Desenho de Sprites

Programação de Jogos

Judson Santos Santiago

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

Carregando Imagens

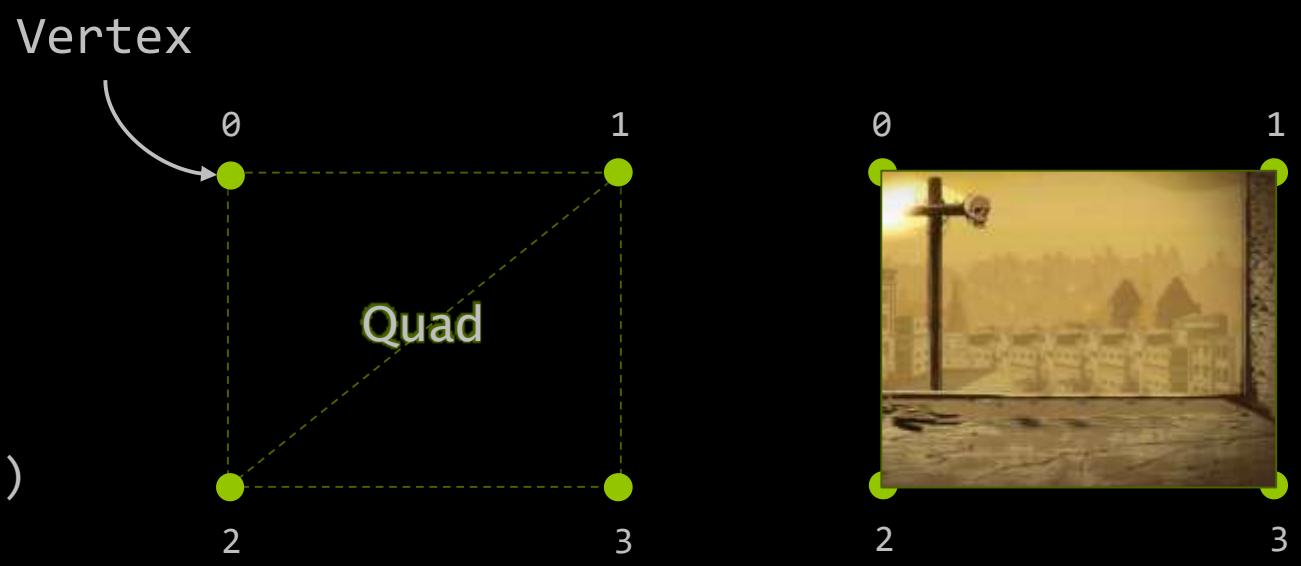
- As imagens precisam ser carregadas do disco
  - O DirectXTK carrega uma imagem em uma Textura D3D usando WIC
    - O Windows Imaging Component é um framework que suporta a manipulação de imagens nos principais formatos:
      TIFF, JPG, PNG, GIF, BMP e HDPhoto

```
// 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
```

Desenho de Sprites

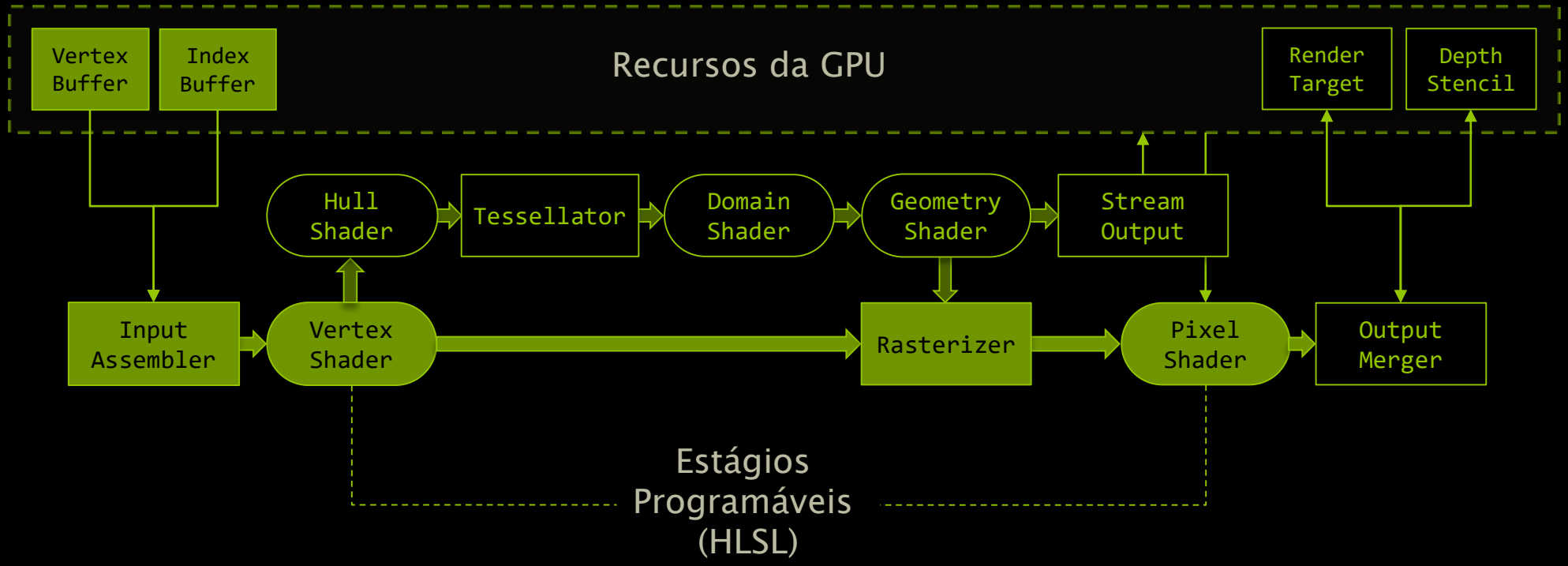
- Para desenhar Sprites no Direct3D é preciso:
  - Criar um **Quad** no espaço tridimensional (vértices e índices)
  - Aplicar uma textura aos vértices

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

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



Desenho de Sprites

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



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);
```

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

Shaders

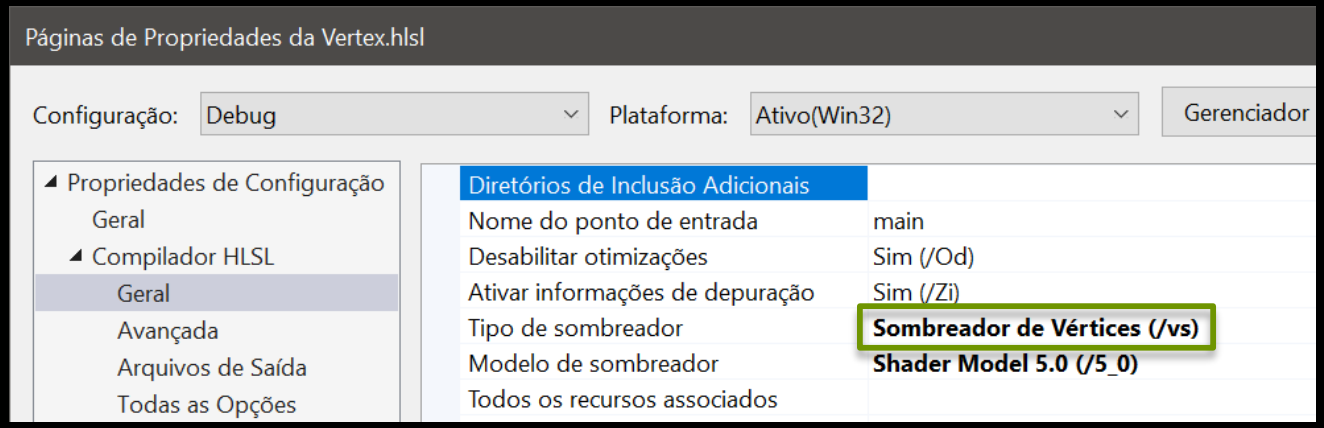
- Vertex Shader

```
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;
}
```

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

Shaders

- Pixel Shader

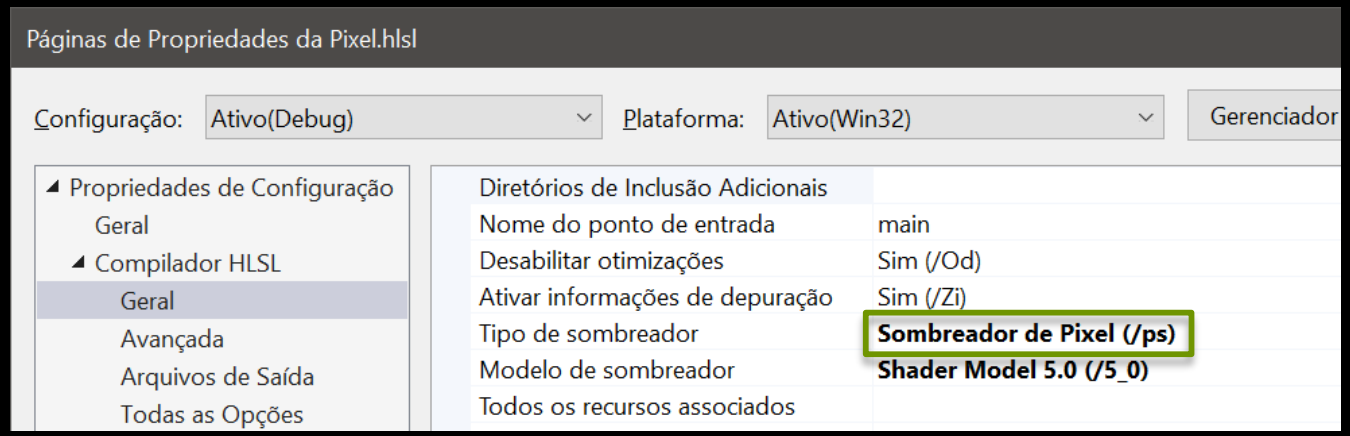
```
Texture2D resource;

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

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

- O rasterizador preenche triângulos

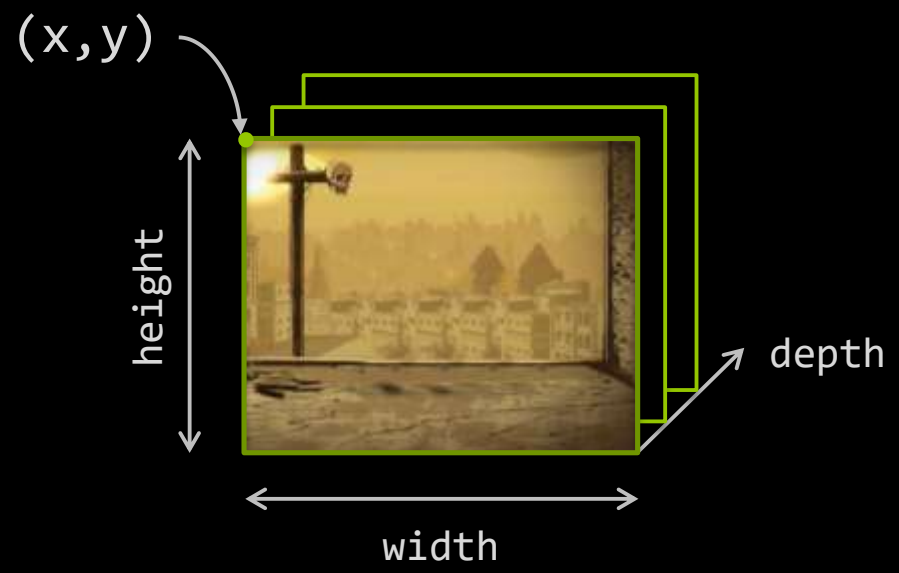
```
D3D11_RASTERIZER_DESC rasterDesc = {0};
rasterDesc.FillMode = D3D11_FILL_SOLID;
//rasterDesc.FillMode = D3D11_FILL_WIREFRAME;
rasterDesc.CullMode = D3D11_CULL_NONE;
rasterDesc.DepthClipEnable = true;

// 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

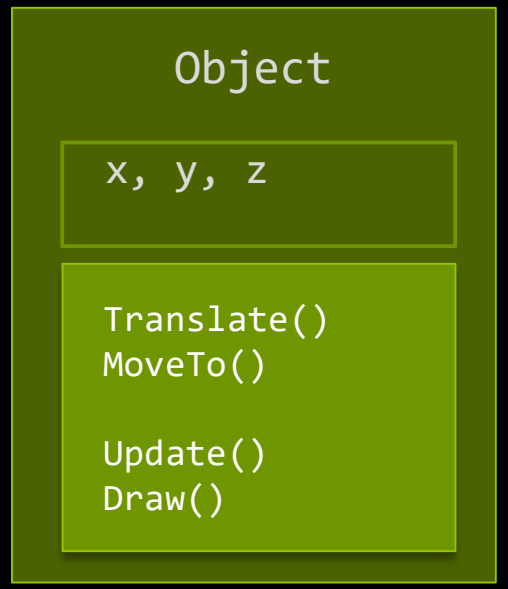
```
RenderBatch(batchTexture, &spriteVector[batchStart], pos - batchStart);
```

Organizando Classes

- Um jogo 2D é baseado no desenho e animação de Sprites
  - Um Sprite precisa guardar mais que apenas uma imagem:
    - Posição (x,y) na tela
    - Profundidade
    - Velocidade
    - Animação
    - Tempo de vida

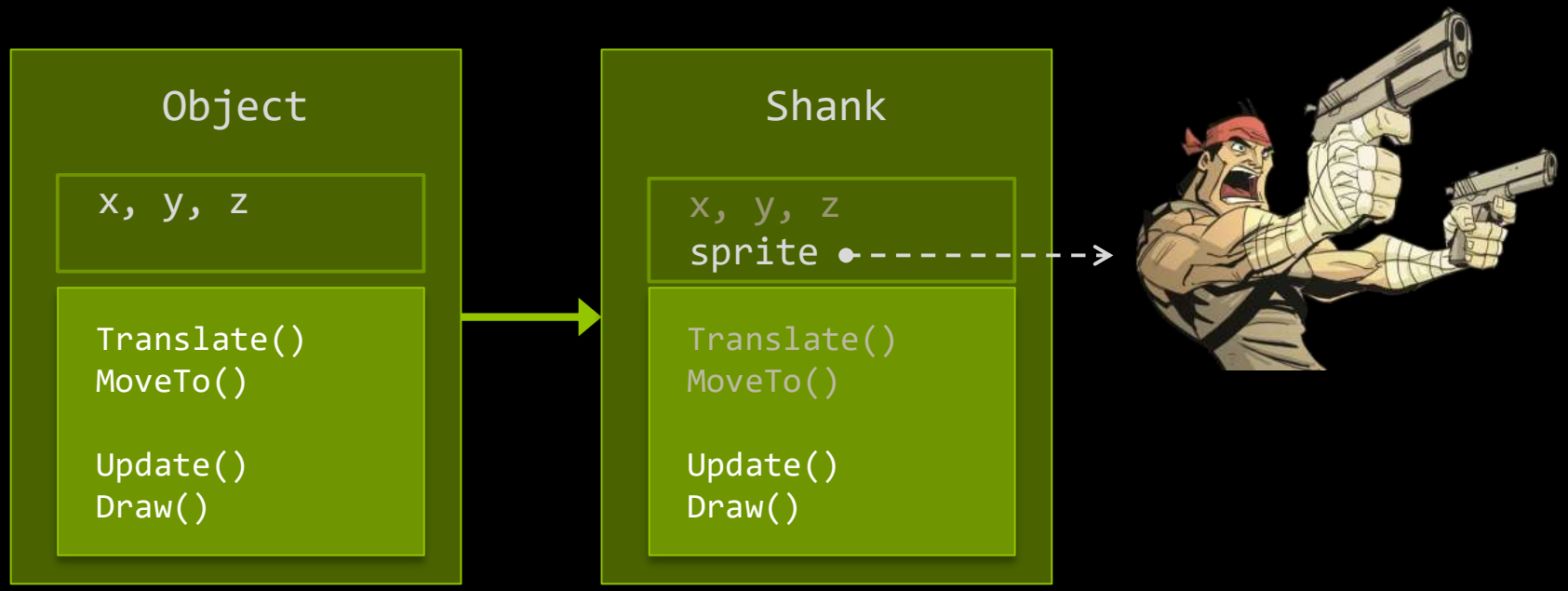
Organizando Classes

- Podemos gerenciar melhor cada Sprite criando uma classe abstrata para representar objetos do jogo



Organizando Classes

- Shank será um Object que contém um sprite
  - A atualização e desenho do objeto Shank será feito nas funções membro Update e Draw



Resumo

- Um Sprite é uma imagem 2D
  - Tipicamente é usado para representar objetos em uma cena
  - O DirectX suporta a manipulação de Sprites
    - Usando formatos de imagens com e sem transparência (bmp, jpg, png, tga, tiff, gif)
    - Através da aplicação de transformações (translação, escala, rotação)
    - Efeitos com cores
  - Uma classe para representar objetos do jogo facilita a atualização e o desenho de sprites