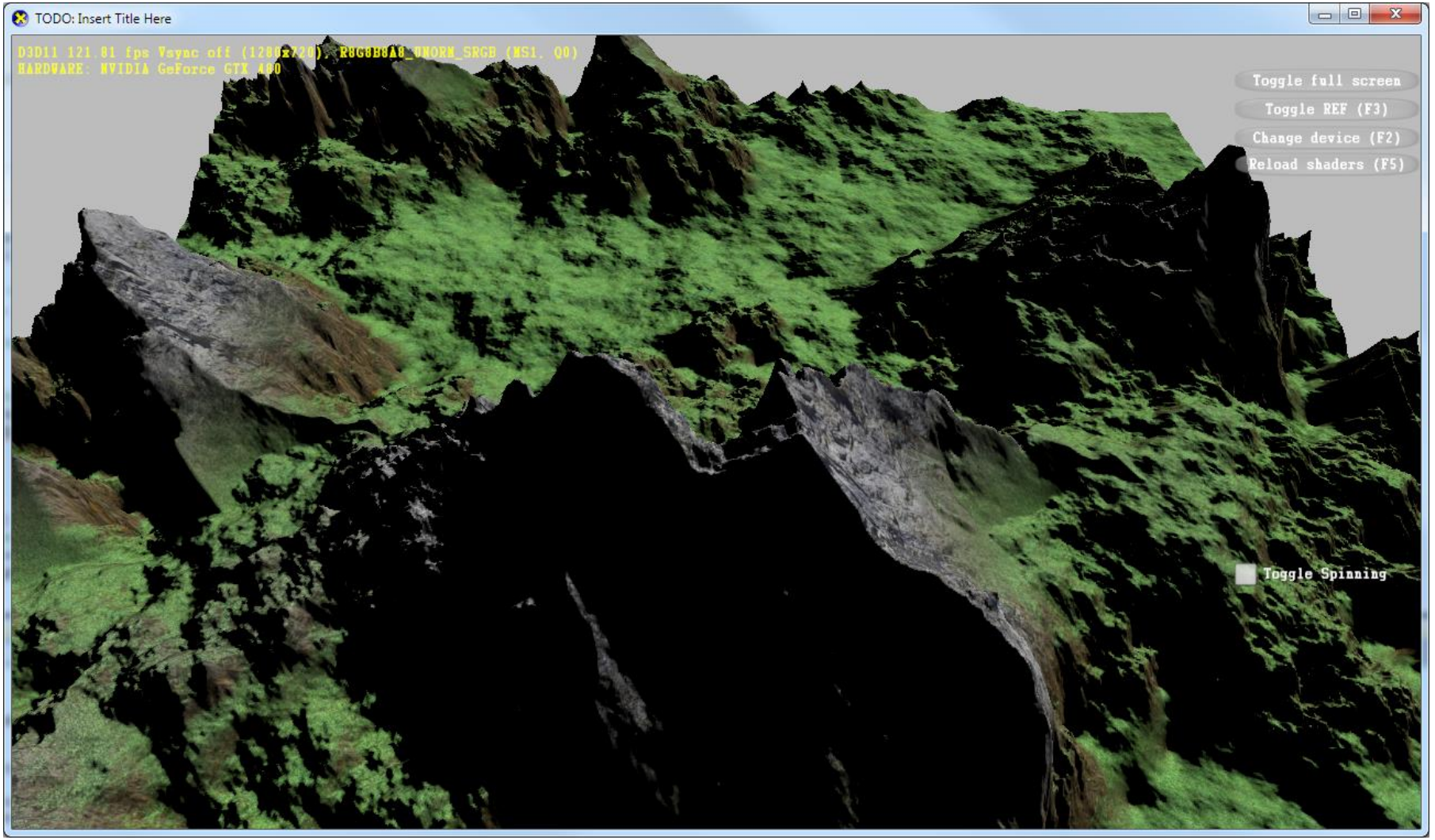
# Game Engine Design

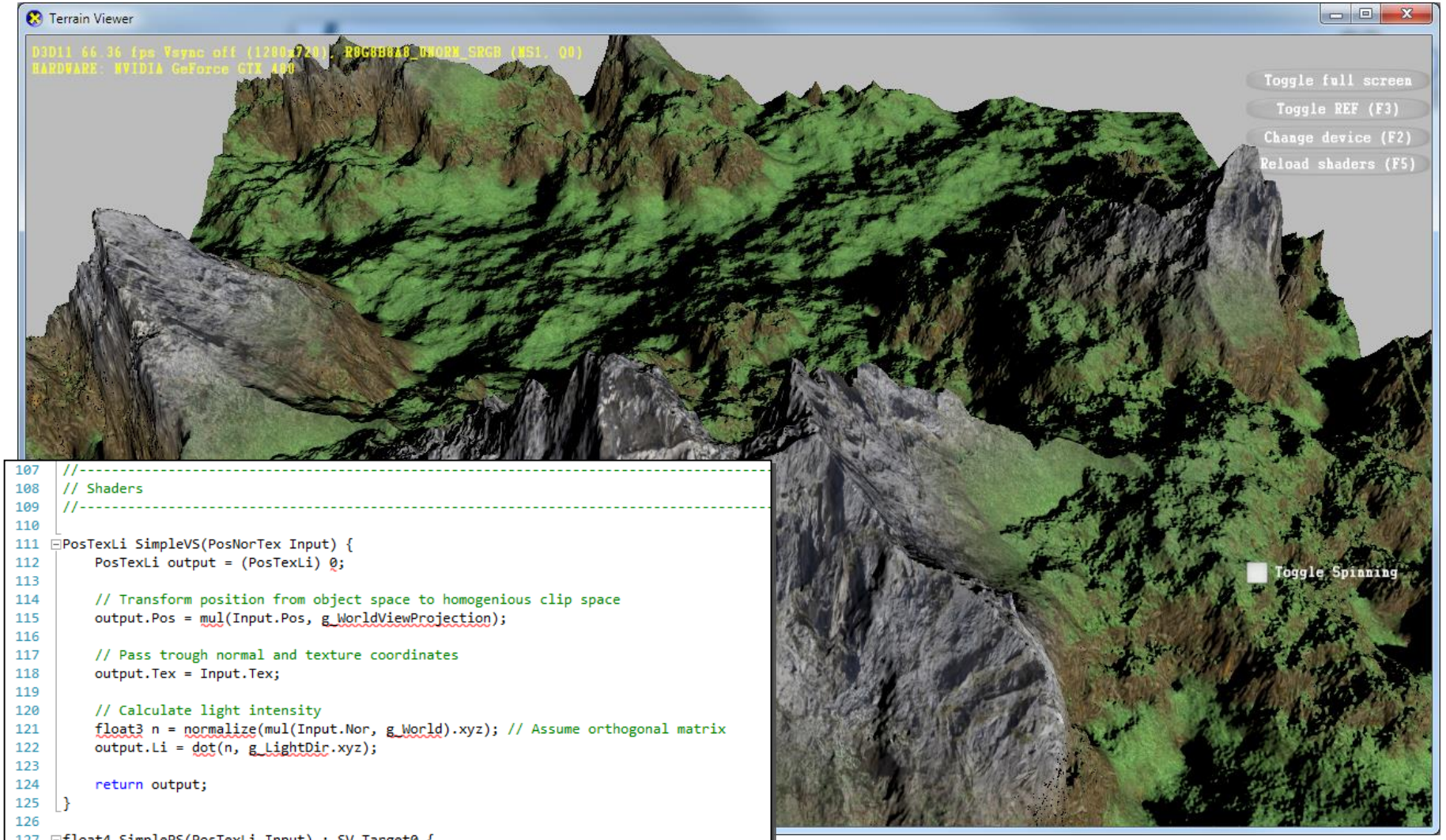
tum.3D

computer graphics & visualization

# This week: Realtime Rendering…



# ...using shaders




```
107 //--------------------------------------------------------------------------------------
108 // Shaders
109 //--------------------------------------------------------------------------------------
110
111 PosTexLi SimpleVS(PosNorTex Input) {
112     PosTexLi output = (PosTexLi) 0;
113
114     // Transform position from object space to homogenious clip space
115     output.Pos = mul(Input.Pos, g_WorldViewProjection);
116
117     // Pass trough normal and texture coordinates
118     output.Tex = Input.Tex;
119
120     // Calculate light intensity
121     float3 n = normalize(mul(Input.Nor, g_World).xyz); // Assume orthogonal matrix
122     output.Li = dot(n, g_LightDir.xyz);
123
124     return output;
125 }
126
127 float4 SimplePS(PosTexLi Input) : SV_Target0 {
128     // Perform lighting in object space, so that we can use the input normal "as it is"
129     float4 matDiffuse = g_Diffuse.Sample(samAnisotropic, Input.Tex);
130     return float4(matDiffuse.rgb * Input.Li, 1);
131 }
```

# Assignment 5



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

# Terrain Renderer Old vs. New

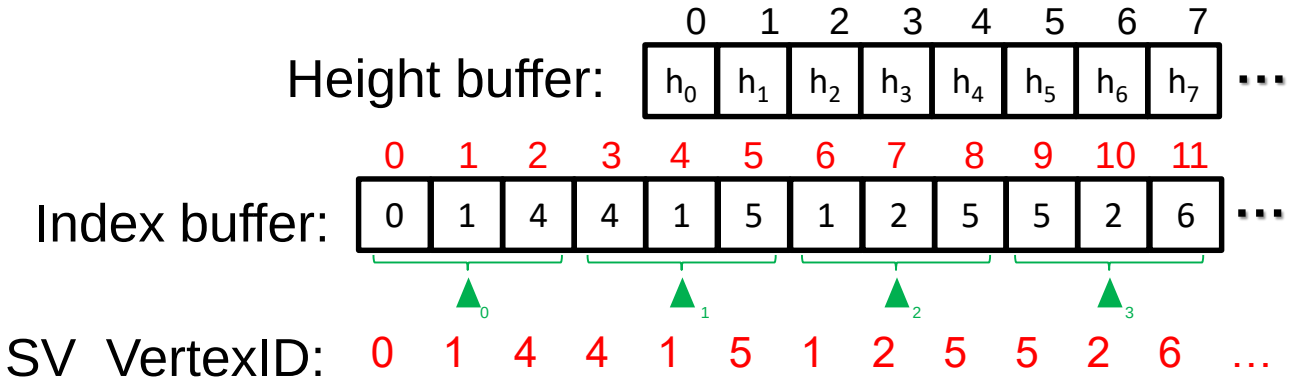
## „Fixed Function“

- 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

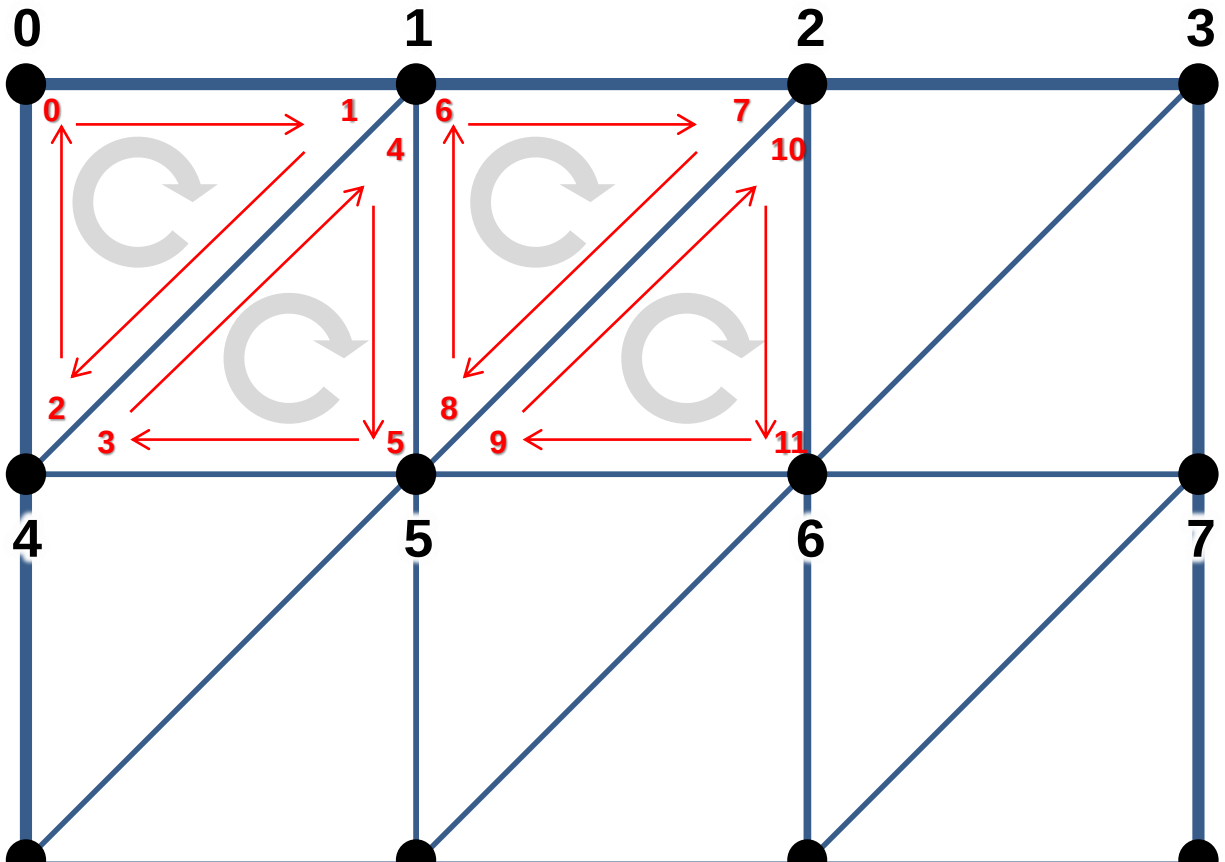



Bind **no** vertex buffer, only system-generated values needed

For each Vertex:

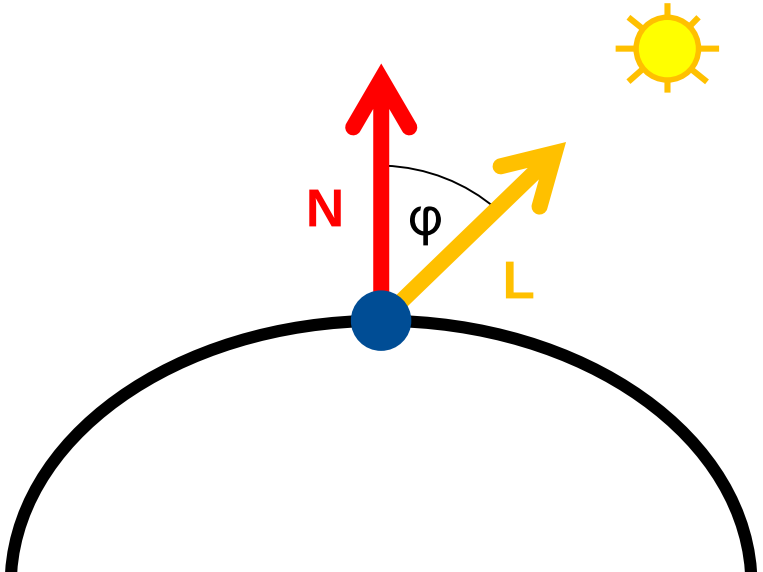
- Calc x / z position
- Look up height value
- Calculate world position
- Transform to clip space

Index buffer stays the same!



# $N \cdot L$ Lighting

- 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



# Rendering Effects

- 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

# Rendering Effects

- 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

# Shaders

- 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);
    }
}
```

# Shaders

- 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

# Shaders

- In HLSL, shaders are defined very similar to functions

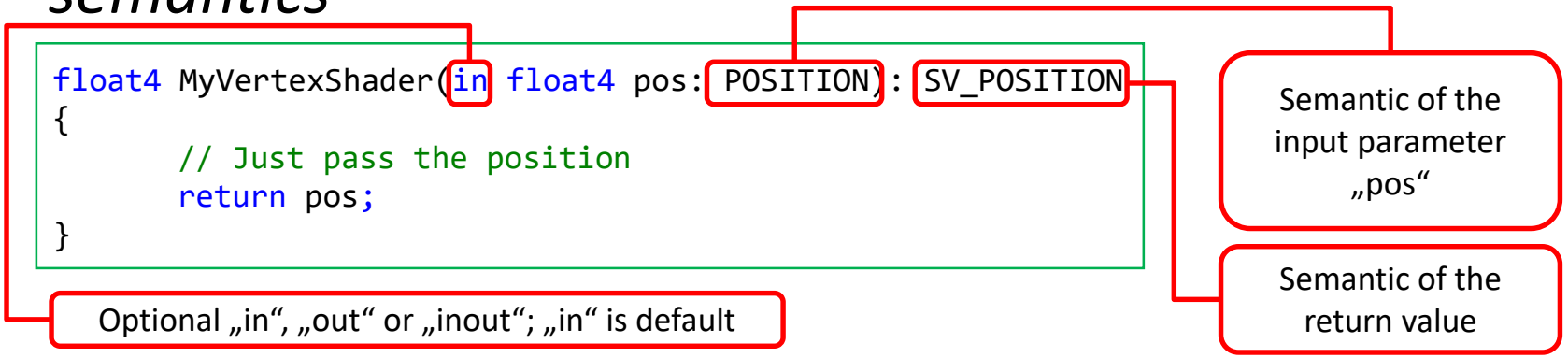
```
//--------------------------------------------------------------------------------------
// Helper functions
//--------------------------------------------------------------------------------------

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 homogenious clip space
    output.Pos = mul(Input.Pos, g_WorldViewProjection);

    // Pass trough 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);
}
```

# Shaders

- 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

# Shader Semantics

- 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…

# Shader Semantics

- 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

- 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 homogenious clip space
output.Pos = mul(Input.Pos, g_WorldViewProjection);
```

# Effect variables

- 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);
```

# D3D11_RESOURCE_VIEW_DESC

- 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
  - 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;
```

# Shader Editing Tips

- 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 programm 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

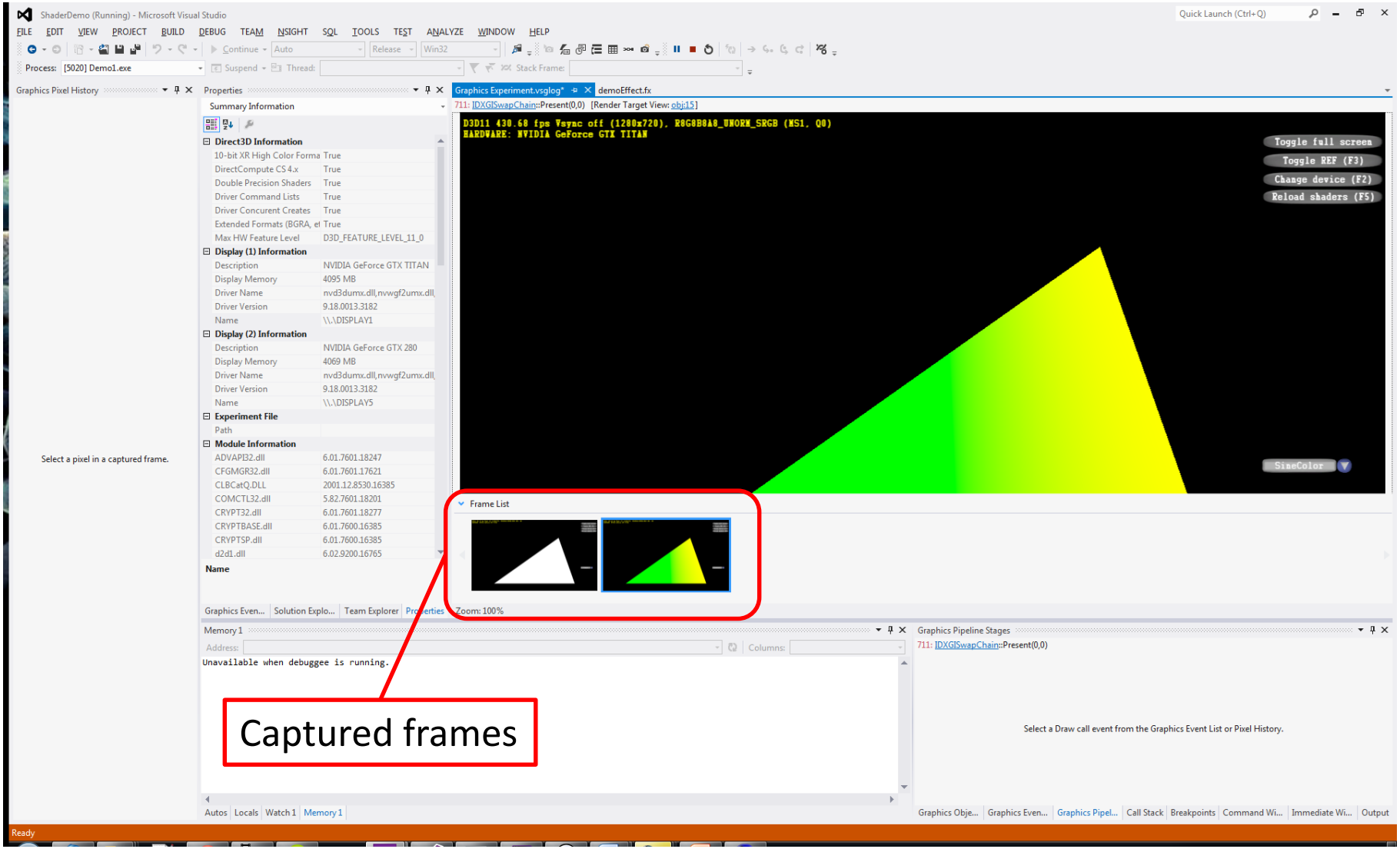
# Graphics Debugging

- Visual Studio 2012 contains a Graphics Debugger!

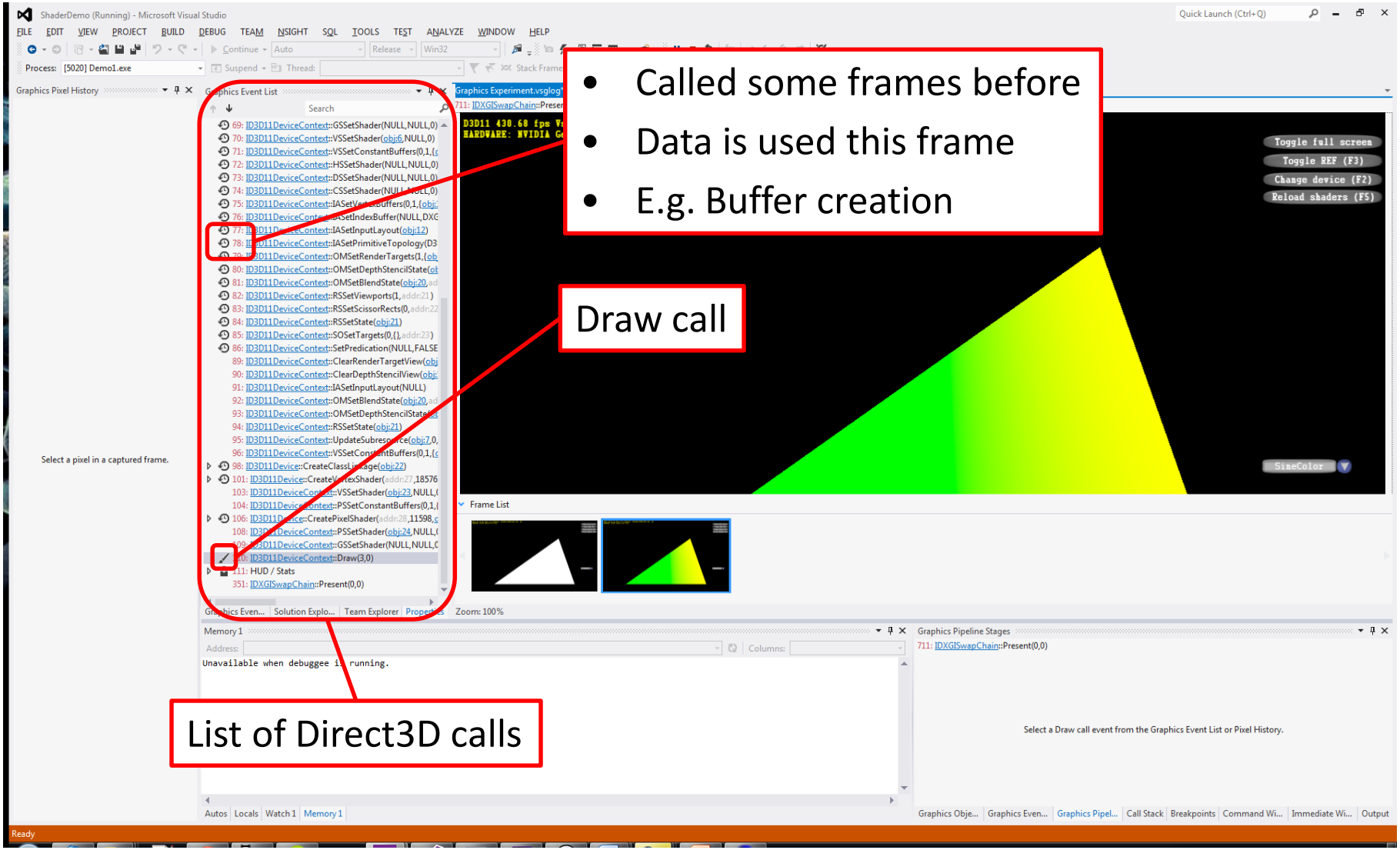
- „Post mortem“ debugging
  - You need to explicitly select a rendered frame for debugging
  - All pipeline states, resources etc. are downloaded to the CPU
  - Each draw call can now be investigated step by step

- Start your program from „Debug -> Graphics -> Start Diagnostics“ (or hit Alt + F5)
  - Press „Print“ to capture the last rendered frame

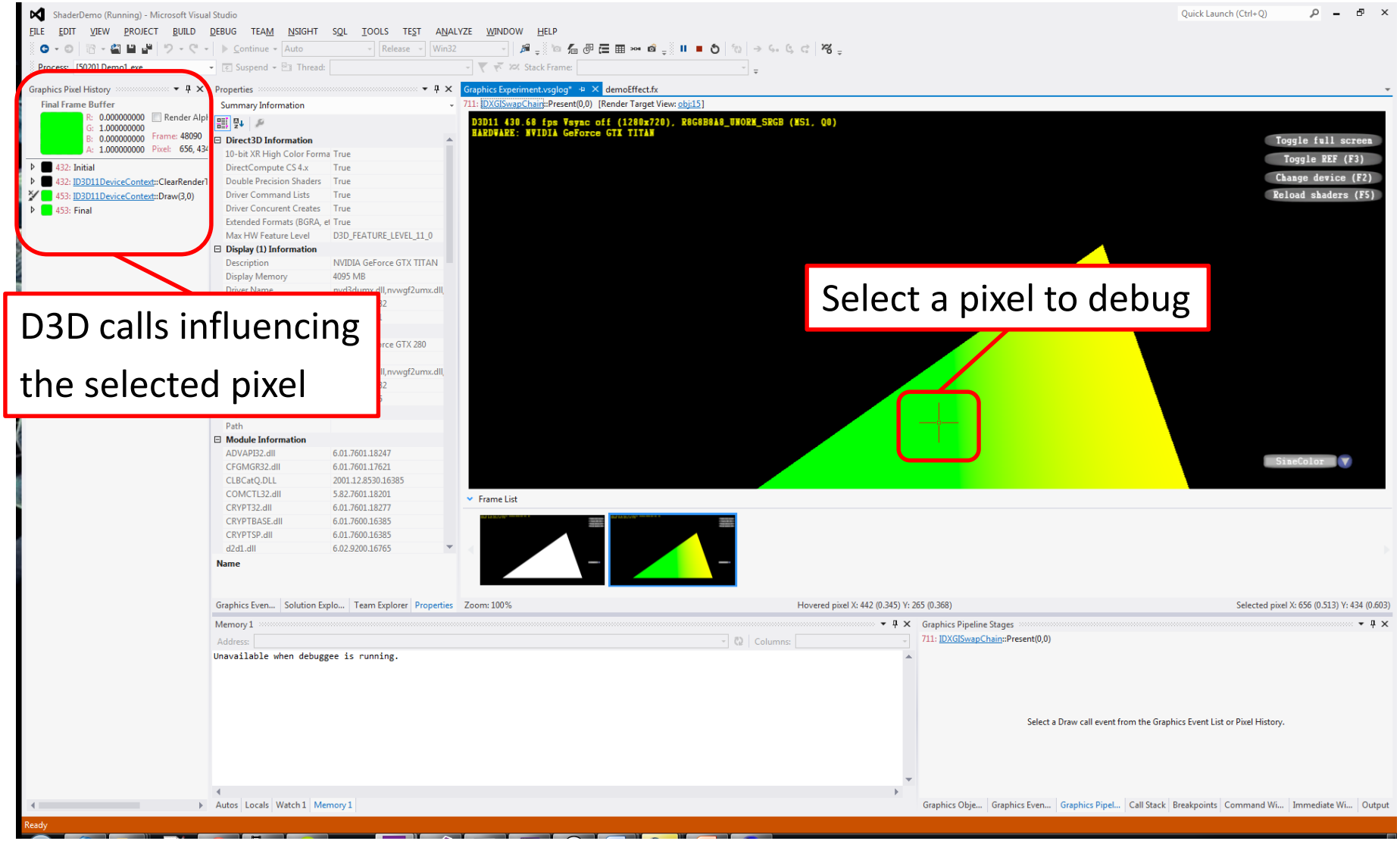
# Graphics Debugging



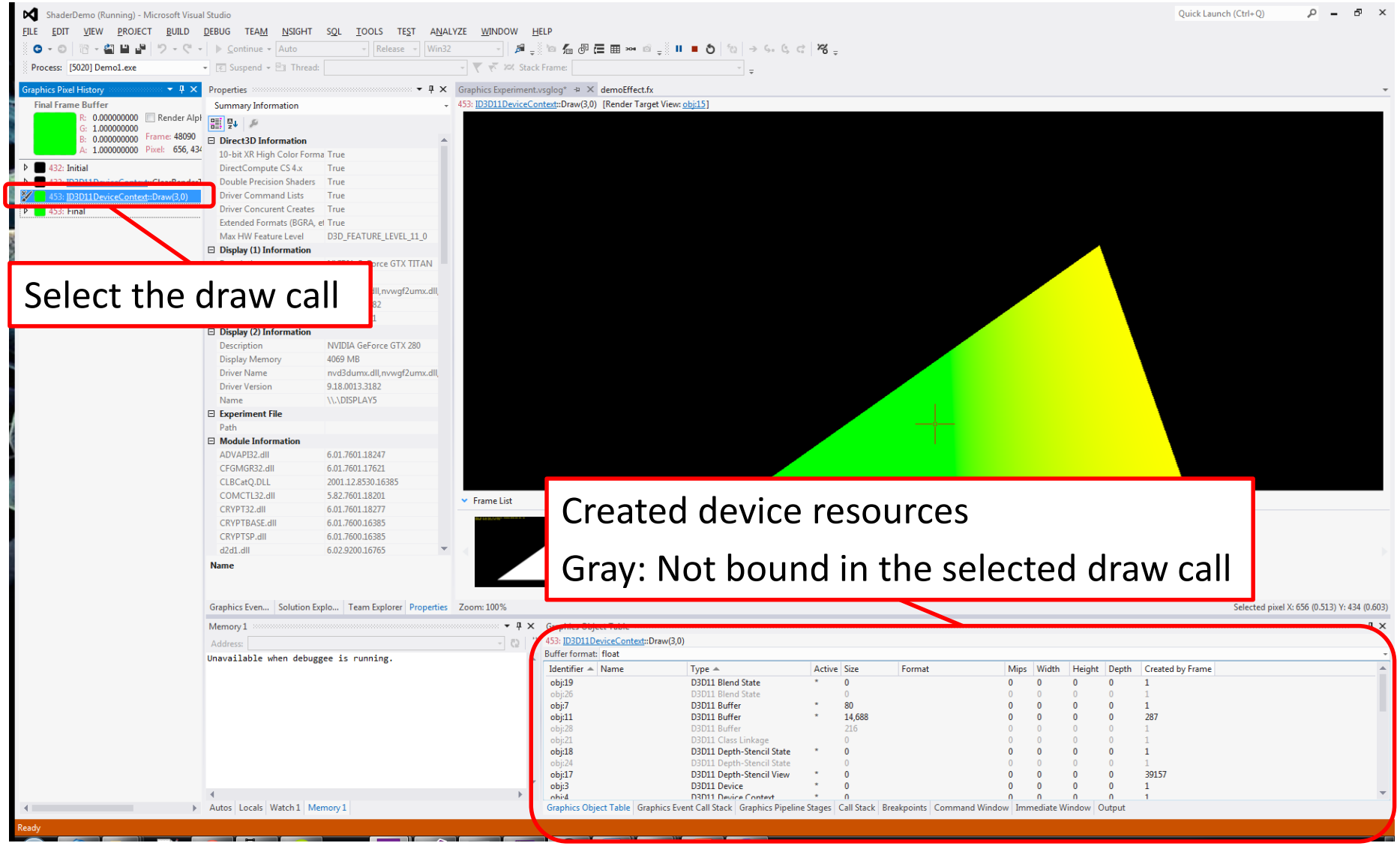
# Graphics Debugging



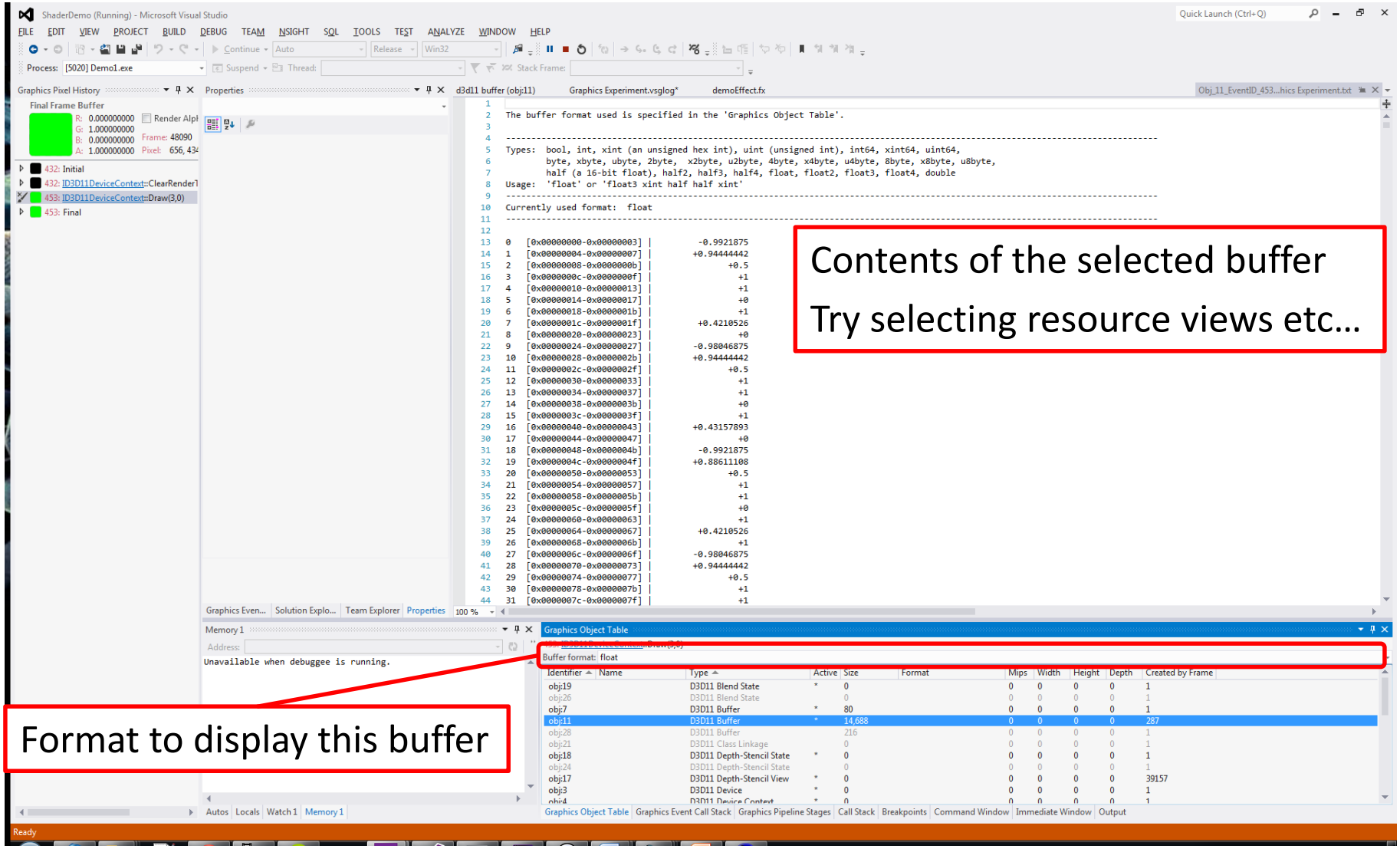
# Graphics Debugging



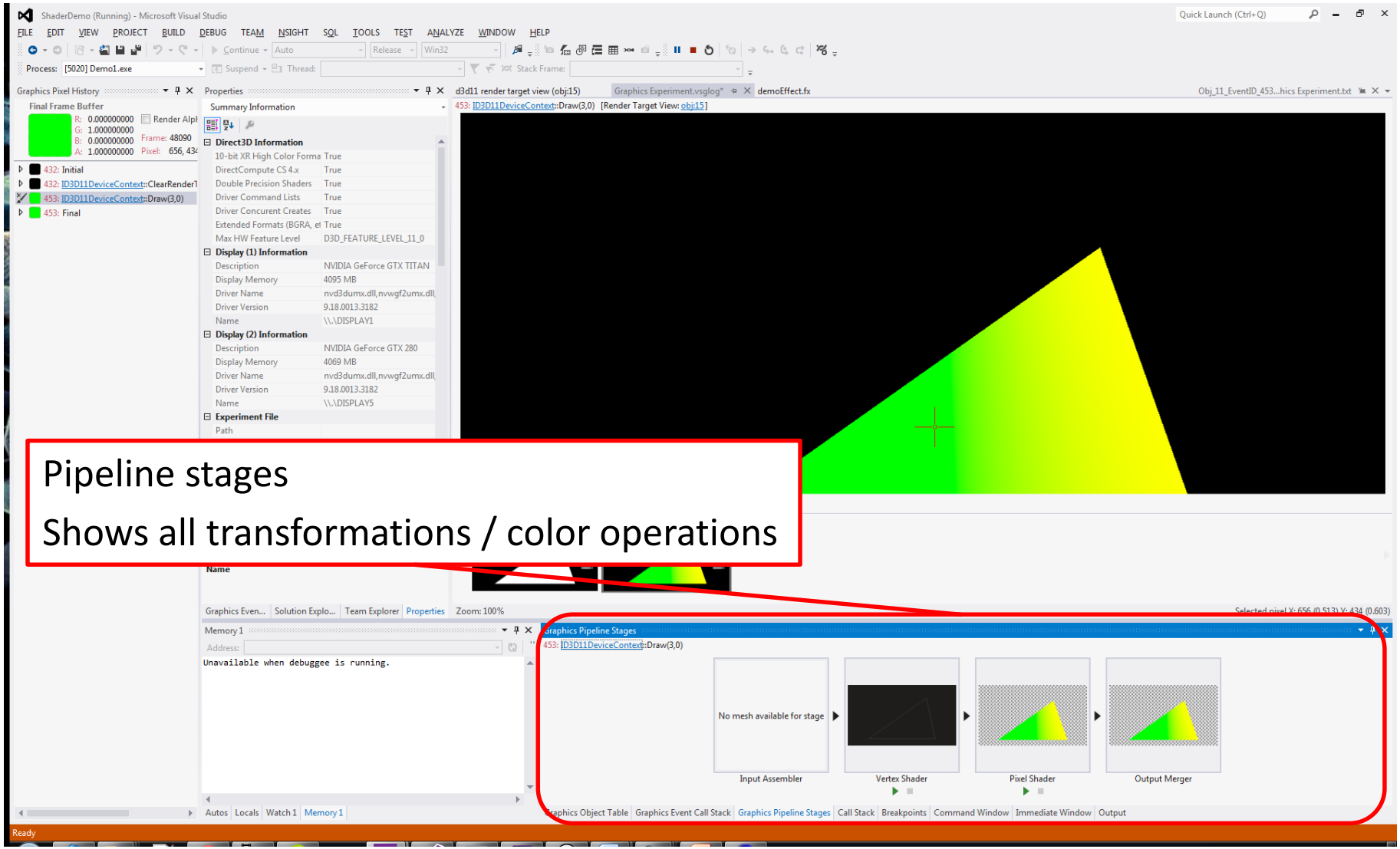
# Graphics Debugging



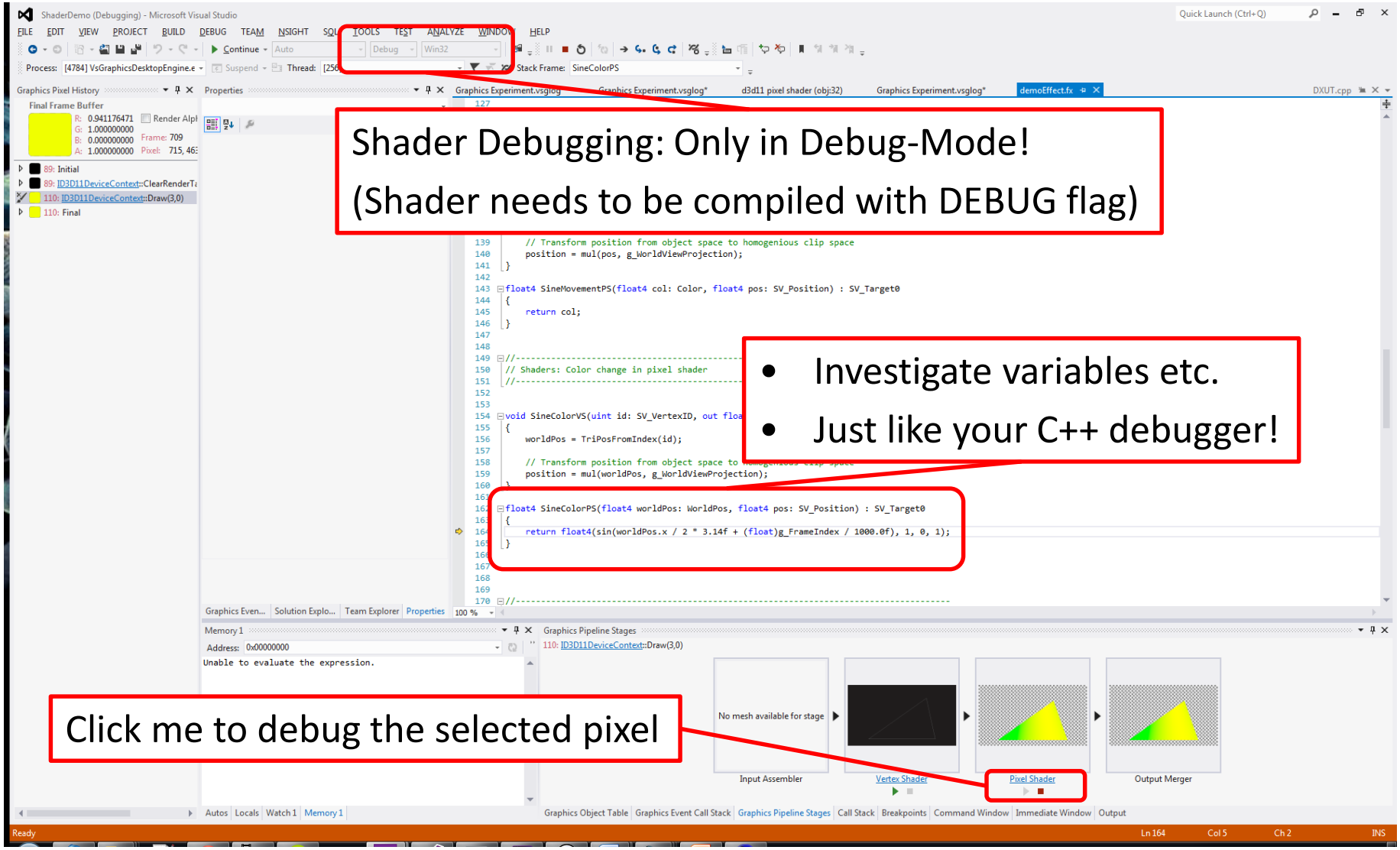
# Graphics Debugging



# Graphics Debugging



# Graphics Debugging



# Questions?