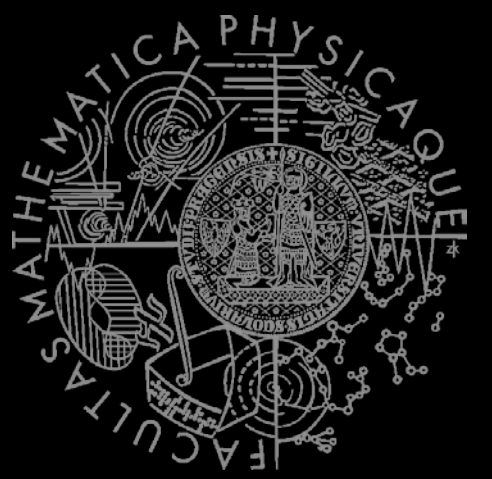
Faculty of Mathematics and Physics
Charles University in Prague
9th March 2016



Graphics for Games

Lab 02 – DirectX 11 Intro

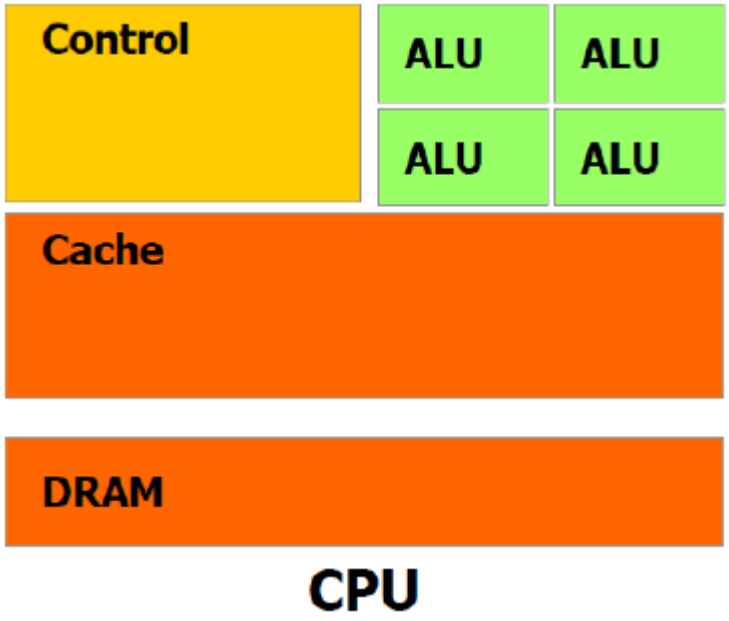
Resources

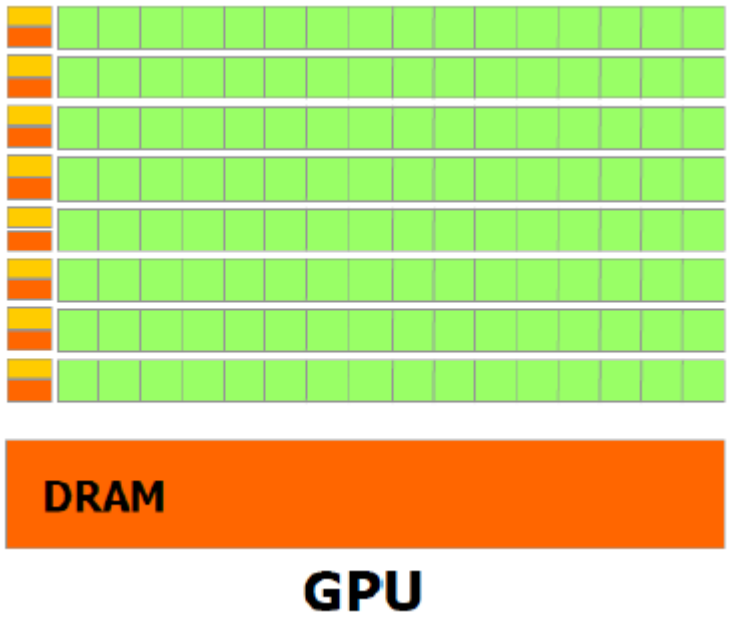
Permanent Slide

- DirectX 11 Pipeline
  - https://msdn.microsoft.com/en-us/library/windows/desktop/ff476882(v=vs.85).aspx
- DirectX Tutorials
  - http://www.rastertek.com/tutdx11.html
- OpenGL 3.3 Tutorials
  - http://www.opengl-tutorial.org/
- OpenGL 3.3 Reference
  - https://www.opengl.org/sdk/docs/man3/
- GLSL 3.3 Specification
  - https://www.opengl.org/registry/doc/GLSLangSpec.3.30.6.pdf
- OpenGL Superbible Book
  - http://www.openglsuperbible.com/
  - http://www.openglsuperbible.com/previous-editions/

CPU vs. GPU

Architecture



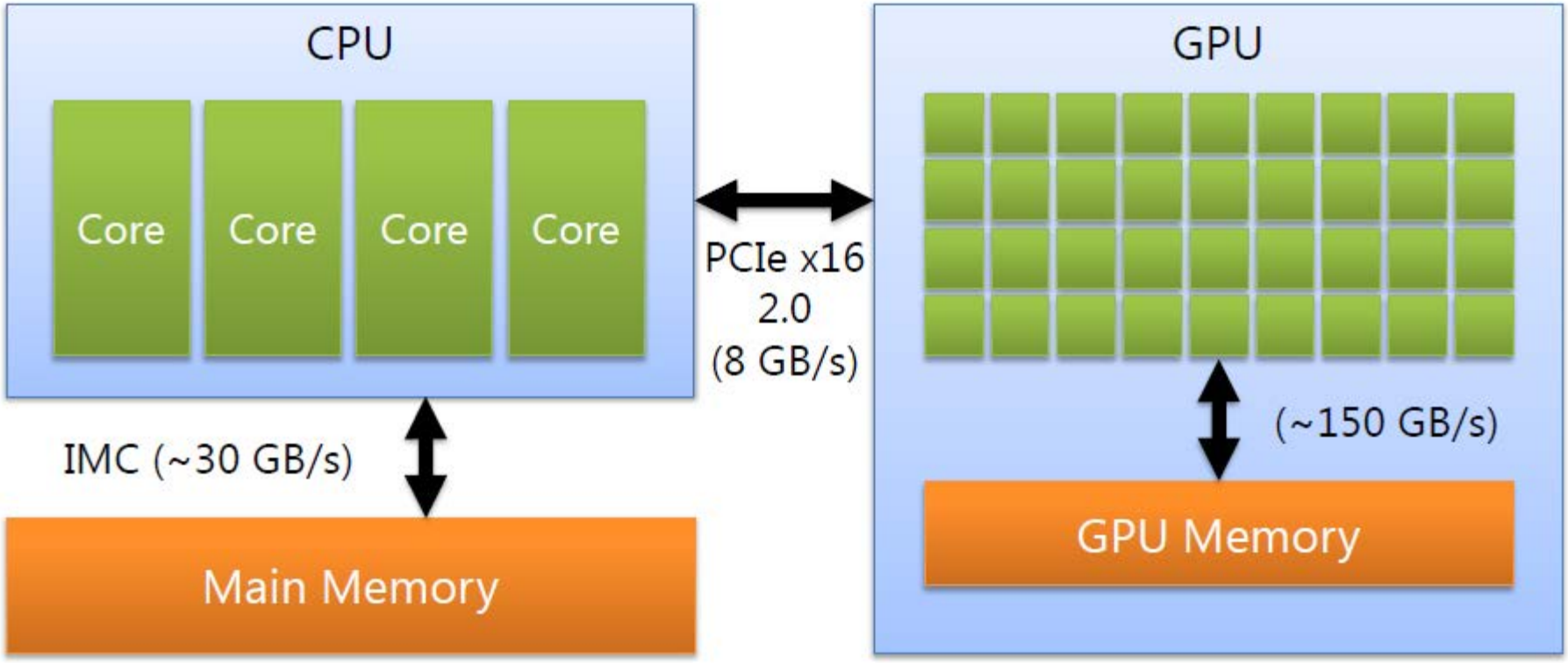


General computations
Lot of instructions
Branching prediction

Massive parallelism
Fewer instructions
Dislikes brancing

CPU vs. GPU

Architecture

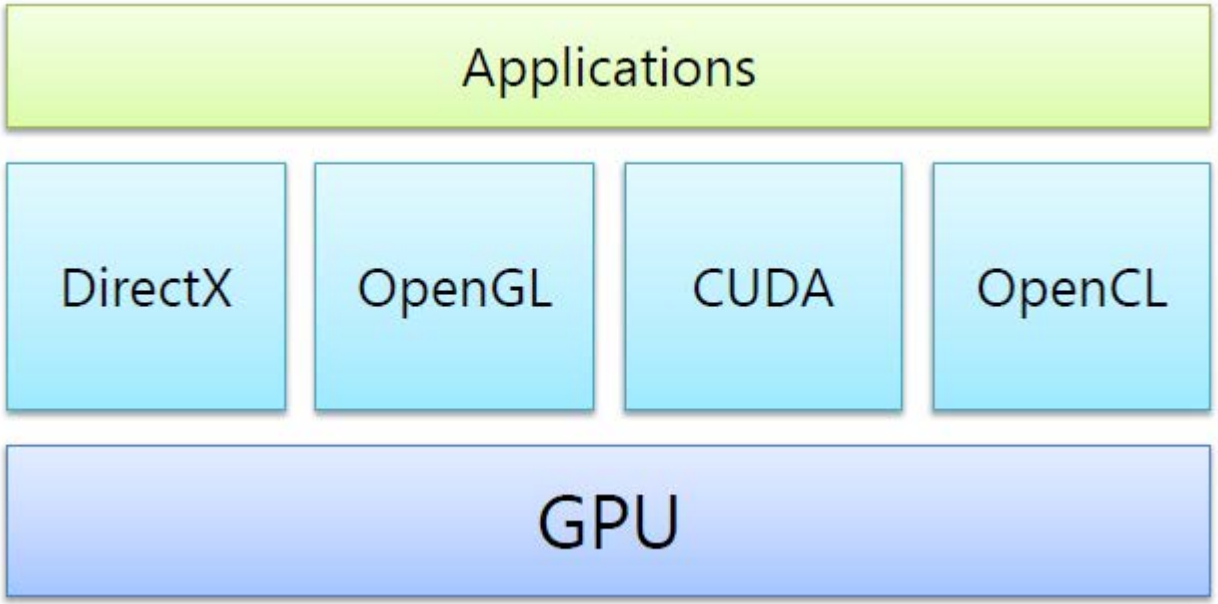


Fast memory access
Slower access to GPU

Extremely fast memory

Programming GPU
Different APIs

- DirectX 11 / 12 ~ OpenGL 4.5 / Vulkan
- CUDA ~ OpenCL



DirectX
Intro

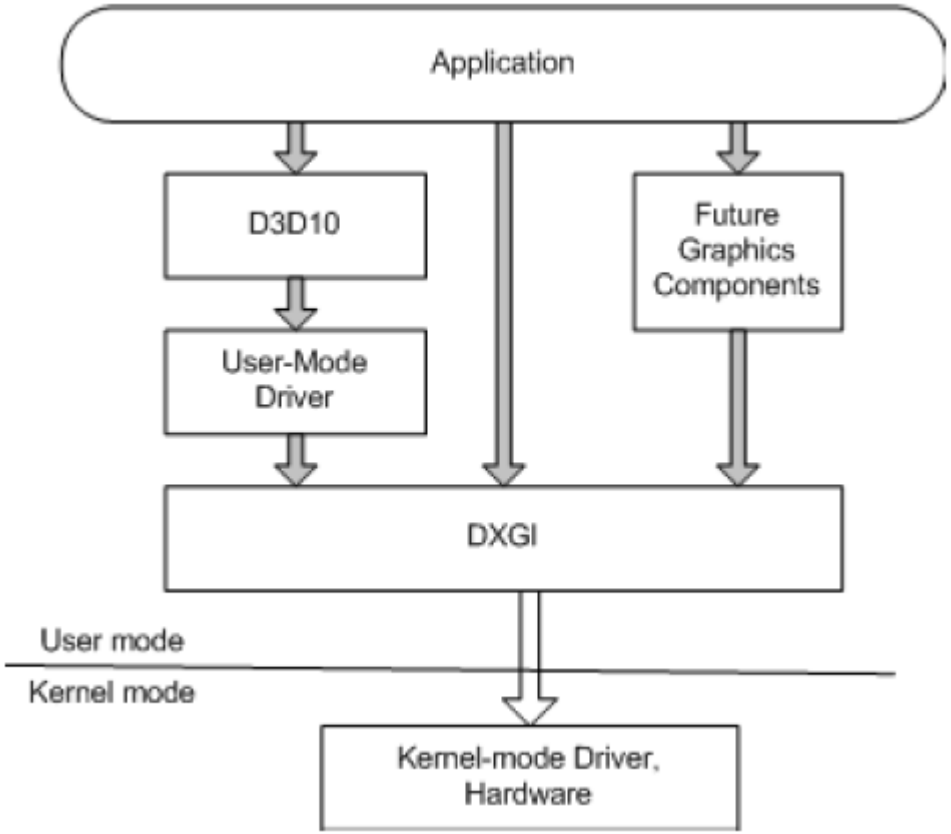
- Set of low-level APIs for creating high-performance multimedia applications, on Microsoft platforms (Windows, Xbox, …)
- Direct3D
  - *Will be covered*
- 2D, 3D Sound and Input
  - *Will NOT be covered*

DirectX

DXGI - DirectX Graphics Infrastructure

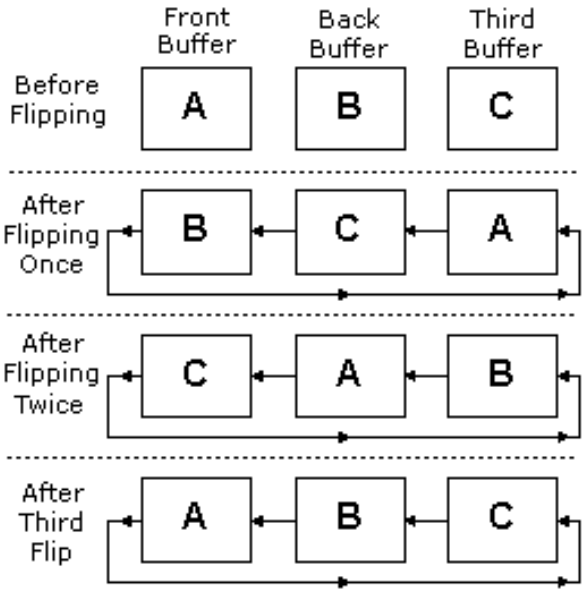
DXGI

- Since Windows Vista
- "Interface to the screen"
- Eases mapping of "framebuffer" to the "monitor screen"
- Managing swap chains
- Handling of window resize



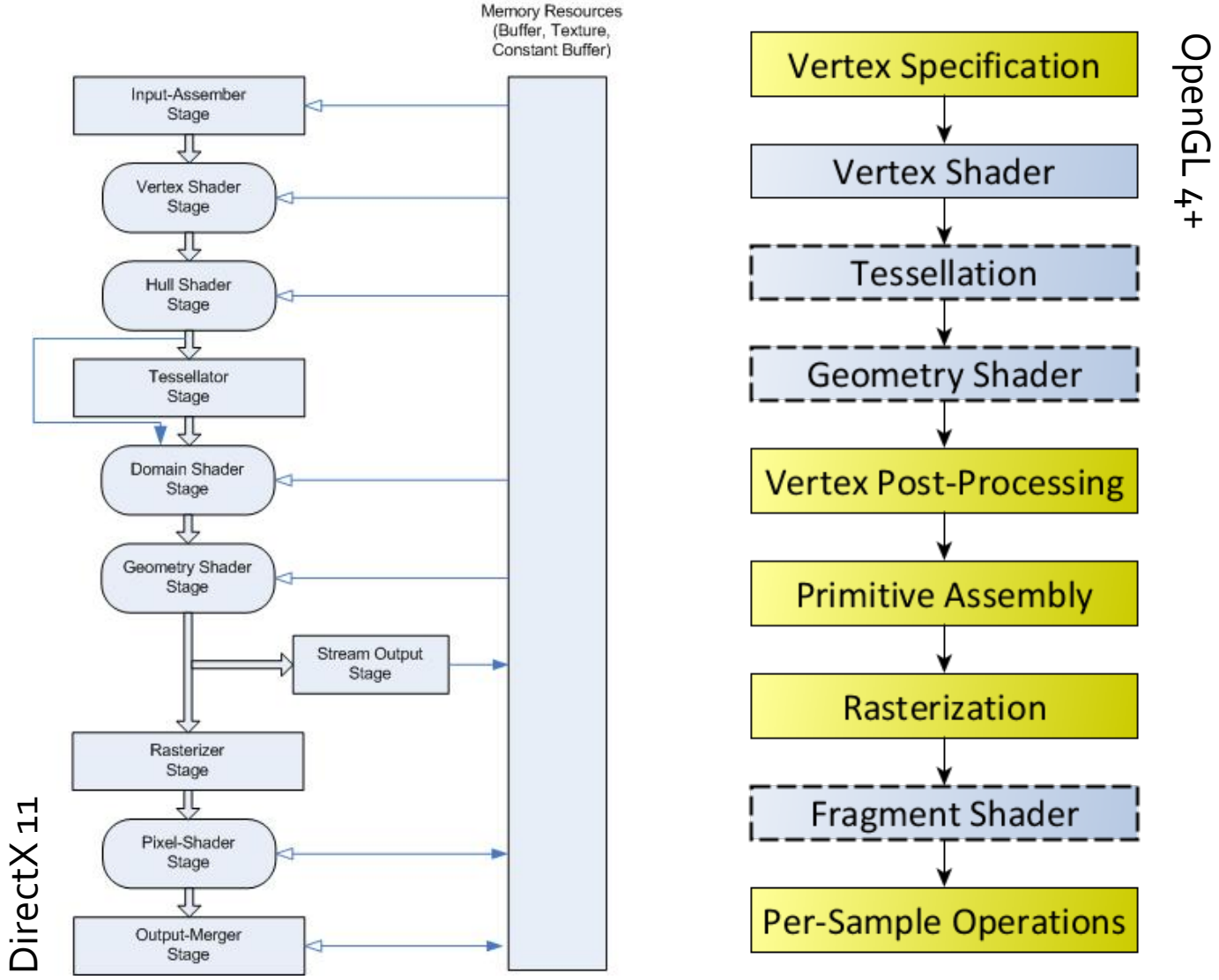
DirectX
Swap Chain

- Encapsulates two or more buffers used for rendering and display
- Double / Triple Buffering
  - Swapping buffers takes time
- Each buffer is "Render Target" a GPU can write to

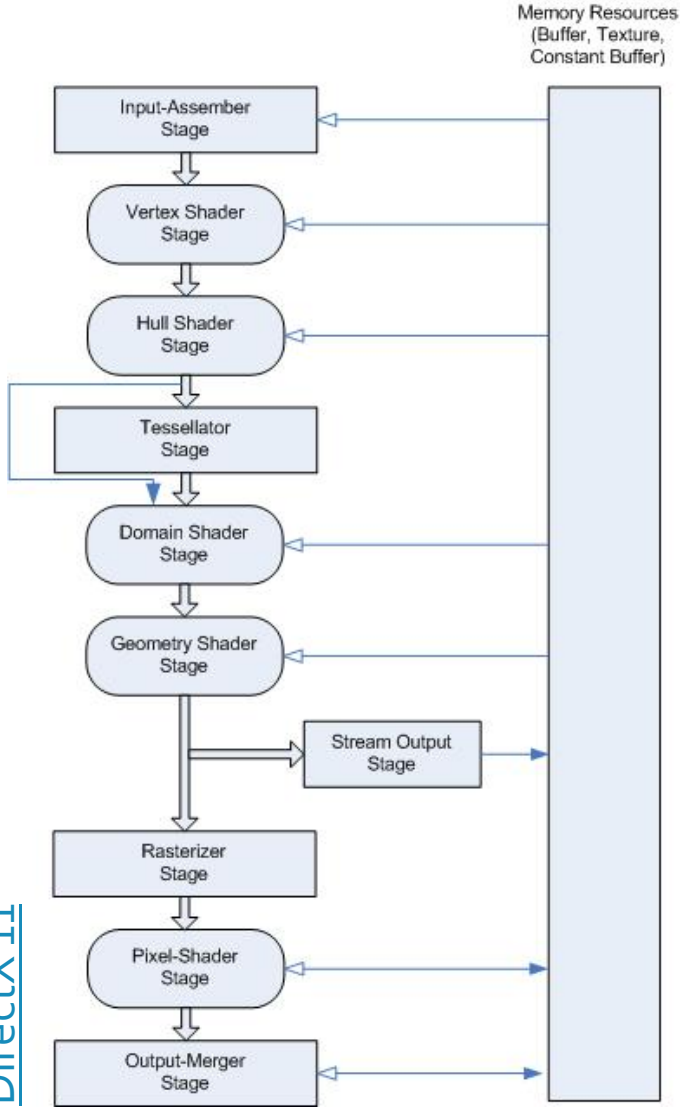


DirectX vs. OpenGL

Graphics Pipeline



DirectX
Graphics Pipeline

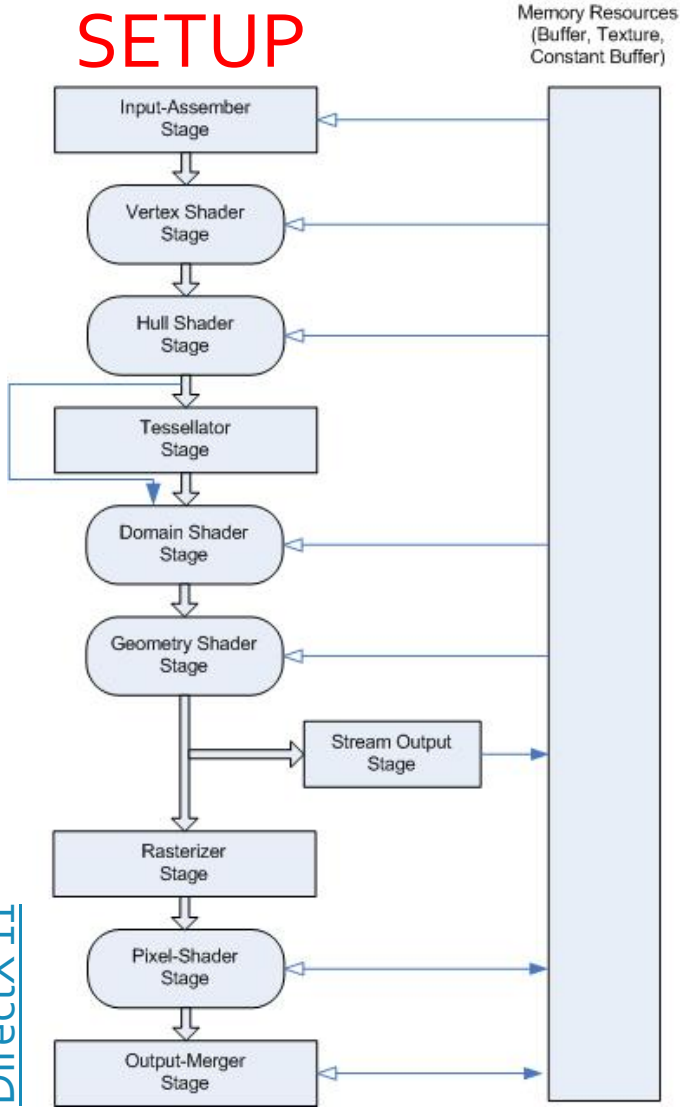


- Programmable (rounded rects) via Shader programs
  - HLSL
  - Profile == Version + Capabilities
    - Program size, Instruction / registry set, …

Shader Model	Shader Profiles	Direct3D support
Shader model 1	vs_1_1	Direct3D 9
Shader model 2	ps_2_0, ps_2_x, vs_2_0, vs_2_x	
Shader model 3	ps_3_0, vs_3_0	
Shader model 4	gs_4_0, ps_4_0, vs_4_0, gs_4_1, ps_4_1, vs_4_1	Direct3D 10
Shader model 5	cs_4_0, cs_4_1, cs_5_0, ds_5_0, gs_5_0, hs_5_0, ps_5_0, vs_5_0	Direct3D 11

  - Each shader must be described in C++ and bound to the pipeline before pipeline can work
  - Mandatory: Vertex + Pixel shader

DirectX Pipeline
SETUP

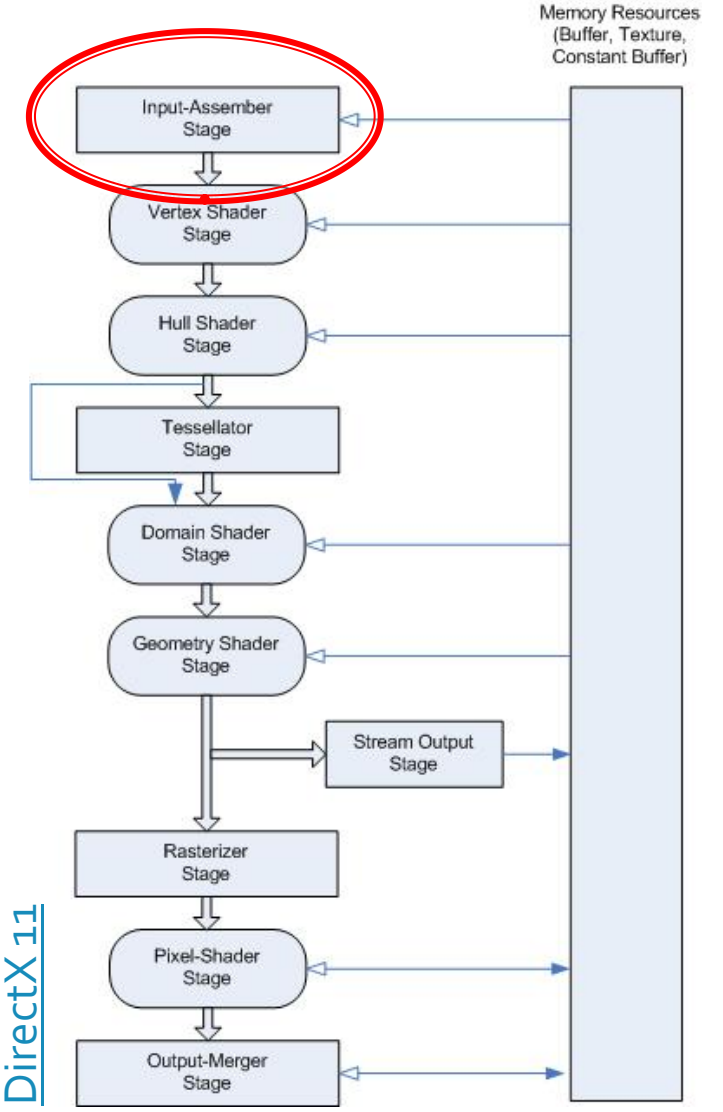


- We have to configure the pipeline before we can use it!

- Load primitives (vertices + indices)
- Load textures
- Compile & describe & attach shaders
- Setup render targets
- …

DirectX Pipeline

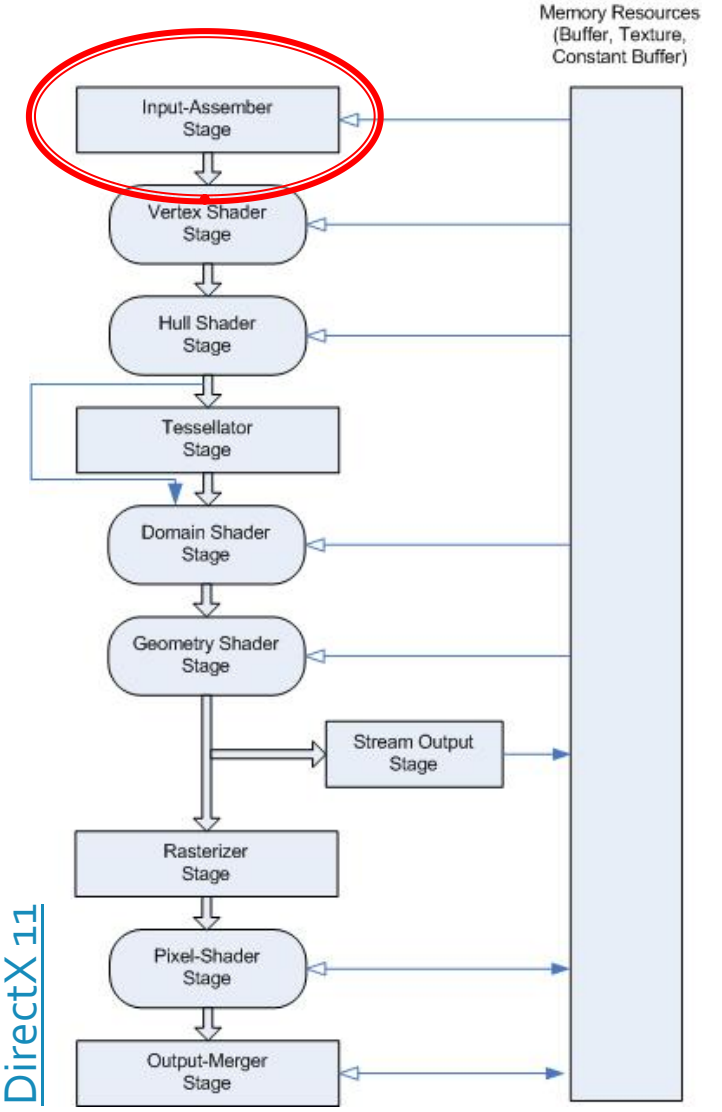
Input-Assembler Stage



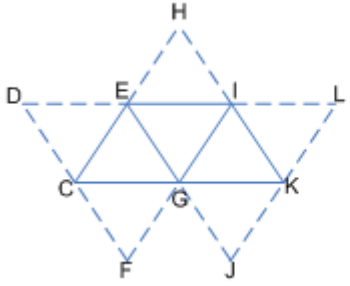
- Read primitive data (points, lines and/or triangles) from user-filled buffers and assemble the data into primitives
- We can provide "primitives with adjacencies"
- Primitives are assigned with system-generated values
  - VertexID, PrimitiveID, InstanceID
  - They can be used within shaders then

DirectX Pipeline

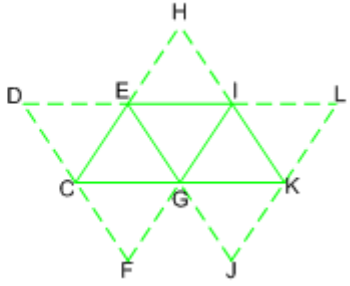
Input-Assembler Stage



- Primitives are assigned with system-generated values



Instance U



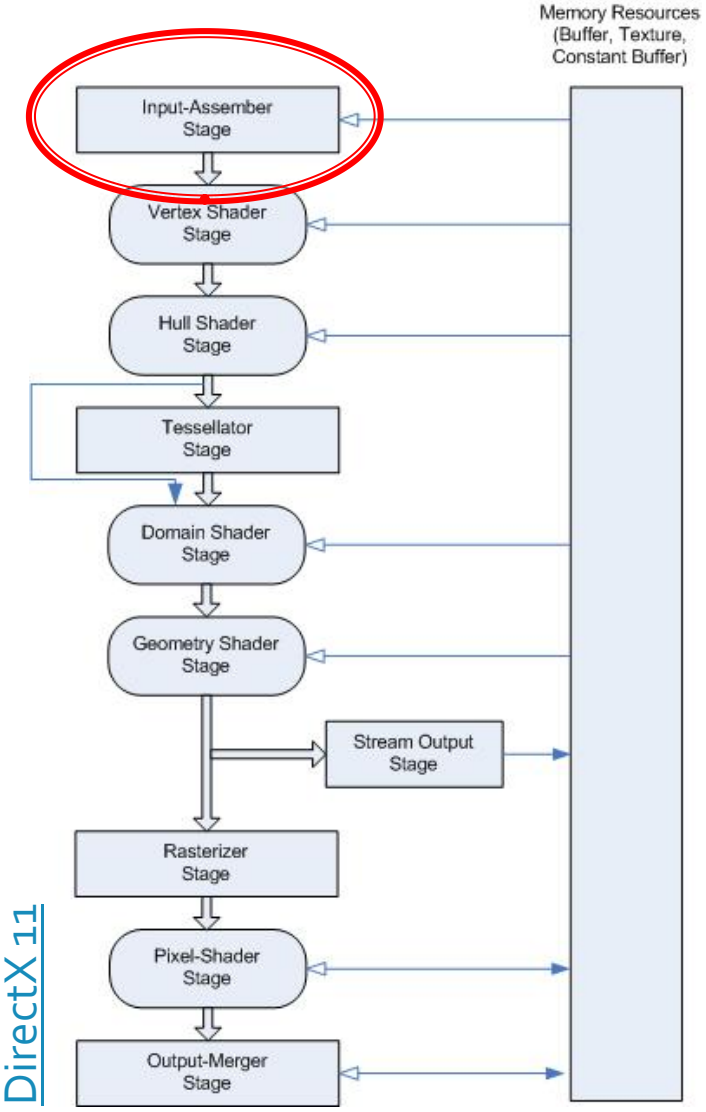
Instance V

Vertex Data	C,U	D,U	E,U	F,U	G,U	H,U	I,U	J,U	K,U	L,U
VertexID	0	1	2	3	4	5	6	7	8	9
InstanceID	0	0	0	0	0	0	0	0	0	0

Vertex Data	C,V	D,V	E,V	F,V	G,V	H,V	I,V	J,V	K,V	L,V
VertexID	0	1	2	3	4	5	6	7	8	9
InstanceID	1	1	1	1	1	1	1	1	1	1

DirectX Pipeline

Input-Assembler Stage



- Creating buffers

```
UINT strides[3];
strides[0] = sizeof(SimpleVertex1);
strides[1] = sizeof(SimpleVertex2);
strides[2] = sizeof(SimpleVertex3);
UINT offsets[3] = { 0, 0, 0 };
g_pd3dDevice->IASetVertexBuffers(
    0,                  //first input slot for binding
    3,                  //number of buffers in the array
    &g_pVertexBuffers, //array of three vertex buffers
    &strides,           //array of stride values, one for each buffer
    &offsets );         //array of offset values, one for each buffer
```

- Specifying primitive topology

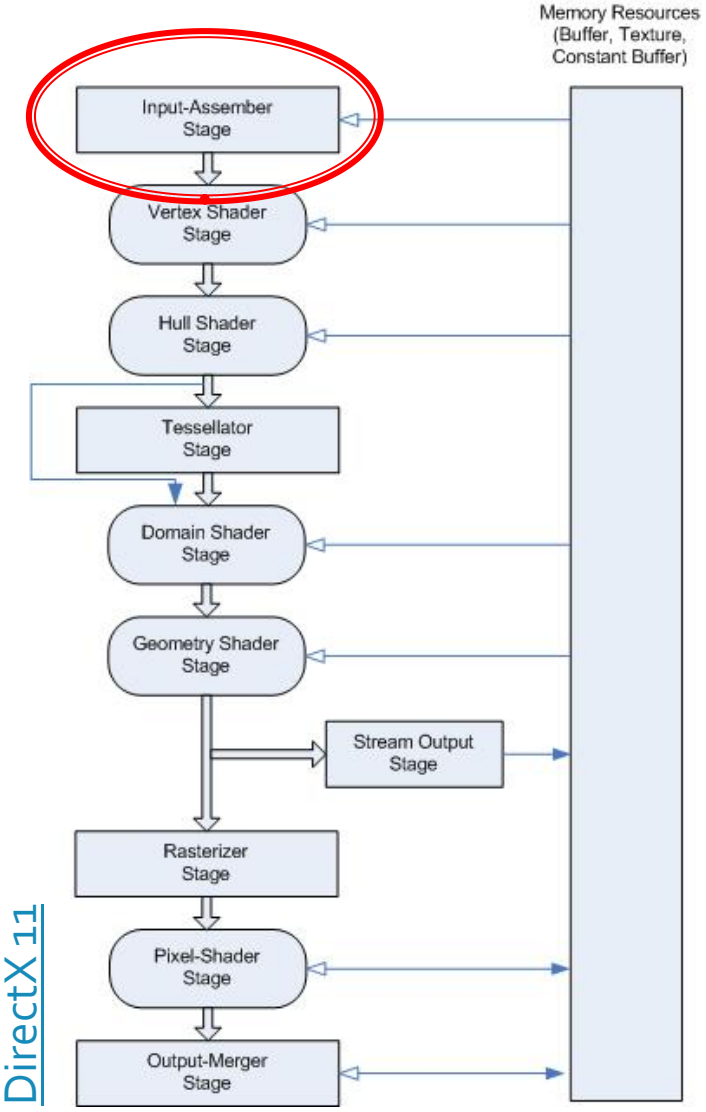
```
g_pd3dDevice->IASetPrimitiveTopology( D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST );
```

- Draw call

ID3D11DeviceContext::Draw

DirectX Pipeline

Input-Assembler Stage



- Creating buffers

```
UINT strides[3];
strides[0] = sizeof(SimpleVertex1);
strides[1] = sizeof(SimpleVertex2);
strides[2] = sizeof(SimpleVertex3);
UINT offsets[3] = { 0, 0, 0 };
g_pd3dDevice->IASetVertexBuffers(
    0,                  //first input slot for binding
    3,                  //number of buffers in the array
    &g_pVertexBuffers, //array of three vertex buffers
    &strides,           //array of stride values, one for each buffer
    &offsets );         //array of offset values, one for each buffer
```

- Specifying primitive topology

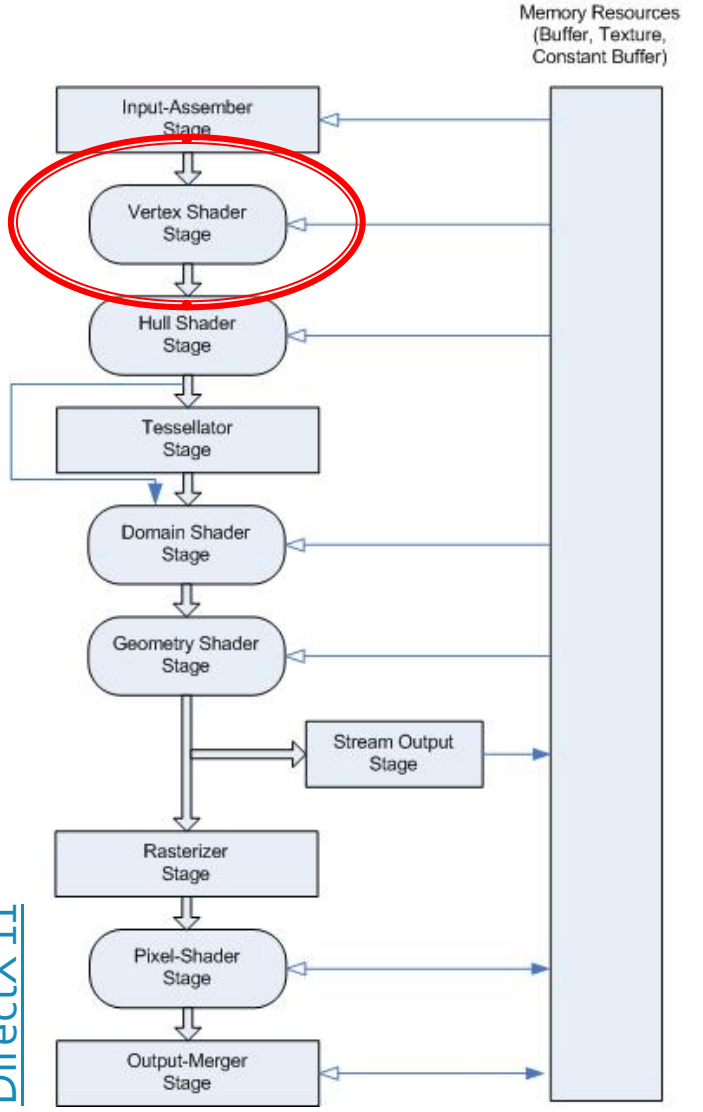
```
g_pd3dDevice->IASetPrimitiveTopology( D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST );
```

- Draw call

ID3D11DeviceContext::Draw

DirectX Pipeline

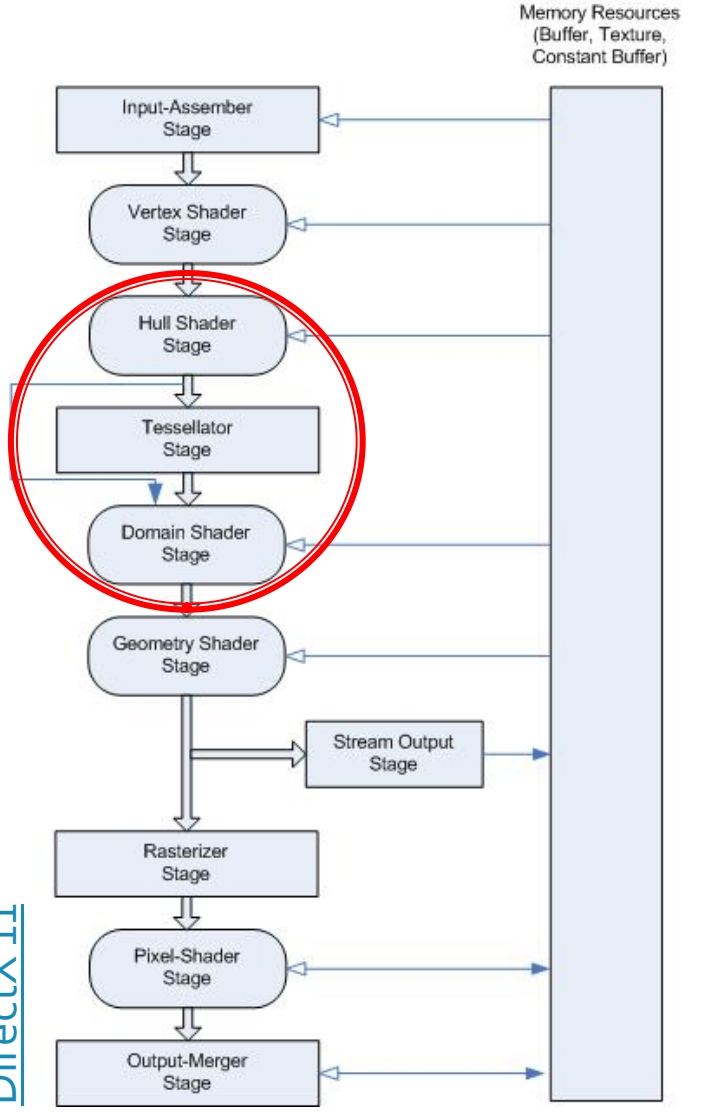
Vertex Shader Stage



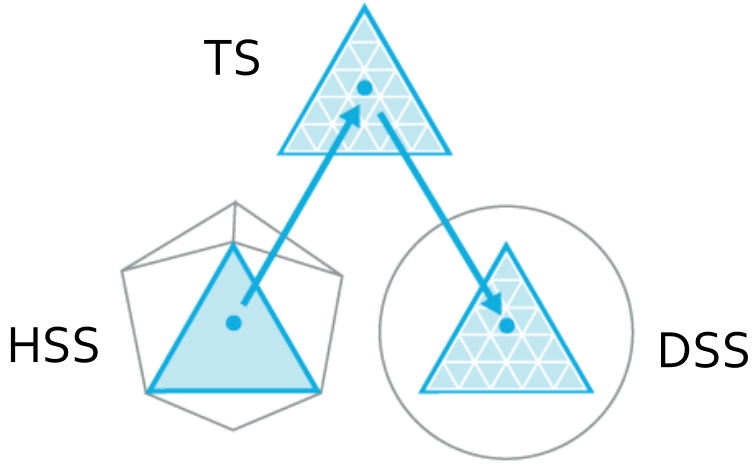
- Processes vertices from the input assembler, performing per-vertex operations such as transformations, skinning, morphing, and per-vertex lighting.
- Always operate over single vertex and produce single output vertex
- May consume VertexID, InstanceID
- Run on adjacent vertices as well
- May already sample textures

DirectX Pipeline

Tesselation

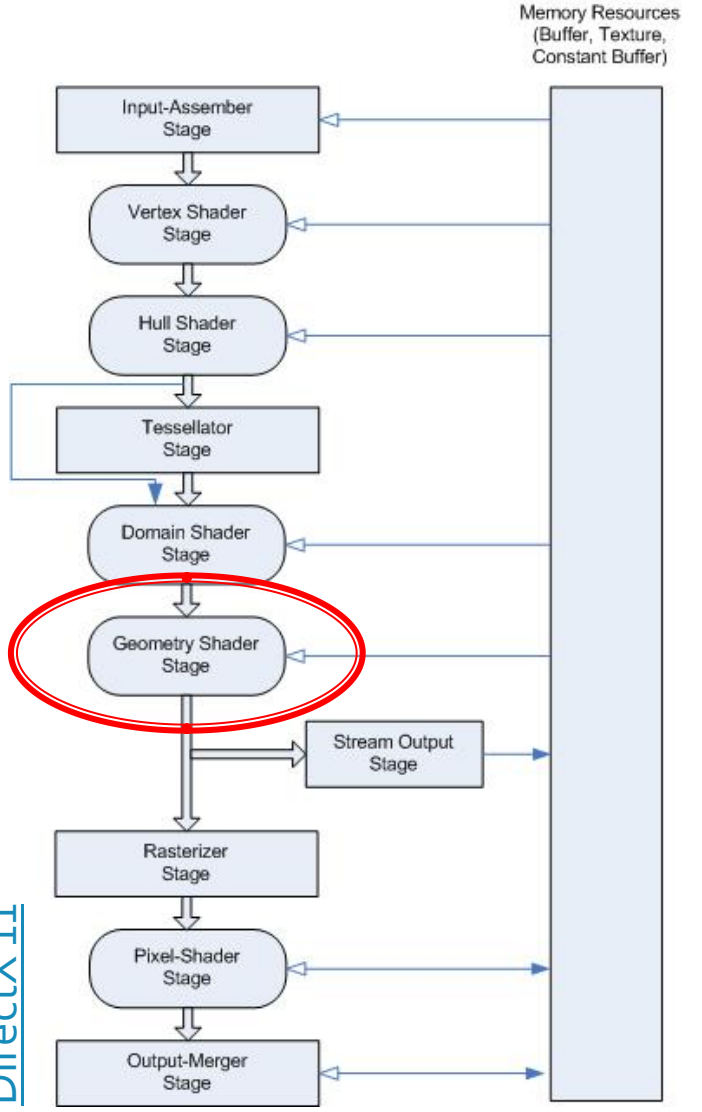


- Save memory & Bandwidth
  - No need to specify that many vertices/indices
- HSS => Patches from vertices
- TS => Divide patches
- DSS => Tweak result vertices

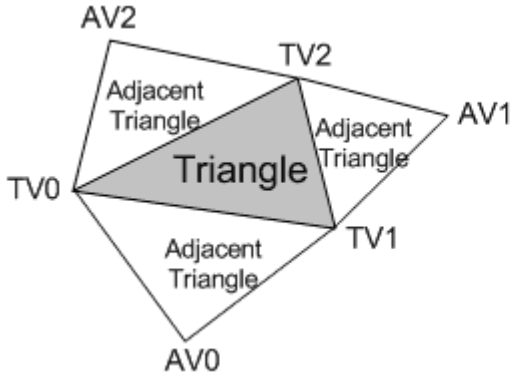


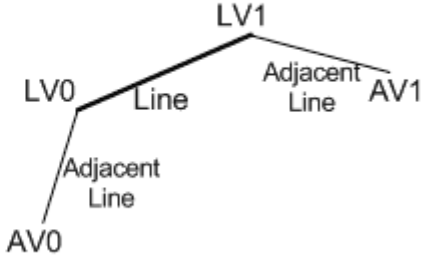
DirectX Pipeline

Geometry Shader Stage



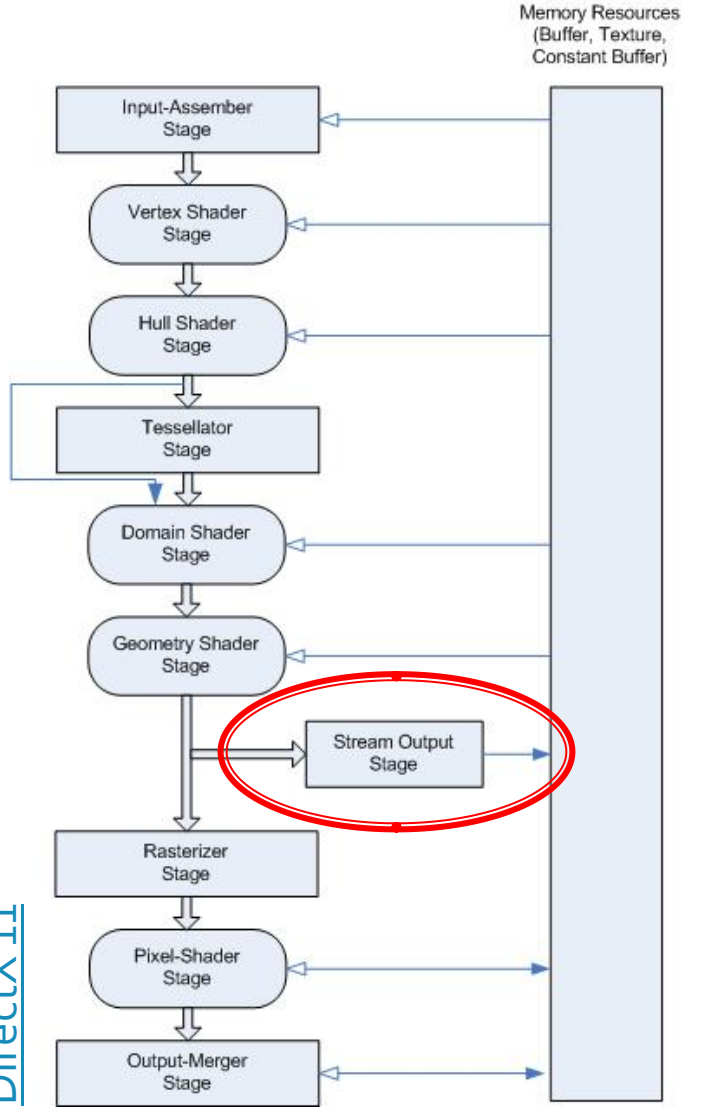
- Input: N vertices
- Output: o-m vertices
- May use PrimitiveID
- Outputs to rasterizer and/or to vertex buffer in GPU memory (can be reused / read by CPU then)
- Example: fur generation / debug





DirectX Pipeline

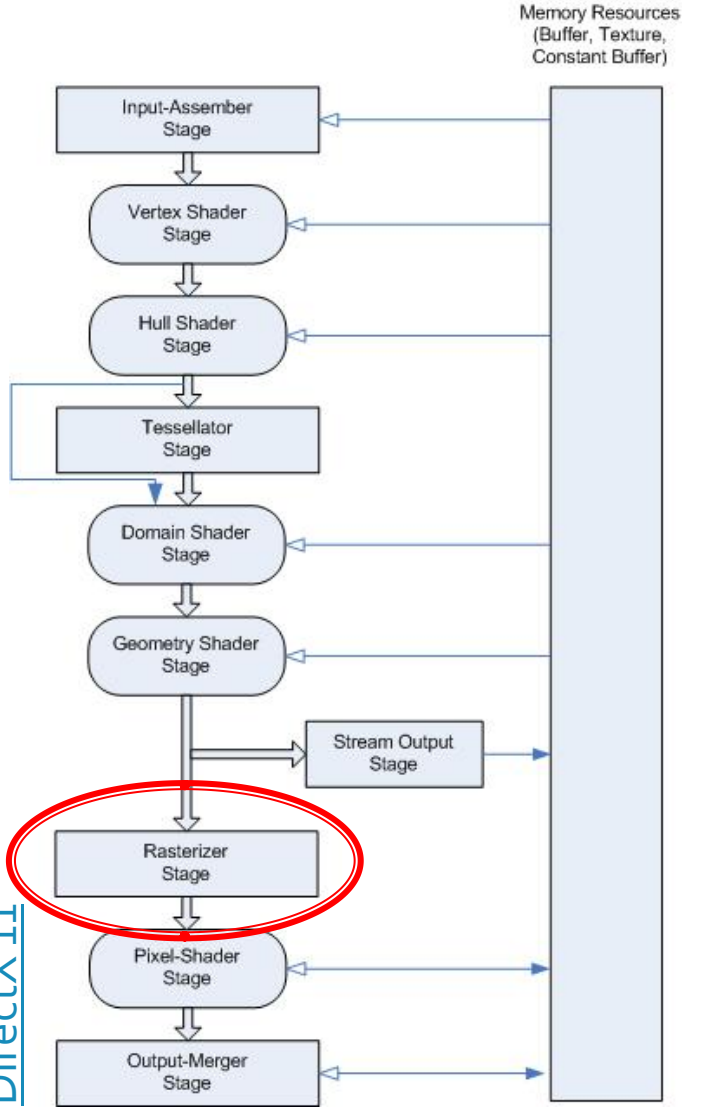
Stream Output Stage



- May be the final step (the rest of the pipeline can be disabled)
- Can output into up-to 4 buffers

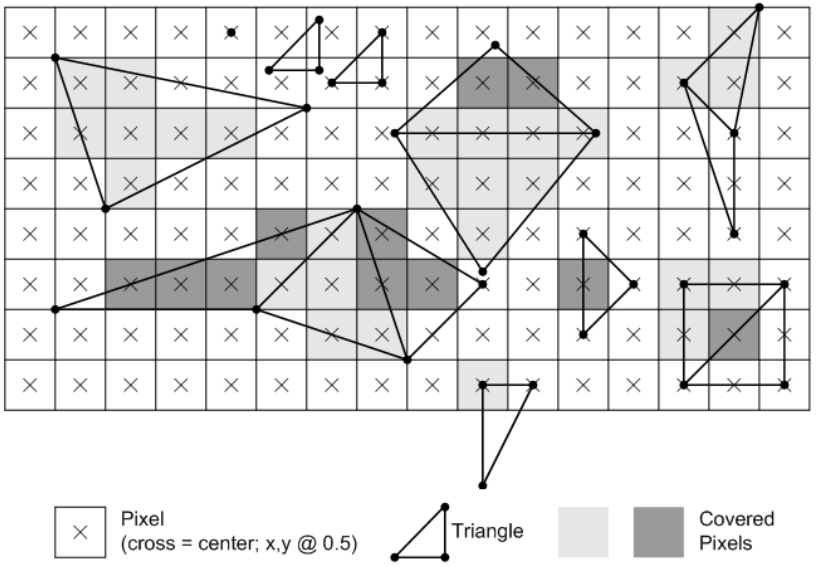
DirectX Pipeline

Rasterizer Stage



DirectX 11

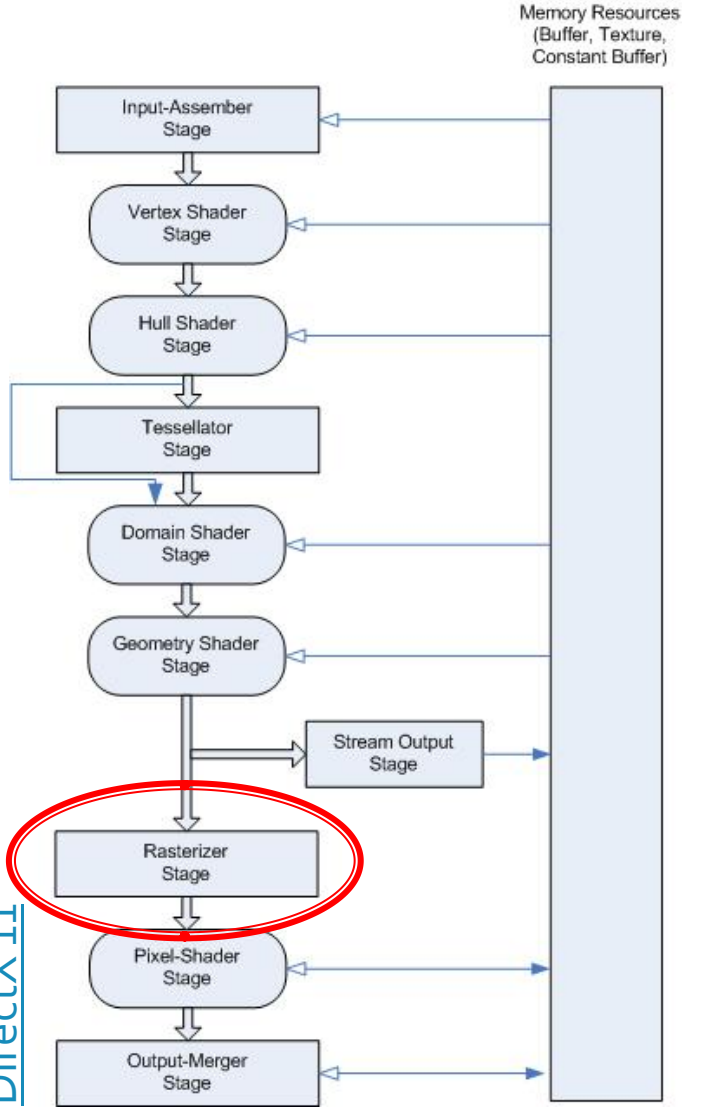
- Vonverts vector information (composed of shapes or primitives) into a raster image (composed of pixels)



Rasterization example w/o multisampling

DirectX Pipeline

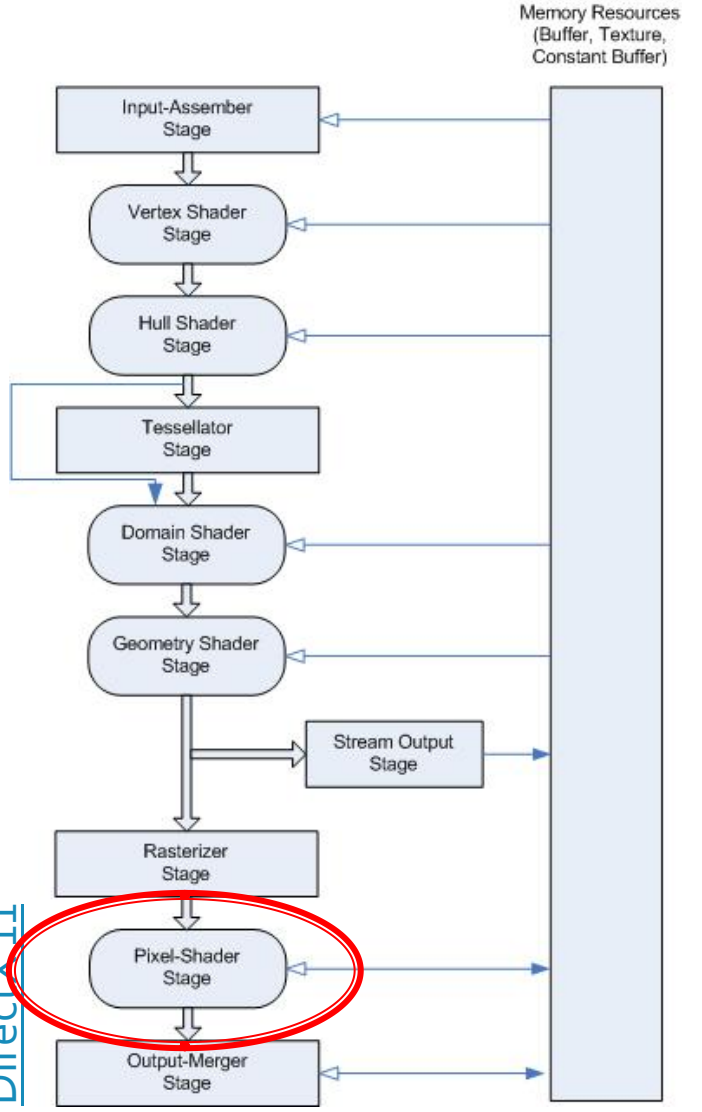
Rasterizer Stage



- Setup
  - Viewport
    - Where to rasterize into the target
  - Scissor test
    - What to rasterize
  - State
    - E.g. Depth-Clip?

DirectX Pipeline

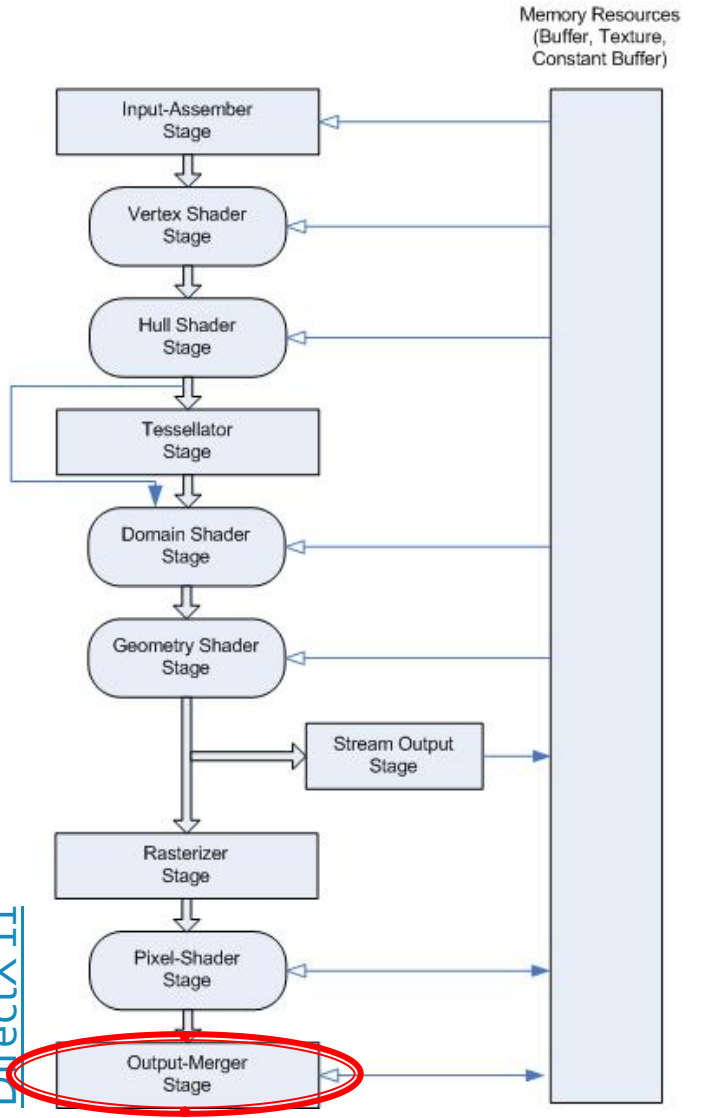
Pixel Shader Stage



- Per-pixel lighting and post-processing
- When the pipeline is configured without a geometry shader, a pixel shader is limited to 16, 32-bit, 4-component inputs. Otherwise, a pixel shader can take up to 32, 32-bit, 4-component inputs.

DirectX Pipeline

Output Merger Stage



- Merge output of Pixel Shaders with Render Target according to the pipeline state
  - Depth / Stencil test
  - Blending with render target
- May render to multiple Render Targets (up to 8)

DirectX
Setup (old way)

- DirectX SDK June 2010
  - http://www.microsoft.com/en-us/download/details.aspx?id=6812
- Configuring Includes + Libraries for VS2010 Project
  - http://www.directxtutorial.com/lessonarticle.aspx?id=4
- D3D Headers + Pragmas + Initialization
  - http://www.directxtutorial.com/Lesson.aspx?lessonid=11-4-2

DirectX
Setup (new way)

- Windows 8 / 8.1 / 10 SDK
  - DirectX SDK has been merged into Windows SDK
    - Where is DirectX SDK (2015) ?
  - Not 1:1 mapping!
    - Living without D3D

⇒ Not so old tutorials not compilable out-of-the-shelf on Windows 8+ even if you install DirectX SDK (June 2010)

⇒ Still RasterTek DX11 tutorials are great to learn from

DirectX
Init – COM Objects

ID3D11Device *dev;

This variable is a pointer to a device. In Direct3D, a device is an object that is intended to be a virtual representation of your video adapter. What this line of code means is that we will create a COM object called ID3D11Device. When COM makes this object, we will ignore it, and access it only indirectly using this pointer. We'll cover how this happens in a moment.

ID3D11DeviceContext *devcon;

A device context is similar to a device, but it is responsible for managing the GPU and the rendering pipeline (the device mostly handles video memory). This object is used to render graphics and to determine how they will be rendered.

IDXGISwapChain *swapchain;

The swap chain is the series of buffers which take turns being rendered on. This variable is a pointer to such a chain.

DirectX

Init – Create D3D COM Objects

```
DXGI_SWAP_CHAIN_DESC scd; // "create D3D context for the window"

// clear out the struct for use
ZeroMemory(&scd, sizeof(DXGI_SWAP_CHAIN_DESC));

// fill the swap chain description struct
scd.BufferCount = 1;
// one back buffer
scd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
// use 32-bit color
scd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
// how swap chain is to be used
scd.OutputWindow = hWnd;
// the window to be used
scd.SampleDesc.Count = 4;
// how many multisamples
scd.Windowed = TRUE;
// windowed/full-screen mode

// create a device, device context and swap chain using the information in the scd struct
D3D11CreateDeviceAndSwapChain(NULL, D3D_DRIVER_TYPE_HARDWARE, NULL, NULL, NULL, NULL,
D3D11_SDK_VERSION, &scd, &swapchain, &dev, NULL, &devcon);
```

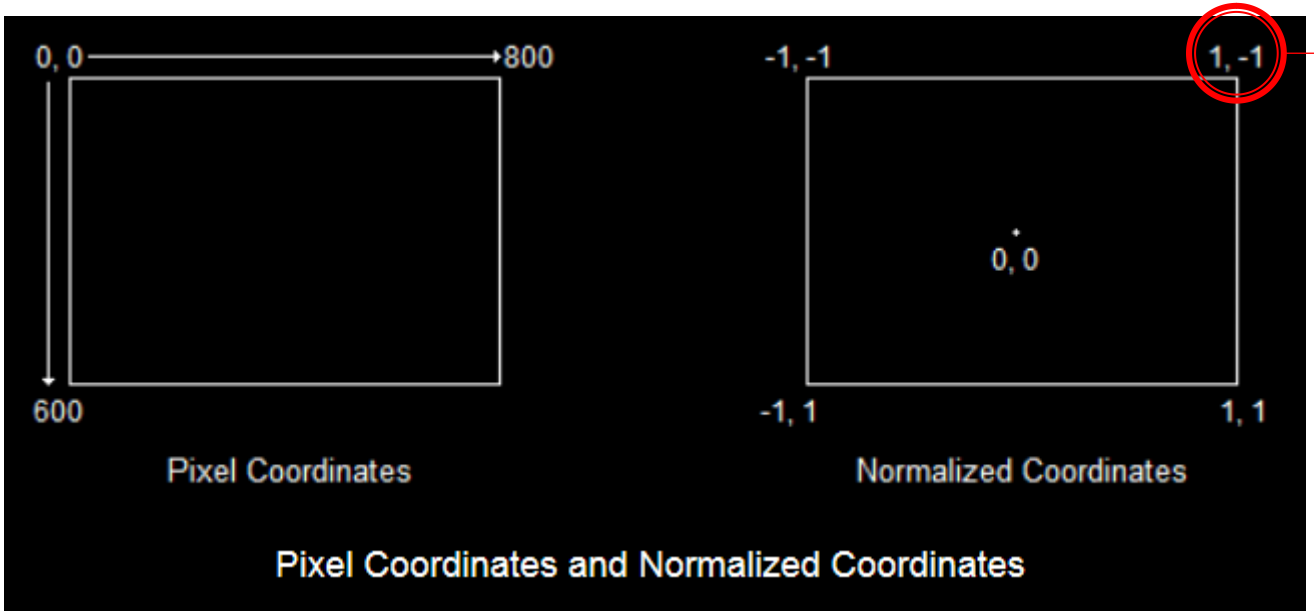
DirectX

Init - Create Render Target for Backbuffer

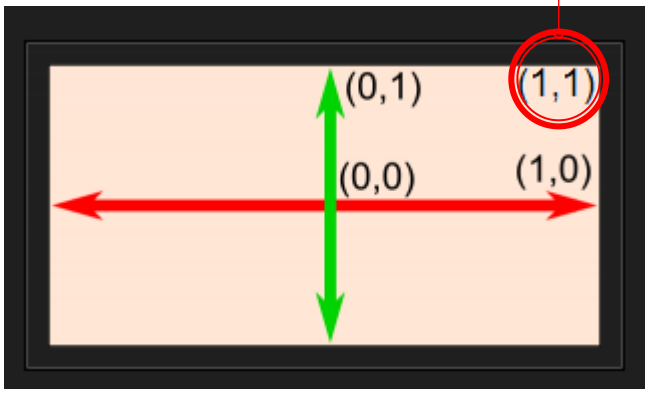
```
// get the address of the back buffer
ID3D11Texture2D *pBackBuffer;
swapchain->GetBuffer(0, __uuidof(ID3D11Texture2D), (LPVOID*)&pBackBuffer);

// use the back buffer address to create the render target
dev->CreateRenderTargetView(pBackBuffer, NULL, &backbuffer);
pBackBuffer->Release();

// set the render target as the back buffer
devcon->OMSetRenderTargets(1, &backbuffer, NULL);
```

DirectX vs. OpenGL

Vertex + Index Buffers



DirectX



OpenGL

DirectX
Vertex + Index Buffers

dx11tuto5-FirstTriangle
- Descriptors (DX) instead of function parameters (OGL)

`modelclass.cpp`
- `ModelClass::InitializeBuffers`
  - Initializing Vertex + Index buffers
- `ModelClass::RenderBuffers`
  - Setting buffers into the pipeline

`graphicsclass.cpp`
- `GraphicsClass::Render`
  - Rendering the triangle

=> dx11tuto6-LightedTriangle

DirectX
Vertex & Pixel Shaders

dx11tuto6-LightedTriangle

`Light.vs, Light.ps`

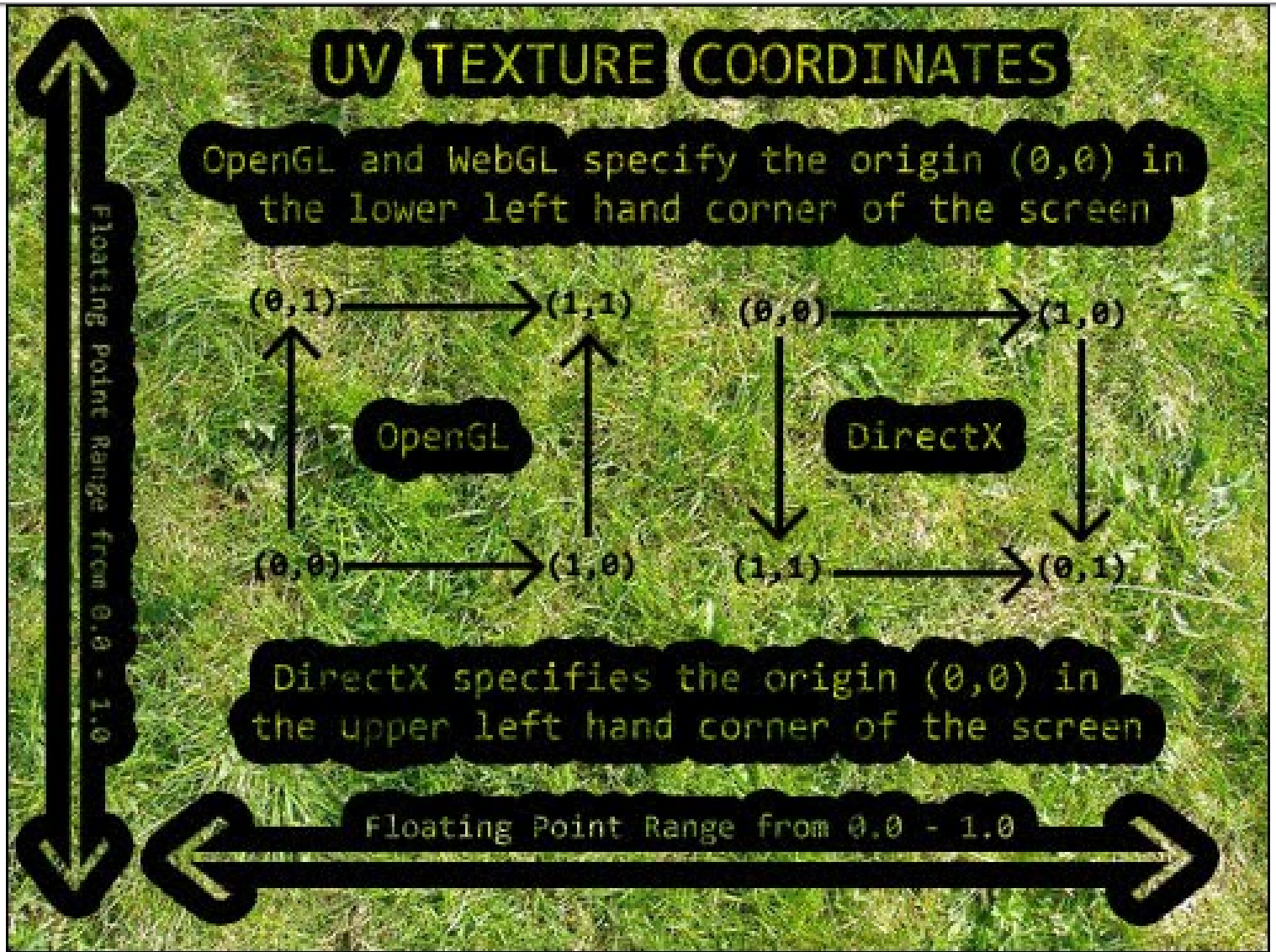
`lightshaderclass.cpp`

- `LightShaderClass::InitializeShader`
  - "Binding shader parameters"
- `LightShaderClass::SetShaderParameters`
  - Setting constants

=> dx11tuto6-LightedTriangle

DirectX

Textures



DirectX vs. OpenGL

Textures

- OpenGL => Texture parameters bound to image
- DirectX => Sampler settings independent on data

`texturearrayclass.cpp`

- `TextureArrayClass::Initialize`
  - Loading texture images

`multitextureshaderclass.cpp`

- `MultiTextureShaderClass::InitializeShader`
  - Sampler setup
- `MultiTextureShaderClass::SetShaderParameters`
  - Setting texture array to PS

DirectX
Rendering to Textures

`rendertextureclass.cpp`

- `RenderTextureClass::Initialize`
  - Loading texture images
- `RenderTextureClass::RenderToTexture`
  - Setting texture as render target (and back)

`graphicsclass.cpp`

- `GraphicsClass::Render`
  - First rendering to the texture
  - Then using rendered texture