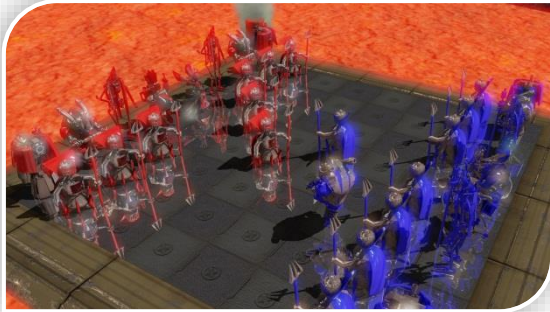
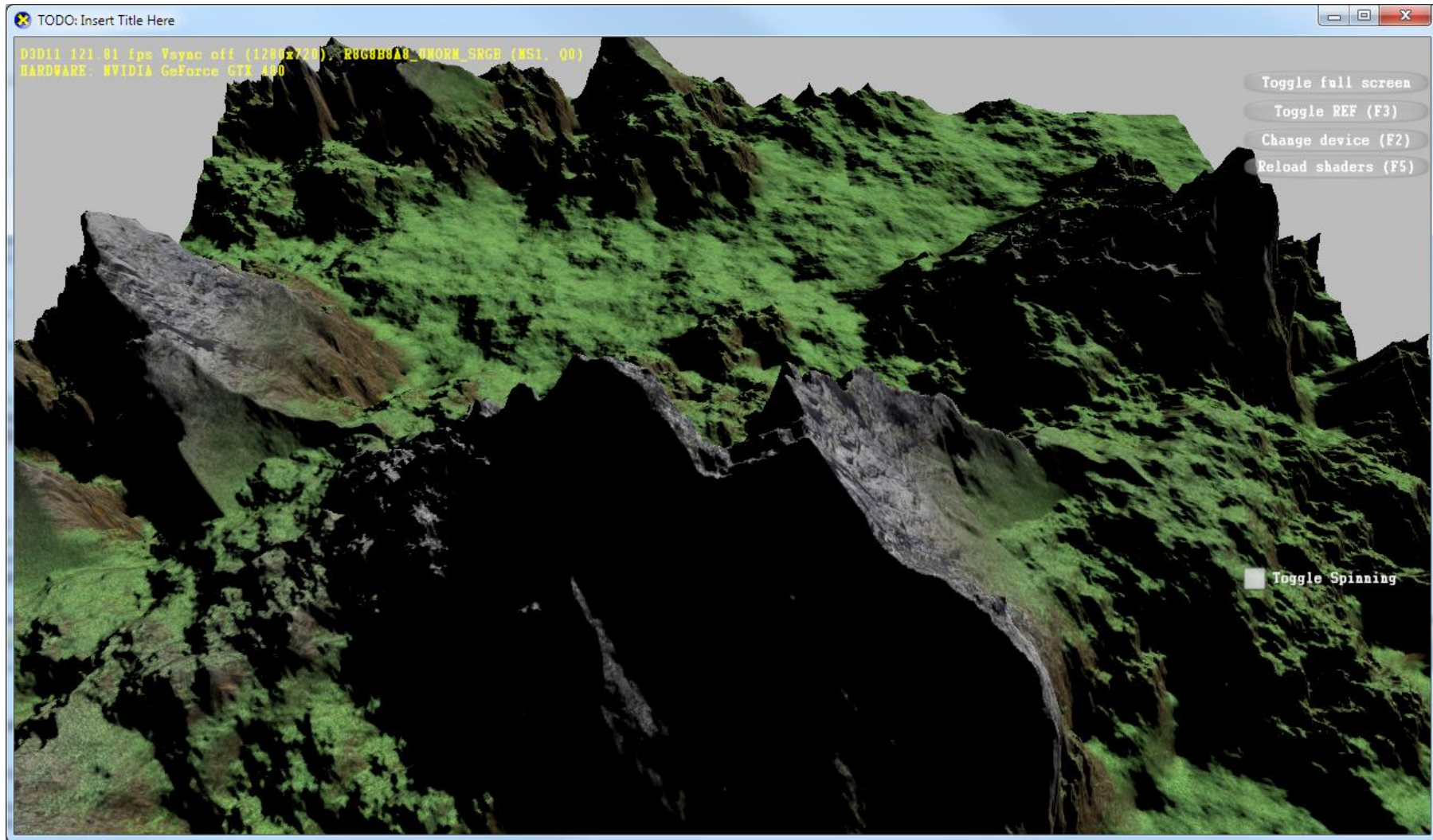


Game Engine Design

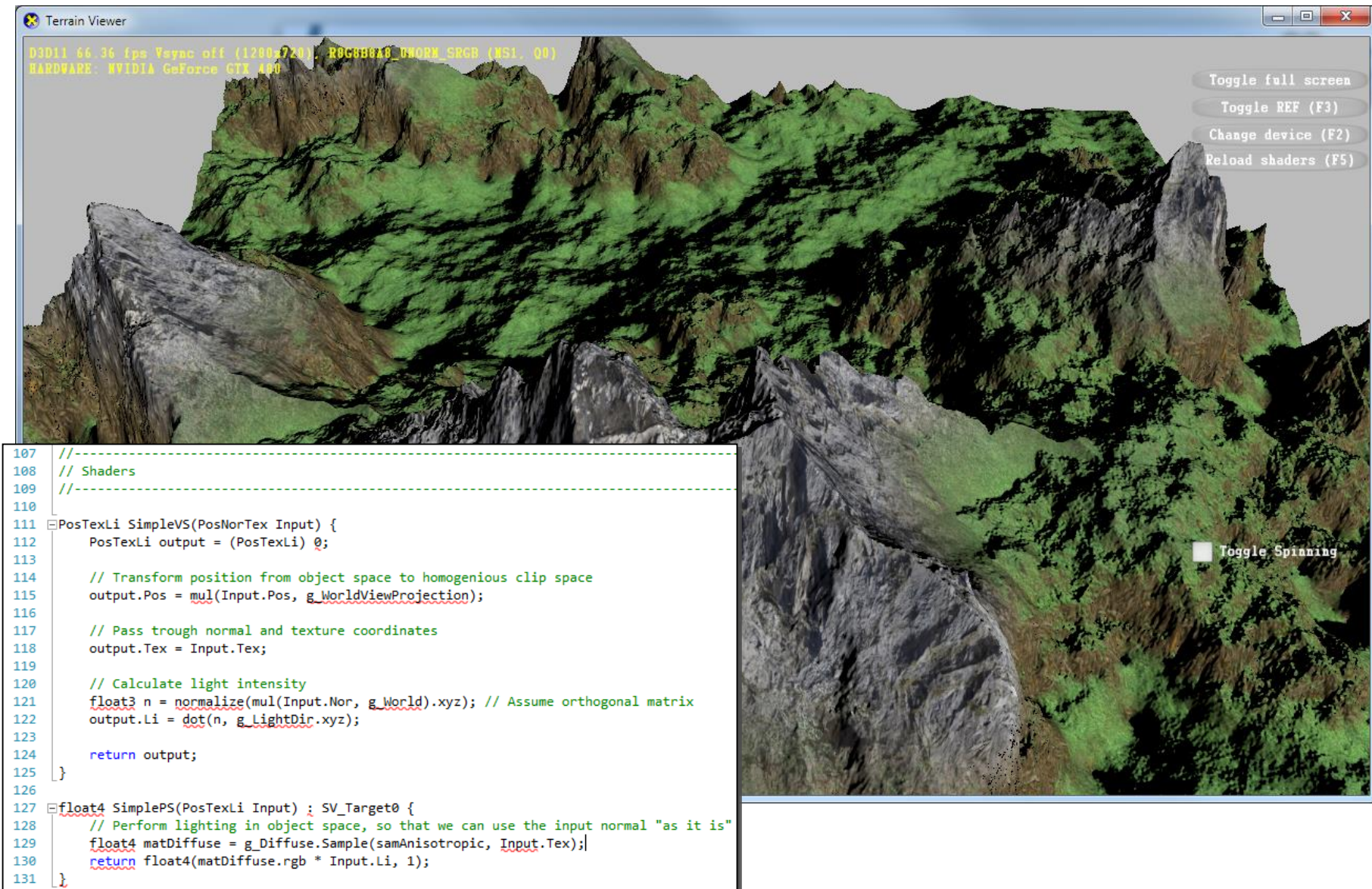


tum.3D
computer graphics & visualization

This week: Realtime Rendering...



...using shaders



Goal:

- Write the vertex shader and the pixel shader programs, which control how the graphics card renders your terrain
- Vertex Shader:
 - Calculation of the Vertex Positions (instead of vertex buffer)
 - Transformation
- Pixel Shader:
 - (Simple) Texturing
 - (Simple) Normal-Mapping
 - (Simple) Lighting

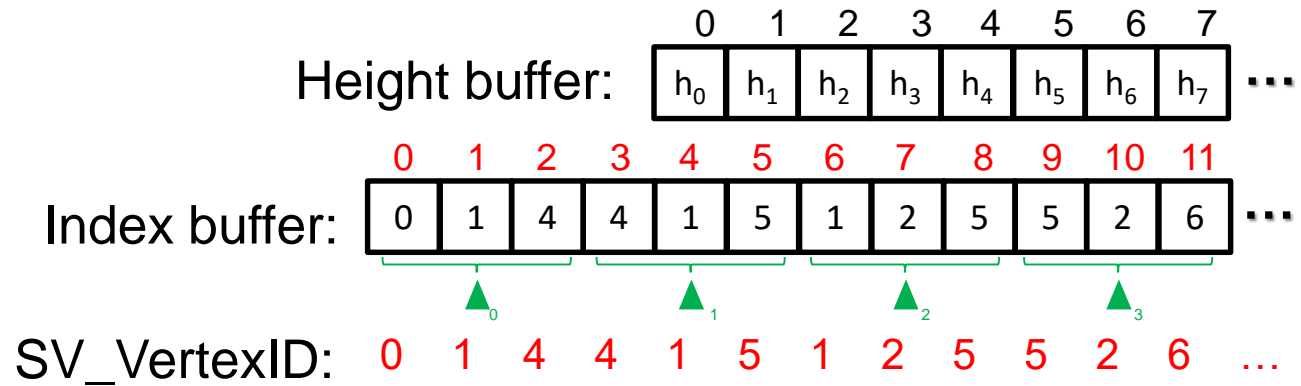
„Fixed Function“

- CPU
 - calculate 3D vertex positions from heightmap
 - calculate 3D vertex normals from heightmap
 - calculate texture coordinates
 - triangulate terrain with index buffer
 - feed geometry as vertex- and index-buffer to the graphics pipeline
- GPU: Vertex Processing
 - transform vertex position and normal
 - calculate per-vertex lighting
- GPU: Fragment Processing
 - sample color texture
 - combine vertex color (lighting) with texture color (material)

Programmable

- CPU
 - triangulate terrain with index buffer, but don't calculate any vertex positions
 - bind terrain maps to the graphics pipeline (height, normal, color)
 - tell the graphics pipeline what indices to draw, but don't feed any geometry
- GPU: Vertex Shader
 - calculate 3D vertex position from heightmap
 - transform vertex position
 - calculate texture coordinates
- GPU: Pixel Shader
 - sample color and normal texture
 - calculate per-pixel lighting
 - Combine light color with texture color (material)

Vertex Displacement in Shader

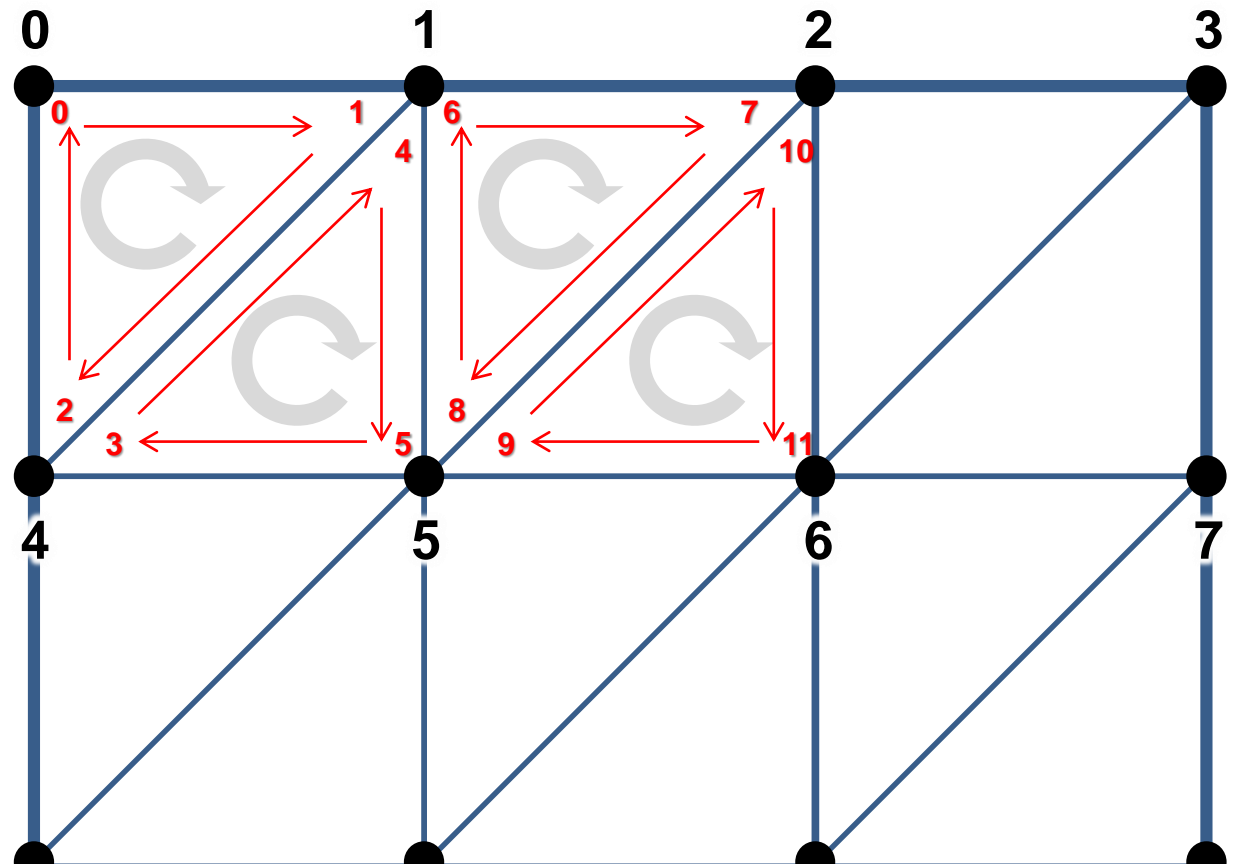


Bind **no** vertex buffer,
only system-generated
values needed

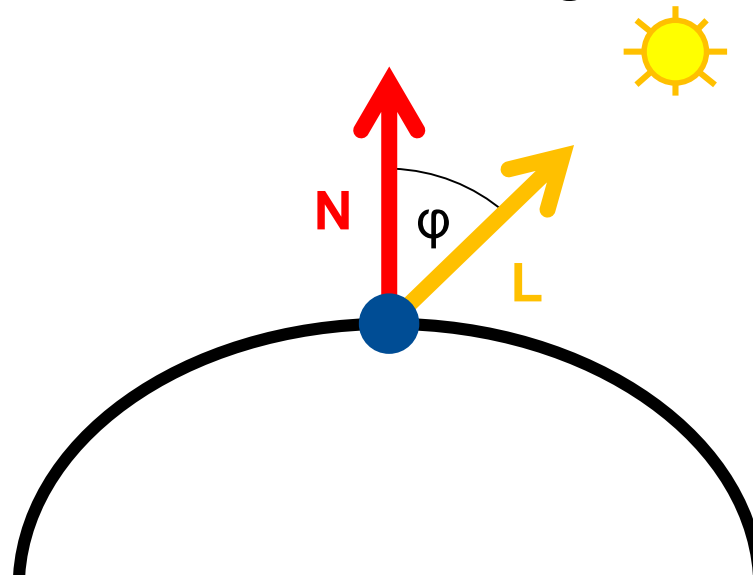
For each Vertex:

- Calc x / z position
- Look up height value
- Calculate world position
- Transform to clip space

Index buffer stays the same!



- Most simple lighting in computer graphics
- More on lighting later this semester
- For now:
 - N is the normal vector of the surface
 - L is the direction vector to the light
 - The scalar product (= dot product) between two normalized vectors yields the cosine of the angle between them



- In Direct3D most of the rendering state is specified in a ***Rendering Effect***
- Each rendering effect is stored in a separate effect source code file (.fx)
- The FX compiler (“fxc”) converts an effect source code file into a compiled effect file (.fxo)
- The function **D3DX11CreateEffectFromMemory** creates an effect object from a compiled effect in main memory
 - This already happens in the template

- A rendering effect can contain multiple **Effect Techniques** to achieve the same rendering effect
 - The effect member function `GetTechniqueByName` retrieves a handle to an effect technique
- An effect technique can contain multiple **Render Passes** to achieve a certain rendering effect
 - The technique member function `GetPassByName` retrieves a handle to a render pass
- A render pass defines the state of the graphics pipeline during a draw
 - It sets the state of the fixed function stages of the graphics pipeline
 - It controls which **Shaders** are used in the programmable stages

- Techniques combine multiple rendering passes
 - A rendering pass specifies which shaders to use (vertex, pixel, geometry, ...)
 - May also specify pipeline states (Rasterizer, DepthStencil, Blend...)

```
//-----  
// Techniques  
//-----  
technique11 Render  
{  
    pass P0  
    {  
        SetVertexShader(CompileShader(vs_4_0, TerrainVS()));  
        SetGeometryShader(NULL);  
        SetPixelShader(CompileShader(ps_4_0, TerrainPS()));  
  
        SetRasterizerState(rsCullNone);  
        SetDepthStencilState(EnableDepth, 0);  
        SetBlendState(NoBlending, float4(0.0f, 0.0f, 0.0f, 0.0f), 0xFFFFFFFF);  
    }  
}
```

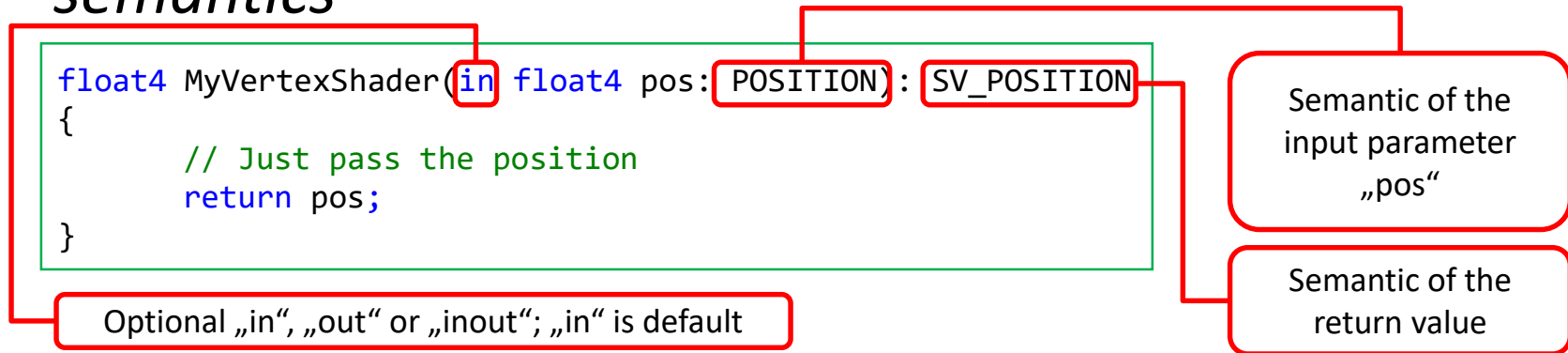
- A shader is a program that is executed on the GPU
- The same shader program is executed for many elements of graphics data in parallel
 - Vertex Shader
 - Pixel Shader
 - etc.
 - SIMD = Single Instruction Multiple Data
- In Direct3D, shaders are written in HLSL (High Level Shading Language)
- Major DirectX versions correspond to major HLSL versions (DirectX 10 -> HLSL 4, DirectX 11 -> HLSL 5)
- HLSL Documentation
 - „Programming Guide for HLSL“ and „Reference for HLSL“ in the „Windows DirectX Graphics Documentation “ help file

- In HLSL, shaders are defined very similar to functions

```
//-----  
// Helper functions  
//-----  
  
float CalcLightingNDotL(float3 n, float3 l) {  
    return dot(n, l);  
}  
  
//-----  
// Shaders  
//-----  
  
PosTexLi SimpleVS(PosNorTex Input) {  
    PosTexLi output = (PosTexLi) 0;  
  
    // Transform position from object space to homogenous clip space  
    output.Pos = mul(Input.Pos, g_WorldViewProjection);  
  
    // Pass through normal and texture coordinates  
    output.Tex = Input.Tex;  
  
    // Calculate light intensity  
    float3 n = normalize(mul(Input.Nor, g_World).xyz); // Assume orthogonal matrix  
    output.Li = CalcLightingNDotL(n, g_LightDir.xyz);  
    .....  
    return output;  
}  
  
float4 SimplePS(PosTexLi Input) : SV_Target0 {  
    // Perform lighting in object space, so that we can use the input normal "as it is"  
    float4 matDiffuse = g_Diffuse.Sample(samAnisotropic, Input.Tex);  
    return float4(matDiffuse.rgb * Input.Li, 1);  
}
```


- Syntax similar to C++
- Special vector datatypes: `float2`, `float4`, `int2`...
 - Component access `.x`, `.y`, `.xyzw`...
 - Swizzling: `.xzy`...
 - Operators `+-/*%` for vector x vector and scalar x vector
 - `int2 v = int2(10, 10) % 5`
 - Common vector operations like `dot()`, `cross()`, `normalize()`...
 - Check the documentation before you write your own function!
- Special matrix types: `float4x4`
 - `mul(float4, float4x4)` to apply matrix transformation to a vector

- Data exchange between pipeline stages is controlled by *semantics*



- A pipeline stage's *output* is the next one's *input*
- Semantics are arbitrary, but must match between stages
 - Input assembler must output a float4 with semantic POSITION
 - This is defined in the input layout!
- Predefined system values SV_*
 - SV_POSITION, SV_TARGET0, SV_VERTEXID...

- Alternative 1:

```
void MyVertexShader(in float4 posIn: POSITION, out float4 posOut: SV_POSITION)
{
    // Just pass the position
    posOut = posIn;
}
```

- Alternative 2:

```
struct MyVertex
{
    float4 pos: POSITION;
};

struct MyFragment
{
    float4 pos: SV_POSITION;
};

void MyVertexShader(in MyVertex input, out MyFragment output)
{
    output = (MyFragment)0;

    // Just pass the position
    output.pos = input.pos;
}
```

- Effect variables are used to pass information from your C++ CPU code to your HLSL shader code
- In the .fx file on HLSL side, texture and buffer resources are defined as global variables while simpler types are combined to constant buffers

```
//-----  
// Shader resources  
//-----  
  
Texture2D    g_Diffuse; // Material albedo color for diffuse lighting  
  
//-----  
// Constant buffers  
//-----  
  
cbuffer cbChangesEveryFrame  
{  
    matrix  g_World; // Object to world space transformation  
    matrix  g_WorldViewProjection; // Object to clip space transformation  
    float4  g_LightDir; // To-Light vector (object space)  
};
```

- In the shaders, both types can be accessed like global variables though

```
// Transform position from object space to homogenous clip space  
output.Pos = mul(Input.Pos, g_WorldViewProjection);
```


- In the .cpp file on C++ side: **effect variables** act as „pointers“ to the variables in the HLSL shader on the GPU

```
ID3DX11EffectMatrixVariable*    g_WorldEV = NULL; // World matrix effect variable
ID3DX11EffectMatrixVariable*    g_WorldViewProjectionEV = NULL; // WorldViewProjection matrix effect variable
ID3DX11EffectShaderResourceVariable* g_DiffuseEV = NULL; // Effect variable for the diffuse color texture
ID3DX11EffectVectorVariable*    g_LightDirEV = NULL; // Light direction in object space
```

- The „**GetVariableByName**“ method of the rendering effect is used to bind the CPU variable to its GPU counterpart

```
g_WorldViewProjectionEV = g_Effect->GetVariableByName("g_WorldViewProjection")->AsMatrix();
if(!g_WorldViewProjectionEV) return E_FAIL;
```

- The „**Set***“ method of an effect variable tells the effect framework the updated value for a variable

```
XMFLOAT4X4 w;
XMStoreFloat4x4(&w, worldViewProj);
V(g_gameEffect.worldViewProjectionEV->SetMatrix((float*)&w));
```

- The upload of the updated values to the GPU happens when the rendering pass is applied

```
// Apply the rendering pass in order to submit the necessary render state changes to the device
g_Pass0->Apply(0, pd3dImmediateContext);
```

- Direct3D can create resource views for textures without further information

```
device->CreateShaderResourceView(diffuseTexture, NULL, &diffuseTextureSRV)
```

- This time, we'll need to pass a description
 - D3D11_SHADER_RESOURCE_VIEW_DESC
[http://msdn.microsoft.com/en-us/library/windows/desktop/ff476211\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ff476211(v=vs.85).aspx)
 - ViewDimension: D3D11_SRV_DIMENSION_BUFFER
 - Format: DXGI_FORMAT_R32_FLOAT
 - Exactly **one** of the remaining structs needs to be filled
 - Buffer in our case
 - FirstElement = 0
 - NumElements = res * res

```
typedef struct D3D11_SHADER_RESOURCE_VIEW_DESC {
    DXGI_FORMAT          Format;
    D3D11_SRV_DIMENSION ViewDimension;
    union {
        D3D11_BUFFER_SRV          Buffer;
        D3D11_TEX1D_SRV           Texture1D;
        D3D11_TEX1D_ARRAY_SRV     Texture1DArray;
        D3D11_TEX2D_SRV           Texture2D;
        D3D11_TEX2D_ARRAY_SRV     Texture2DArray;
        D3D11_TEX2DMS_SRV         Texture2DMS;
        D3D11_TEX2DMS_ARRAY_SRV   Texture2DMSArray;
        D3D11_TEX3D_SRV           Texture3D;
        D3D11_TEXCUBE_SRV         TextureCube;
        D3D11_TEXCUBE_ARRAY_SRV   TextureCubeArray;
        D3D11_BUFFEREX_SRV        BufferEx;
    };
} D3D11_SHADER_RESOURCE_VIEW_DESC;
```

- The template project is configured to compile .fx files with FXC when you build the project
 - If your shader doesn't compile, the build fails
 - Double-click the error message to jump into the shader file
 - The compiled .fxo effect is loaded when the program starts
 - You can also compile the .fx file manually (Strg + F7)
- The template project includes a „Reload Shader“ button (F5) which allows shader editing at runtime
 - Loads the compiled .fxo effect at runtime and immediately uses the updated shader for rendering
 - This only works if all effect variables are retrieved in „ReloadShader()“ and updated („Set*()“) after the reload
 - Compiling doesn't work during debugging, so start without debugging (Strg+F5) if you work on your shader code

- Visual Studio 2012 contains a Graphics Debugger!
- „Post mortem“ debugging
 - You need to explicitly select a rendered frame for debugging
 - All pipeline states, resources etc. are downloaded to the CPU
 - Each draw call can now be investigated step by step
- Start your program from „Debug -> Graphics -> Start Diagnostics“ (or hit Alt + F5)
 - Press „Print“ to capture the last rendered frame

Graphics Debugging

The screenshot shows the Microsoft Visual Studio IDE with the ShaderDemo application running. The interface is divided into several panes:

- Process:** [5020] Demo1.exe
- Graphics Pixel History:** A list of captured frames. A red box highlights the "Captured frames" section, which shows two frames: a white triangle and a yellow triangle.
- Properties:** A pane showing various system and driver information, including Direct3D, Display, and Module information.
- Frame List:** A pane showing a list of frames. A red box highlights the "Frame List" section, which shows two frames: a white triangle and a yellow triangle.
- Graphics Pipeline Stages:** A pane showing the current stage of the graphics pipeline, which is "IDXGISwapChain::Present(0,0)".

The main window displays a 3D scene with a yellow triangle on a black background. The status bar at the bottom indicates "Ready".

Captured frames

The screenshot displays the Microsoft Visual Studio Graphics Debugger interface. The main window is divided into several panes. On the left, the 'Graphics Event List' pane shows a chronological list of Direct3D API calls. A red box highlights a specific sequence of calls, with a red arrow pointing from a text box to the 'Draw(3,0)' call at index 109. Below this, the 'Graphics Pipeline Stages' pane shows the current stage as '711: IDXGISwapChain::Present(0,0)'. On the right, a 3D scene is rendered, showing a green and yellow triangular prism. A red box highlights the 'Draw call' in the scene, with a red arrow pointing from a text box to it. At the bottom, the 'Memory' pane shows the address of the selected call, and the 'Autos' pane shows the current state of the debugger.

- Called some frames before
- Data is used this frame
- E.g. Buffer creation

Draw call

List of Direct3D calls

The screenshot displays the Visual Studio Graphics Debugger interface. The main window shows a 3D scene with a red box highlighting a pixel. A red callout box points to the pixel history, stating: "D3D calls influencing the selected pixel". The pixel history shows the following calls:

- 432: Initial
- 432: ID3D11DeviceContext->ClearRenderTargetView
- 453: ID3D11DeviceContext->Draw(3,0)
- 453: Final

The main window also displays the Direct3D Information, Display (1) Information, and Module Information. The Direct3D Information includes details about the GPU (NVIDIA GeForce GTX TITAN) and the driver (NVIDIA GeForce GTX 280). The Display (1) Information shows the display memory (4095 MB) and the driver name (nvidia3dumx.dll). The Module Information lists the loaded modules and their addresses.

A red box highlights a pixel in the 3D scene, with a red callout box stating: "Select a pixel to debug". The pixel is located at X: 442 (0.345) Y: 265 (0.368). The selected pixel is at X: 656 (0.513) Y: 434 (0.603).

The bottom of the interface shows the Graphics Pipeline Stages, which includes the call ID3D11DeviceContext->Present(0,0).

Graphics Debugging

Select the draw call

Created device resources
Gray: Not bound in the selected draw call

Identifier	Name	Type	Active	Size	Format	Mips	Width	Height	Depth	Created by Frame
obj19	D3D11 Blend State	D3D11 Blend State	*	0		0	0	0	0	1
obj26	D3D11 Blend State	D3D11 Blend State	*	0		0	0	0	0	1
obj7	D3D11 Buffer	D3D11 Buffer	*	80		0	0	0	0	1
obj11	D3D11 Buffer	D3D11 Buffer	*	14,688		0	0	0	0	287
obj28	D3D11 Buffer	D3D11 Buffer	*	216		0	0	0	0	1
obj21	D3D11 Class Linkage	D3D11 Class Linkage	*	0		0	0	0	0	1
obj18	D3D11 Depth-Stencil State	D3D11 Depth-Stencil State	*	0		0	0	0	0	1
obj24	D3D11 Depth-Stencil State	D3D11 Depth-Stencil State	*	0		0	0	0	0	1
obj17	D3D11 Depth-Stencil View	D3D11 Depth-Stencil View	*	0		0	0	0	0	39157
obj3	D3D11 Device	D3D11 Device	*	0		0	0	0	0	1
obj4	D3D11 Device Context	D3D11 Device Context	*	0		0	0	0	0	1

Graphics Debugging

ShaderDemo (Running) - Microsoft Visual Studio

Process: [5020] Demo1.exe Thread: d3d11 buffer (obj11) Graphics Experiment.vsglog* demoEffect.fx

Graphics Pixel History: Final Frame Buffer

Properties: d3d11 buffer (obj11)

1 The buffer format used is specified in the 'Graphics Object Table'.
2
3
4
5 Types: bool, int, xint (an unsigned hex int), uint (unsigned int), int64, xint64, uint64,
6 byte, xbyte, ubyte, 2byte, x2byte, u2byte, 4byte, x4byte, u4byte, 8byte, x8byte, u8byte,
7 half (a 16-bit float), half2, half3, half4, float, float2, float3, float4, double
8 Usage: 'float' or 'float3 xint half half xint'
9
10 Currently used format: float
11
12
13 0 [0x00000000-0x00000003] -0.9921875
14 1 [0x00000004-0x00000007] +0.94444442
15 2 [0x00000008-0x0000000b] +0.5
16 3 [0x0000000c-0x0000000f] +1
17 4 [0x00000010-0x00000013] +1
18 5 [0x00000014-0x00000017] +0
19 6 [0x00000018-0x0000001b] +1
20 7 [0x0000001c-0x0000001f] +0.4210526
21 8 [0x00000020-0x00000023] +0
22 9 [0x00000024-0x00000027] -0.98046875
23 10 [0x00000028-0x0000002b] +0.94444442
24 11 [0x0000002c-0x0000002f] +0.5
25 12 [0x00000030-0x00000033] +1
26 13 [0x00000034-0x00000037] +1
27 14 [0x00000038-0x0000003b] +0
28 15 [0x0000003c-0x0000003f] +1
29 16 [0x00000040-0x00000043] +0.43157893
30 17 [0x00000044-0x00000047] +0
31 18 [0x00000048-0x0000004b] -0.9921875
32 19 [0x0000004c-0x0000004f] +0.8861108
33 20 [0x00000050-0x00000053] +0.5
34 21 [0x00000054-0x00000057] +1
35 22 [0x00000058-0x0000005b] +1
36 23 [0x0000005c-0x0000005f] +0
37 24 [0x00000060-0x00000063] +1
38 25 [0x00000064-0x00000067] +0.4210526
39 26 [0x00000068-0x0000006b] +1
40 27 [0x0000006c-0x0000006f] -0.98046875
41 28 [0x00000070-0x00000073] +0.94444442
42 29 [0x00000074-0x00000077] +0.5
43 30 [0x00000078-0x0000007b] +1
44 31 [0x0000007c-0x0000007f] +1

Contents of the selected buffer
Try selecting resource views etc...

Graphics Object Table

Buffer format: float

Identifier	Name	Type	Active	Size	Format	Mips	Width	Height	Depth	Created by Frame
obj19	D3D11 Blend State	*	0	0	0	0	0	0	1	
obj26	D3D11 Blend State	*	0	0	0	0	0	0	1	
obj7	D3D11 Buffer	*	80	0	0	0	0	0	1	
obj11	D3D11 Buffer	*	14,688	0	0	0	0	0	287	
obj28	D3D11 Buffer	*	216	0	0	0	0	0	1	
obj21	D3D11 Class Linkage	*	0	0	0	0	0	0	1	
obj18	D3D11 Depth-Stencil State	*	0	0	0	0	0	0	1	
obj24	D3D11 Depth-Stencil State	*	0	0	0	0	0	0	1	
obj17	D3D11 Depth-Stencil View	*	0	0	0	0	0	0	39157	
obj3	D3D11 Device	*	0	0	0	0	0	0	1	
obj4	D3D11 Device Context	*	0	0	0	0	0	0	1	

Format to display this buffer

Graphics Debugging

ShaderDemo (Running) - Microsoft Visual Studio

Process: [5020] Demo1.exe Thread: d3d11 render target view (obj15)

Graphics Pixel History

Final Frame Buffer

R: 0.000000000
G: 1.000000000
B: 0.000000000
A: 1.000000000

Frame: 4890
Pixel: 656, 434

Properties

Summary Information

Direct3D Information

10-bit XR High Color Format True
DirectCompute CS 4.x True
Double Precision Shaders True
Driver Command Lists True
Driver Concurrent Creates True
Extended Formats (BIGRA, ei True
Max HW Feature Level D3D_FEATURE_LEVEL_11_0

Display (1) Information

Description NVIDIA GeForce GTX TITAN
Display Memory 4095 MB
Driver Name nvd3dumx.dll,nvwgf2umx.dll
Driver Version 9.18.0013.3182
Name \\.\DISPLAY1

Display (2) Information

Description NVIDIA GeForce GTX 280
Display Memory 4069 MB
Driver Name nvd3dumx.dll,nvwgf2umx.dll
Driver Version 9.18.0013.3182
Name \\.\DISPLAY5

Experiment File

Path

Graphics Pipeline Stages

453: ID3D11DeviceContext::Draw(3,0)

No mesh available for stage

Input Assembler

Vertex Shader

Pixel Shader

Output Merger

Selected pixel X: 656 (0.513) Y: 434 (0.603)

Ready

Game Engine Design

Mathias Kanzler, Prof. Dr. R. Westermann

Pipeline stages
Shows all transformations / color operations

The screenshot shows the Visual Studio interface for debugging a shader demo. The top menu bar includes FILE, EDIT, VIEW, PROJECT, BUILD, DEBUG, TEAM, NSIGHT, SOLUTION, TOOLS, TEST, ANALYZE, and WINDOW. The 'DEBUG' menu is highlighted, and the 'Debug' option is selected. The 'Stack Frame' dropdown shows 'SineColorPS'. The 'Graphics Pixel History' window on the left shows a final frame buffer with a yellow square. The 'Graphics Experiment.vsglog' window shows a log of graphics events, including 'ID3D11DeviceContext::ClearRenderTargetView' and 'ID3D11DeviceContext::Draw(3,0)'. The 'Graphics Pipeline Stages' window at the bottom shows a sequence of stages: Input Assembler, Vertex Shader, Pixel Shader, and Output Merger. The 'Pixel Shader' stage is selected, and a red box highlights the 'Click me to debug the selected pixel' text. A large red box in the center contains the text 'Shader Debugging: Only in Debug-Mode! (Shader needs to be compiled with DEBUG flag)'. Another red box on the right contains a list of bullet points: 'Investigate variables etc.' and 'Just like your C++ debugger!'. The code editor shows the shader code for 'SineColorPS' and 'SineColorVS'.

Shader Debugging: Only in Debug-Mode!
(Shader needs to be compiled with DEBUG flag)

- Investigate variables etc.
- Just like your C++ debugger!

Click me to debug the selected pixel

Graphics Pipeline Stages

Input Assembler → Vertex Shader → Pixel Shader → Output Merger

Questions?

