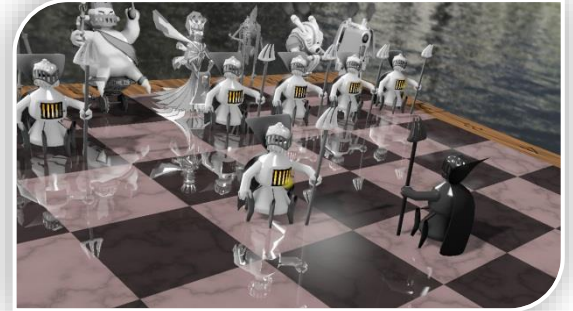
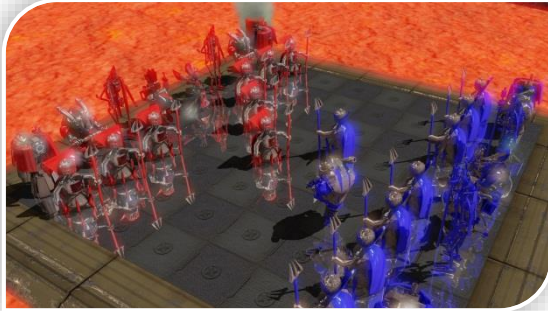
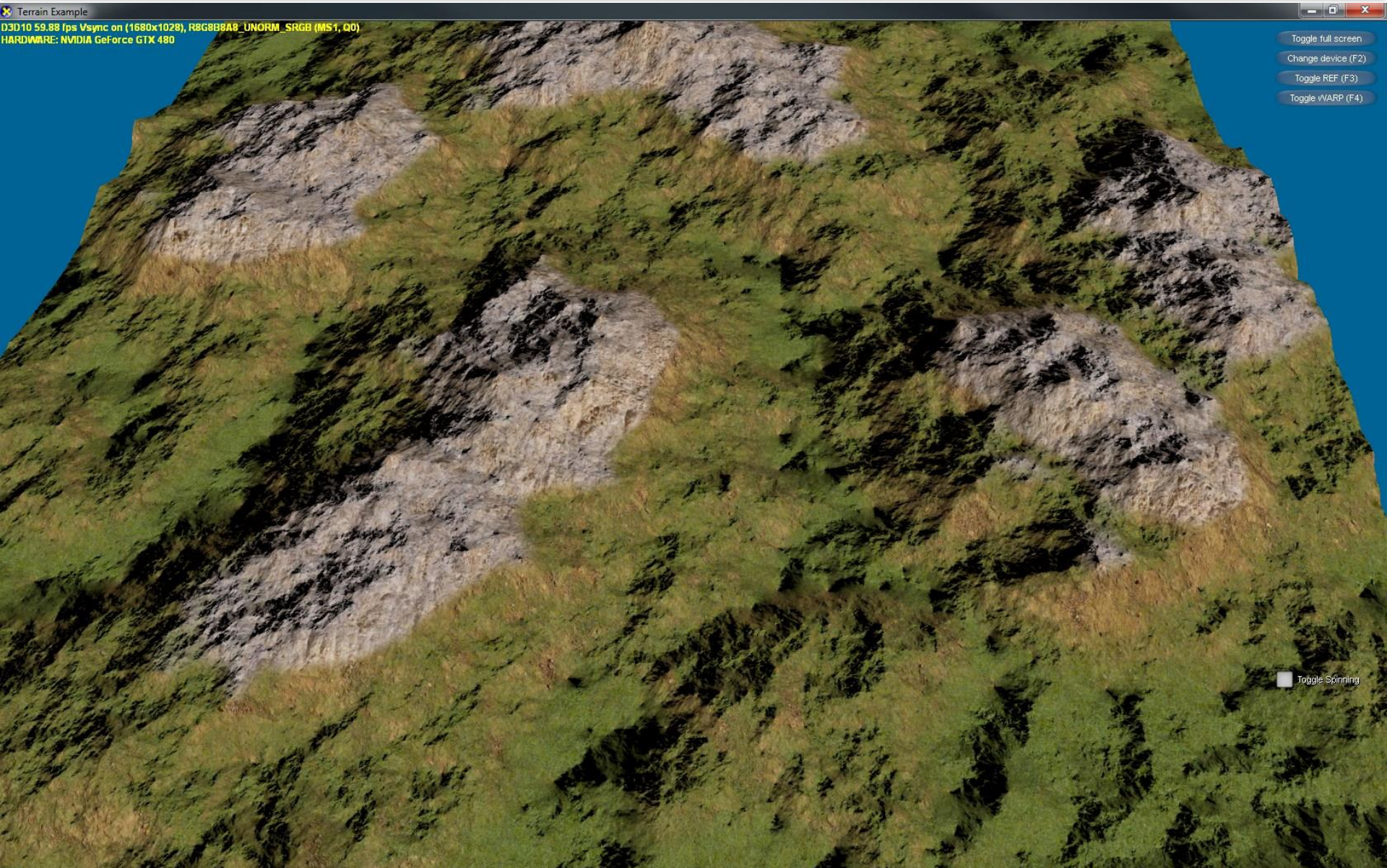


Praktikum: Echtzeit Computergrafik



tum.3D
computer graphics & visualization

This week: Realtime Rendering



Goal:

- Render the terrain yourself (implement the viewer)
- Input: height map + color texture + normal texture
- Output: realtime rendering of a lit and textured landscape
- Using only the “fixed” function pipeline (no shader programs)

- Direct3D is a graphics API to access the underlying hardware
 - Part of DirectX
 - Provides GPU memory allocation / deallocation („GPU resources“), pipeline configuration, draw calls and much more...

Microsoft®
DirectX®



- DirectX API is based on abstract interfaces
 - You never work on real class instances
 - DirectX allocates and deallocates its resources
 - Available through pointers to `IUnknown` (or a descendent of it)
 - Lifecycles management: Internal reference counters via `IUnknown::AddRef()` and `IUnknown::Release()`

- AddRef / Release: Similar to new and delete
 - Each AddRef() requires a Release()
 - DirectX calls AddRef() automatically whenever a pointer to an Interface is created

```
ID3D11Device* device;  
...  
ID3D11DeviceContext* context;  
device->GetImmediateContext(&context);           // Calls AddRef() on the created context
```

- Two main Direct3D interfaces

ID3D11Device

- Access to a Direct3D device (usually a single GPU)
- Used to **create** all resources for a GPU

ID3D11DeviceContext

- Manages the current pipeline state
- Used to **bind** resources to various pipeline stages
- Used for draw calls

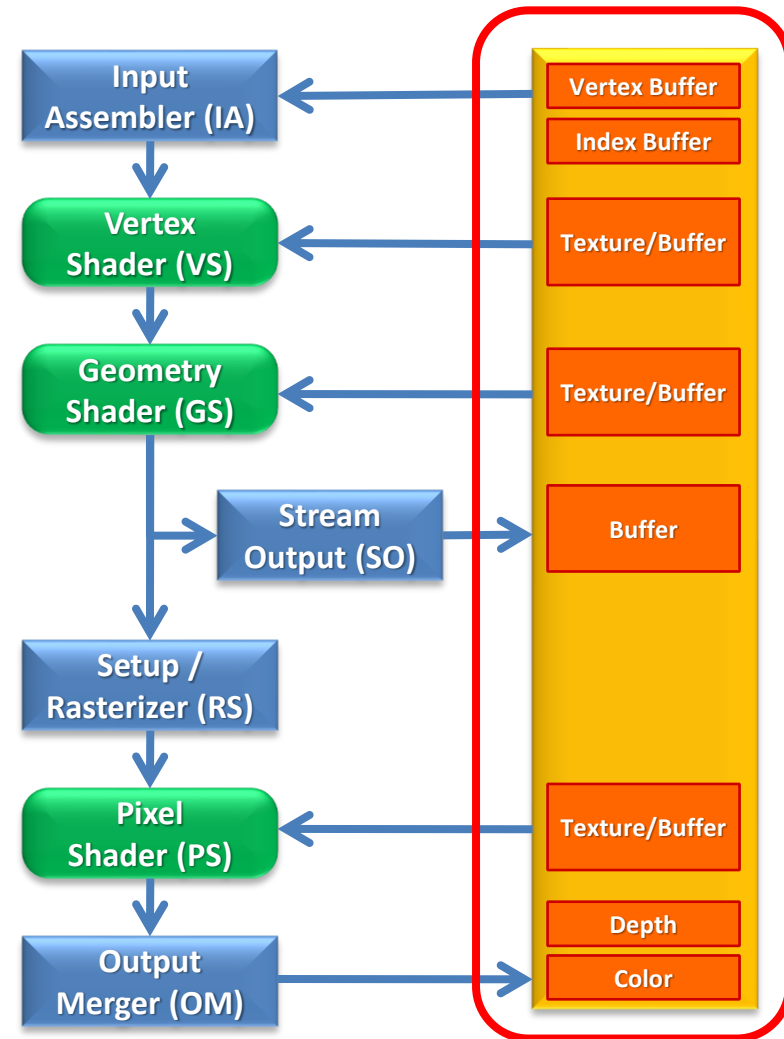
- ID3D11DeviceContext: State machine



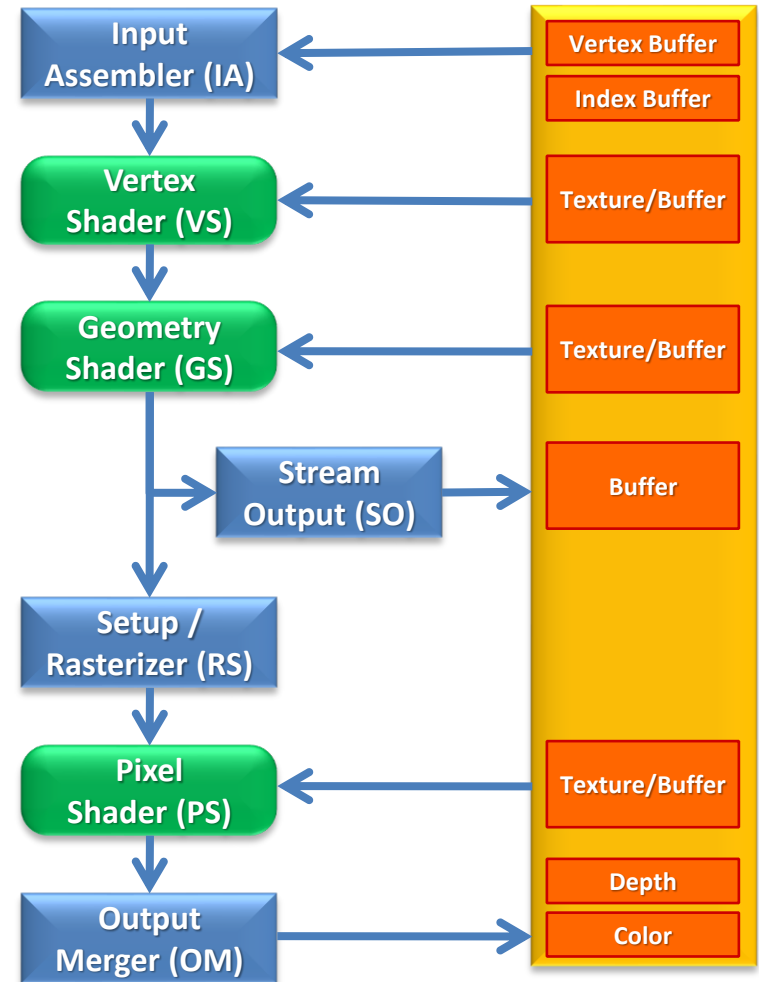
- **Resource:** An object allocated in GPU memory
 - Buffers
 - Vertex / Index Buffer: Vertex data and connectivity
 - ConstantBuffer: Variables accessed in GPU shader programs
 - Textures
 - Texture1D, Texture2D, Texture3D
 - Like buffers with additional functionality
 - Multi-dimensional access, hardware interpolation
- All these resources are read-only for the GPU

- All resources descend from `ID3D11Resource`
 - `ID3D11Buffer`
 - `ID3D11Texture2D`
- Creation:
 - `ID3D11Device::CreateBuffer()`
 - `ID3D11Device::CreateTexture2D()`
 - ...

- Resources are **bound** to the graphics pipeline
 - Draw call: Process a fixed number of vertices
 - Resources cannot change during a draw call
 - Different stages may have access to different resources
- Pipeline state:
`ID3D11DeviceContext`



- Purpose:
 - Generate vertex data from input
- Input:
 - Vertex Buffers + Index Buffer
- Output:
 - Vertices with attributes
- Controllable through:
 - `IASetVertexBuffers / IASetIndexBuffer`
 - `IASetInputLayout`
 - `IASetPrimitiveTopology`



The *Input Assembler* stage supplies geometry data (e.g. Lines or Triangles) for the rest of the pipeline

- It reads user defined data blocks and
 - Uses an *Input Layout* to interpret the data
 - Generates a set of geometric primitives controlled by `D3D11_PRIMITIVE_TOPOLOGY`
 - `D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST`
 - Supplies the assembled primitives to the rest of the pipeline
- The elemental unit thereby is the edge point (vertex), which can carry arbitrary user defined attributes (e.g. position, normal, color, ...)

- Vertices contain arbitrary user defined attributes
- Most common:
 - Position(s)
 - Normal(s)
 - Color(s)
 - Texture coordinate(s)
 - Material parameter(s)

- C++ side: vertices are stored in linear arrays

```
float triangle[] = {  
    // Vertex 0  
    0.0f, 0.0f, 0.0f, 1.0f, // Position  
    0.0f, 1.0f, 0.0f, 0.0f, // Normal  
    0.0f, 0.0f, // Texcoords  
  
    // Vertex 1  
    1.0f, 0.0f, 0.0f, 1.0f, // Position  
    0.0f, 1.0f, 0.0f, 0.0f, // Normal  
    0.0f, 1.0f, // Texcoords  
  
    // Vertex 2  
    0.0f, 0.0f, 1.0f, 1.0f, // Position  
    0.0f, 1.0f, 0.0f, 0.0f, // Normal  
    1.0f, 0.0f, // Texcoords  
};
```

- Usually represented by a **struct** for cleaner code:

```
– struct SimpleVertex  
{  
    DirectX::XMFLOAT4 Pos;           // Position  
    DirectX::XMFLOAT4 Normal;       // Normal  
    DirectX::XMFLOAT2 UV;           // Texture coordinates  
};
```

- The GPU gets the raw data in one (or several) buffer(s)
- Additionally it needs an ***Input-Layout*** to interpret the data
- A **D3D11_INPUT_ELEMENT_DESC** thereby describes each vertex attribute
- Must be consistent with the vertex on C++ (struct) and HLSL (GPU vertex shader input) sides!
- Is already defined in our template!

- Example:

```
// Define the input layout
const D3D11_INPUT_ELEMENT_DESC layout[] = // http://msdn.microsoft.com/en-us/library/bb205117%28v=vs.85%29.aspx
{
    { "SV_POSITION",    0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 0, D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "NORMAL",         0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 16, D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "TEXCOORD",       0, DXGI_FORMAT_R32G32_FLOAT,        0, 32, D3D11_INPUT_PER_VERTEX_DATA, 0 },
};
UINT numElements = sizeof( layout ) / sizeof( layout[0] );

// Create the input layout
D3DX11_PASS_DESC pd;
V_RETURN(g_gameEffect.pass0->GetDesc(&pd));
V_RETURN( pd3dDevice->CreateInputLayout( layout, numElements, pd.pIAInputSignature,
    pd.IAInputSignatureSize, &g_terrainVertexLayout ) );
```

- To create the InputLayout, the *input signature* of the vertex shader is required (here: signature of a *pass*)
- The input layout is bound to the pipeline via:

```
// Set input layout
pd3dImmediateContext->IASetInputLayout( g_terrainVertexLayout );
```

- Now the GPU knows how to interpret the data, but it still needs the data itself.
- Therefore we create a ***Vertex Buffer*** on the GPU and fill it with our data.
- Creation requires two additional structures:
 - `D3D11_BUFFER_DESC` describes the buffer
 - `D3D11_SUBRESOURCE_DATA` to deliver the data

- Vertex buffer creation:

```
// Create and fill description
D3D11_SUBRESOURCE_DATA id;
id.pSysMem = &triangle[0];
id.SysMemPitch = 10 * sizeof(float); // Stride
id.SysMemSlicePitch = 0;

// Define initial data
D3D11_BUFFER_DESC bd;
bd.BindFlags = D3D11_BIND_VERTEX_BUFFER;
bd.ByteWidth = triangle.size() * sizeof(float);
bd.CPUAccessFlags = 0;
bd.MiscFlags = 0;
bd.Usage = D3D11_USAGE_DEFAULT;

// Create buffer
V(device->CreateBuffer(&bd, &id, &vertexBuffer));
```

- Vertex buffer binding:

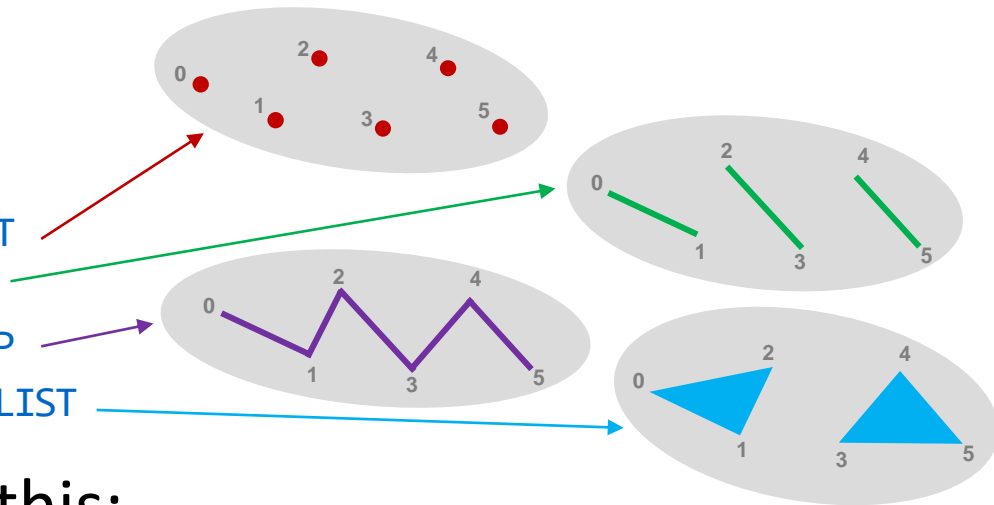
```
context->IASetVertexBuffers(0, 1, vbs, strides, offsets);
```

- Stride: size of one element
- Offset: offset of the first element in the buffer

- Now the GPU has the vertex data and knows how to interpret it
- Now we tell it the kind of geometric primitive we want to describe

- Examples:

- `D3D11_PRIMITIVE_TOPOLOGY_POINTLIST`
- `D3D11_PRIMITIVE_TOPOLOGY_LINELIST`
- `D3D11_PRIMITIVE_TOPOLOGY_LINESTRIP`
- `D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST`



- We set the topology like this:

```
context->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_LINESTRIP);
```

Execute the Rendering (Draw)

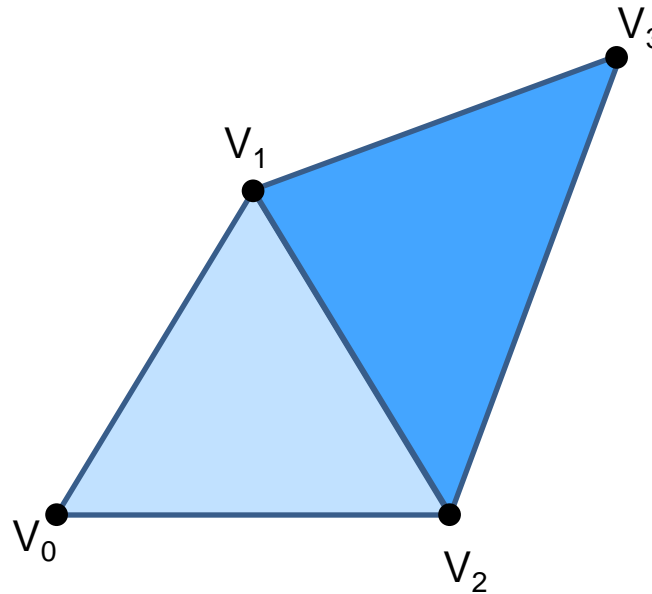
- When the pipeline state is completely set, we „draw“ our data using a certain *pass* of an effect *technique*:

```
// Render
D3D11_TECHNIQUE_DESC techDesc;
g_pTechnique->GetDesc( &techDesc );
for( UINT p = 0; p < techDesc.Passes; ++p ) {
    // A technique may require multiple rendering passes
    g_pTechnique->GetPassByIndex( p )->Apply(0, context);
    context->Draw(3, 0);
}
```

Number of vertices
to draw

Index of the first
vertex within the
buffer to draw

Draw() vs. DrawIndexed()



- Draw
 - $VB[] = \{V_0, V_1, V_2, V_2, V_1, V_3\};$
 - Implicit topology
 - **Redundant data, e.g.**
 - Position, Normal, UV ...
- DrawIndexed
 - $VB[] = \{V_0, V_1, V_2, V_3\};$
 - $IB[] = \{0, 1, 2, 2, 1, 3\};$
 - Explicit topology

- Creation similar to Vertex Buffer:

```
// Create and fill description
ZeroMemory(&bd, sizeof(bd));
bd.Usage = D3D11_USAGE_DEFAULT;
bd.ByteWidth = sizeof(unsigned int) * indices.size();
bd.BindFlags = D3D11_BIND_INDEX_BUFFER;
bd.CPUAccessFlags = 0;
bd.MiscFlags = 0;

// Define initial data
ZeroMemory(&id, sizeof(id));
id.pSysMem = &indices[0];

// Create Buffer
V(device->CreateBuffer( &bd, &id, &indexBuffer ));
```

The main
difference

- Binding:

```
context->IASetIndexBuffer(indexBuffer, DXGI_FORMAT_R32_UINT, 0);
```

- Slightly different draw command:

...

```
context->DrawIndexed( 6 , 0 , 0 );
```

...

Number of indices to
draw

Index of the first
vertex within the
vertex buffer to draw
("Index" is relative to
this)

Index of the first
index within the index
buffer to draw

- Shader programs do not access resources directly
- Instead: **ResourceViews** are bound to the pipeline
 - `ID3D11ShaderResourceView`
 - `ID3D11Device::CreateShaderResourceView()`
 - Each resource needs at least one ResourceView when it is accessed by a shader
- ResourceViews offer more flexible control
 - E.g. only bind a specific mip-map of a texture

```
HRESULT CreateShaderResourceView(  
    [in]    ID3D11Resource *pResource,  
    [in]    const D3D11_SHADER_RESOURCE_VIEW_DESC *pDesc,  
    [out]   ID3D11ShaderResourceView **ppSRView  
);
```

- **D3D11_SHADER_RESOURCE_VIEW_DESC** describes the resource view to be created (width, height, etc)
- Common usecase: Textures
 - You can pass **nullptr** for pDesc!

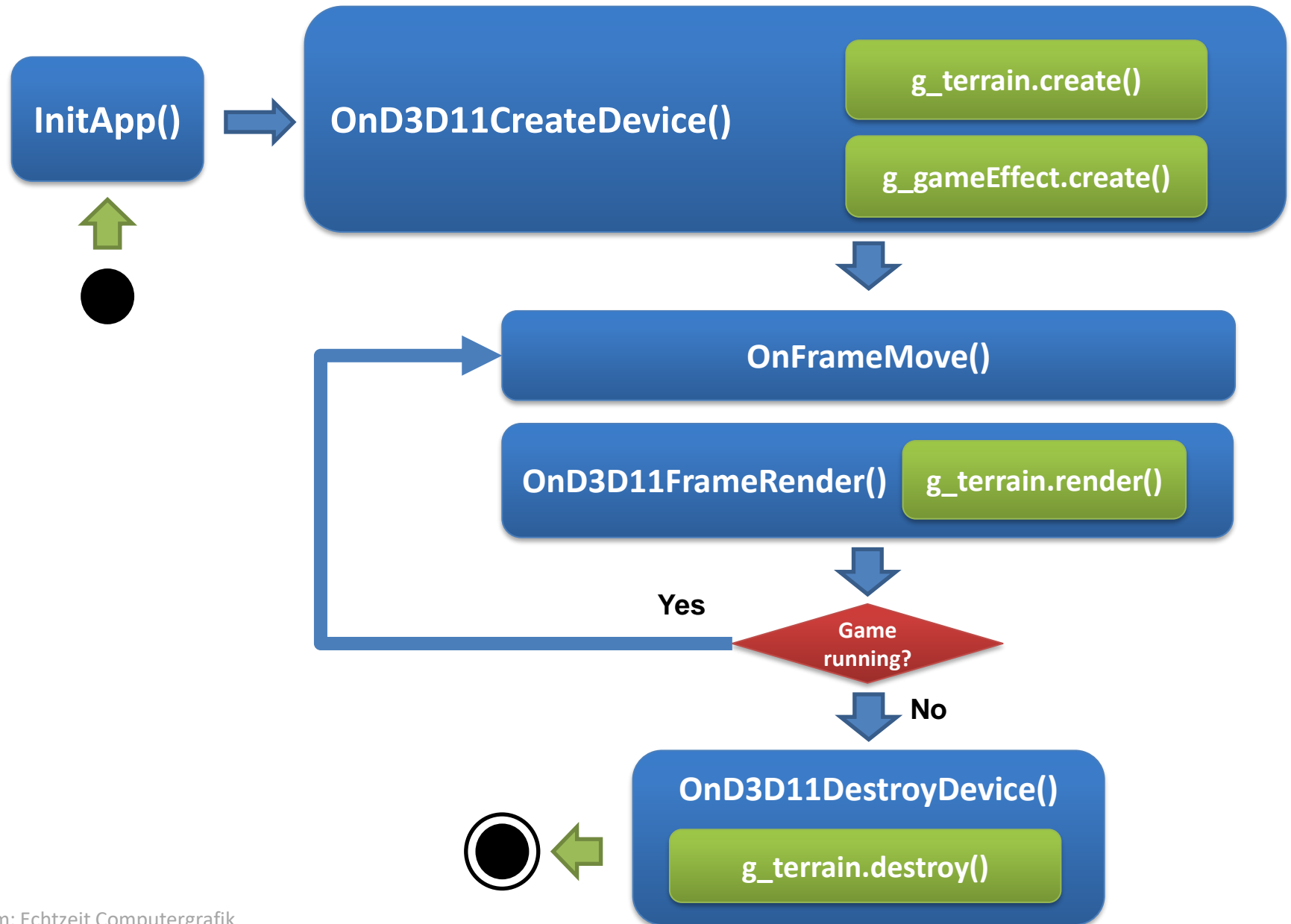
```
DirectX::CreateDDSTextureFromFile(device, L"resources\\debug_green.dds", nullptr, &debugSRV)
```

- DDSTextureLoader.h: Simple DDS loading
 - Creates a resource and a resource view
 - Can also be `nullptr` if you do not need both
 - Top example: Resource is also created, of course!
 - Released upon release of the resource view (by reference counting)
 - You'll need to release the resource view (debugSRV) yourself

- A sample project to get started fast
 - Initializes Direct3D
 - Manages the Window
 - Manages all Graphics Resources
 - Implements the Rendering Loop
 - Handles Mouse/Keyboard Inputs
 - Draws the User Interface
 - and much more
 - Looks complicated first, but you'll get used to it
- Alternative: Starting from scratch
 - Next three weeks would be Direct3D initialization...

- Important files
 - game.cpp
 - (De-)Initialization
 - Main rendering loop, calls all necessary functions
 - Keyboard / mouse callbacks
 - Terrain.h / Terrain.cpp
 - Terrain loading, (GPU) creation and rendering
 - game.fx
 - GPU shader code for terrain rendering
 - Not relevant for the current assignment!
 - GameEffect.h
 - The „CPU part“ of our shader (game.fx)
 - Access to „GPU variables“

Program Flow (Simplified)



- g_gameEffect is globally accessible
 - As long as you include „GameEffect.h“
 - In GameEffect.h:

```
extern GameEffect g_gameEffect;
```

- Tells the Linker: „g_gameEffect is instantiated in some other .obj file“
- Exactly one .cpp file must contain an instantiation of g_gameEffect
- See game.cpp:

```
GameEffect g_gameEffect;
```

- The template project makes heavy use of DXUT
 - The DirectX Utility Library (DXUT) simplifies the usage of the WINDOWS and D3D APIs
- DXUT was developed for the DXSDK samples
 - Keeps sample code clean and easy to understand
- DXUT helps with:
 - Window / device creation, window events (e.g. user input)
 - Simple user interfaces
 - ...

The template project: Main

```
int WINAPI wWinMain(...) {  
  
    DXUTSetCallbackMsgProc( MsgProc );  
    DXUTSetCallbackKeyboard( OnKeyboard );  
    DXUTSetCallbackFrameMove( OnFrameMove );  
    DXUTSetCallbackDeviceChanging( ModifyDeviceSettings );  
    DXUTSetCallbackD3D11DeviceAcceptable( IsD3D11DeviceAcceptable );  
    DXUTSetCallbackD3D11DeviceCreated( OnD3D11CreateDevice );  
    DXUTSetCallbackD3D11SwapChainResized( OnD3D11ResizedSwapChain );  
    DXUTSetCallbackD3D11SwapChainReleasing( OnD3D11ReleasingSwapChain );  
    DXUTSetCallbackD3D11DeviceDestroyed( OnD3D11DestroyDevice );  
    DXUTSetCallbackD3D11FrameRender( OnD3D11FrameRender );  
  
    InitApp();  
  
    DXUTInit( true, true, NULL );  
  
    DXUTSetCursorSettings( true, true );  
  
    DXUTCreateWindow( L"My Game" );  
  
    DXUTCreateDevice( D3D_FEATURE_LEVEL_10_0, true, 1280, 720 );  
  
    DXUTMainLoop();  
  
    return DXUTGetExitCode();  
}
```

- Interaction with DirectX and Windows is controlled via callback functions:
 - **DXUTSetCallbackD3D11DeviceCreated:**
Create (device->CreateX) and initialize everything which does **not** depend on the window size.
 - **DXUTSetCallbackD3D11SwapChainResized:**
Create and initialize everything which depends on the window size.
 - **DXUTSetCallbackD3D11SwapChainReleasing:**
Release everything which was *Created* in SwapChainResized.
 - **DXUTSetCallbackD3D11DeviceDestroyed:**
Release everything which was *Created* in DeviceCreated.

- Callback-functions continued:
 - **DXUTSetCallbackD3D11FrameRender:**
Render your scene (context->DrawX). Unless you control all device states yourself, also place all your context->SetX calls here.
 - **DXUTSetCallbackFrameMove:**
Called **before** the rendering. Place your sound, physics, game-logic, etc. here.
 - **DXUTSetCallbackMsgProc:**
Handle window messages (e.g. mouse/UI events).
 - **DXUTSetCallbackKeyboard:**
Handle keyboard events

- Additional features:
 - Camera-Classes
 - `CFirstPersonCamera` („First Person“)
 - `CModelViewerCamera` („Third Person“)
 - Graphical User Interface
 - Text rendering

- Load the config file
 - Parser provided (by you :P)
- Load the terrain files
 - Everything provided
- Convert the terrain heightfield into a vertex buffer
- Triangulate the terrain using an index buffer
- Create the terrain color texture
 - File format loader provided
- Draw the terrain
 - Complete the provided renderer
 - Some small modifications necessary
- Everything else:
 - Provided

- Heightfield:
 - Entry count
 - $N = \text{Res} * \text{Res}$
 - Entry
 - Height: h

Vertex position:	$p.x = [-\text{width} / 2,$ $p.y = [0,$ $p.x = [-\text{depth} / 2,$	$\text{width} / 2]$ $\text{height}]$ $\text{depth} / 2]$
Texture:	$t.u, t.v = [0.0,$	$1.0]$
Heightfield:	$i.x, i.y = [0,$	$\text{Res}-1]$

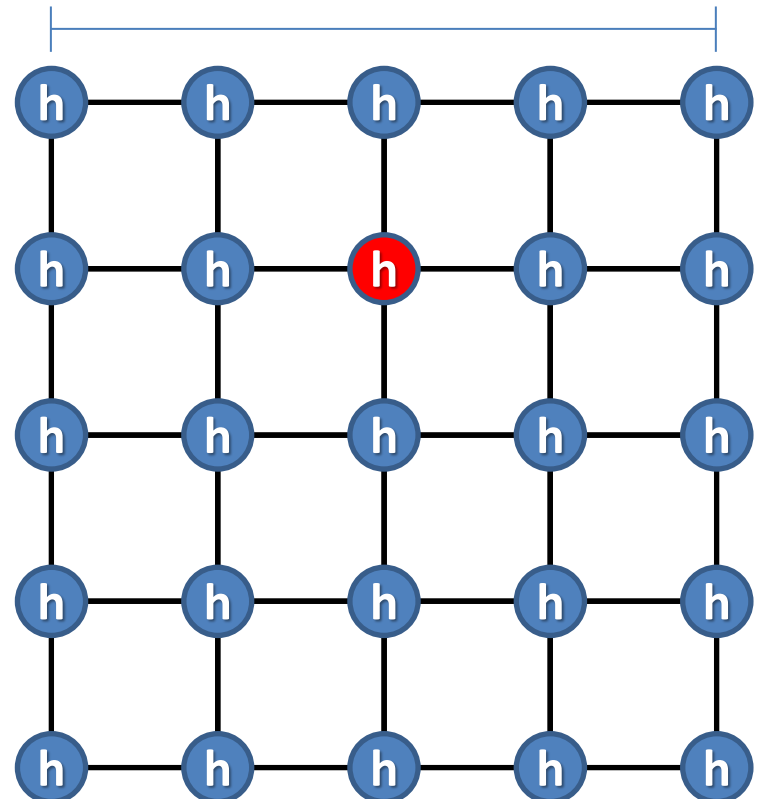
- Vertex buffer:

- Entry count
 - $N = \text{Res} * \text{Res}$
- Entry
 - Position: $p = (p.x, p.y, p.z, 1)$
 - Normal: $n = (n.x, n.y, n.z, 0)$
 - Tex Coord: $t = (t.u, t.v)$

For now, just recalculate this – the normalmap will be used next assignment!

- Conversion necessary

- Vertex buffer: „height“ is stored in y!
- $h = \text{hf}[\text{IDX}(i.x, i.y, \text{Res})]$
- $t.u = i.x / (\text{Res} - 1)$
- $p.x = (i.x / \text{Res} - 0.5) * \text{width}$
- $p.y = h * \text{height}$



Vertex buffer:


0	1	2	3	4	5	6	7
v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7

 ...

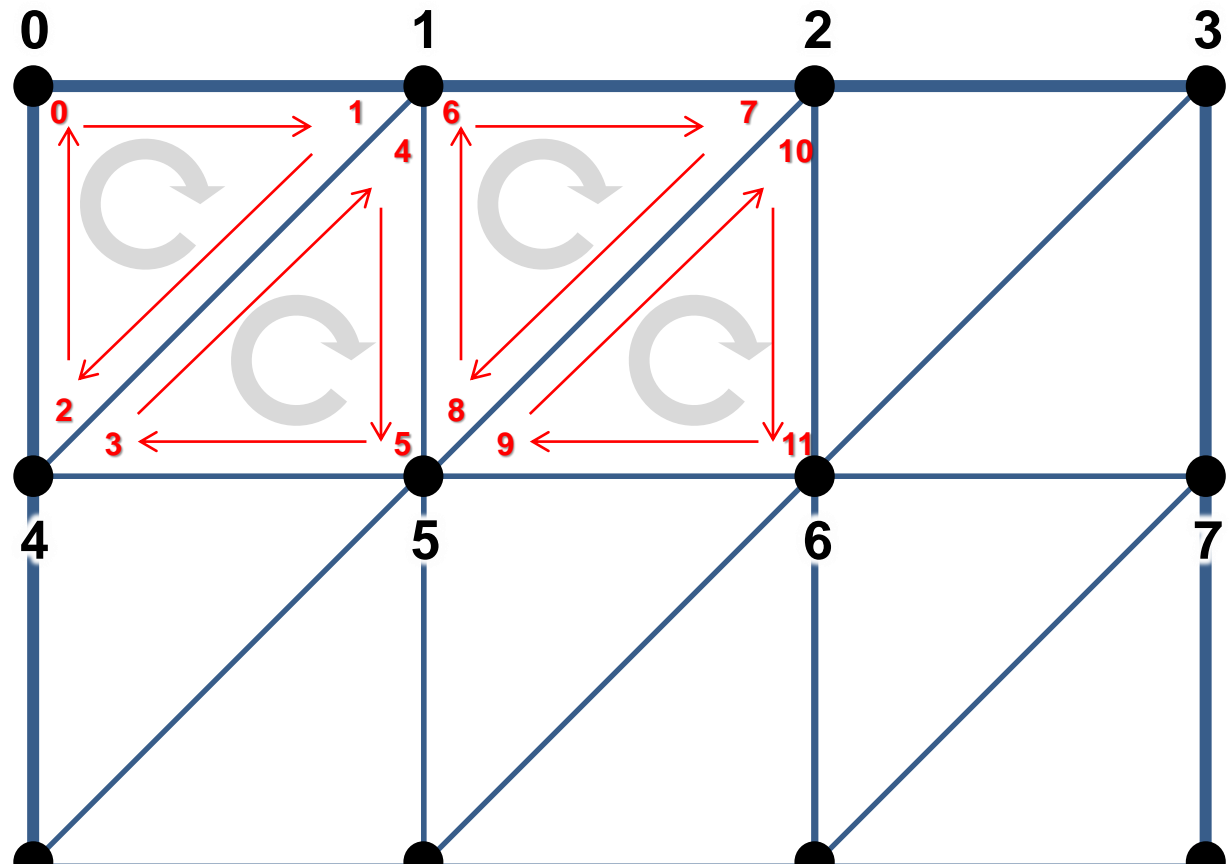
Index buffer:

0	1	2	3	4	5	6	7	8	9	10	11
0	1	4	4	1	5	1	2	5	5	2	6

 ...



- Index buffer
 - List of triangles
 - Triangle:
 - 3 vertex indices



- The Normal-Map isn't used at this point
 - For now, recalculate the normals from the heightfield
- The normals need to be scaled depending on your terrain configuration
 - Calculate them as if depth == width == height
 - Rescale the normals before storage in the vertex buffer

- Normal transformation: DirectXMath
 - [http://msdn.microsoft.com/en-us/library/windows/desktop/ee415574\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ee415574(v=vs.85).aspx)
 - #include <DirectXMath.h>
 - Contains Matrix / Vector operations in the DirectX namespace (DirectX::XM...)
 - Less typing effort: `using namespace DirectX;`
 - Only do this in .cpp files, not in header files!

`DirectX::XMVECTOR`: 4-component 32bit float vector

`DirectX::XMMATRIX`: 4x4 32bit float matrix

- Optimized for speed
 - You can not access member of those classes directly

```
XMATRIX matTest = XMMatrixIdentity();  
XMVECTOR vTest = XMVectorSet(x, y, z, 1);  
  
XMVECTOR vectorDest;  
XMStoreFloat4(&vectorDest, vTest);  
  
XMVECTOR4X4 matrixDest;  
XMStoreFloat4x4(&matrixDest, matTest);
```

- Important: Normals need to be scaled using the inverted, transposed matrices!

```
XMMATRIX matNormalScaling = XMMatrixScaling(x, y, z);  
matNormalScaling = XMMatrixTranspose(XMMatrixInverse(nullptr, matNormalScaling));
```

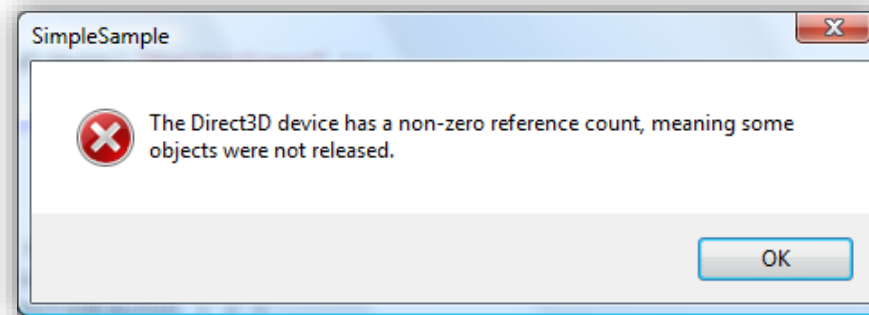
- Transform each normal vector

```
vNormal = XMVector4Transform(vNormal, matNormalScaling);
```

- Don't forget to normalize afterwards

```
vNormal = XMVector3Normalize(vNormal);
```

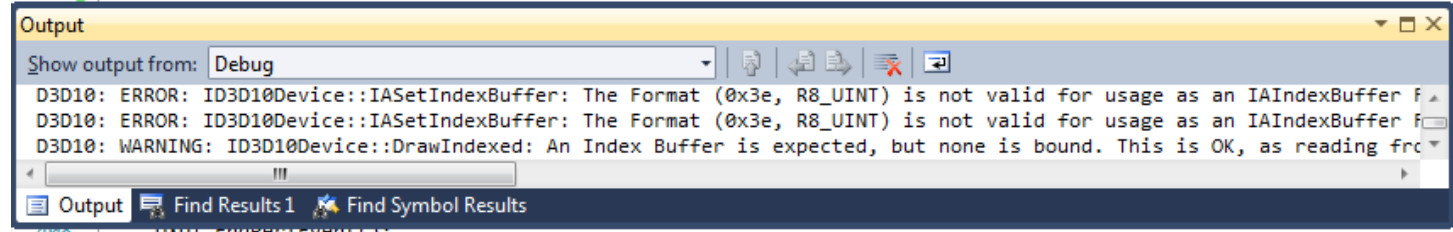
- All allocated GPU memory must also be freed.
- DXUT convenience macro: `SAFE_RELEASE` (*Pointer*)
- In debug builds the following message is displayed if resources are not yet released when the program is closed:



- We NEVER want to see this message
 - Whenever you write `device->CreateX` you should immediately also write `SAFE_RELEASE(x)`

- At Runtime (in debug builds):
 - Direct3D emits warnings and error messages
 - When you see them popping up every frame, your code is almost certainly doing something wrong

```
398 pd3dDevice->IASetIndexBuffer(0, DXGI_FORMAT_R8_UINT, 0);
399 pd3dDevice->DrawIndexed(3, 0, 0);
```



Output

Show output from: Debug

D3D10: ERROR: ID3D10Device::IASetIndexBuffer: The Format (0x3e, R8_UINT) is not valid for usage as an IAIndexBuffer f

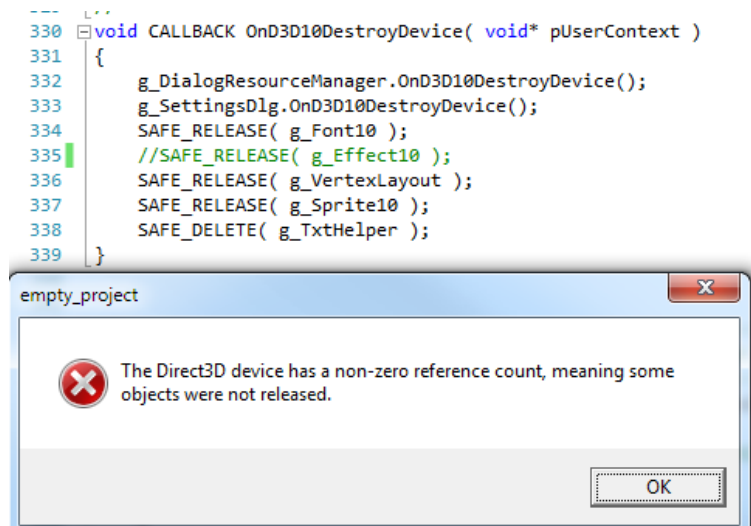
D3D10: ERROR: ID3D10Device::IASetIndexBuffer: The Format (0x3e, R8_UINT) is not valid for usage as an IAIndexBuffer f

D3D10: WARNING: ID3D10Device::DrawIndexed: An Index Buffer is expected, but none is bound. This is OK, as reading fro

Output Find Results 1 Find Symbol Results

- DXUT reports GPU memory leaks when the process terminates

```
330 void CALLBACK OnD3D10DestroyDevice( void* pUserContext )
331 {
332     g_DialogResourceManager.OnD3D10DestroyDevice();
333     g_SettingsDlg.OnD3D10DestroyDevice();
334     SAFE_RELEASE( g_Font10 );
335     //SAFE_RELEASE( g_Effect10 );
336     SAFE_RELEASE( g_VertexLayout );
337     SAFE_RELEASE( g_Sprite10 );
338     SAFE_DELETE( g_TxtHelper );
339 }
```



empty_project

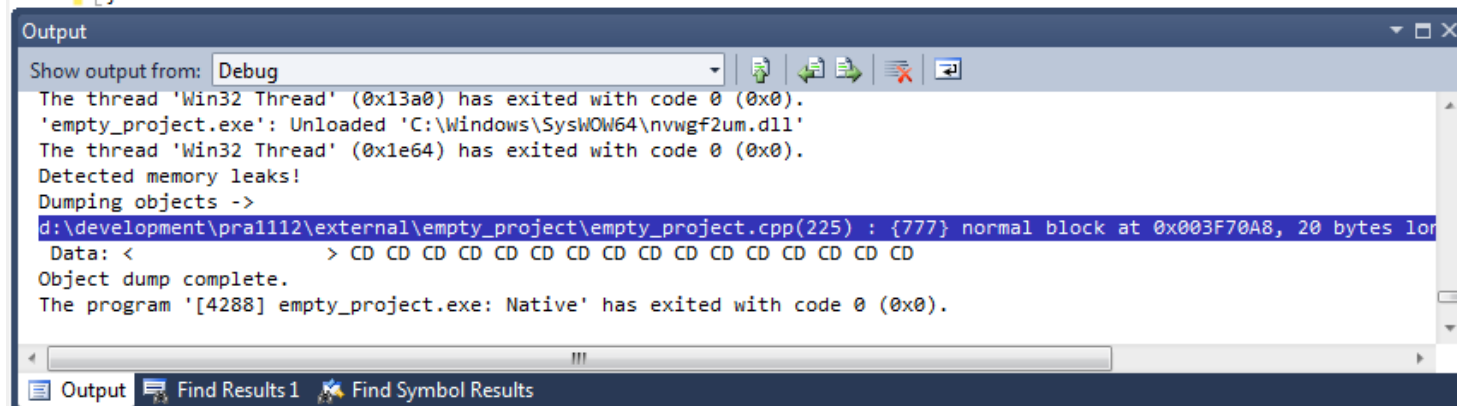
The Direct3D device has a non-zero reference count, meaning some objects were not released.

OK

- At Runtime (in debug builds):
 - Memory leak detection when the process terminates can be enabled (already in the template project):
- By including “debug.h” as last include in all .cpp files, memory leaks can be found with one click

```
81 int WINAPI wWinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance, LPWSTR lpCmdLine, int nCmdShow )
82 {
83     // Enable run-time memory check for debug builds.
84 #if defined(DEBUG) | defined(_DEBUG)
85     _CrtSetDbgFlag( _CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF );
86 #endif
87 }
```

```
225 char* foo = new char[20];
226 return;
227 }
```



- Direct3D Reference and programming guide:
<http://msdn.microsoft.com/en-us/library/windows/desktop/hh309466%28v=vs.85%29.aspx>
- DXUT samples and documentation:
<http://dxut.codeplex.com/documentation>
- DirectXMath:
[http://msdn.microsoft.com/en-us/library/windows/desktop/ee415574\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ee415574(v=vs.85).aspx)

Questions?

