

# Desenho de Sprites

## Programação de Jogos

### Introdução

- ▶ Um **Sprite** é uma imagem que compõe uma cena 2D



Shank

# Introdução



## ► Os Sprites são constituídos a partir de **imagens**:

- **Opacas**: cores sólidas sem transparência, normalmente usadas como pano de fundo da cena
- **Com transparência**: cada pixel contém um quarto componente indicando o grau de transparência



pixels sem transparência



pixels transparentes

**Atenção, imagens com transparência não tem apenas o RGB**, pois para existir a transparência precisa existir o quarto componente (imagem acima).

# Carregando Imagens



## ► As imagens precisam ser **carregadas do disco**

- O DirectXTK carrega uma imagem em uma Textura D3D usando WIC
  - O Windows Imaging Component é um framework que suporta a manipulação de imagens nos principais formatos: TIFF, JPG, PNG, GIF, BMP e HDPhoto

```
// cria shader resource view da imagem em disco
D3D11CreateTextureFromFile(
    Graphics::Device(),           // dispositivo Direct3D
    Graphics::Context(),         // contexto do dispositivo
    filename.c_str(),             // nome do arquivo de imagem
    nullptr,                     // habilita retorno da textura
    &textureView,                // retorna view da textura
    width,                       // retorna largura da imagem
    height);                     // retorna altura da imagem
```

O **DirectXTK** é um projeto (open source) da Microsoft para **simplificar** o uso de todas as versões do DirectX, uma dessas formas é a **criação de funções básicas**. Uma dessas funções básicas é, por exemplo, essa facilidade mostrada na última imagem acima, usada **para carregar imagens a partir do disco rígido** que é carregada em uma textura do Direct3D.

O **DirectXTK** tem versões específicas para cada versão do DirectX.

A imagem abaixo é um zoom na função do **DirectXTK** sendo usada no nosso projeto (imagem acima).

```
// cria shader resource view da imagem em disco
D3D11CreateTextureFromFile(
    Graphics::Device(),           // dispositivo Direct3D
    Graphics::Context(),         // contexto do dispositivo
    filename.c_str(),            // nome do arquivo de imagem
    nullptr,                     // habilita retorno da textura
    &textureView,                // retorna view da textura
    width,                       // retorna largura da imagem
    height);                     // retorna altura da imagem
```

Linha 7 - **Não** está retornando o endereço da textura, apenas a **view**.

O que é a “**view da textura**”? É um **registro** que contém **informações** para a Direct3D/GPU de como **ACESSAR** e **USAR** a textura.

---

Resumo <https://youtu.be/sVXwZuLeejs?t=547>

Usaremos essa função PRONTA do **DirectXTK** citada, para realizar o CARREGAMENTO DE TEXTURAS, ou melhor dizendo, CARREGAMENTO DE IMAGENS **EM** TEXTURAS DO **DIRECT3D**.

---

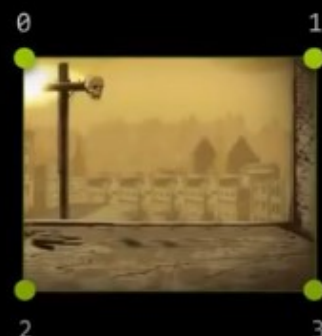
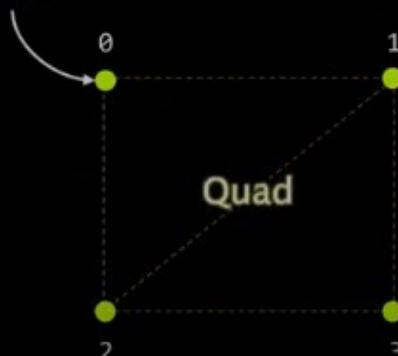
## Desenho de Sprites

- ▶ Para desenhar **Sprites no Direct3D** é preciso:
  - Criar um Quad no espaço tridimensional (vértices e índices)
  - Aplicar uma textura aos vértices

```
struct Vertex
{
    XMFLOAT3 pos;
    XMFLOAT4 color;
    XMFLOAT2 tex;
};
```

Quad = (0, 1, 2, 1, 3, 2)

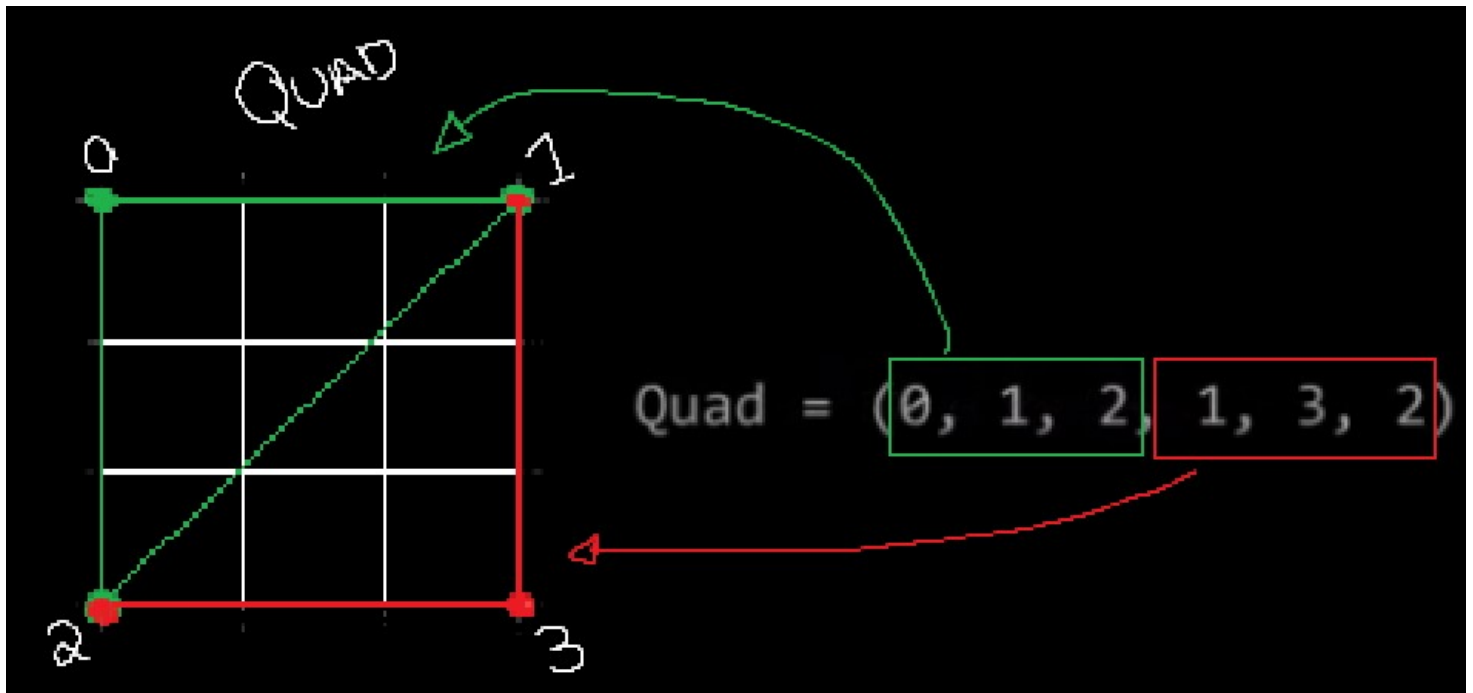
Vertex



As placas de vídeo **apenas suportam** o 3D, pois é o suficiente para usar o 2D. Embora o curso seja para a criação de um jogo 2D, **usaremos recursos 3D**:  
**Por isso que a imagem acima diz** “Criar um **Quad** no espaço tridimensional”.

Um **Quad** é uma **figura quadrilátera** (quadrado), para criá-lo precisamos definir os vértices (Vertex), assim como mostra a imagem acima.

Note as propriedades de cada vértice que estão definidas dentro da estrutura **Vertex**, na primeira linha dentro do **Vertex** está definido o **XMFLOAT3** que é a posição 3D (x, y, z) que o vértice se encontra no “universo 3D”.



]

Nota: O hardware trabalha polígonos, mais especificamente TRIÂNGULOS.

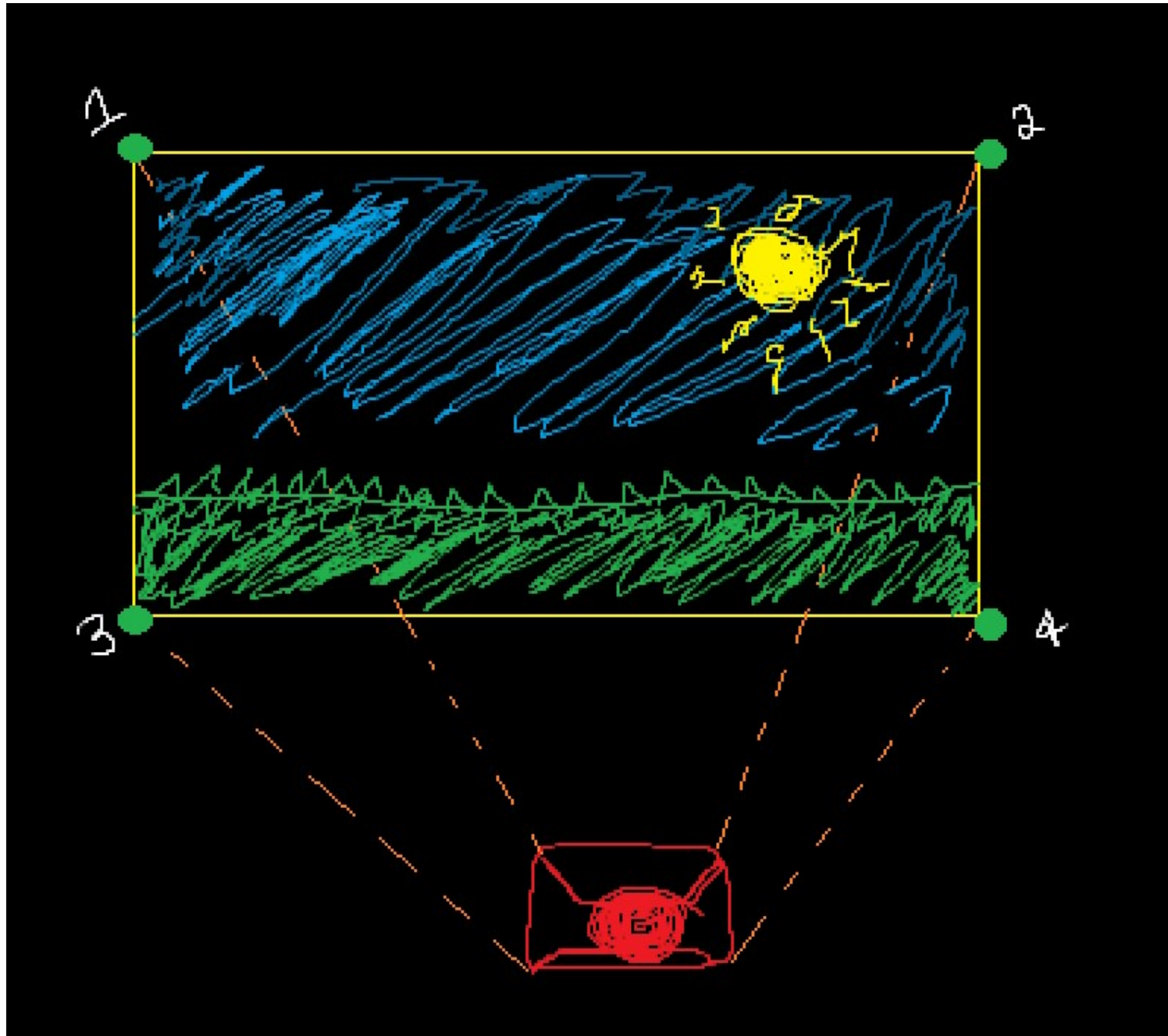
Nota: O **QUAD** é formado por triângulos, como mostra a imagem acima.

Para facilitar o entendimento, pense em uma textura 2D flutuando no meio da escuridão. A escuridão representa o “universo 3D”.

Para imaginar como vemos isso, pense em uma câmera acompanhando essa textura.

A imagem abaixo exemplifica isso.





Descrição da imagem acima:

Essa figura em **amarelo** é a textura e seus vértices funcionando dentro do universo 3D.

Essa figura **vermelha** é a câmera apontando para a textura 2D.

Basicamente o que faremos é, criar o **quad** composto por triângulos, mapearmos uma textura para as posições dos vértices e depois colocarmos na tela.

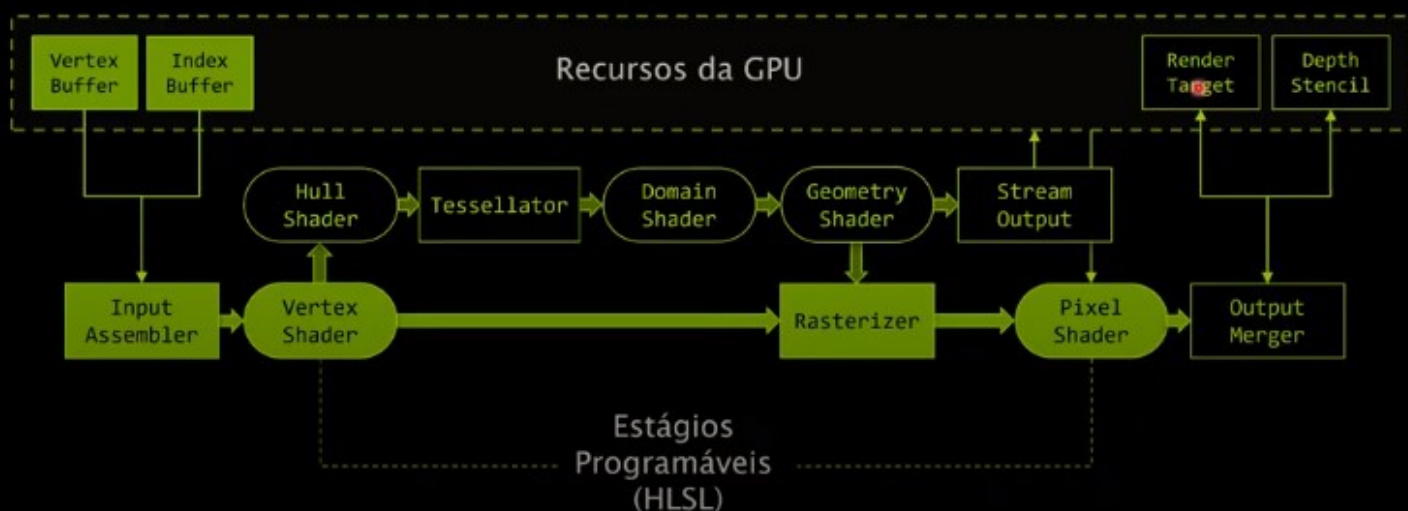
---

Na imagem abaixo estamos relembrando o processo do **pipeline** para identificar os próximos passos a partir de agora.

# Desenho de Sprites



- ▶ O Sprite passa pelo **Pipeline do Direct3D**
  - Vários estágios precisam ser configurados:



Lembrando que anteriormente havíamos configurado uma **Swap Chain** para a troca de buffers (**frontbuffer** e **backbuffer**) o qual teria seu conteúdo criado através da **Render Target**.

**Pergunta:** Como faremos para colocar o nosso **Quad** e texturas para o **Pipeline**?

**Resposta:**

1º - **Vertex Buffer** - Adicionar os vértices na memória.

2º - **Input Assembler** - Trata da posição dos vértices e informa **isso** ao Direct3d.

3º - **Vertex Shader** - Posiciona os vértices **DENTRO DA CENA**. O nosso Vertex Shader que criaremos será bem básico.

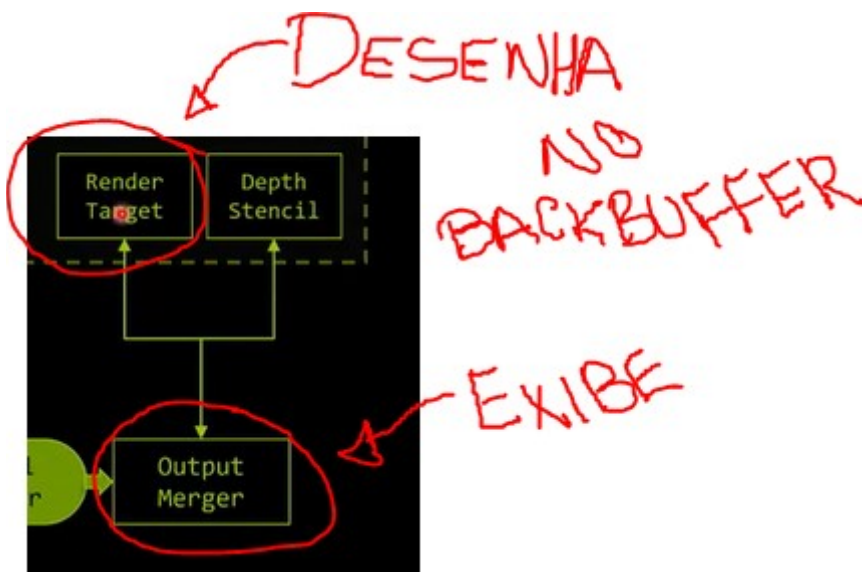
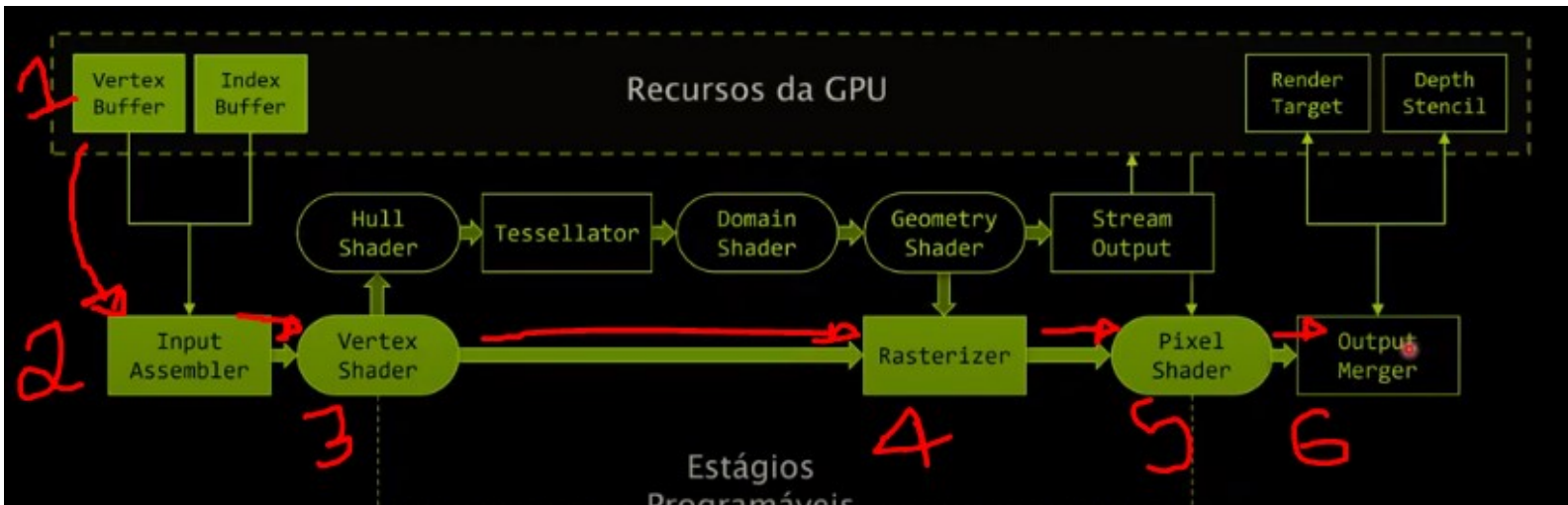
4º - **Rasterizer** Capta os vértices (que por enquanto estão no mundo 3D) e joga eles em um **PLANO**. Que também faz a pintura da **área limitada** pelos vértices (colorização),



mas que nesse caso ele só vai preencher essa pintura com a **textura** que escolhermos.

5º - **Pixel Shader** Cada pixel passará pelo processo. É nessa parte que podemos adicionar um efeito, um filtro. Na aula, o filtro que colocaremos é para redimensionar do tamanho que quisermos.

6º - A imagem é gerada e jogada no Backbuffer (Output Merger).



<https://youtu.be/sVXwZuLeejs?t=960> Criação dos Buffers

## Input Layout

- Um Vertex Buffer é um vetor de vértices

```
D3D11_BUFFER_DESC vertexBufferDesc = { 0 };
vertexBufferDesc.ByteWidth = sizeof(Vertex) * VerticesPerSprite * MaxBatchSize;
vertexBufferDesc.Usage = D3D11_USAGE_DYNAMIC;
vertexBufferDesc.BindFlags = D3D11_BIND_VERTEX_BUFFER;
vertexBufferDesc.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;
graphics->Device()->CreateBuffer(&vertexBufferDesc, nullptr, &vertexBuffer);
```

- Um Index Buffer é um vetor de índices

```
D3D11_BUFFER_DESC indexBufferDesc = { 0 };
indexBufferDesc.ByteWidth = sizeof(short) * IndicesPerSprite * MaxBatchSize;
indexBufferDesc.Usage = D3D11_USAGE_DEFAULT;
indexBufferDesc.BindFlags = D3D11_BIND_INDEX_BUFFER;
graphics->Device()->CreateBuffer(&indexBufferDesc, &indexData, &indexBuffer);
```

Primeiro precisa definir um buffer de vértices. Não será abordado sobre o que faz todas as linhas. Abaixo temos a função da segunda linha:

```
vertexBufferDesc.ByteWidth = sizeof(Vertex)* VerticesPerSprite * MaxBatchSize;
```

O **VerticesPerSprite** trata do número de vértices que têm cada sprite (lembre-se do Quad). **MaxBatchSize** é a quantidade máxima de sprites que o Direct3D suporta (atualmente é 4096 mas é possível expandir até 32.768 (exatamente do tamanho em memória de um tipo **short**)) por cena.

```
indexBufferDesc.ByteWidth = sizeof(short)* IndicesPerSprite * MaxBatchSize;
```

Então a nossa engine vai suportar, por padrão, 4096 Sprits, e só.

---

O uso dos vértices é dinâmico, pois a cada frame em que os objetos se movem na cena, a posição deles muda (imagem abaixo).

```
vertexBufferDesc.Usage = D3D11_USAGE_DYNAMIC;
```

Já quando falamos de índices, esses não mudam, são valores fixos, são os rótulos dos vértices.

Nota: A criação de vértices fica na memória de vídeo, não na memória ram. Como os **sprites** estão se movimentando na tela, é preciso atualizá-los para que isso aconteça. Então novos vértices substituirão os antigos vértices (**igual uma animação**). Para que isso aconteça, o método de subscrição deverá estar habilitado para que a CPU faça os desenhos, como mostra a imagem abaixo.

```
vertexBufferDesc.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;
```

Qual a diferença fundamental entre **IndexBuffer** e o **VertexBuffer**?  
**VertexBuffer**, por tratar de desenhos, seu conteúdo pode ser alterado.

**IndexBuffer**, por se tratar da identificação em memória, seu conteúdo não é alterado. Sempre que um novo triângulo é criado, segue a sequência numérica, veja o exemplo abaixo:  
vértice 1,2,3 == triângulo / vértice 4,5,6 == triângulo(2) / vértice 7,8,9 == triângulo(3)

O registro abaixo mostra como bloquear o método de escrita da CPU no Index Buffer. É uma forma de otimizar a velocidade de execução da engine, pois a tarefa da placa de vídeo é apenas leitura, sem interferência da CPU.

```
indexBufferDesc.Usage = D3D11_USAGE_DEFAULT;
```



# Input Layout



## ► Um Input Layout descreve os vértices

```
// descreve o input layout dos vértices
D3D11_INPUT_ELEMENT_DESC layoutDesc[] =
{
    { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 }
};

// cria o input layout
graphics->Device()->CreateInputLayout(layoutDesc, 3, vShader->GetBufferPointer(), vShader->GetBufferSize(), &inputLayout);
```

- O layout recebe também um ponteiro para o buffer do **Vertex Shader**
  - Contém código a ser executado sobre cada vértice

Até agora colocamos os vértices em um Buffer. Falta agora informar qual é o layout (imagem acima e abaixo), que é o **conteúdo** dentro da forma 'formada' pelos vértices, que é o conteúdo de seu preenchimento.

```
{ "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 },
{ "COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 },
{ "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, D3D11_APPEND_ALIGNED_ELEMENT, D3D11_INPUT_PER_VERTEX_DATA, 0 }
```

R32R32 etc. Informa que o valor é **float** e de 32 bits.

```
{ "POSITION", 0,
{ "COLOR", 0,
{ "TEXCOORD", 0,
```

(imagem acima) Passa as informações das propriedades do vértice:

**Position** é a posição do vértice.

**Color** trata da cor do vértice.

**TexCoord** é a coordenada da textura.

```
// cria o input layout
graphics->Device()->CreateInputLayout(layoutDesc, 3, vShader->GetBufferPointer(), vShader->GetBufferSize(), &inputLayout);
```

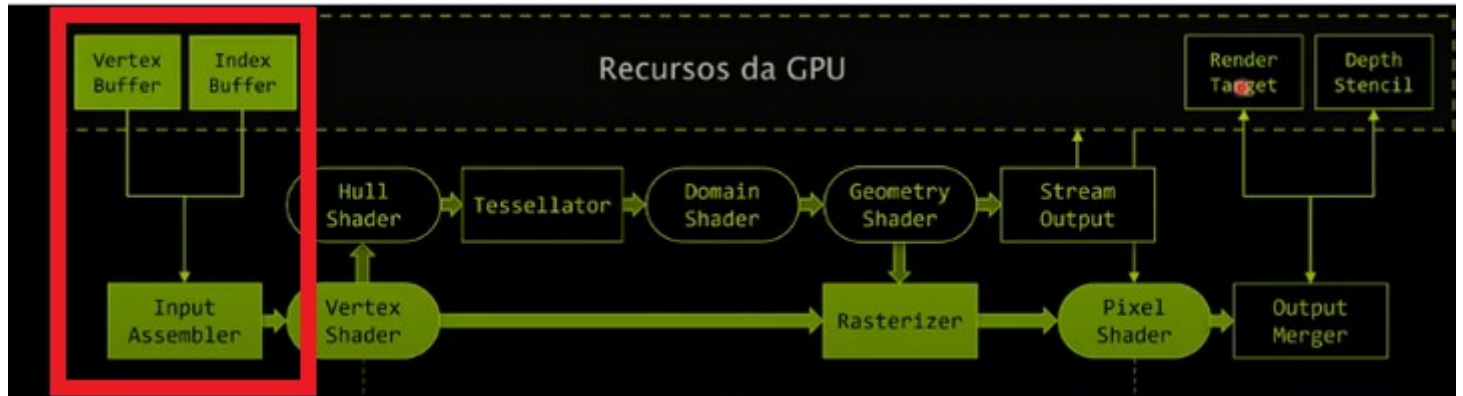
Informa ao Direct3d qual é o formato do layout com base nas informações das últimas 3 imagens. É como se fosse uma **instância**.

- O layout recebe também um ponteiro para o buffer do **Vertex Shader**
  - Contém código a ser executado sobre cada vértice

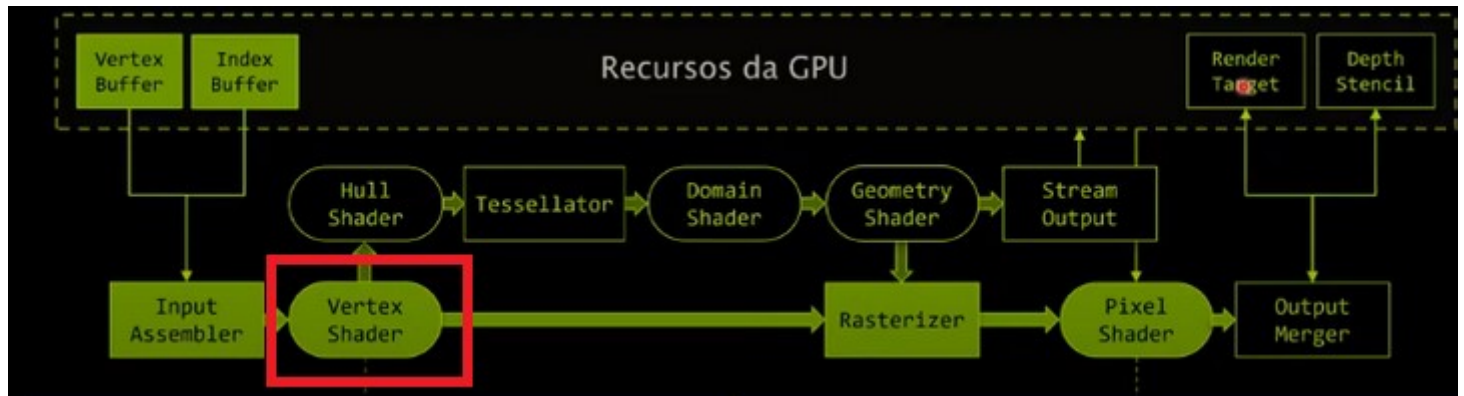
Vertex Shader é aquele outro estágio do pipeline. Para o desenho existir, essa passa a ser outra parte importante do fluxo.

[https://www.youtube.com/watch?v=sVXwZuLeejs&t=1232s&ab\\_channel=JudsonSantiago](https://www.youtube.com/watch?v=sVXwZuLeejs&t=1232s&ab_channel=JudsonSantiago)

Informação: Executamos a criação da estrutura para a criação dos vértices e também a re-organização deles (Vertex Buffer e Index Buffer / Input Assembler).



Próximo passo: Vertex Shader.



**Vertex Shader**, o que é?

Posiciona os vértices **DENTRO DA CENA**. O nosso Vertex Shader que criaremos será bem básico. Ele contém o código que será executado **nos** vértices. Então precisamos criá-lo.

# Shaders

## ▶ Vertex Shader

```
cbuffer ConstantBuffer
{
    float4x4 WorldViewProj;
}

struct VertexIn
{
    float3 Pos    : POSITION;
    float4 Color  : COLOR;
    float2 Tex    : TEXCOORD;
};

struct VertexOut
{
    float4 Pos    : SV_POSITION;
    float4 Color  : COLOR;
    float2 Tex    : TEXCOORD;
};

VertexOut main( VertexIn vIn )
{
    VertexOut vOut;

    // transforma vértices para coordenadas da tela
    vOut.Pos = mul(float4(vIn.Pos, 1.0f), WorldViewProj);

    // mantém as cores inalteradas
    vOut.Color = vIn.Color;

    // mantém as coordenadas da textura inalteradas
    vOut.Tex = vIn.Tex;

    return vOut;
}
```

A imagem acima mostra a criação de um Vertex Shader bem simples. A direita é a criação da **main**, que recebe um vertex na entrada da função:

```
VertexOut main( VertexIn vIn )
```

E retorna um vertex na saída (imagem abaixo):

```

VertexOut main( VertexIn vIn )
{
    VertexOut vOut;

    // transforma vértices para coordenadas da tela
    vOut.Pos = mul(float4(vIn.Pos, 1.0f), WorldViewProj);

    // mantém as cores inalteradas
    vOut.Color = vIn.Color;

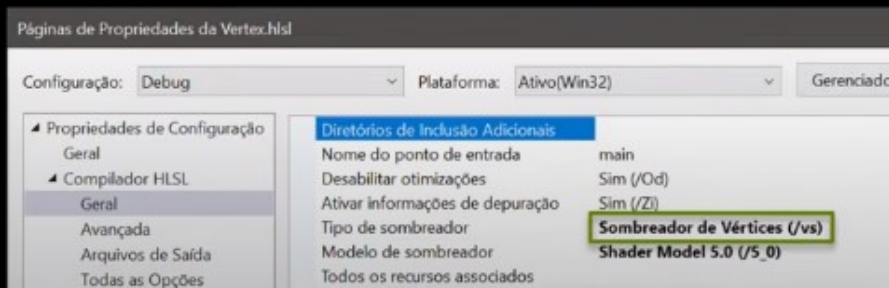
    // mantém as coordenadas da textura inalteradas
    vOut.Tex = vIn.Tex;

    return vOut;
}

```

# Shaders

- ▶ O **Vertex Shader** é um programa de alto nível
  - Executado pela placa gráfica
  - Precisa ser compilado
    - Compilador disponível em d3dcompiler.lib
    - Gera arquivo Vertex.cso



Arquivo precisa ser configurado como um **Sombreador de Vértices**

O **Vertex Shader** é executado pela placa gráfica, **não** pela CPU. O projeto dependerá de bibliotecas para serem configuradas no Visual Studio. Também lidaremos com a configuração de outro estágio do **pipeline**, que é o Pixel Shader.



# Shaders

## ► Pixel Shader

```
Texture2D resource;

SamplerState linearfilter
{
    Filter = MIN_MAG_MIP_LINEAR;
};

SamplerState anisotropic
{
    Filter = ANISOTROPIC;
    MaxAnisotropy = 4;
};

struct pixelIn
{
    float4 Pos    : SV_POSITION;
    float4 Color  : COLOR;
    float2 Tex    : TEXCOORD;
};

float4 main(pixelIn pIn) : SV_TARGET
{
    return resource.Sample(linearfilter, pIn.Tex) * pIn.Color;
}
```

- Diferentes filtros podem ser aplicados sobre as texturas

O **Vertex Shader** realiza uma operação sobre vértices

O *Pixel Shader* recebe um pixel na função e devolve um pixel (última imagem acima)

```
float4 main(pixelIn pIn) : SV_TARGET
{
    return resource.Sample(linearfilter, pIn.Tex) * pIn.Color;
}
```

(imagem acima)

Em vermelho: Pega a textura do pixel

Em azul: Aplica um filtro linear

Em branco: São a cor dos pixels de entrada. A cor da textura será baseada nela, então se alteramos uma textura para a cor verde, através desse método, o resultado será esverdeado:



## ► Pixel Shader

```
Texture2D resource;  
  
SamplerState linearfilter  
{  
    Filter = MIN_MAG_MIP_LINEAR;  
};  
  
SamplerState anisotropic  
{  
    Filter = ANISOTROPIC;  
    MaxAnisotropy = 4;  
};
```

### MIN\_MAG\_MIP\_LINEAR;

É um filtro linear, aumenta e diminui a imagem de **forma linear**

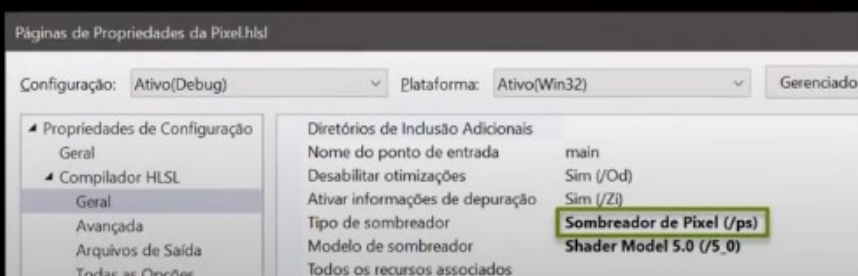
### Anisotropic

É notável em texturas que começa perto do jogador e se estendem conforme o tamanho. Por exemplo, uma rua. Veja como o BORRÃO some quando é usada.



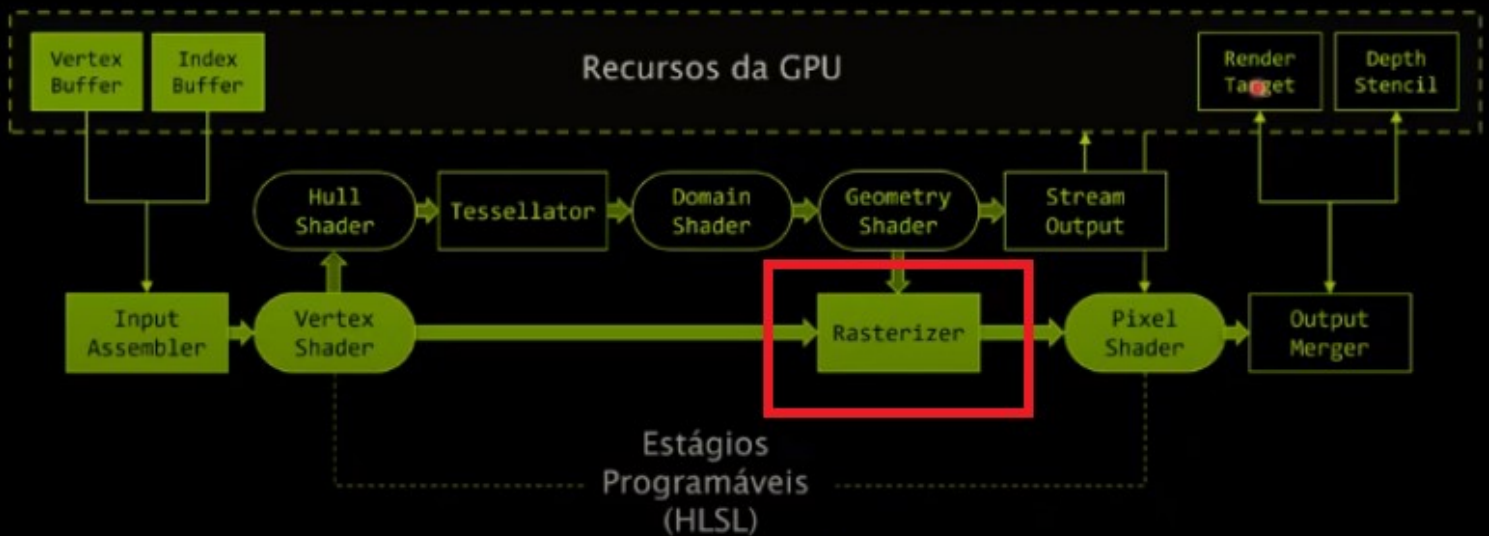
## Shaders

- O **Pixel Shader** é um programa de alto nível
  - Executado pela placa gráfica
  - Precisa ser compilado
    - Compilador disponível em d3dcompiler.lib
    - Gera arquivo **Pixel.cso**



Arquivo precisa ser configurado como um **Sombreador de Pixel**

# Rasterizer



# Rasterizer

## ► O rasterizador preenche triângulos

```
D3D11_RASTERIZER_DESC rasterDesc = {0};
rasterDesc.FillMode = D3D11_FILL_SOLID;
//rasterDesc.FillMode = D3D11_FILL_WIREFRAME;
rasterDesc.CullMode = D3D11_CULL_NONE;
rasterDesc.DepthClipEnable = true;

// cria estado do rasterizador
graphics->Device()->CreateRasterizerState(&rasterDesc, &rasterState);
```

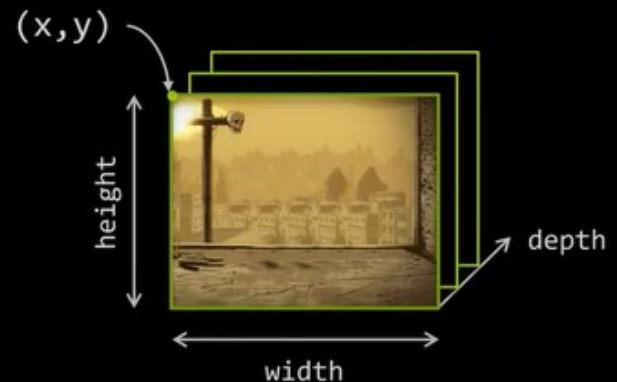
- Transforma vértices no espaço 3D para pixels na tela
  - Elimina partes não visíveis da cena
    - Culling
    - Clipping

# Renderizador de Sprites



- ▶ Para simplificar criaremos uma classe **Renderer**
  - Ela recebe e armazena as informações dos Sprites
  - Desenha um conjunto de sprites na tela

```
struct SpriteData
{
    float x, y;
    float depth;
    uint width;
    uint height;
    ID3D11ShaderResourceView* texture;
};
```



# Renderizador de Sprites

- ▶ Configurando Sprite e enviando para desenho

```
// configura registro sprite
spriteData.x = x;
spriteData.y = y;
spriteData.scale = 1.0f;
spriteData.depth = z;
spriteData.rotation = 0.0f;
spriteData.width = image->Width();
spriteData.height = image->Height();
spriteData.texture = image->View();

// envia sprite para ser desenhado
Renderer::Draw(&spriteData);
```



# Renderizador de Sprites

- ▶ Os Sprites **não são desenhados imediatamente**
  - Eles são adicionados em um vetor

```
void Renderer::Draw(SpriteData * sprite)
{
    spriteVector.push_back(sprite);
}
```

- O vetor em seguida é:
  - Ordenado por profundidade
  - Agrupado por textura

```
RenderBatch(batchTexture, &spriteVector[batchStart], pos - batchStart);
```

`spriteVector.push_back(sprite)` joga o **sprite** para o final do **vetor** (eis o **parallax**).

---

## Código da aula (abaixo)

```
SpriteDemo.cpp  SpriteDemo
SpriteDemo
1  // *****
2  // SpriteDemo
3  //
4  // Criação: 09 Mai 2014
5  // Atualização: 14 Ago 2021
6  // Compilador: Visual C++ 2019
7  //
8  // Descrição: O programa carrega e desenha imagens opacas e transparentes,
9  //             demonstrando como usar as classes Image e Sprite com os novos
10 //             recursos de desenho de Sprites da classe Renderer
11 //
12 // *****
13
14 #include "Engine.h"
15 #include "Game.h"
16 #include "Image.h"
17 #include "Sprite.h"
18 #include "Resources.h"
19 class SpriteDemo : public Game
20 {
21 private:
22     Sprite * backg = nullptr; // sprite do fundo de tela
23     Sprite * shank = nullptr; // sprite do personagem
24
25     Image * logoImg = nullptr; // imagem do logotipo
26     Sprite * logol = nullptr; // sprite 1 do logotipo
27     Sprite * logo2 = nullptr; // sprite 2 do logotipo
28
29     float x = 0, y = 0; // posição x,y do shank
30
31 public:
32     void Init();
33     void Update();
34     void Draw();
35     void Finalize();
36 };
37 void SpriteDemo::Init()
38 {
39     backg = new Sprite("Resources/Background.jpg");
40     shank = new Sprite("Resources/Shank.png");
41
42     logoImg = new Image("Resources/Logo.png");
43     logol = new Sprite(logoImg);
44     logo2 = new Sprite(logoImg);
45
46     // Posição do personagem na tela
47     x = 80.0f;
48     y = 90.0f;
49 }
50 void SpriteDemo::Update()
51 {
52     // sai com o pressionar do ESC
53     if (window->KeyDown(VK_ESCAPE))
54         window->Close();
55
56     // desloca personagem
57     if (window->KeyDown(VK_LEFT))
58         x -= 50.0f * gameTime;
59     if (window->KeyDown(VK_RIGHT))
60         x += 50.0f * gameTime;
61     if (window->KeyDown(VK_UP))
62         y -= 50.0f * gameTime;
63     if (window->KeyDown(VK_DOWN))
64         y += 50.0f * gameTime;
65 }
```

Está na pasta resource

gameTime é baseado no Timer (contador de tempo que fizemos na aula passada)

**Pergunta:** Por que mesmo estamos usando a variável gameTime (contador de tempo) para atualizar os movimentos do sprite na tela?

**Resposta:** Porque, baseado no tempo da realidade, os sprites se moverão em uma velocidade igual em qualquer computador que esse jogo esteja funcionando.

Para descobrir esse tempo, usamos uma função (explicada na aula passada) que converte o **tempo de clock** em um tipo de um relógio.

```

void SpriteDemo::Finalize()
{
    // remove sprites da memória
    delete backg;
    delete shank;
    delete logo1;
    delete logo2;

    // remove imagem da memória
    delete logoImg;
}

```

No final, os sprites e imagens são apagados da memória.

Ainda no mesmo arquivo, aqui é definido a profundidade dos sprites, eis o parallax.

```

66
67 // -----
68
69 void SpriteDemo::Draw()
70 {
71     backg->Draw(0.0f, 0.0f, Layer::BACK);
72     shank->Draw(x, y);
73     logo1->Draw(40.0f, 60.0f, Layer::UPPER);
74     logo2->Draw(400.0f, 450.0f, Layer::LOWER);
75 }
76
77 // -----
78

```

**backg** é o fundo.

**shank** é o personagem.

**logo1** é a logo.

**logo2** é a logo.



Informação extra:



O **backg**, **shank** foi feito através do carregamento de uma imagem em um sprite (carregado no sprite).

O **logo1** e **logo2**, primeiro foi carregado uma imagem **E DEPOIS** foi criado um sprite a partir dessa imagem. E por qual razão fizemos isso? Porque isso economiza memória.

<https://youtu.be/sVXwZuLeejs?t=2729>

Explicação: Na classe sprite temos essas duas opções de como carregar imagens, se o objetivo for economizar memória é possível referenciar a imagem com um ponteiro, dessa forma não precisando criar um sprite e ocupar ainda mais recursos, eles apenas apontam para onde está a imagem.

**Foi isso que fizemos no logo1 e logo2.**

Pense em uma situação com 10 sprites, o quanto de memória economizamos com essa técnica.



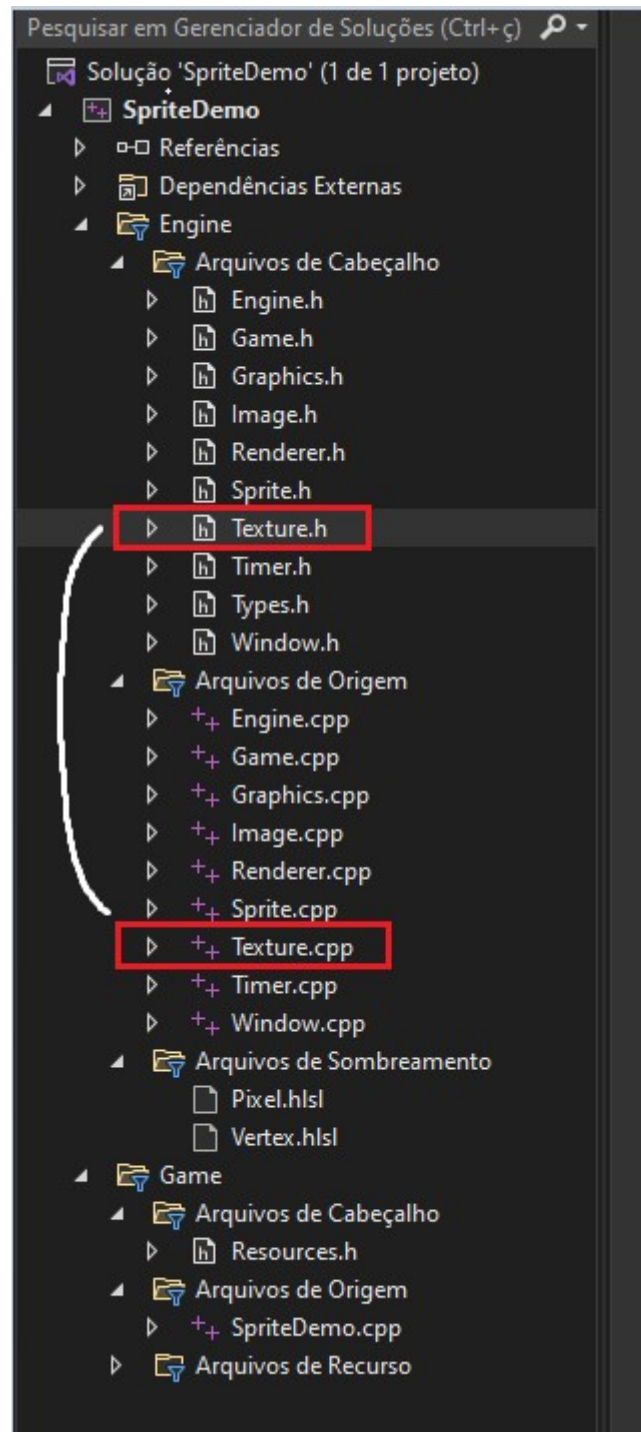
```
69 void SpriteDemo::Draw()  
70 {  
71     backg->Draw(0.0f, 0.0f, Layer::BACK);  
72     shank->Draw(x, y);  
73     logo1->Draw(40.0f, 60.0f, Layer::UPPER);  
74     logo2->Draw(400.0f, 450.0f, Layer::LOWER);  
75 }
```

A profundidade varia entre 0 à 1.

**Back**, **Upper** e **Lower** são constantes, podemos usar eles para definir a profundidade também.



Informações sobre os nossos arquivos da engine até o momento.



Esses dois novos arquivos (**Texture.h** e **Texture.cpp**) são do DirectX Tool Kit (DirectXTK). Texture.cpp é um arquivo extenso, não será abordado por completo. O que precisamos saber dele é que é carregado uma imagem do HD através dele.

```
10  *****/
11
12  #ifndef _PROGJOGOS_IMAGE_H_
13  #define _PROGJOGOS_IMAGE_H_
14
15  // -----
16  // Inclusões
17
18  #include "Types.h" // tipos específicos do motor
19  #include "Texture.h" // função para carregar textura
20  #include <string> // classe string de C++
21  using std::string; // classe pode ser usada sem std::
22
23  // -----
24
25  class Image
26  {
27  private:
28      ID3D11ShaderResourceView * textureView; // view associada a textura
29      uint width; // altura da imagem
30      uint height; // largura da imagem
31
32  public:
33      Image(string filename); // constroi imagem a partir de um arquivo
34      ~Image(); // destrutor
35
36      uint Width() const; // retorna largura da imagem
37      uint Height() const; // retorna altura da imagem
38      ID3D11ShaderResourceView * View() const; // retorna ponteiro para a view da imagem
39  };
40
41  // -----
42  // Métodos Inline
43
44  // retorna largura da textura
45  inline uint Image::Width() const
46  { return width; }
47
48  // retorna altura da textura
49  inline uint Image::Height() const
50  { return height; }
51
52  // retorna ponteiro para textura D3D
53  inline ID3D11ShaderResourceView * Image::View() const
54  { return textureView; }
55
56  // -----
57
58  #endif
```

```
Image.cpp SpriteDemo (Escopo Global)
10  [*****]
11
12  #include "Image.h"
13  #include "Graphics.h"
14
15  // -----
16
17  Image::Image(string filename) : textureView(nullptr), width(0), height(0)
18  {
19      // cria sharer resource view da imagem em disco
20      D3D11CreateTextureFromFile(
21          Graphics::device,           // dispositivo Direct3D
22          Graphics::context,         // contexto do dispositivo
23          filename.c_str(),          // nome do arquivo de imagem
24          nullptr,                   // retorna textura
25          &textureView,              // retorna view da textura
26          width,                     // retorna largura da imagem
27          height);                   // retorna altura da imagem
28  }
29
30  // -----
31
32  Image::~Image()
33  {
34      // libera memória ocupada pela texture view
35      if (textureView)
36      {
37          // pega ponteiro para recurso
38          ID3D11Resource * resource = nullptr;
39          textureView->GetResource(&resource);
40
41          // liberando a view não libera automaticamente
42          // o recurso que foi criado junto com a view
43          if (resource)
44          {
45              resource->Release();
46              resource = nullptr;
47          }
48
49          textureView->Release();
50          textureView = nullptr;
51      }
52  }
53
54  // -----
```

Nota importante. Para liberar a imagem da memória ram, é preciso usar ponteiros, como mostra a imagem acima.

<https://youtu.be/sVXwZuLeejs?t=3326>

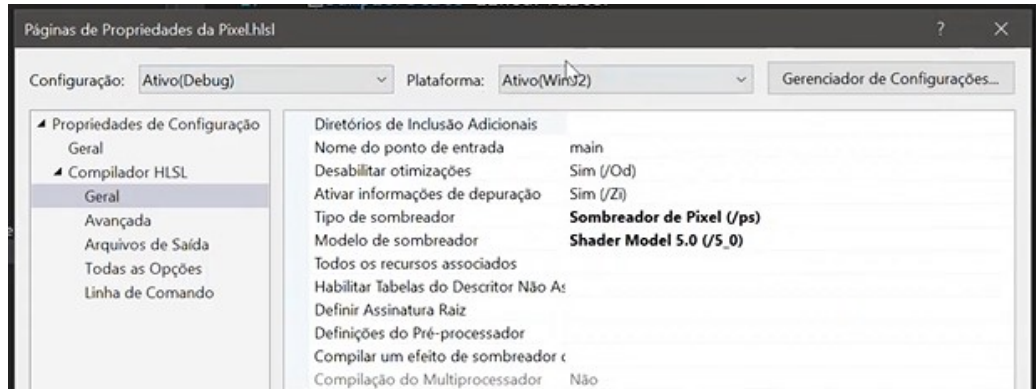
Nota: Aplicações x86 rodam em x64. Pois x64 é uma expansão de x86.

Aplicações x86 atingem um maior número de clientes, pois rodam até em máquinas x86 e x64.

Aplicações x64 só rodam em máquina x64.

Nota 2: A configuração do Visual Studio deve ser essa, para rodar o projeto:





Info: O projeto do professor já vem configurado corretamente.

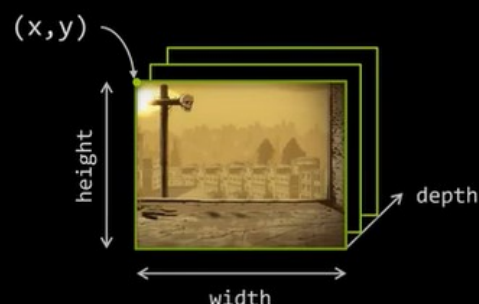
## Vamos falar sobre o Sprites

### Renderizador de Sprites

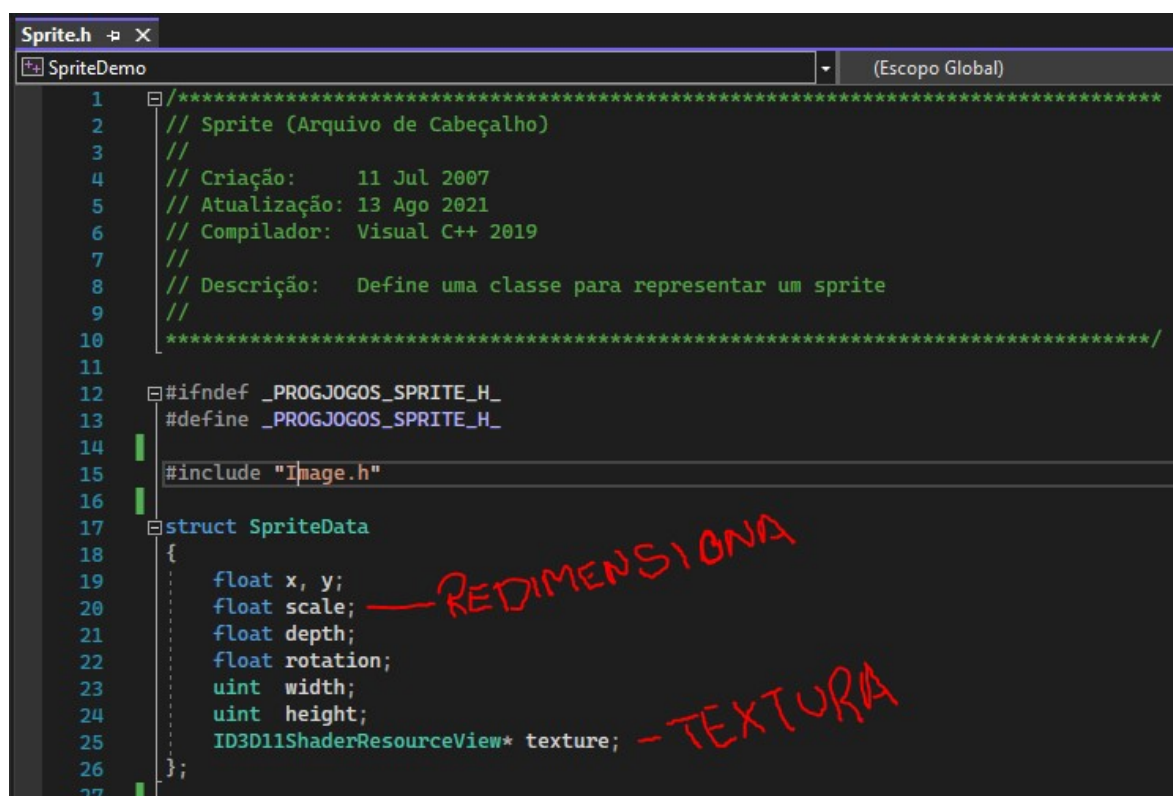
- ▶ Para simplificar criaremos uma classe **Renderer**

- Ela recebe e armazena as informações dos Sprites
- Desenha um conjunto de sprites na tela

```
struct SpriteData
{
    float x, y;
    float depth;
    uint width;
    uint height;
    ID3D11ShaderResourceView* texture;
};
```

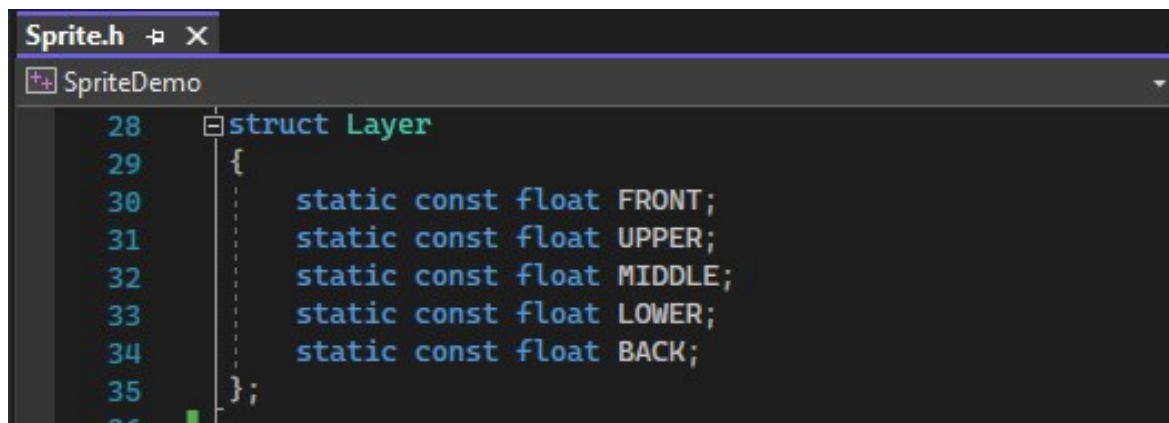


Já falamos sobre as camadas anteriormente, abaixo temos ela aplicada no projeto.



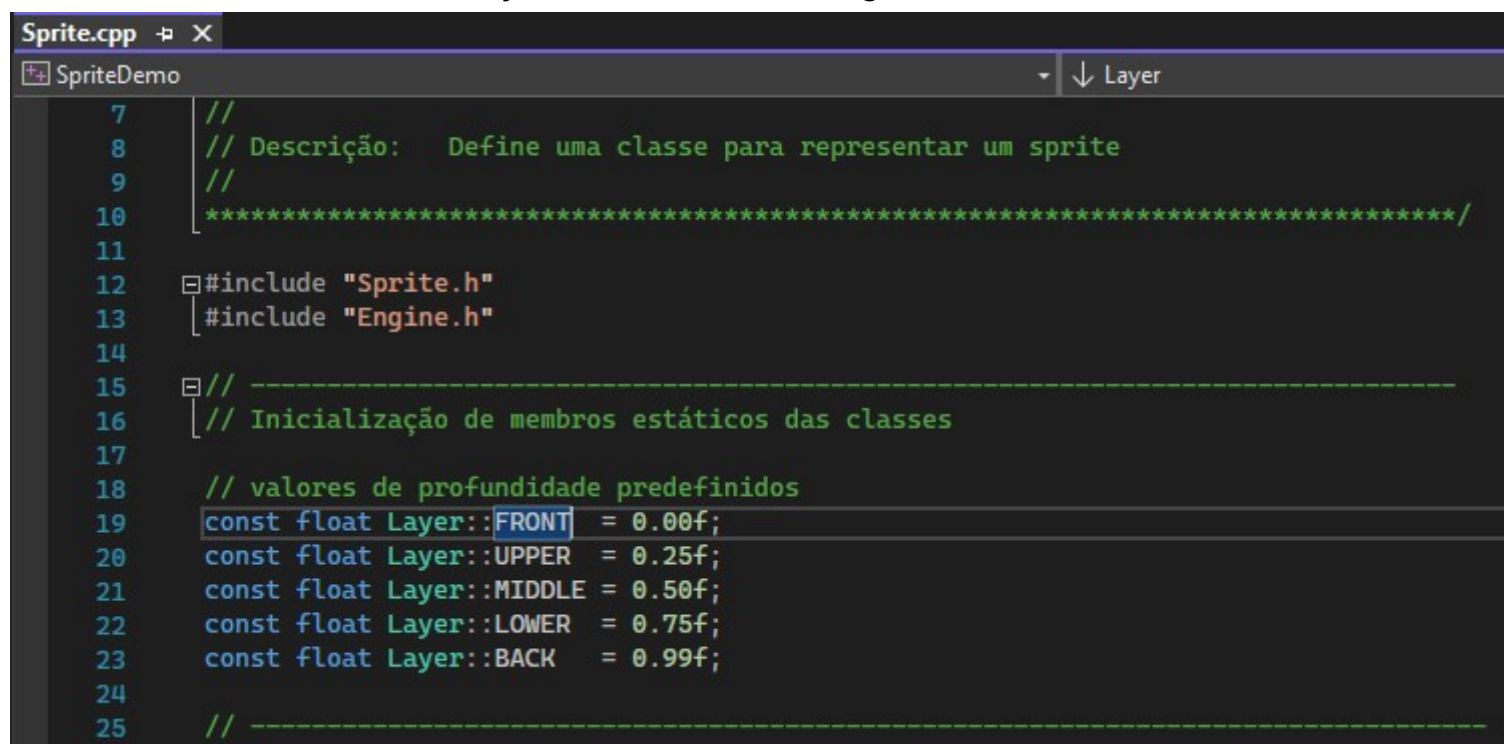


A imagem abaixo é apenas uma conveniência, para facilitar a definição de profundidade do Sprite.  
Ele se trata de um ponto flutuante.



```
Sprite.h  X
SpriteDemo
28 struct Layer
29 {
30     static const float FRONT;
31     static const float UPPER;
32     static const float MIDDLE;
33     static const float LOWER;
34     static const float BACK;
35 };
36
```

Veja o valor dele na imagem abaixo:



```
Sprite.cpp  X
SpriteDemo
7 //
8 // Descrição: Define uma classe para representar um sprite
9 //
10 *****/
11
12 #include "Sprite.h"
13 #include "Engine.h"
14
15 // -----
16 // Inicialização de membros estáticos das classes
17
18 // valores de profundidade predefinidos
19 const float Layer::FRONT = 0.00f;
20 const float Layer::UPPER = 0.25f;
21 const float Layer::MIDDLE = 0.50f;
22 const float Layer::LOWER = 0.75f;
23 const float Layer::BACK = 0.99f;
24
25 // -----
```

É isso que ele vale.

Profundidade 0.00f, sobrepõe todas as camadas.

A profundidade máxima é 1.00f.

---

<https://youtu.be/sVXwZuLeejs?t=3892>