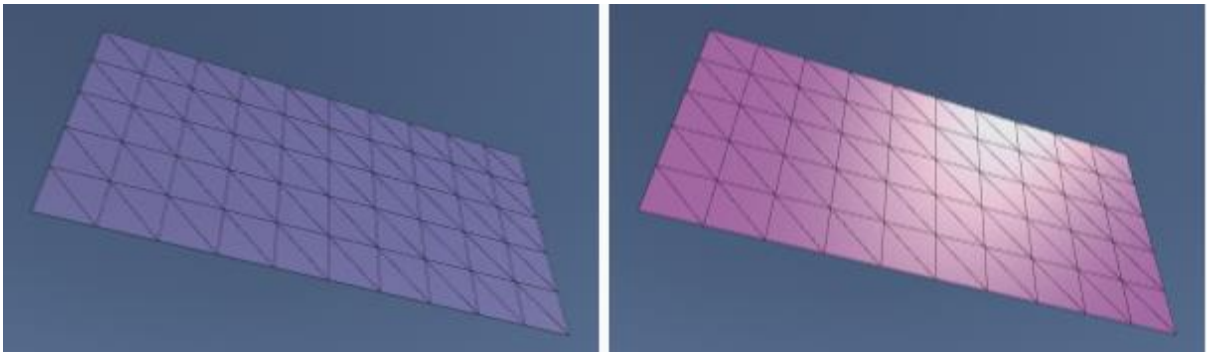
```
	private void Generate () {
		….
		int[] triangles = new int[xSize * ySize * 6];
		for (int ti = 0, vi = 0, y = 0; y < ySize; y++, vi++) {
			for (int x = 0; x < xSize; x++, ti += 6, vi++) {
				triangles[ti] = vi;
				triangles[ti + 3] = triangles[ti + 2] = vi + 1;
				triangles[ti + 4] = triangles[ti + 1] = vi + xSize + 1;
				triangles[ti + 5] = vi + xSize + 2;
			}
		}
		mesh.triangles = triangles;
	}
```

Gerando dados adicionais para os vértices

A iluminação da *mesh* é dada por suas normais. Entretanto, a direção normal padrão é (0,0,1). As normais são definidas pelo vértice, então é necessário a utilização de outro vetor de Vector3. O Mesh Filter contém um método que calcula as próprias normais com base em seus triângulos.

```
	private void Generate () {
```

```
		…
		mesh.triangles = triangles;
		mesh.RecalculateNormals();
	}
```



Existem também as **coordenadas UV**. A *grid* atualmente tem uma cor uniforme, mesmo que use um material com uma textura de albedo. Isto porque não são fornecidas as coordenadas UV, estas ainda contém os valores zerados.

Na *grid*, para configurar as coordenadas UV é necessário dividir a posição do vértice pelas dimensões da mesma. Assim, a textura será projetada sobre toda sua extensão. O resultado será de valores entre 0 e 1.

```
		vertices = new Vector3[(xSize + 1) * (ySize + 1)];
		Vector2[] uv = new Vector2[vertices.Length];
		for (int i = 0, y = 0; y <= ySize; y++) {
			for (int x = 0; x <= xSize; x++, i++) {
				vertices[i] = new Vector3(x, y);
				uv[i] = new Vector2((float)x / xSize, (float)y / ySize);
			}
		}
		mesh.vertices = vertices;
		mesh.uv = uv;
```

Uma maneira de adicionar detalhes mais aparentes a uma superfície é usar um **normal mapping**. Estes mapas contêm vetores normais codificados como cores. Aplicá-los a uma superfície resultará em efeitos de luz mais detalhados. Para isso devemos recalcular, após a geração dos vértices, as normais e tangentes da mesh.

```
private void Generate () {
		...
		mesh.uv = uv;
		mesh.RecalculateTangents();
```

```
        mesh.RecalculateNormals();
}
```

Conclusão

Nesse material abordamos conhecimentos básicos sobre shader na Unity, com enfoque nos estágios de vertex e fragment shader. Também abordamos a criação de uma mesh via código, comumente feito na geração de terrenos procedurais. Basicamente buscou-se explorar como texturizar objetos através de Customs Shaders bem como a manipulação de vértices. Inicialmente, este tutorial parece simples e sem muitas aplicações, no entanto, toda estrutura de shader se comporta da mesma forma. Ainda falando sobre a simplicidade, shaders são estruturas bastante simples, sua maior complexidade dá-se pelo nível de abstração, uma vez que temos que criar uma lógica que será executada para diversos vértices e pixels.