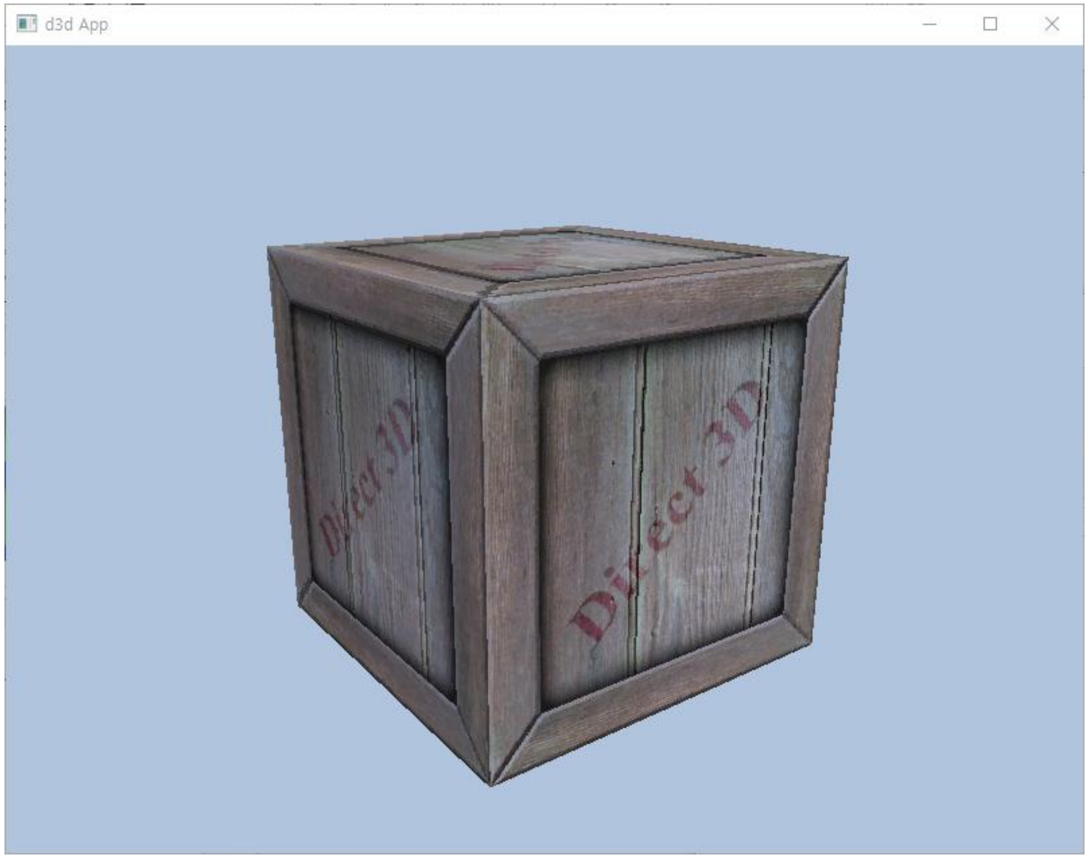
II. Direct3D Foundations
9. Texturing

Game Graphic Programming
Kyung Hee University
Software Convergence
Prof. Daeho Lee

Texturing



Kyung Hee University
nize@khu.ac.kr

DXGI_FORMAT

- **DXGI_FORMATF**
  - **DXGI_FORMAT_R32G32B32_FLOAT**: Each element has three 32-bit floating-point components.
  - **DXGI_FORMAT_R16G16B16A16_UNORM**: Each element has four 16-bit components mapped to the [0, 1] range.
  - **DXGI_FORMAT_R32G32_UINT**: Each element has two 32-bit unsigned integer components.
  - **DXGI_FORMAT_R8G8B8A8_UNORM**: Each element has four 8-bit unsigned components mapped to the [0, 1] range.
  - **DXGI_FORMAT_R8G8B8A8_SNORM**: Each element has four 8-bit signed components mapped to the [-1, 1] range.
  - **DXGI_FORMAT_R8G8B8A8_SINT**: Each element has four 8-bit signed integer components mapped to the [−128, 127] range.
  - **DXGI_FORMAT_R8G8B8A8_UINT**: Each element has four 8-bit unsigned integer components mapped to the [0, 255] range.

  - Textures are different than buffer resources, which just store arrays of data; textures can have mipmap levels, and the GPU can do special operations on them, such as apply filters and multisampling.

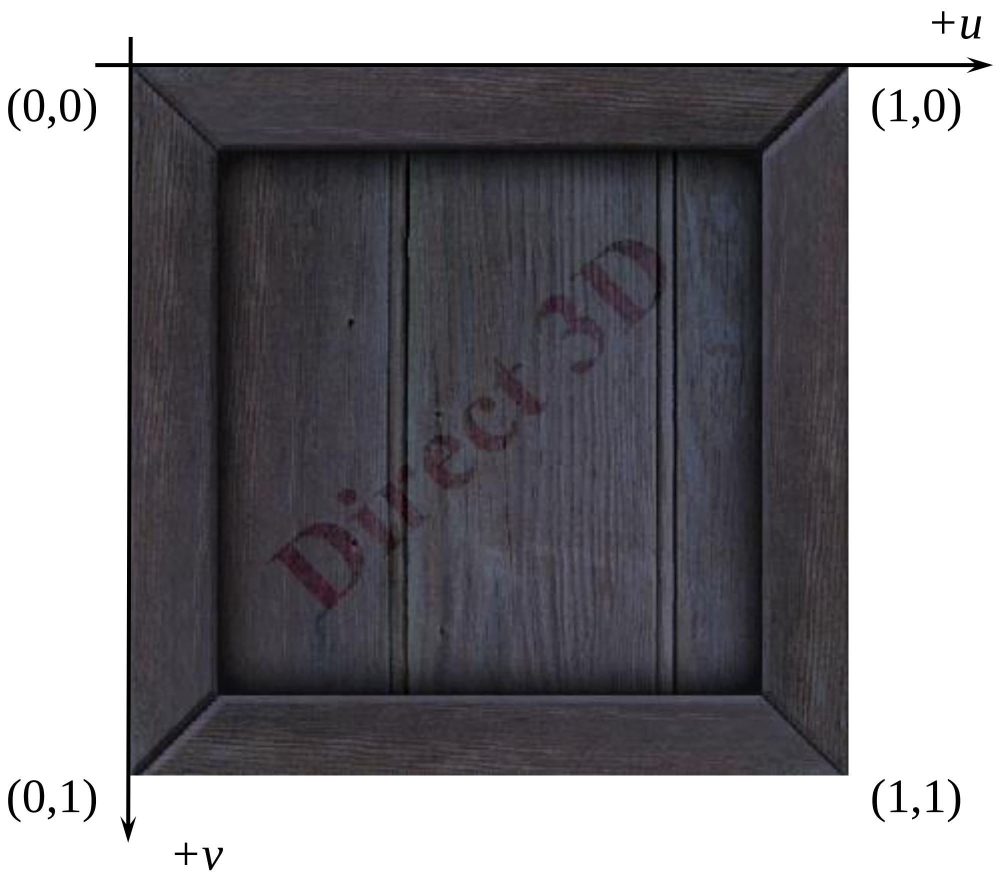
Texture Resources

- For a texture to be used as both a render target and a shader resource, we would need to create two descriptors for the texture resource.
  - One that lives in a render target heap (i.e., **`D3D12_DESCRIPTOR_HEAP_TYPE_RTV`**)
  - One that lives in a shader resource heap (i.e., **`D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV`**)
- Texture Binding

```
// Bind as render target.
CD3DX12_CPU_DESCRIPTOR_HANDLE rtv = …;
CD3DX12_CPU_DESCRIPTOR_HANDLE dsv = …;
cmdList->OMSetRenderTargets(1, &rtv, true, &dsv);

// Bind as shader input to root parameter.
CD3DX12_GPU_DESCRIPTOR_HANDLE tex = …;
cmdList->SetGraphicsRootDescriptorTable(rootParamIndex,
    tex);
```

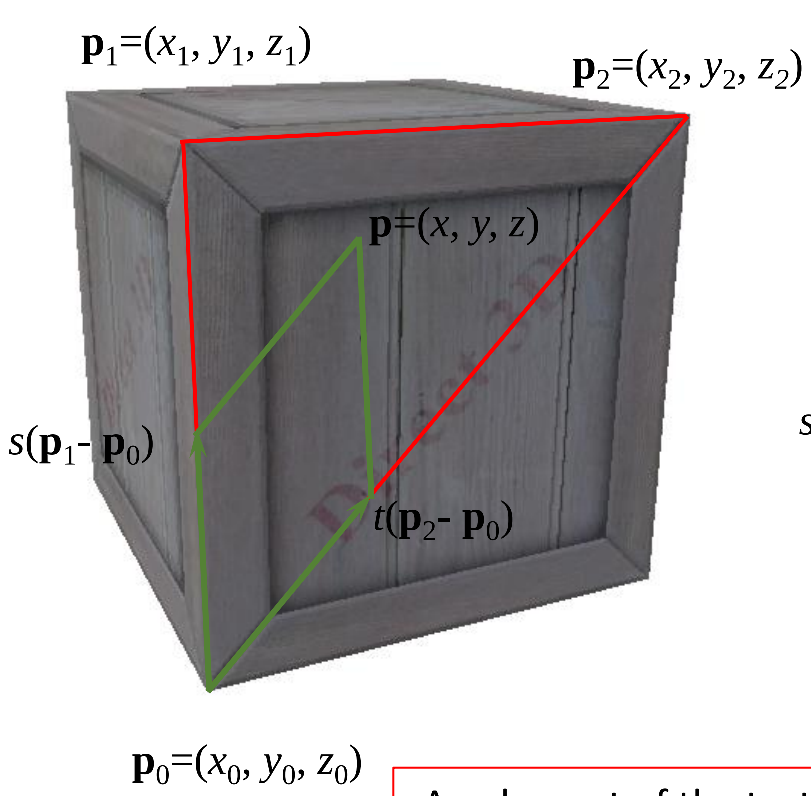
Texture Coordinates (1)

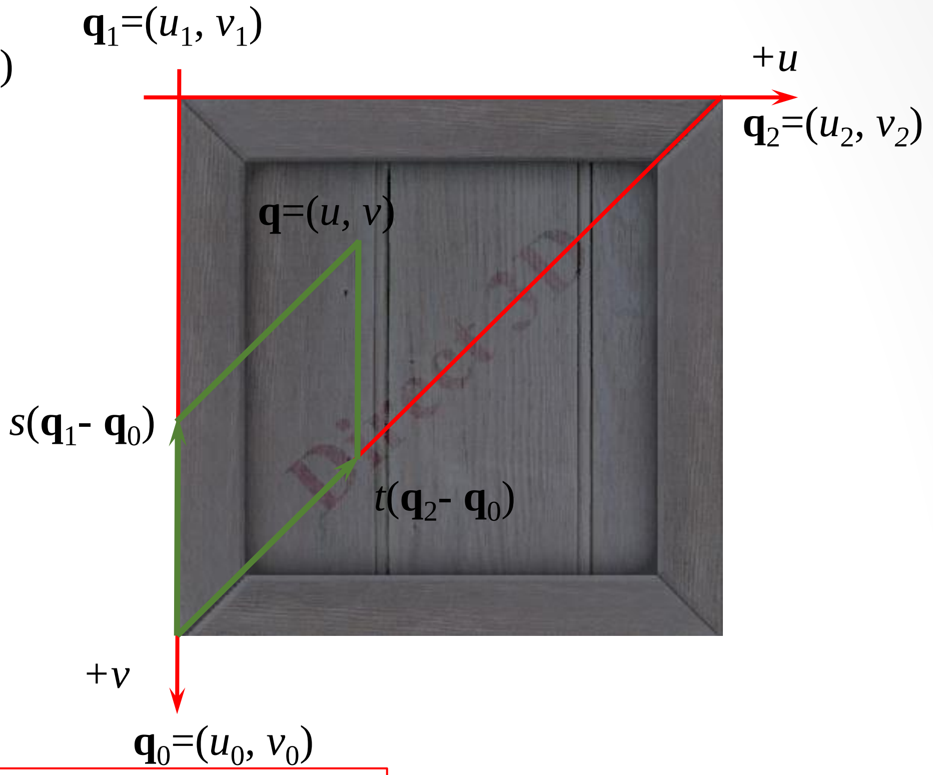
- Direct3D uses a texture coordinate system that consists of a u-axis that runs horizontally to the image and a v-axis that runs vertically to the image.



Texture Coordinates (2)

- $\mathbf{q}_0$, $\mathbf{q}_1$, and $\mathbf{q}_2$ are the corresponding texture coordinates of the vertices, $\mathbf{p}_0$, $\mathbf{p}_1$, and $\mathbf{p}_2$.
  - An arbitrary point $\mathbf{p}=(x, y, z)$ on the 3D triangle, its texture coordinates $\mathbf{q}=(u, v)$ are found by linearly interpolation.


$\mathbf{p}_1=(x_1, y_1, z_1)$
$\mathbf{p}_2=(x_2, y_2, z_2)$
$\mathbf{p}=(x, y, z)$
$s(\mathbf{p}_1 - \mathbf{p}_0)$
$t(\mathbf{p}_2 - \mathbf{p}_0)$
$\mathbf{p}_0=(x_0, y_0, z_0)$


$\mathbf{q}_1=(u_1, v_1)$
$+u$
$\mathbf{q}_2=(u_2, v_2)$
$\mathbf{q}=(u, v)$
$s(\mathbf{q}_1 - \mathbf{q}_0)$
$t(\mathbf{q}_2 - \mathbf{q}_0)$
$+v$
$\mathbf{q}_0=(u_0, v_0)$

An element of the texture is called a **texel**.

Kyung Hee University
nize@khu.ac.kr

Texture Coordinates (3)

- The vertex structure should have texture coordinates that identify a point on the texture.
  - FrameResource.h

```
struct Vertex {
    DirectX::XMFLOAT3 Pos;
    DirectX::XMFLOAT3 Normal;
    DirectX::XMFLOAT2 TexC;
};
```

  - **void CrateApp::BuildShadersAndInputLayout()**

```
// …
    mInputLayout // std::vector<D3D12_INPUT_ELEMENT_DESC>
        = {
        { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
        D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 },
        { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12,
        D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 },
        { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 24,
        D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 }};
// …
```

DDS Overview (1)

- The most prevalent way of creating textures for games is for an artist to make them in Photoshop or some other image editor, and then save them as an image file like BMP, DDS, TGA, or PNG.
- Then the game application will load the image data at load time into an ID3D12Resource object. For real-time graphics applications, the **DDS (DirectDraw Surface format)** image file format is preferred, as it supports a variety of image formats that are natively understood by the GPU; in particular, it supports compressed image formats that can be natively decompressed by the GPU.

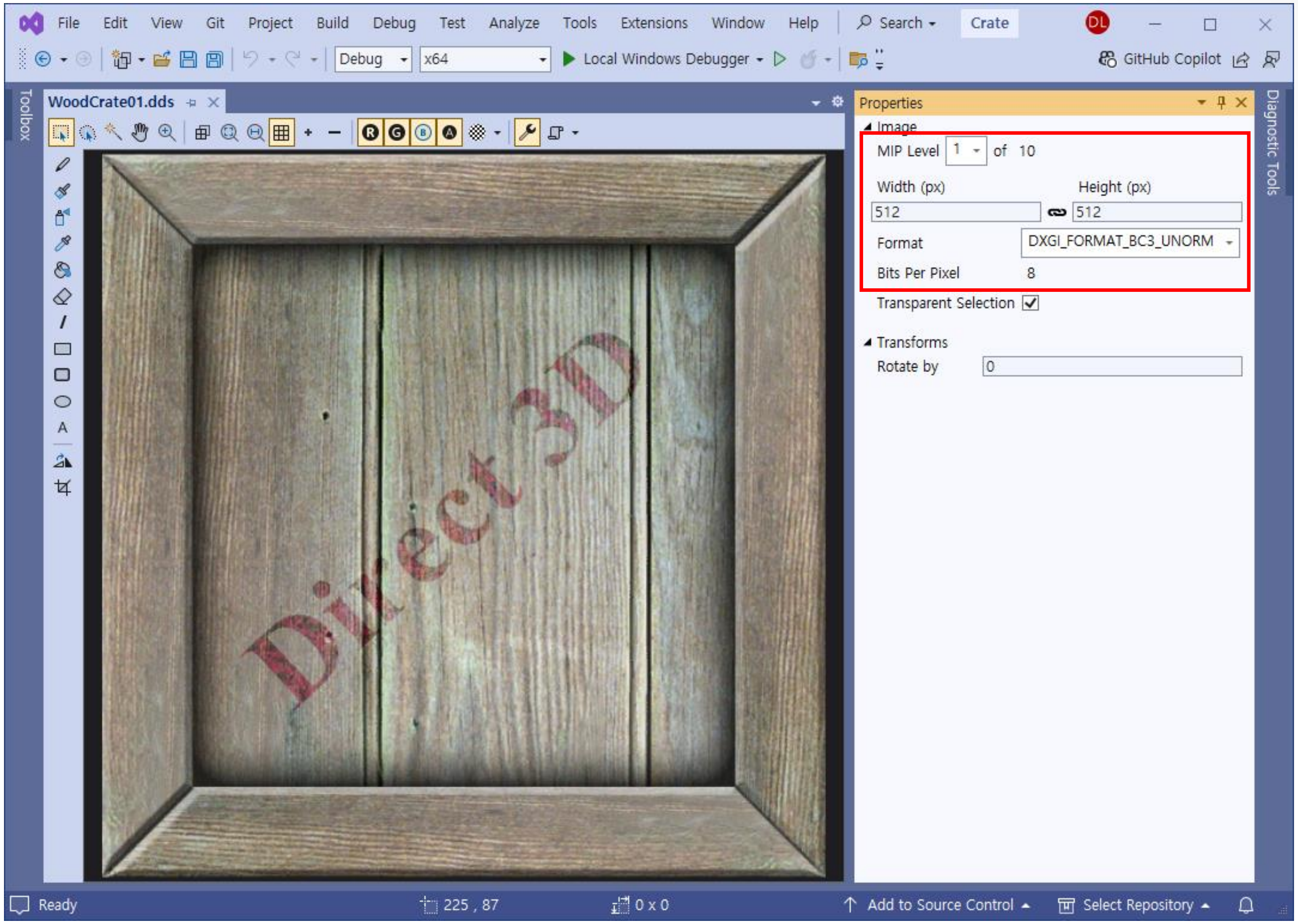
DDS Overview (2)

- DDS textures support the following features used in 3D graphics development
  - Mipmaps
  - Compressed formats that the GPU can natively decompress
  - Texture arrays
  - Cube maps
  - Volume textures

- Typically, for uncompressed image data, we will use the following formats.
  - **`DXGI_FORMAT_B8G8R8A8_UNORM`** or **`DXGI_FORMAT_B8G8R8X8_UNORM`**: For low-dynamic-range images.
  - **`DXGI_FORMAT_R16G16B16A16_FLOAT`**: For high-dynamic-range images.

- Compressed formats
  - **`DXGI_FORMAT_BC1_UNORM, DXGI_FORMAT_BC2_UNORM, , DXGI_FORMAT_BC3_UNORM,`** …

DDS Overview (3)



Properties

Image

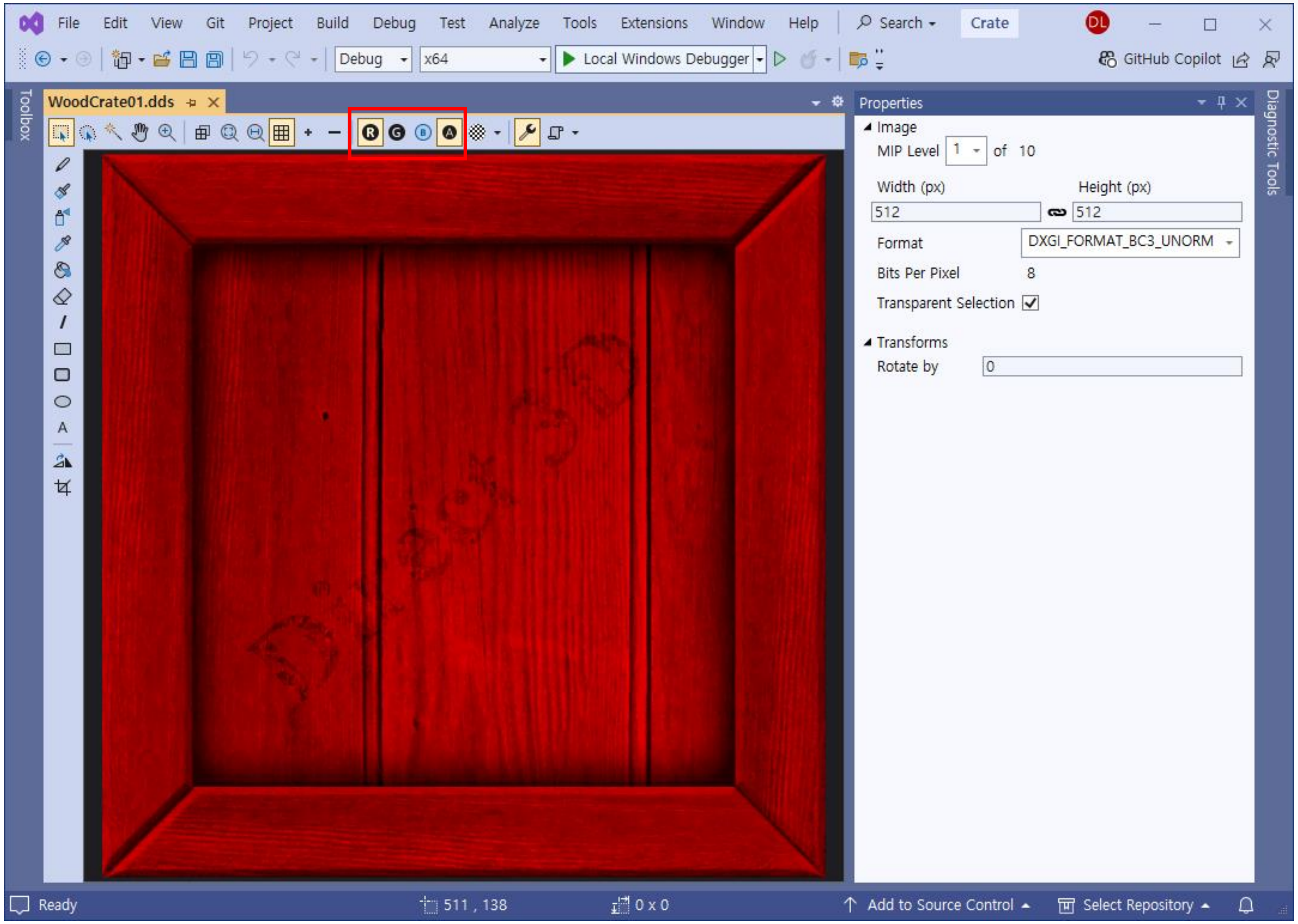
MIP Level	1 of 10
Width (px)	512
Height (px)	512
Format	DXGI_FORMAT_BC3_UNORM
Bits Per Pixel	8
Transparent Selection	✓

Transforms

Rotate by	0

225 , 87   0 x 0

DDS Overview (4)



Properties

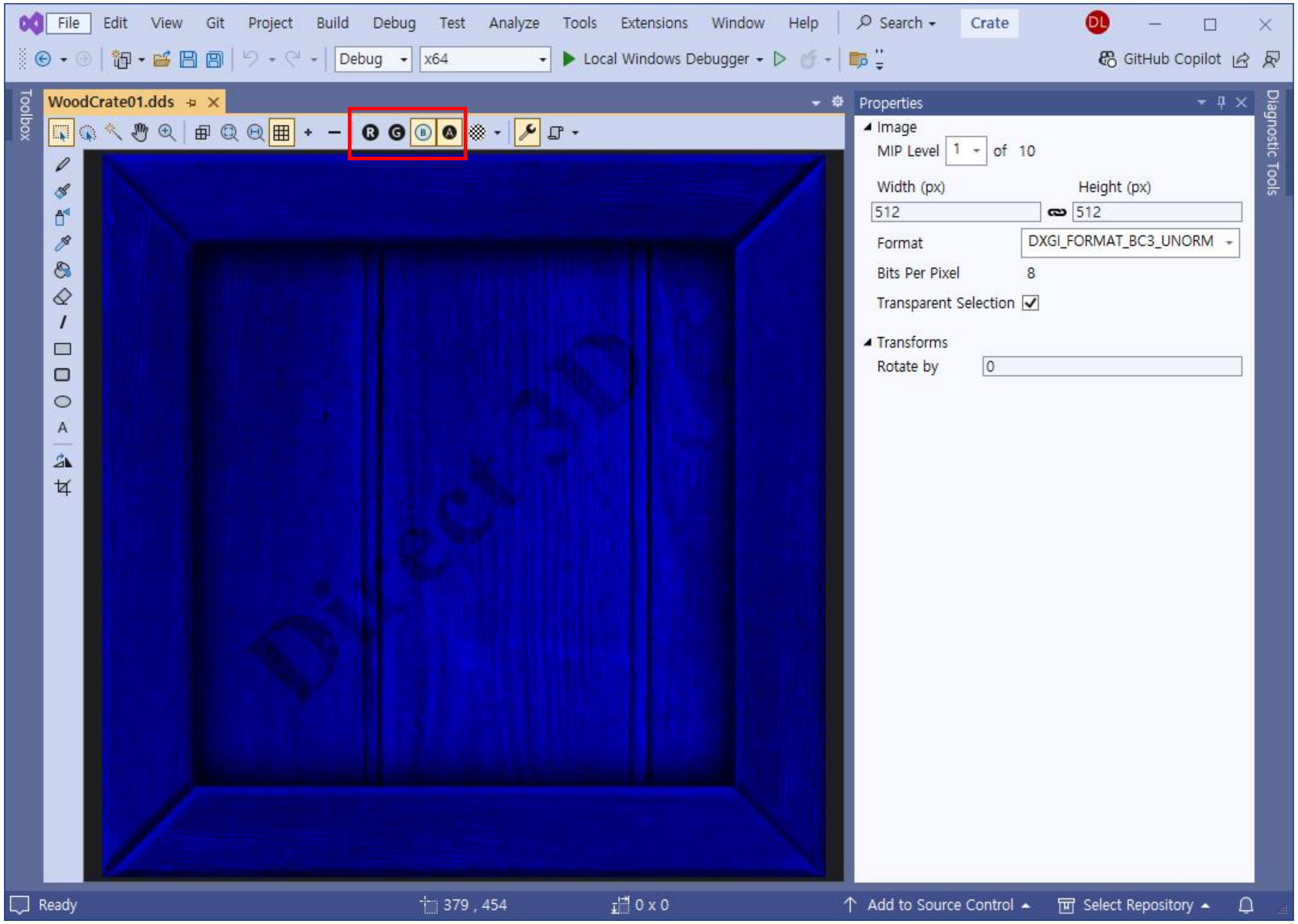
Image

MIP Level	1 of 10
Width (px)	512
Height (px)	512
Format	DXGI_FORMAT_BC3_UNORM
Bits Per Pixel	8
Transparent Selection	✓

Transforms

Rotate by	0

DDS Overview (5)



Properties

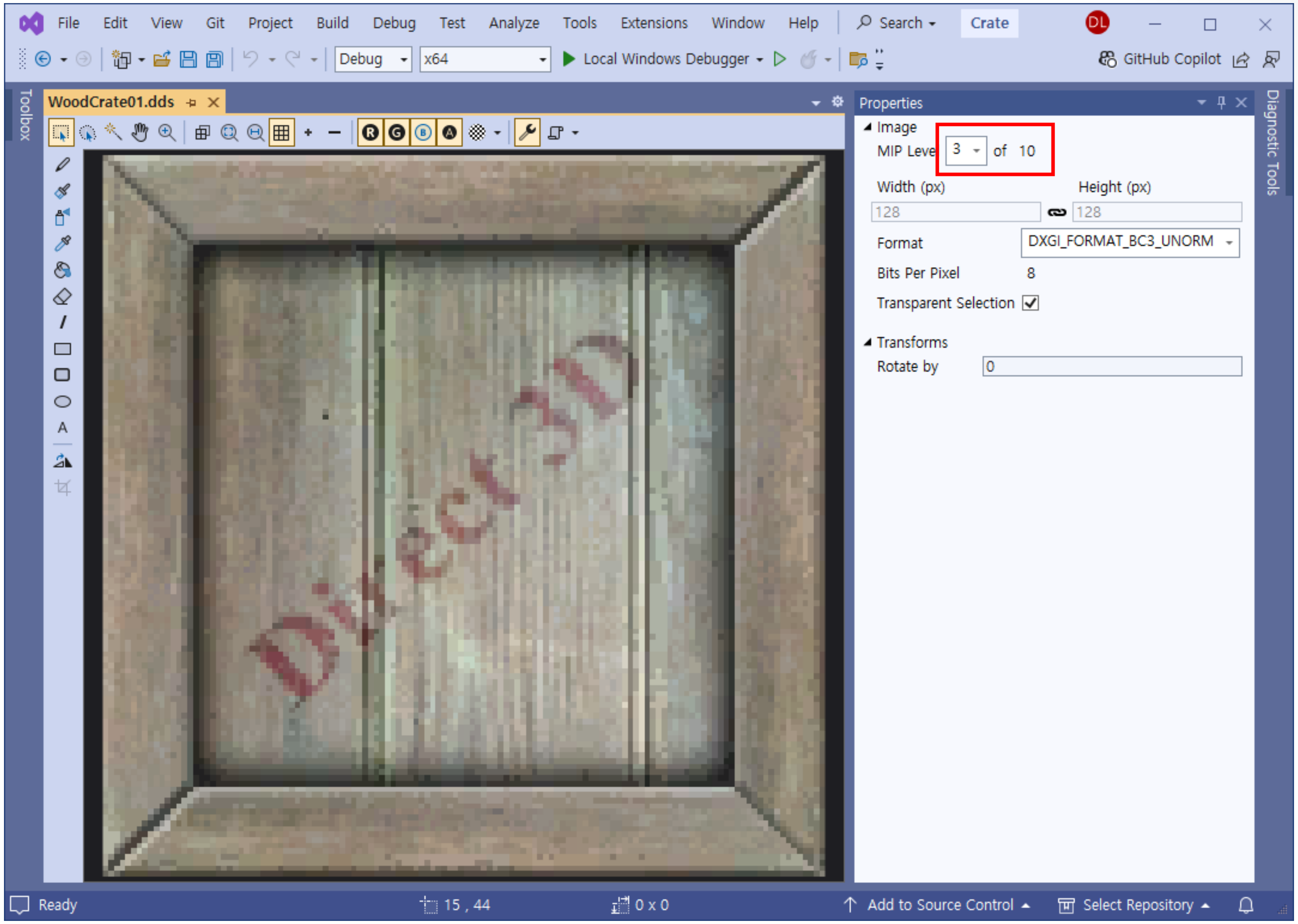
Image

MIP Level	1 of 10
Width (px)	512
Height (px)	512
Format	DXGI_FORMAT_BC3_UNORM
Bits Per Pixel	8
Transparent Selection	✓

Transforms

Rotate by	0

DDS Overview (6)



Properties

Image

MIP Level 3 of 10

Width (px)	Height (px)
128	128

Format DXGI_FORMAT_BC3_UNORM

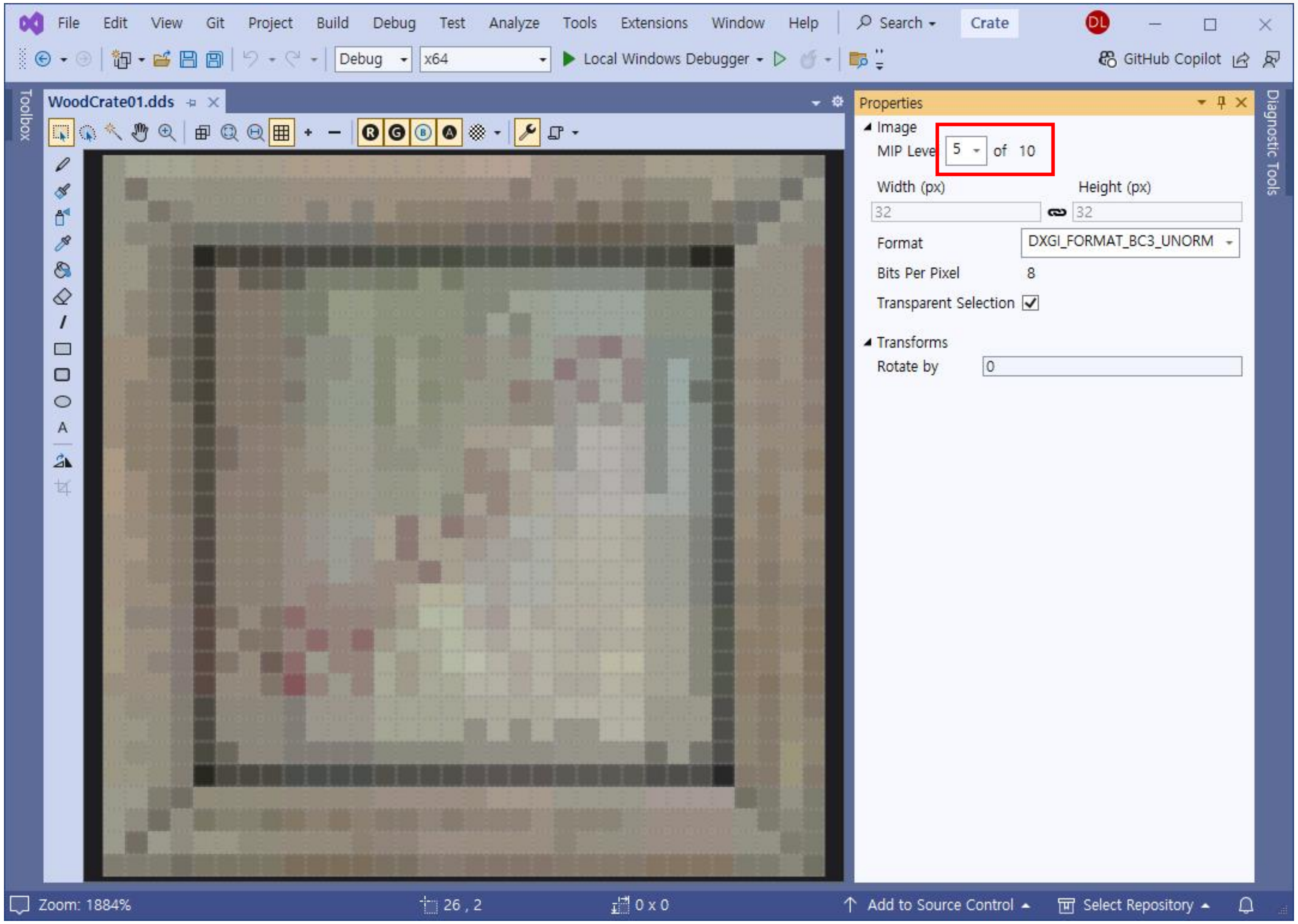
Bits Per Pixel 8

Transparent Selection ✓

Transforms

Rotate by 0

DDS Overview (7)



Properties

Image

MIP Level 5 of 10

Width (px) 32

Height (px) 32

Format DXGI_FORMAT_BC3_UNORM

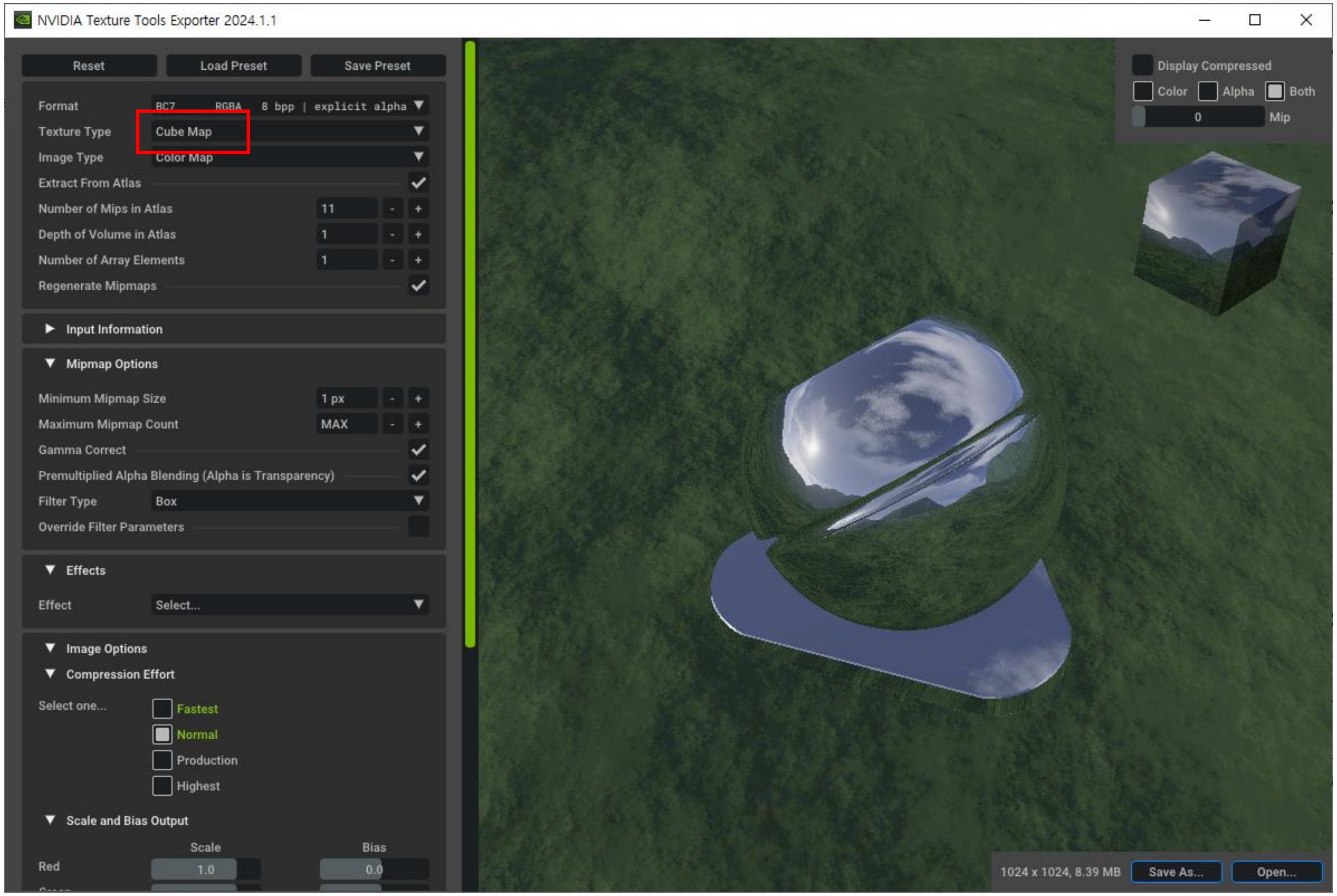
Bits Per Pixel 8

Transparent Selection ✓

Transforms

Rotate by 0

DDS Overview (8)



NVIDIA Texture Tools Exporter 2024.1.1

Reset | Load Preset | Save Preset

Format	BC7 RGBA 8 bpp | explicit alpha
Texture Type	Cube Map
Image Type	Color Map
Extract From Atlas	✓
Number of Mips in Atlas	11
Depth of Volume in Atlas	1
Number of Array Elements	1
Regenerate Mipmaps	✓

Input Information

Mipmap Options

Minimum Mipmap Size	1 px
Maximum Mipmap Count	MAX
Gamma Correct	✓
Premultiplied Alpha Blending (Alpha is Transparency)	✓
Filter Type	Box
Override Filter Parameters	

Effects

Effect: Select...

Image Options

Compression Effort

Select one... Fastest [✓ Normal] Production Highest

Scale and Bias Output

	Scale	Bias
Red	1.0	0.0

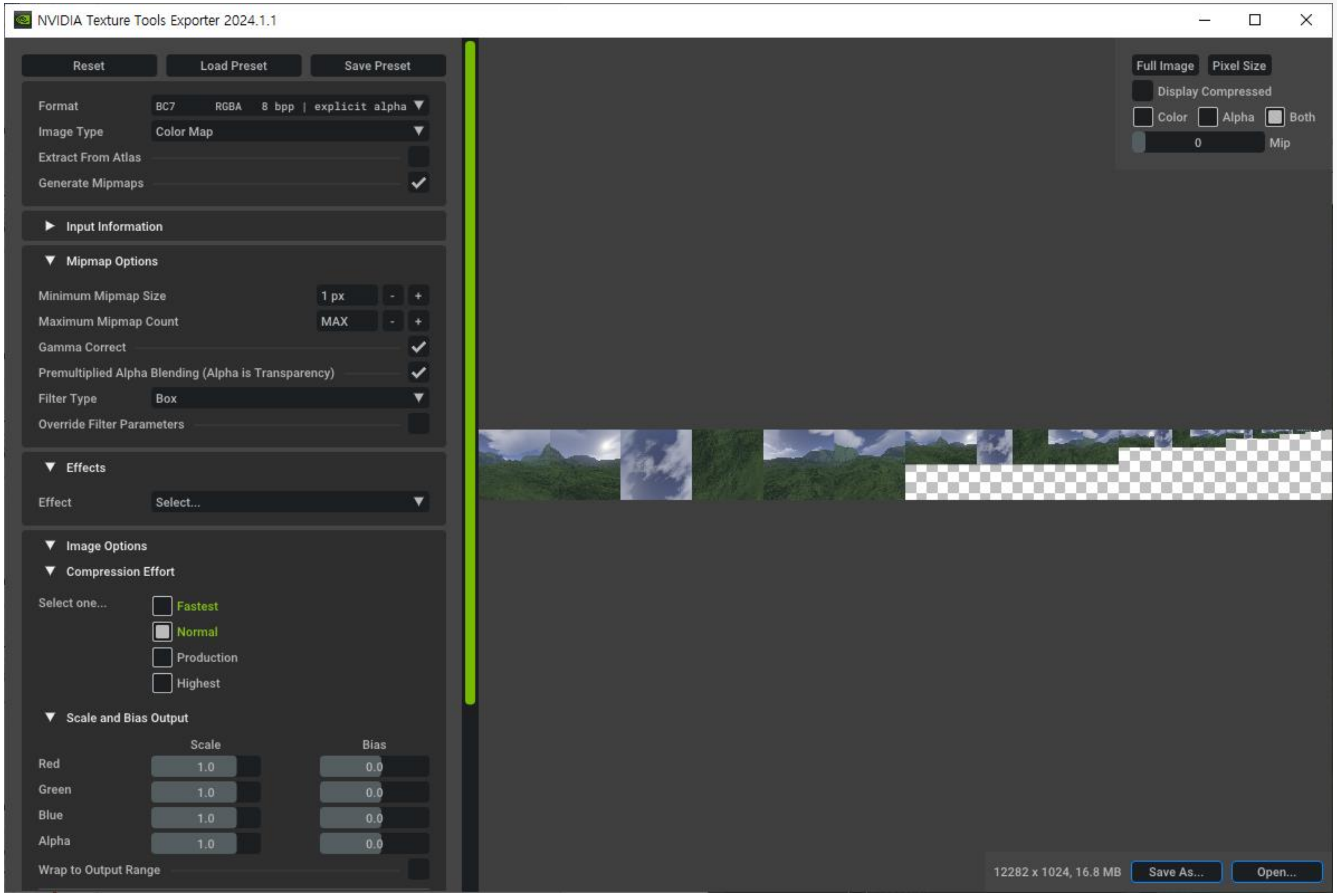
Display Compressed

Color Alpha [✓ Both]

0 Mip

1024 x 1024, 8.39 MB | Save As... | Open...

DDS Overview (9)


NVIDIA Texture Tools Exporter 2024.1.1

Reset | Load Preset | Save Preset

Format: BC7 RGBA 8 bpp | explicit alpha
Image Type: Color Map
Extract From Atlas: ☐
Generate Mipmaps: ✓

▶ Input Information

▼ Mipmap Options
Minimum Mipmap Size: 1 px
Maximum Mipmap Count: MAX
Gamma Correct: ✓
Premultiplied Alpha Blending (Alpha is Transparency): ✓
Filter Type: Box
Override Filter Parameters: ☐

▼ Effects
Effect: Select...

▼ Image Options
▼ Compression Effort
Select one...
- ☐ Fastest
- [✓ Normal]
- ☐ Production
- ☐ Highest

▼ Scale and Bias Output

	Scale	Bias
Red	1.0	0.0
Green	1.0	0.0
Blue	1.0	0.0
Alpha	1.0	0.0

Wrap to Output Range: ☐

Full Image | Pixel Size
Display Compressed: ☐
☐ Color ☐ Alpha [✓ Both]
0 Mip

12282 x 1024, 16.8 MB | Save As... | Open...

Creating DDS Files

- Two ways to convert traditional image formats to the DDS format:
  - NVIDIA Texture Tools Exporter
    - https://developer.nvidia.com/texture-tools-exporter
  - Microsoft provides a command line tool called **Texconv** that can be used to convert traditional image formats to DDS.
    - https://github.com/microsoft/DirectXTex/wiki/Texconv

Loading DDS Files (1)

- Microsoft provides lightweight source code to load DDS files at:
  - https://github.com/Microsoft/DirectXTK/wiki/DDSTextureLoader

- We have modified the **./Common/DDSTextureLoader.h/cpp** files and provided an additional method for DirectX 12.

```
HRESULT DirectX::CreateDDSTextureFromFile12(
    _In_ ID3D12Device* device,
    _In_ ID3D12GraphicsCommandList* cmdList,
    _In_z_ const wchar_t* szFileName,
    _Out_ ComPtr<ID3D12Resource>& texture,
    _Out_ ComPtr<ID3D12Resource>& textureUploadHeap,
    _In_ size_t maxsize = 0,
    _Out_opt_ DDS_ALPHA_MODE* alphaMode = nullptr
);
```

Loading DDS Files (2)

- d3dUtil.h

```
struct Texture{
  std::string Name;
  std::wstring Filename;
  Microsoft::WRL::ComPtr<ID3D12Resource> Resource = nullptr;
  Microsoft::WRL::ComPtr<ID3D12Resource> UploadHeap = nullptr;
};
```

- Application.cpp

```
std::unordered_map<std::string, std::unique_ptr<Texture>>
mTextures;

mTextures["woodCrateTex"] = std::make_unique<Texture>();
mTextures["woodCrateTex"]->Filename; // = L"";
ThrowIfFailed(DirectX::CreateDDSTextureFromFile12(
    md3dDevice.Get(), mCommandList.Get(),
    mTextures["woodCrateTex"]->Filename.c_str(),
    mTextures["woodCrateTex"]->Resource,
    mTextures["woodCrateTex"]->UploadHeap));
```

Creating an SRV Heap

- Once a texture resource is created, we need to create an SRV descriptor to it which we can set to a root signature parameter slot for use by the shader programs.
- In order to create an SRV descriptor, we first need to create a descriptor heap with **`ID3D12Device::CreateDescriptorHeap`** to store the SRV descriptors.

```
D3D12_DESCRIPTOR_HEAP_DESC srvHeapDesc = {};
srvHeapDesc.NumDescriptors = 1; // # of textures
srvHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV;
srvHeapDesc.Flags
    = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;
ThrowIfFailed(
    md3dDevice->CreateDescriptorHeap(&srvHeapDesc,
           IID_PPV_ARGS(&mSrvDescriptorHeap)));
```

Creating SRV Descriptors (1)

```
typedef struct D3D12_SHADER_RESOURCE_VIEW_DESC {
  DXGI_FORMAT         Format;
  D3D12_SRV_DIMENSION ViewDimension;
  UINT                Shader4ComponentMapping;
  union {
    D3D12_BUFFER_SRV                            Buffer;
    D3D12_TEX1D_SRV                             Texture1D;
    D3D12_TEX1D_ARRAY_SRV                       Texture1DArray;
    D3D12_TEX2D_SRV                             Texture2D;
    D3D12_TEX2D_ARRAY_SRV                       Texture2DArray;
    D3D12_TEX2DMS_SRV                           Texture2DMS;
    D3D12_TEX2DMS_ARRAY_SRV                     Texture2DMSArray;
    D3D12_TEX3D_SRV                             Texture3D;
    D3D12_TEXCUBE_SRV                           TextureCube;
    D3D12_TEXCUBE_ARRAY_SRV                     TextureCubeArray;
    D3D12_RAYTRACING_ACCELERATION_STRUCTURE_SRV RaytracingAccelerationStructure;
  };
} D3D12_SHADER_RESOURCE_VIEW_DESC;
```

Creating SRV Descriptors (2)

- **`D3D12_SRV_DIMENSION ViewDimension:`**
  - **`D3D12_SRV_DIMENSION_TEXTURE2D,`** …
- **`UINT Shader4ComponentMapping`**
  - A value, constructed using the **`D3D12_ENCODE_SHADER_4_COMPONENT_MAPPING`** macro. The **`D3D12_SHADER_COMPONENT_MAPPING`** enumeration specifies what values from memory should be returned when the texture is accessed in a shader via this shader resource view (SRV).
  - In this lecture, we just specify **`D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING`** (default 1:1 mapping).

Creating SRV Descriptors (3)

- `D3D12_TEX2D_SRV Texture2D;`

```
typedef struct D3D12_TEX2D_SRV {
  UINT  MostDetailedMip;
  // the index of the most detailed mipmap level
  UINT  MipLevels;     // # of mipmap levels
  UINT  PlaneSlice;    // plane index
  FLOAT ResourceMinLODClamp;
  // Specifies the minimum mipmap level that can be accessed.
  // 0.0 means all the mipmap levels can be accessed.
  // Specifying 3.0 means mipmap levels 3.0 to MipCount-1 can
  //   be accessed.
} D3D12_TEX2D_SRV;
```

Creating SRV Descriptors (4)

```
ID3D12Resource* ExTex;// Created texture resource

CD3DX12_CPU_DESCRIPTOR_HANDLE hDescriptor
(mSrvDescriptorHeap->GetCPUDescriptorHandleForHeapStart());

D3D12_SHADER_RESOURCE_VIEW_DESC srvDesc = {};
srvDesc.Shader4ComponentMapping
    = D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING;
srvDesc.Format = ExTex->GetDesc().Format;
srvDesc.ViewDimension = D3D12_SRV_DIMENSION_TEXTURE2D;
srvDesc.Texture2D.MostDetailedMip = 0;
srvDesc.Texture2D.MipLevels = ExTex->GetDesc().MipLevels;
srvDesc.Texture2D.ResourceMinLODClamp = 0.0f;

md3dDevice->CreateShaderResourceView(ExTex.Get(),
    &srvDesc, hDescriptor);
```

Binding Textures to the Pipeline (1)

- HLSL
  - Chap. 8

```
float4 PS(VertexOut pin) : SV_Target {
    // …
    float4 ambient = gAmbientLight*gDiffuseAlbedo;
    Material mat = { gDiffuseAlbedo, gFresnelR0, shininess };
    // …
}
```

  - Chap. 9

```
float4 PS(VertexOut pin) : SV_Target {
    float4 diffuseAlbedo
          = gDiffuseMap.Sample(gsamLinear, pin.TexC) * gDiffuseAlbedo;
    // …
    float4 ambient = gAmbientLight*diffuseAlbedo;
    Material mat = { diffuseAlbedo, gFresnelR0, shininess };
    // …
}
```

Binding Textures to the Pipeline (2)

- Shapes application
  - Chap. 8

```
void Application::BuildMaterials() {
        auto bricks0 = std::make_unique<Material>();
        bricks0->Name = "bricks0";
        bricks0->MatCBIndex = 0;
        bricks0->DiffuseSrvHeapIndex = 0;
        bricks0->DiffuseAlbedo = XMFLOAT4(Colors::ForestGreen);
        bricks0->FresnelR0 = XMFLOAT3(0.02f, 0.02f, 0.02f);
        bricks0->Roughness = 0.1f;
```

  - Chap. 9

```
void Application::BuildMaterials() {
        auto bricks0 = std::make_unique<Material>();
        bricks0->Name = "bricks0";
        bricks0->MatCBIndex = 0;
        bricks0->DiffuseSrvHeapIndex = 0;
        bricks0->DiffuseAlbedo = XMFLOAT4(1.0f, 1.0f, 1.0f, 1.0f);
        bricks0->FresnelR0 = XMFLOAT3(0.02f, 0.02f, 0.02f);
        bricks0->Roughness = 0.1f;
```

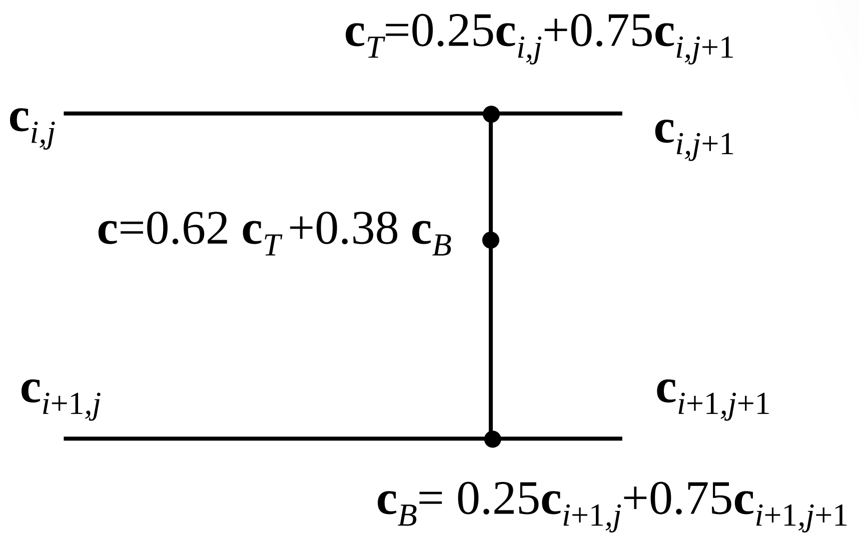
Binding Textures to the Pipeline (3)

- DrawRenderItems

```
void Applicatoin::DrawRenderItems(ID3D12GraphicsCommandList* cmdList,
     const std::vector<RenderItem*>& ritems) {
     // …
     for(size_t i = 0; i < ritems.size(); ++i)     {
          auto ri = ritems[i];
          // …
          CD3DX12_GPU_DESCRIPTOR_HANDLE tex(
               mSrvDescriptorHeap->GetGPUDescriptorHandleForHeapStart());
         tex.Offset(ri->Mat->DiffuseSrvHeapIndex, mCbvSrvDescriptorSize);
          D3D12_GPU_VIRTUAL_ADDRESS objCBAddress
           = objectCB->GetGPUVirtualAddress()+ri->ObjCBIndex*objCBByteSize;
          D3D12_GPU_VIRTUAL_ADDRESS matCBAddress
           = matCB->GetGPUVirtualAddress()+ri->Mat->MatCBIndex*matCBByteSize;

          cmdList->SetGraphicsRootDescriptorTable(0, tex);
          cmdList->SetGraphicsRootConstantBufferView(1, objCBAddress);
          cmdList->SetGraphicsRootConstantBufferView(3, matCBAddress);
          cmdList->DrawIndexedInstanced(ri->IndexCount, 1,
               ri->StartIndexLocation, ri->BaseVertexLocation, 0);
     }
}
```

Magnification

- Magnification filtering is used when the texture image is larger than the surface it is mapped onto, and it needs to be scaled up.
  - 1D texture with 256 samples, texture coordinate u = 0.126484375
    - 256×0.126484375 = 32.38 texel
    - We must use interpolation to approximate it.
      - Nearest neighbor interpolation
      - Linear interpolation
  - 2D texture
    - Bilinear interpolation


$\mathbf{c}_T=0.25\mathbf{c}_{i,j}+0.75\mathbf{c}_{i,j+1}$
$\mathbf{c}_{i,j}$
$\mathbf{c}_{i,j+1}$
$\mathbf{c}=0.62\,\mathbf{c}_T+0.38\,\mathbf{c}_B$
$\mathbf{c}_{i+1,j}$
$\mathbf{c}_{i+1,j+1}$
$\mathbf{c}_B= 0.25\mathbf{c}_{i+1,j}+0.75\mathbf{c}_{i+1,j+1}$

Texture **sampling** is the process of reading textures through the GPU.

Minification

- Minification filtering is used when the texture image is smaller than the surface it is mapped onto, and it needs to be scaled down.
  - Pick and use the mipmap level that best matches the projected screen geometry resolution for texturing, applying constant or linear interpolation as needed. (point filtering)
  - Pick the two nearest mipmap levels that best match the projected screen geometry resolution for texturing (one will be bigger and one will be smaller than the screen geometry resolution). Next, apply constant or linear filtering to both of these mipmap levels to produce a texture color for each one. Finally, interpolate between these two texture color results. (linear filtering)

Anisotropic Filtering

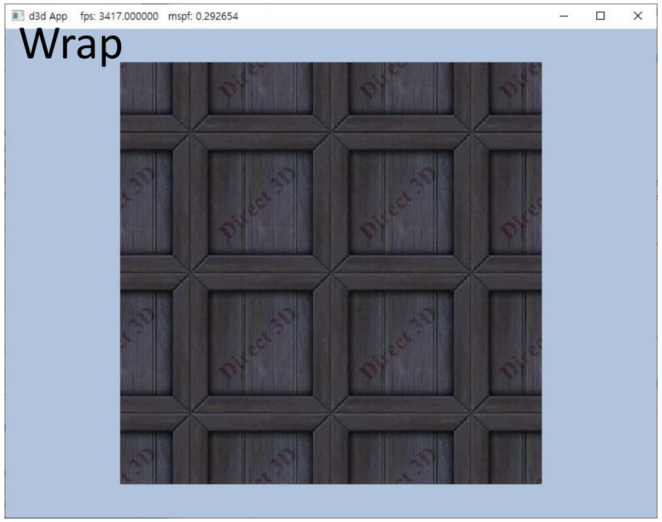
- Anisotropic filtering works by applying different amounts of filtering in different directions.
  - This filter helps alleviate the distortion that occurs when the angle between a polygon's normal vector and the camera's look vector is wide (e.g., when a polygon is orthogonal to the view window)

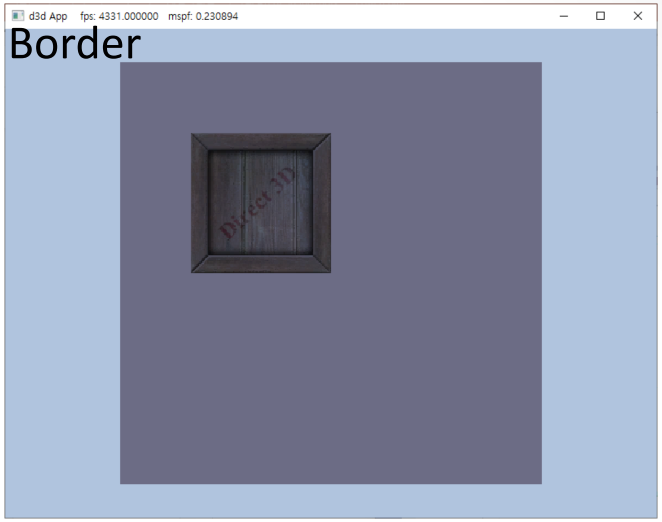
Anisotropic

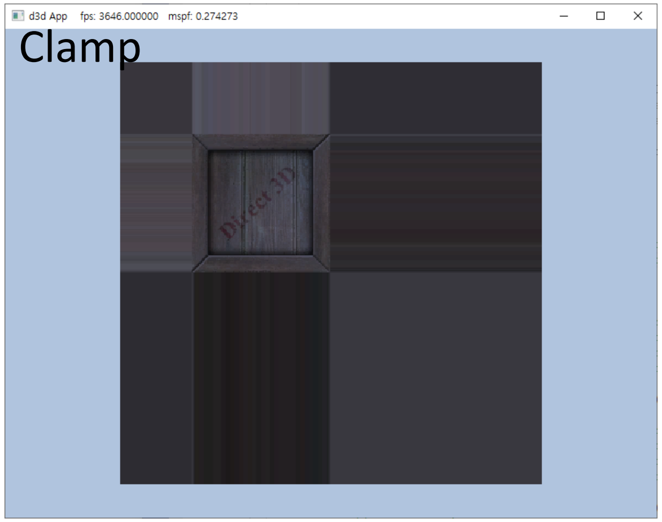
Address Modes (1)

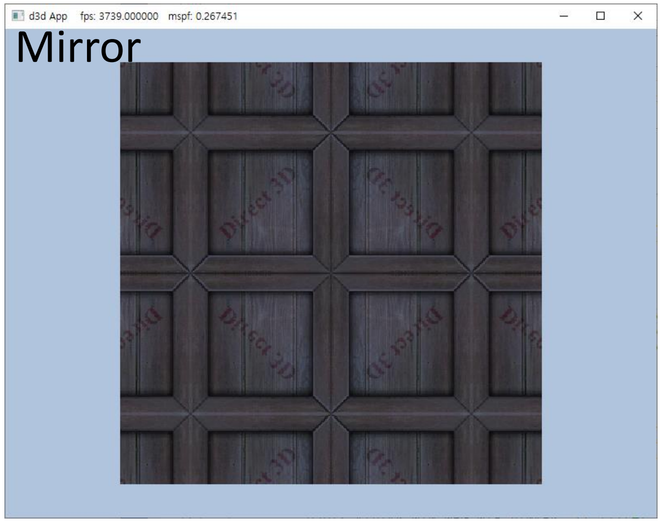
- A texture, combined with constant or linear interpolation, defines a vector-valued function $T(u, v) = (r, g, b, a)$.
- That is, given the texture coordinates $(u, v) \in [0, 1]^2$, the texture function T returns a color (r, g, b, a). Direct3D allows us to extend the domain of this function in four different ways (called address modes): wrap, border color, clamp, and mirror.
  - Wrap extends the texture function by repeating the image at every integer junction.
  - Border color extends the texture function by mapping each (u, v) not in $[0, 1]^2$ to some color specified by the programmer.
  - Clamp extends the texture function by mapping each (u, v) not in $[0, 1]^2$ to the color $T(u_0, v_0)$, where (u_0, v_0) is the nearest point to (u, v) contained in $[0, 1]^2$.
  - Mirror extends the texture function by mirroring the image at every integer junction.

Address Modes (2)



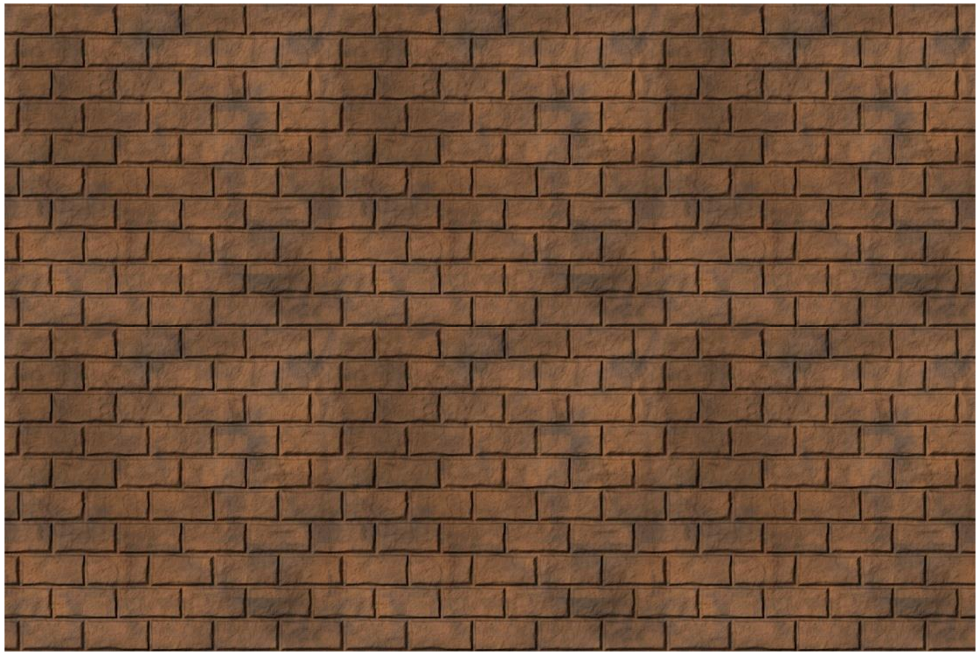






[−0.5, 2.5]

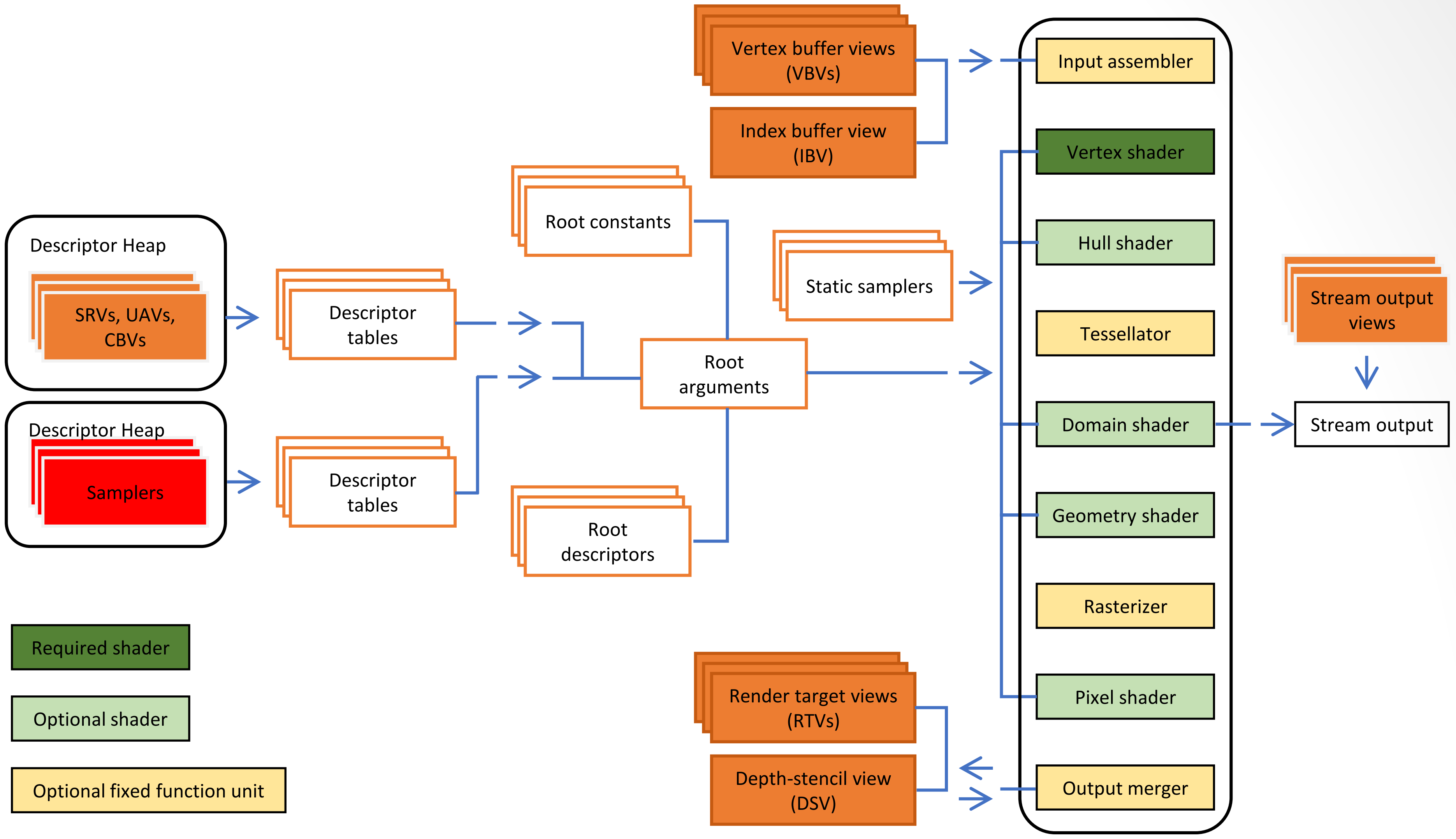
Address Modes (3)



Address Modes (4)

```
typedef enum D3D12_TEXTURE_ADDRESS_MODE {
        D3D12_TEXTURE_ADDRESS_MODE_WRAP      = 1,
        D3D12_TEXTURE_ADDRESS_MODE_MIRROR    = 2,
        D3D12_TEXTURE_ADDRESS_MODE_CLAMP     = 3,
        D3D12_TEXTURE_ADDRESS_MODE_BORDER    = 4,
        D3D12_TEXTURE_ADDRESS_MODE_MIRROR_ONCE       = 5
} D3D12_TEXTURE_ADDRESS_MODE;
```

Creating Samplers (1)



Creating Samplers (2)

- Samplers are used in shaders for texture sampling.
- In order to bind samplers to shaders for use, we need to bind descriptors to sampler objects.
- Root signature using descriptor tables

```
CD3DX12_DESCRIPTOR_RANGE descRange[3];
descRange[0].Init(D3D12_DESCRIPTOR_RANGE_TYPE_SRV, 1, 0);
descRange[1].Init(D3D12_DESCRIPTOR_RANGE_TYPE_SAMPLER, 1, 0);
descRange[2].Init(D3D12_DESCRIPTOR_RANGE_TYPE_CBV, 1, 0);

CD3DX12_ROOT_PARAMETER rootParameters[3];
rootParameters[0].InitAsDescriptorTable(1, &descRange[0],
    D3D12_SHADER_VISIBILITY_PIXEL);
rootParameters[1].InitAsDescriptorTable(1, &descRange[1],
    D3D12_SHADER_VISIBILITY_PIXEL);
rootParameters[2].InitAsDescriptorTable(1, &descRange[2],
    D3D12_SHADER_VISIBILITY_ALL);

CD3DX12_ROOT_SIGNATURE_DESC descRootSignature;
descRootSignature.Init(3, rootParameters, 0, nullptr,
    D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);
```

Creating Samplers (3)

- Sampler heap for using the sampler descriptor

```
D3D12_DESCRIPTOR_HEAP_DESC descHeapSampler = {};
descHeapSampler.NumDescriptors = 1;
descHeapSampler.Type = D3D12_DESCRIPTOR_HEAP_TYPE_SAMPLER;
descHeapSampler.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;

ComPtr mSamplerDescriptorHeap;
mDevice->CreateDescriptorHeap(&descHeapSampler,
    __uuidof(ID3D12DescriptorHeap), (void**)&mSamplerDescriptorHeap);
```

- Sampler descriptor

```
typedef struct D3D12_SAMPLER_DESC {
 D3D12_FILTER Filter;
 D3D12_TEXTURE_ADDRESS_MODE AddressU;
 D3D12_TEXTURE_ADDRESS_MODE AddressV;
 D3D12_TEXTURE_ADDRESS_MODE AddressW;
 FLOAT MipLODBias;
 UINT MaxAnisotropy;
 D3D12_COMPARISON_FUNC ComparisonFunc;
 FLOAT BorderColor[4];
 FLOAT MinLOD;
 FLOAT MaxLOD;
} D3D12_SAMPLER_DESC;
```

Creating Samplers (4)

```
D3D12_SAMPLER_DESC samplerDesc = {};
samplerDesc.Filter = D3D12_FILTER_MIN_MAG_MIP_LINEAR;
samplerDesc.AddressU = D3D12_TEXTURE_ADDRESS_MODE_WRAP;
samplerDesc.AddressV = D3D12_TEXTURE_ADDRESS_MODE_WRAP;
samplerDesc.AddressW = D3D12_TEXTURE_ADDRESS_MODE_WRAP;
samplerDesc.MinLOD = 0;
samplerDesc.MaxLOD = D3D12_FLOAT32_MAX;
samplerDesc.MipLODBias = 0.0f;
samplerDesc.MaxAnisotropy = 1;
samplerDesc.ComparisonFunc = D3D12_COMPARISON_FUNC_ALWAYS;
md3dDevice->CreateSampler(&samplerDesc,
   mSamplerDescriptorHeap->GetCPUDescriptorHandleForHeapStart());

commandList->SetGraphicsRootDescriptorTable(1,
   samplerDescriptorHeap->GetGPUDescriptorHandleForHeapStart());
```

Static Samplers (1)

- Graphics applications usually only use a handful of samplers.
  - Therefore, Direct3D provides a special shortcut to define an array of samplers and set them without going through the process of creating a sampler heap.
  - The **`Init`** function of the **`CD3DX12_ROOT_SIGNATURE_DESC`** class has two parameters that allow you to define an array of so-called static samplers your application can use.
- Static samplers are described by the D3D12_STATIC_SAMPLER_DESC structure.
  - This structure is very similar to **`D3D12_SAMPLER_DESC`**, with the following exceptions:
    - There are some limitations on what the border color can be
    - It contains additional fields to specify the shader register, register space, and shader visibility, which would normally be specified as part of the sampler heap.
    - In addition, you can only define 2032 number of static samplers, which is more than enough for most applications.

Static Samplers (2)

- In this lecture, we use the following static samplers.

```
std::array TexColumnsApp::GetStaticSamplers() {
  const CD3DX12_STATIC_SAMPLER_DESC pointWrap(
    0, // shaderRegister
    D3D12_FILTER_MIN_MAG_MIP_POINT,   // filter
    D3D12_TEXTURE_ADDRESS_MODE_WRAP,  // addressU
    D3D12_TEXTURE_ADDRESS_MODE_WRAP,  // addressV
    D3D12_TEXTURE_ADDRESS_MODE_WRAP); // addressW
  const CD3DX12_STATIC_SAMPLER_DESC pointClamp(
    1, // shaderRegister
    D3D12_FILTER_MIN_MAG_MIP_POINT,   // filter
    D3D12_TEXTURE_ADDRESS_MODE_CLAMP, // addressU
    D3D12_TEXTURE_ADDRESS_MODE_CLAMP, // addressV
    D3D12_TEXTURE_ADDRESS_MODE_CLAMP);// addressW
  // …
  return { pointWrap, pointClamp, /* … */  };
}
```

Static Samplers (3)

```
void Application::BuildRootSignature() {
    CD3DX12_DESCRIPTOR_RANGE texTable;
    texTable.Init(D3D12_DESCRIPTOR_RANGE_TYPE_SRV,
        1,  // number of descriptors
        0); // register t0

    CD3DX12_ROOT_PARAMETER slotRootParameter[4];
    slotRootParameter[0].InitAsDescriptorTable(1, &texTable,
       D3D12_SHADER_VISIBILITY_PIXEL);
    slotRootParameter[1].InitAsConstantBufferView(0); // register b0
    slotRootParameter[2].InitAsConstantBufferView(1); // register b1
    slotRootParameter[3].InitAsConstantBufferView(2); // register b2

    auto staticSamplers = GetStaticSamplers();
    // A root signature is an array of root parameters.
    CD3DX12_ROOT_SIGNATURE_DESC rootSigDesc(4, slotRootParameter,
            (UINT)staticSamplers.size(), staticSamplers.data(),
    D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);

    // …
}
```

Sampling Textures in a Shader

- A texture object is defined in HLSL and assigned to a texture register with the following syntax:

```
Texture2D gDiffuseMap : register(t0);
```

- Sampler objects are defined in HLSL and assigned to a sampler register with the following syntax:

```
SamplerState gsamPointWrap : register(s0);
SamplerState gsamPointClamp : register(s1);
// …
SamplerState gsamAnisotropicClamp : register(s5);
```

- For a pair of texture coordinates (u, v) for a pixel in the pixel shader, we actually sample a texture using the **Texture2D::Sample** method:

```
float4 diffuseAlbedo =
  gDiffuseMap.Sample(gsamAnisotropicWrap, pin.TexC) *
  gDiffuseAlbedo; // pin is the input parameter of PS
```

Specifying Texture Coordinates

```
// Common/GeometryGenerator.cpp
GeometryGenerator::MeshData GeometryGenerator::CreateBox(
 float width, float height, float depth, uint32 numSubdivisions) {
   MeshData meshData;
   Vertex v[24];
   float w2 = 0.5f*width;
   float h2 = 0.5f*height;
   float d2 = 0.5f*depth;
   v[0] = Vertex(-w2, -h2, -d2, /* … */, 0.0f, 1.0f); // u, v
   v[1] = Vertex(-w2, +h2, -d2, /* … */, 0.0f, 0.0f);
   v[2] = Vertex(+w2, +h2, -d2, /* … */, 1.0f, 0.0f);
   v[3] = Vertex(+w2, -h2, -d2, /* … */, 1.0f, 1.0f);
   // …
}
```

Kyung Hee University
nize@khu.ac.kr

Creating the Texture

```
// CrateApp.cpp
std::unordered_map<std::string, std::unique_ptr<Texture>> mTextures;

void CrateApp::LoadTextures() {
  auto woodCrateTex = std::make_unique<Texture>();
  woodCrateTex->Name = "woodCrateTex";
  woodCrateTex->Filename = L"Textures/WoodCrate01.dds";
  DirectX::CreateDDSTextureFromFile12(md3dDevice.Get(),
      mCommandList.Get(), woodCrateTex->Filename.c_str(),
      woodCrateTex->Resource, woodCrateTex->UploadHeap);

  mTextures[woodCrateTex->Name] = std::move(woodCrateTex);
}
```

Creating the Materials

```
// CrateApp.cpp
std::unordered_map<std::string, std::unique_ptr<Material>> mMaterials;

void CrateApp::BuildMaterials() {
   auto woodCrate = std::make_unique<Material>();
   woodCrate->Name = "woodCrate";
   woodCrate->MatCBIndex = 0; // cbuffer cbMaterial : register(b2)
   woodCrate->DiffuseSrvHeapIndex = 0;
                           // Texture2D    gDiffuseMap : register(t0);
   woodCrate->DiffuseAlbedo = XMFLOAT4(1.0f, 1.0f, 1.0f, 1.0f);
   woodCrate->FresnelR0 = XMFLOAT3(0.05f, 0.05f, 0.05f);
   woodCrate->Roughness = 0.2f;

   mMaterials["woodCrate"] = std::move(woodCrate);
}
```

Setting the Texture

```
// CrateApp.cpp
void CrateApp::DrawRenderItems(ID3D12GraphicsCommandList* cmdList,
   const std::vector<RenderItem*>& ritems) {
    // …
    for(size_t i = 0; i < ritems.size(); ++i) {
        auto ri = ritems[i];

        // …
      CD3DX12_GPU_DESCRIPTOR_HANDLE tex(
         mSrvDescriptorHeap->GetGPUDescriptorHandleForHeapStart());
      tex.Offset(ri->Mat->DiffuseSrvHeapIndex,mCbvSrvDescriptorSize);

      // …
      cmdList->SetGraphicsRootDescriptorTable(0, tex);
      // …
    }
}
```

Updated HLSL (1)

```
// hlsl
// …
Texture2D    gDiffuseMap : register(t0);
SamplerState gsamLinear  : register(s0);
// …

cbuffer cbPerObject : register(b0) {
    float4x4 gWorld;
    float4x4 gTexTransform; // This has not been discussed until now.
};

// cbuffer cbPass : register(b1)

cbuffer cbMaterial : register(b2) {
    float4 gDiffuseAlbedo;
    float3 gFresnelR0;
    float  gRoughness;
    float4x4 gMatTransform; // This has not been discussed until now.
};
```

Updated HLSL (2)

```
// hlsl
struct VertexIn {
    float3 PosL    : POSITION;
    float3 NormalL : NORMAL;
    float2 TexC    : TEXCOORD;
};

struct VertexOut {
    float4 PosH    : SV_POSITION;
    float3 PosW    : POSITION;
    float3 NormalW : NORMAL;
    float2 TexC    : TEXCOORD;
};

VertexOut VS(VertexIn vin) {
   VertexOut vout = (VertexOut)0.0f;

   // …

    float4 texC = mul(float4(vin.TexC, 0.0f, 1.0f), gTexTransform);
    vout.TexC = mul(texC, gMatTransform).xy;

    return vout;
}
```

Updated HLSL (3)

```
// hlsl
float4 PS(VertexOut pin) : SV_Target {
    float4 diffuseAlbedo
       = gDiffuseMap.Sample(gsamLinear, pin.TexC) * gDiffuseAlbedo;

    pin.NormalW = normalize(pin.NormalW);

    float3 toEyeW = normalize(gEyePosW - pin.PosW);

    float4 ambient = gAmbientLight*diffuseAlbedo;

    const float shininess = 1.0f - gRoughness;
    Material mat = { diffuseAlbedo, gFresnelR0, shininess };
    float3 shadowFactor = 1.0f;
    float4 directLight = ComputeLighting(gLights, mat, pin.PosW,
        pin.NormalW, toEyeW, shadowFactor);

    float4 litColor = ambient + directLight;

    litColor.a = diffuseAlbedo.a;

    return litColor;
}
```

Transforming Textures (1)

- Texture coordinates represent 2D points in the texture plane. Thus, we can translate, rotate, and scale them like we could any other point.
- Here are some example uses for transforming textures:
    - 1. A brick texture is stretched along a wall. The wall vertices currently have texture coordinates in the range [0, 1]. We scale the texture coordinates by 4 to scale them to the range [0, 4], so that the texture will be repeated four-by-four times across the wall.
    - 2. We have cloud textures stretching over a clear blue sky. By translating the texture coordinates as a function of time, the clouds are animated over the sky.
    - 3. Texture rotation is sometimes useful for particle-like effects, where we rotate a fireball texture over time.

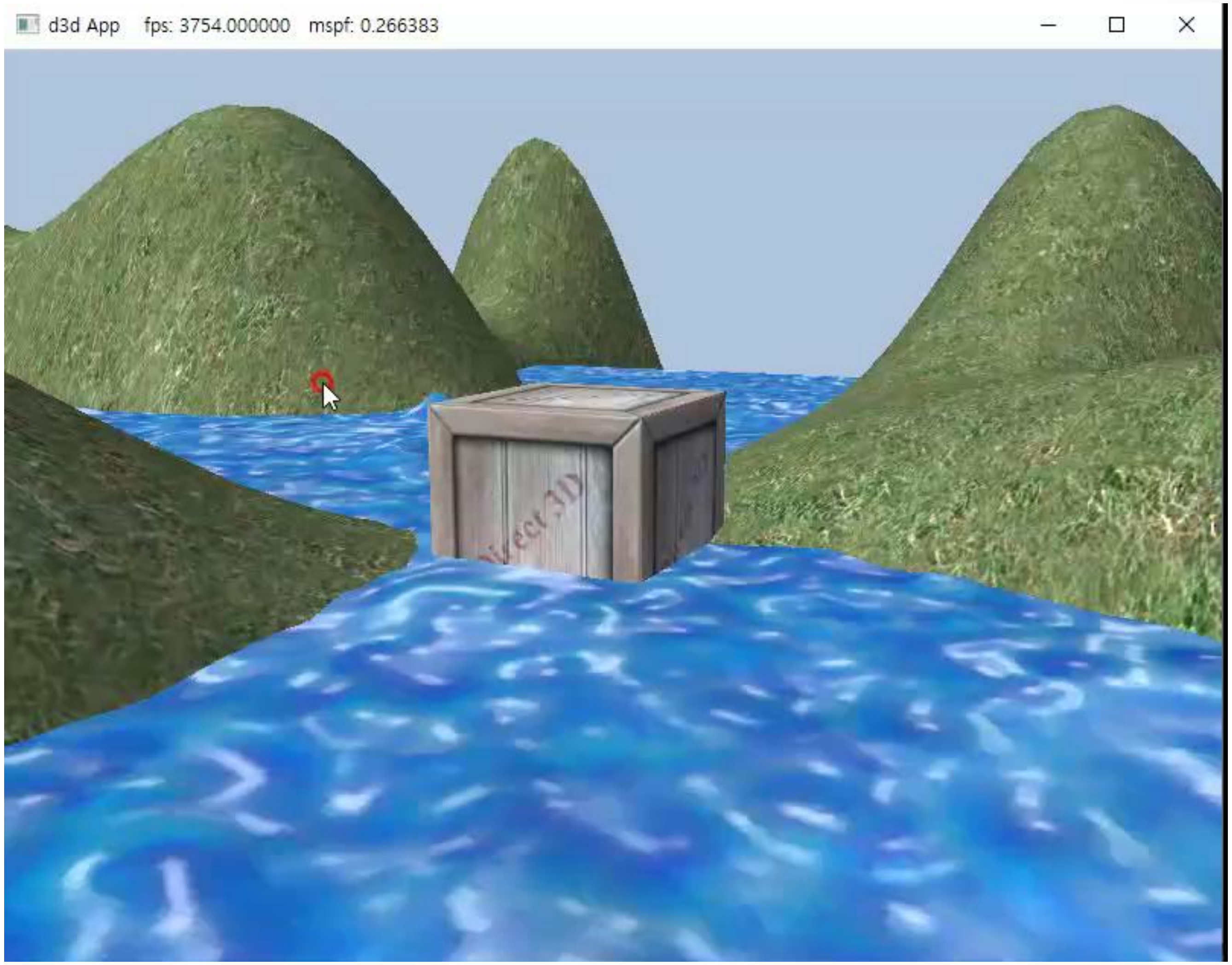
Transforming Textures (2)

```
cbuffer cbPerObject : register(b0) {
    float4x4 gWorld;
    float4x4 gTexTransform;
};
// std::vector<std::unique_ptr<RenderItem>> mAllRitems;
void Application::UpdateObjectCBs(const GameTimer& gt) {
    auto currObjectCB = mCurrFrameResource->ObjectCB.get();
    for(auto& e : mAllRitems) {
        if(e->NumFramesDirty > 0) {
            XMMATRIX world = XMLoadFloat4x4(&e->World);
            XMMATRIX texTransform = XMLoadFloat4x4(&e->TexTransform);

            ObjectConstants objConstants;
            XMStoreFloat4x4(&objConstants.World, XMMatrixTranspose(world));
            XMStoreFloat4x4(&objConstants.TexTransform,
                XMMatrixTranspose(texTransform));

            currObjectCB->CopyData(e->ObjCBIndex, objConstants);
            e->NumFramesDirty--;
        }
    }
}
```

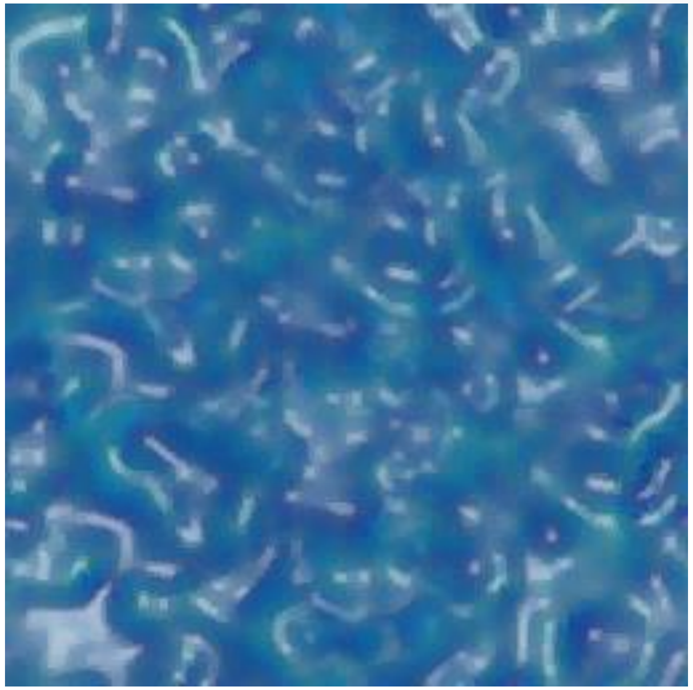
Transforming Textures (3)

```
cbuffer cbMaterial : register(b2) {
    float4 gDiffuseAlbedo;
    float3 gFresnelR0;
    float  gRoughness;
    float4x4 gMatTransform;
};
std::unordered_map<std::string, std::unique_ptr<Material>> mMaterials;
void Application::UpdateMaterialCBs(const GameTimer& gt) {
   auto currMaterialCB = mCurrFrameResource->MaterialCB.get();
   for(auto& e : mMaterials) {
      Material* mat = e.second.get();
      if(mat->NumFramesDirty > 0) {
         XMMATRIX matTransform = XMLoadFloat4x4(&mat->MatTransform);
         MaterialConstants matConstants;
         matConstants.DiffuseAlbedo = mat->DiffuseAlbedo;
         matConstants.FresnelR0 = mat->FresnelR0;
         matConstants.Roughness = mat->Roughness;
         XMStoreFloat4x4(&matConstants.MatTransform,
            XMMatrixTranspose(matTransform));
         currMaterialCB->CopyData(mat->MatCBIndex, matConstants);
         mat->NumFramesDirty--;
      }
   }
}
```

Textured Hills and Waves Demo (1)



Textured Hills and Waves Demo (2)

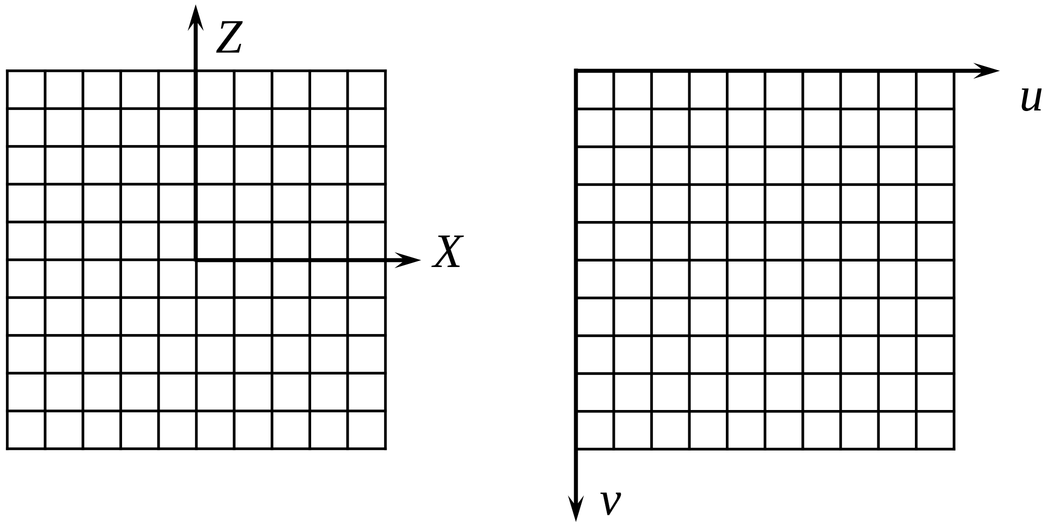


Grid Texture Coordinate Generation (1)

- [$m \times n$ grid in the xz-plane]
  - Texture space coordinates of the *ij*th vertex are:

$$u_{ij} = j \cdot \Delta u \qquad \Delta u = \frac{1}{n-1}$$

$$v_{ij} = i \cdot \Delta v \qquad \Delta v = \frac{1}{m-1}$$



Grid Texture Coordinate Generation (2)

```
// Common/GeometryGenerator.cpp
GeometryGenerator::MeshData GeometryGenerator::CreateGrid(
    float width, float depth, uint32 m, uint32 n) {
    MeshData meshData;

    uint32 vertexCount = m*n;       uint32 faceCount   = (m-1)*(n-1)*2;
    float halfWidth = 0.5f*width;   float halfDepth = 0.5f*depth;
    float dx = width / (n-1);       float dz = depth / (m-1);

    float du = 1.0f / (n-1);        float dv = 1.0f / (m-1);
    meshData.Vertices.resize(vertexCount);
    for(uint32 i = 0; i < m; ++i) {
        float z = halfDepth - i*dz;
        for(uint32 j = 0; j < n; ++j) {
            float x = -halfWidth + j*dx;
            meshData.Vertices[i*n+j].Position = XMFLOAT3(x, 0.0f, z);
            meshData.Vertices[i*n+j].Normal   = XMFLOAT3(0.0f, 1.0f, 0.0f);
            meshData.Vertices[i*n+j].TangentU = XMFLOAT3(1.0f, 0.0f, 0.0f);
            meshData.Vertices[i*n+j].TexC.x = j*du;
            meshData.Vertices[i*n+j].TexC.y = i*dv;
        }
    }
     // …
}
```

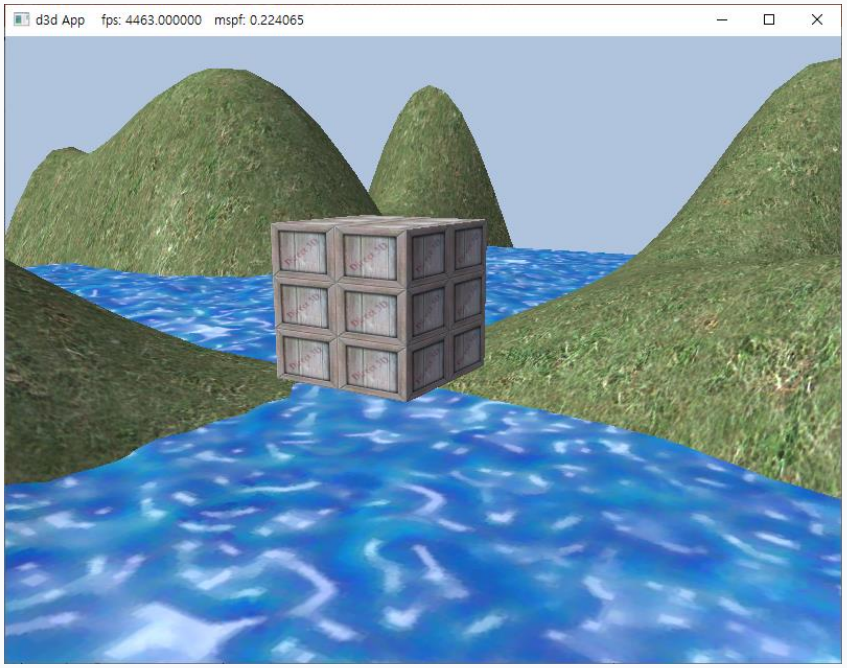
Texture Tiling (1)

- To tile the texture, we specify the wrap address mode and scale the texture coordinates by 5 using a texture transformation matrix.
  - $[0,1]^2 \rightarrow [0,5]^2$

```
void TexWavesApp::BuildRenderItems(){
   auto wavesRitem = std::make_unique<RenderItem>();
   wavesRitem->World = MathHelper::Identity4x4();
   XMStoreFloat4x4(&wavesRitem->TexTransform,
      XMMatrixScaling(5.0f, 5.0f, 1.0f));
   // …
}

// hlsl
float4 PS(VertexOut pin) : SV_Target{
   float4 diffuseAlbedo =
      gDiffuseMap.Sample(gsamAnisotropicWrap, pin.TexC) *
      gDiffuseAlbedo;
   // …
}
```

Texture Tiling (2)

```
void TexWavesApp::BuildRenderItems() {
// …
    auto boxRitem = std::make_unique<RenderItem>();
    XMStoreFloat4x4(&boxRitem->World, XMMatrixTranslation(3.0f, 5.0f, -9.0f));
    XMStoreFloat4x4(&boxRitem->TexTransform, XMMatrixScaling(2.0f, 3.0f, 1.0f));
// …
}
```

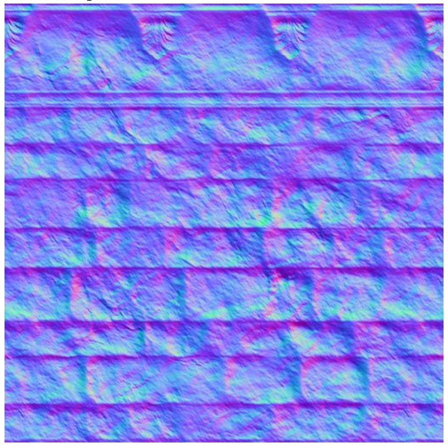


Texture Animation

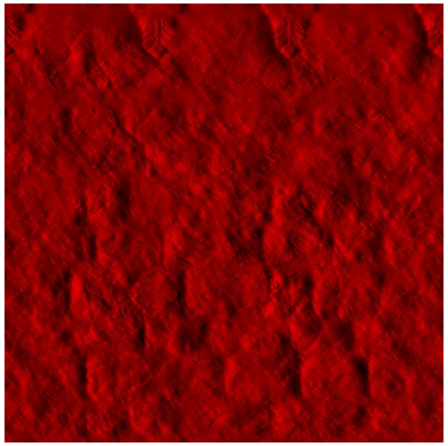
```
void TexWavesApp::AnimateMaterials(const GameTimer& gt) {
    auto waterMat = mMaterials["water"].get();

    float& tu = waterMat->MatTransform(3, 0);
    float& tv = waterMat->MatTransform(3, 1);

    tu += 0.1f * gt.DeltaTime();
    tv += 0.02f * gt.DeltaTime();

    if(tu >= 1.0f)
        tu -= 1.0f;

    if(tv >= 1.0f)
        tv -= 1.0f;

//  waterMat->MatTransform(3, 0) = tu;
//  waterMat->MatTransform(3, 1) = tv;

    waterMat->NumFramesDirty = gNumFrameResources;
}
```

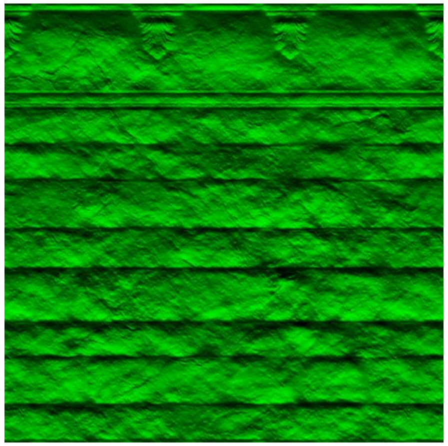
Normal Mapping (1)

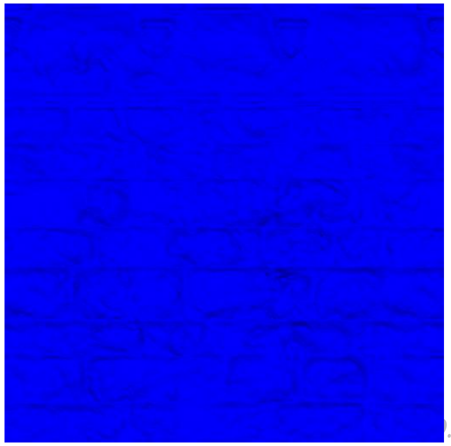
- A normal map is a texture, but instead of storing RGB data at each texel, we store a compressed x-coordinate, y-coordinate, and z-coordinate in the red component, green component, and blue component, respectively.



In the textbook, normal mapping is introduced in Chapter 19.

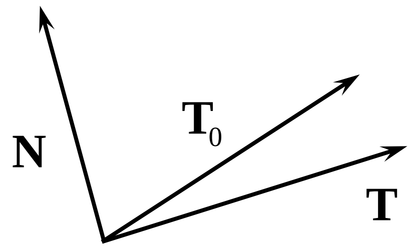






Normal Mapping (2)

- For 24-bit image, x coordinate in the range $[-1, 1]$: $f(x) = 255(0.5x+0.5)$
    - The x value is calculated by $f^{-1}(r)=2x/255-1$
    - Normalized RGB $[0, 1]$: $f^{-1}(r)=2x-1$
    - $\mathbf{n}_{tangent}$, texture space (tangent space) → We should transform the tangent space to the world space.
- The vertex normal vector $\mathbf{N}$.
- The tangent vector $\mathbf{T}$.
    - After the interpolation, the tangent vector ($\mathbf{T}_0$) and normal vector ($\mathbf{N}$) may not be orthonormal.
    - $\mathbf{T} = \text{normal}(\mathbf{T}_0 - (\mathbf{T}_0 \cdot \mathbf{N})\mathbf{N})$
- The bitangent vector $\mathbf{B}$.
    - $\mathbf{B} = \mathbf{N} \times \mathbf{T}$

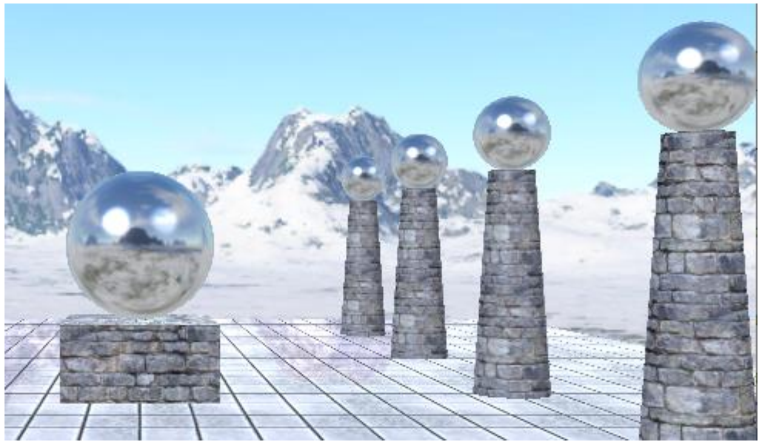


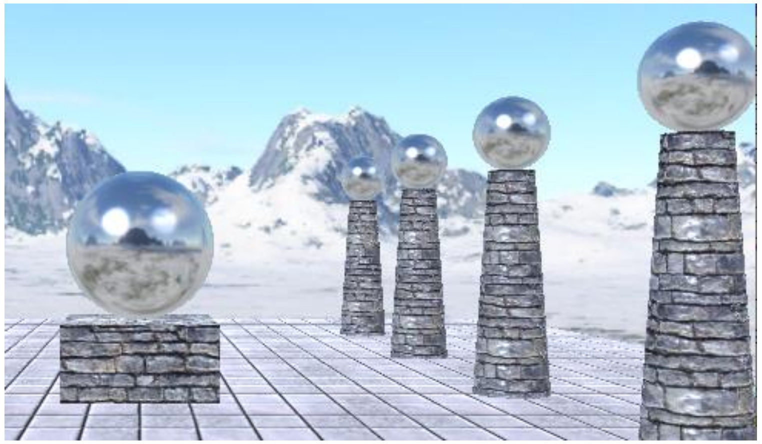
```
struct VertexIn{
     float3 PosL     : POSITION;
     float3 NormalL : NORMAL;
     float2 TexC     : TEXCOORD;
     float3 TangentU : TANGENT; };
struct VertexOut{
     float4 PosH     : SV_POSITION;
     float3 PosW     : POSITION;
     float3 NormalW : NORMAL;        // N
     float3 TangentW : TANGENT;      // T0
     float2 TexC     : TEXCOORD; };
```

Normal Mapping (3)

- World normal: $\mathbf{n}_{tangent}\,(\mathbf{M}_{object}\mathbf{M}_{world})$

$$M_{object} = \begin{pmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \\ N_x & N_y & N_z \end{pmatrix}$$


Without normal mapping


With normal mapping