# OpenTK - Renderizando Triângulos

**O que é um triângulo?**

Um triângulo é uma figura formada pelo ligamento de 3 pontos (vértices). Mas eu aposto que você já sabia disso. Então, para criarmos um triângulo, primeiro precisamos definir seus vértices:

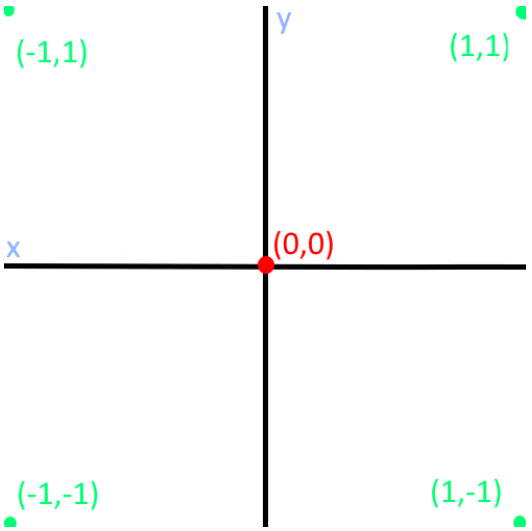
**Definindo os vértices:**

```
// criando um novo array
float[] vertices =
{
        0f, 0,5f, 0f, // Vértice 1 (Topo)
        0.5f, -0.5f, 0f, // Vértice 2 (Inferior direito)
        -0.5f, -0.5f, 0f // Vértice 3 (Inferior esquerdo)
}
```

Aqui criamos um array que contém todos os 3 pontos, ordenados pela sequência x, y, z.

Agora, imagine que o seu monitor (retangular) seja um plano cartesiano, onde a origem é o centro. Cada um dos 4 cantos da tela representa uma coordenada normalizada, ou seja, com valores variando entre -1, e 1, obrigatoriamente.

O canto superior esquerdo é (-1,1), o inferior direito é (1, -1), e o resto você já consegue adivinhar. *(-1,-1) inf esq / (1,1) sup dir.




Depois de definirmos os vértices do triângulo, que nesse caso terá metade do tamanho da tela, pois especificamos que os vértices estão a 0.5 unidades da origem, o que é exatamente metade, agora precisamos passar essa informação para a nossa *Placa de Vídeo*, e faremos isso através de um *Vertex Buffer*.

**Passando dados para a placa de vídeo por um Vertex Buffer:**

Vamos criar primeiramente um int que conterá o valor do buffer:

```
private int vertexBuffer;
```

Geramos então um Buffer que irá definir os valores da variável que nós criamos, podemos fazer isso pelo seguinte método:

```
// Gerando um buffer e atribuindo seu valor ao int
vertexBuffer = GL.GenBuffer();
```

Depois de gerar o buffer, precisamos atribuí-lo para depois passar os valores para a nossa placa de vídeo:

```
// Atribuindo o buffer
GL.BindBuffer(BufferTarget.ArrayBuffer, vertexBuffer);

// Passando os dados para a placa
GL.BufferData(BufferTarget.ArrayBuffer, vertices.Length * sizeof(float), vertices, BufferUsageHint.StaticDraw);
```

Nessa função primeiramente passamos o buffer alvo, depois o tamanho dos dados em bytes (vértices em bytes), depois o valor em si (vértices), e por fim o uso do buffer, definimos StaticDraw pois não ficaremos alterando seu valor, e sim o deixaremos estático.

Podemos agora desvincular o buffer, já que já o usamos:

```
GL.BindBuffer(BufferTarget.ArrayBuffer, 0); // Vincular a 0 = desvincular
```

**Criando um Vertex Shader:**

Depois, temos que criar um vertex shader, e faremos isso usando GLSL (GL Shader Language). Há duas formas de fazer isso, ou podemos programar o código em uma string formatada dentro do C#, ou criamos um arquivo separado e o integramos em nossa aplicação através de um StreamReader, o que não mostrarei aqui já que está fora do conteúdo deste pdf.

```
// especificando a versão
#version 330 core

// definindo um vetor3 da posição
layout (location = 0) in vec3 aPosition; /* esse location = 0 é para podermos identificar a variável pelo seu índice no c# */

// criando o método principal
void main
{
	gl_Position = vec4(aPosition, 1.0);
}
```

**Criando um Pixel Shader:**

```
#version 330 core

// criamos uma variável do tipo vetor4, que especificará uma cor, essa cor será
// passada para todos os pixels dos vértices

out vec4 pixelColor;
void main()
{
	// passamos a cor em r,g,b,a
	pixelColor = vec4(0,0,1,1); // definindo a cor azul
}
```

**Compilando os Shaders:**

Voltando para o C#, temos que dizer ao OpenGL que queremos criar pixel shaders, e depois que queremos compilar os pixel shaders, vamos fazer isso então:

```
int vertexShader = GL.CreateShader(ShaderType.VertexShader)
GL.ShaderSource(vertexShader,  LoadShaderFile("VertShader.vert")); /* obs: aqui estou pegando o código do shader através de uma função StreamReader que lê o meu arquivo de shader, você pode simplesmente referenciar a string formatada que mencionei anteriormente */

GL.CompileShader(vertexShader);
```

faremos o mesmo com o pixelShader:

```
// criando o shader do tipo PixelShader
int pixelShader = GL.CreateShader(ShaderType.PixelShader)
// vinculando o shader com um código de shader
GL.ShaderSource(pixelShader, LoadShaderFile("PxShader.vert"))
// compilando o shader
GL.CompileShader(pixelShader);
```

**Criando o ShaderProgram:**

Depois de criarmos e compilamos os shaders, precisamos juntá-los em um só Programa de Shader, vamos fundir os 2 criando uma única entidade.

Vamos começar definindo o int do shaderProgram:

```
private int shaderProgram
```

Agora, seguindo os mesmos passos de antes, vamos criar o ShaderProgram:

```
shaderProgram = GL.CreateProgram();

GL.AttachShader(shaderProgram, vertexShader);
GL.AttachShader(shaderProgram, pixelShader);

GL.LinkProgram(shaderProgram);
```

Criaremos também um int para o vertexArray, e o configuramos dentro da função OnLoad, junto com os outros:

```
private int vertexArray; // fora da função

vertexArray = GL.GenVertexArray();
GL.BindBuffer(BufferTarget.ArrayBuffer, vertexBuffer);
GL.VertexAttribPointer(0, 3, VertexAttribPointerType.Float, false, 0, 0);
GL.EnableVertexAttribArray(0);
```

Como fundimos os shaders no programa, podemos nos livrar deles para economizar recursos:

```
GL.BindVertexArray(0);
GL.DetatchShader(shaderProgram, vertexShader);
GL.DetatchShader(shaderProgram, pixelShader);
GL.DeleteShader(vertexShader);
GL.DeleteShader(pixelShader);
```

**Desenhando os vértices:**

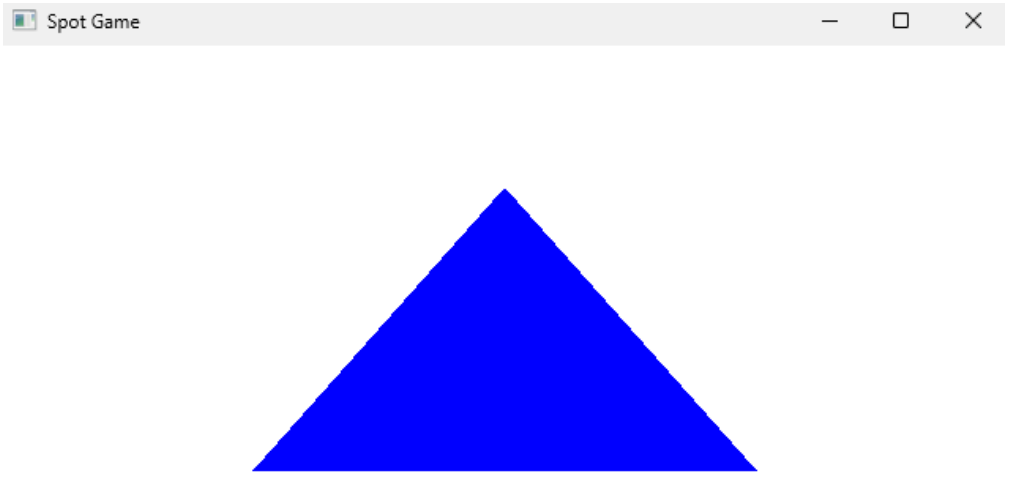
Depois de concluir essas etapas, podemos finalmente começar a desenhar os vértices, faremos isso na função OnRenderFrame nativa do OpenTK:

```
        protected override void OnRenderFrame(FrameEventArgs e)
        {
          GL.Clear(ClearBufferMask.CollorBufferBit);
          GL.UseProgram(shaderProgram);
         GL.DrawArrays(PrimitiveType.Triangles, 0, 3); /* Um triângulo, então 3
vértices */


          Context.SwapBuffers();
          base.OnRenderFrame(e);

      }
```

**Um triângulo!**

Ao implementar tudo corretamente, você já deve ser capaz de rodar o código e ver um triângulo na sua tela, que ocupa exatamente metade da janela. Caso o triângulo não tenha sido propriamente carregado, verifique se todos aqueles códigos onde criamos e vinculamos os shaders e vértices esteja no método
correto (Ele tem que ser chamado quando o programa inicia, de preferência no próprio OnLoad). Também certifique-se de que o código que desenha o triângulo esteja propriamente colocado no método OnRenderFrame, que irá constantemente atualizar a cena.

Você deve ver algo assim:




E é assim que se renderiza um triângulo usando OpenTK. Realmente impressionante!

**Resumo para os preguiçosos:**

O primeiro passo para se renderizar um triângulo é definir os vértices (pontos) do mesmo em um array de floats.
Depois, é necessário criar e atribuir um VertexBuffer em um int, que será usado para transmitir os dados à GPU.

Com os dados no VertexBuffer, deve-se carregar e compilar os shaders básicos necessários para se renderizar o triângulo, e depois criar um ShaderProgram, que é a união dos dois shaders.

Além do ShaderProgram, é nesserário criar um VertexArray com o VertexBuffer do tipo BufferTarget.Array criado anteriormente.

Depois, basta desenhar na tela o triângulo usando GL.DrawArrays na função de renderização!

***Importante: Depois de criar e usar os buffers, é necessário desatribuí-los para liberar memória!**

**Source Code**

https://docs.lucasof.com/tutorials/opentk/Triangle.cs