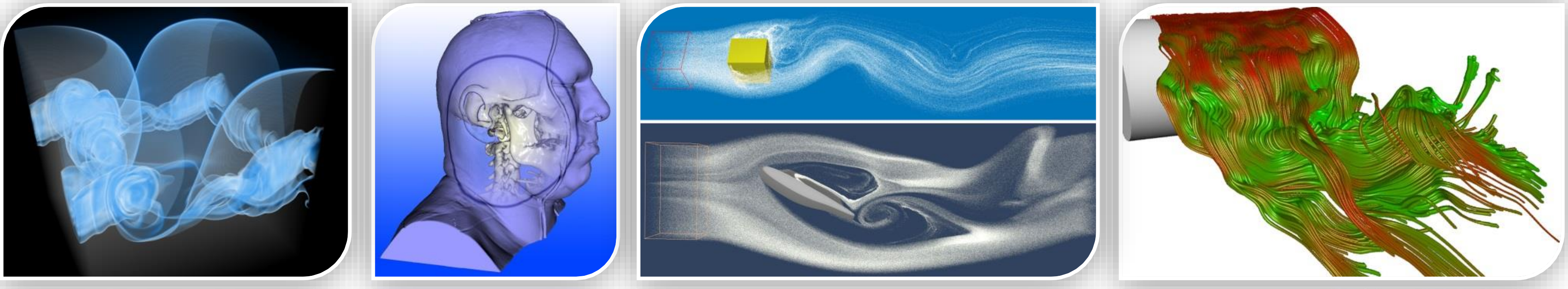
Master Practical Course

Interactive Visual Data Analysis



tum.3D

computer graphics & visualization

Overview - Today

- **Organizational Issues**
- Template Project & Assignment 1
- DXUT
- Effects & HLSL
- Transformations

Organizational Issues

- If not already done, **register in TUMonline** for this course!
- Weekly assignment cycle
  - Presentation of new assignment every Wednesday, 3pm-4pm, room MI 02.13.010
  - Submission deadline every Wednesday, 9am
- Groups of two or three students (two preferred)
  - Everybody should understand the whole code
- To pass the course:
  - Presence at the weekly assignment presentations (at least one student per group)
  - **Timely** and **complete** solution of **all** assignments!

Grading

- The grade depends on
  - The quality of your solutions
  - A short oral examination at the end of the semester
- You‘ll get weekly evaluation emails:
  - Rating is based on pluses (+) / minuses (–) for things **we liked and optional exercises** / **didn‘t like**
  - Pluses cancel out minuses
  - Final Grade:
    Sum of zero or more pluses = 1.0
    3 minuses ≈ +1/3 grade
    (subject to change)

Computer Room

- You can use the machines in our lab, room MI 02.13.036 (choose wisely: not all machines have a DX11 graphics card)
- We‘ll send you an email with your account for lab & SVN
  - **You have to activate your account by logging in once in the lab**

Required Soft- & Hardware

- If you want to work at home, you‘ll need
  - Windows
  - DX11 graphics card (NVIDIA 4xx +, AMD 5xxx +)
  - Microsoft Visual Studio 2012
    http://dreamspark.rbg.tum.de/
    (includes everything you need for working with DirectX)
  - Subversion Client (TortoiseSVN, AnkhSVN)

SVN Submission

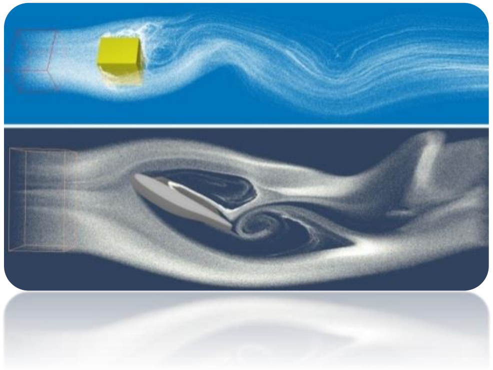
- Submission of your weekly solutions via SVN commit
- One SVN repository for all
  - One subfolder for every group
  - Several shared folders, containing assignments, slides, datasets, libraries, etc.
- We will rate
  - the content of the „solution“ folder within your group folder
  - the last revision before the deadline
- Write a readme.txt in your „solution“ folder
  - Should we rate a specific SVN-Revision?
  - Does something not work as expected?
  - Hotkeys, instructions on the usage of your tool, …

SVN Submission

- Code has to compile and start **out-of-the-box**
  - Check-In all dependencies
  - If we cannot even start your program, no pass!
- In your code:
  - Mark lines in your code that correspond to individual exercises, e.g.
    // assignment 3.2
  - Help us understand important parts of your code by writing comments (short and concise)

Outline

- **Reminder – Goal of this course:** Development of an interactive tool for the visualization of 3D/4D scalar and vector fields with C++ and DirectX
- Structure
  - **Introduction into Direct3D 11 and shader programming**
  - Volume Rendering
  - Flow Visualization
- Not included!
  - C++
  - Computer graphics fundamentals

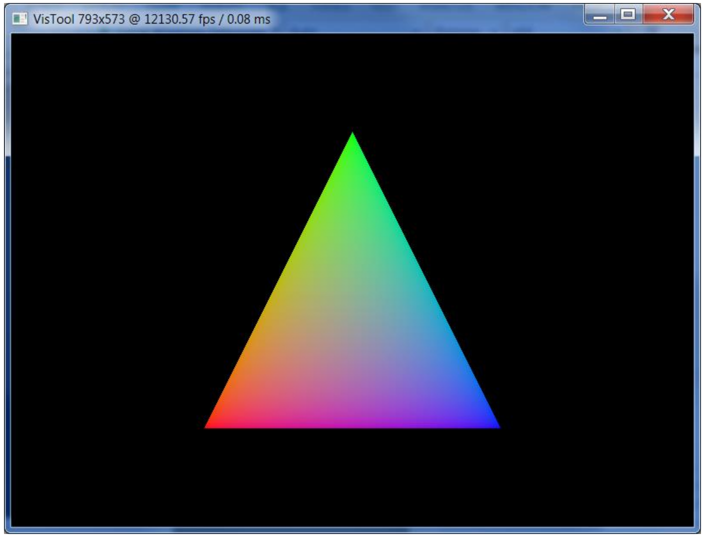


Overview - Today

- Organizational Issues
- **Template Project & Assignment 1**
- DXUT
- Effects & HLSL
- Transformations

Assignment 1

- What you get from us:

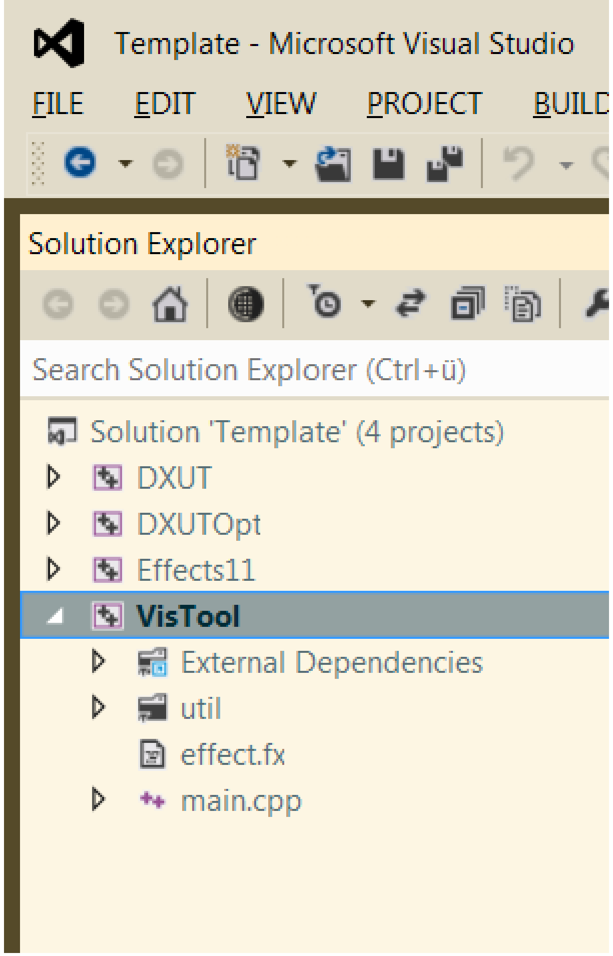


- Task: Add a user-controlled 3D camera and render a bounding box
  - Not that much code to write, but much to read, understand and familiarize with
  - Modify and extend given code, try to understand as much as possible

Template Project

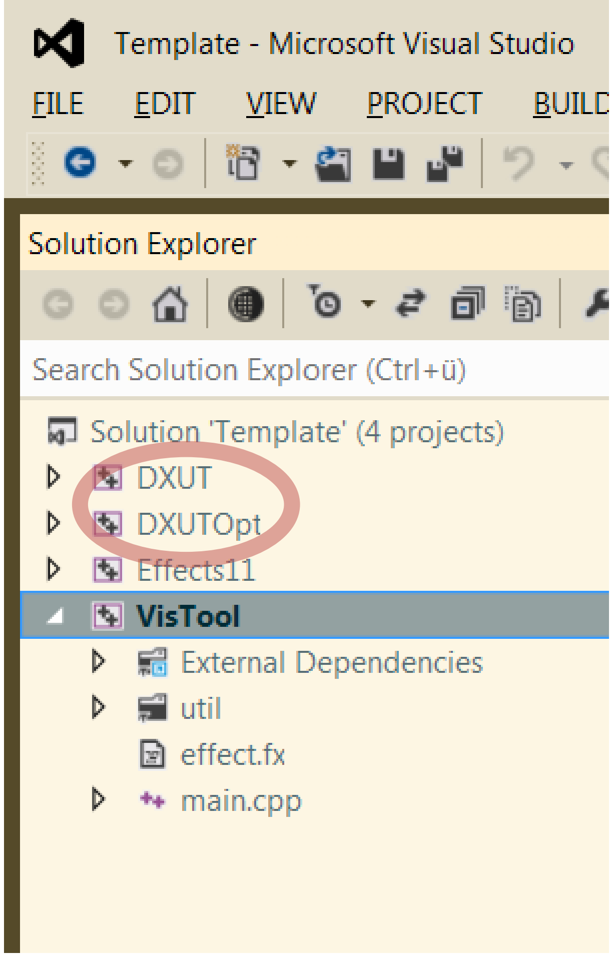
- Minimal starting point ++
  - Based on DirectX Empty Project sample
  - Everything you need to get started immediately (e.g. a working effect file)
  - Runs out-of-the-box ☺

- VS Solution contains four projects
  - DirectX Utility Library (DXUT)
  - DirectX 11 Effect Framework
  - VisTool (your playground!)



Template Project

- Minimal starting point ++
  - Based on DirectX Empty Project sample
  - Everything you need to get started immediately (e.g. a working effect file)
  - Runs out-of-the-box ☺
- VS Solution contains four projects
  - DirectX Utility Library (DXUT)
  - DirectX 11 Effect Framework
  - VisTool (your playground!)



Template - Microsoft Visual Studio

Solution Explorer

Search Solution Explorer (Ctrl+ü)

- Solution 'Template' (4 projects)
  - DXUT
  - DXUTOpt
  - Effects11
  - VisTool
    - External Dependencies
    - util
    - effect.fx
    - main.cpp

DXUT

- The DirectX Utility Library (DXUT) simplifies the usage of the Windows and D3D APIs
- DXUT helps with:
  - Window creation
  - Direct3D device creation
  - Main message and render loop
  - Handling of device and windows events (e.g. user input)
  - ...
- Additional features:
  - Camera-Classes
    - `CFirstPersonCamera` („First Person“)
    - `CModelViewerCamera` („Third Person“)
  - Simple graphical user interface (which we won‘t use, it‘s bad...)
  - Text rendering
  - ...

DXUT: Main method

- Simplified main() from template project:

```
int WINAPI wWinMain(…) {
      // Set DXUT callbacks
      DXUTSetCallbackKeyboard( OnKeyboard );
      DXUTSetCallbackMouse   ( OnMouse );
      DXUTSetCallbackMsgProc ( MsgProc );

      DXUTSetCallbackD3D11DeviceCreated     ( OnD3D11CreateDevice );
      DXUTSetCallbackD3D11SwapChainResized  ( OnD3D11ResizedSwapChain );
      DXUTSetCallbackD3D11SwapChainReleasing( OnD3D11ReleasingSwapChain );
      DXUTSetCallbackD3D11DeviceDestroyed   ( OnD3D11DestroyDevice );

      DXUTSetCallbackFrameMove( OnFrameMove );
      DXUTSetCallbackD3D11FrameRender( OnD3D11FrameRender );

      //Application initialization
      DXUTInit( true, true, NULL );
      DXUTCreateWindow( L"VisTool" );
      DXUTCreateDevice( D3D_FEATURE_LEVEL_11_0, true, 640, 480 );

      // Enter into the DXUT render loop
      DXUTMainLoop();

      //Application deinitialization
      DXUTShutdown();
      return DXUTGetExitCode();
}
```

DXUT

- Interaction with Direct3D and Windows is controlled via callback functions:
  - `DXUTSetCallbackD3D11DeviceCreated`:

    Create (device->CreateX) and initialize everything which does **not** depend on the window size.
  - `DXUTSetCallbackD3D11SwapChainResized`:

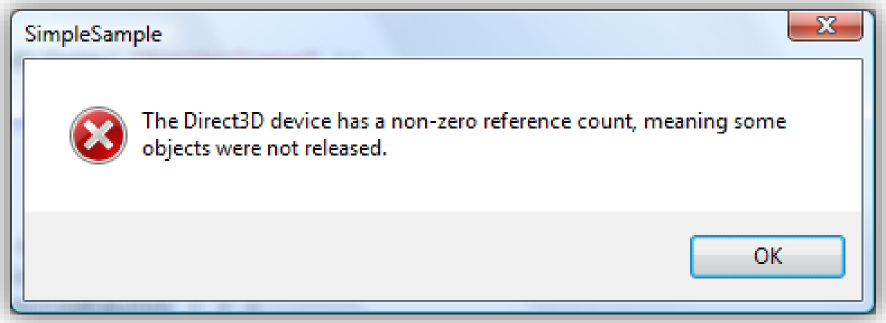
    Create and initialize everything which depends on the window size.
  - `DXUTSetCallbackD3D11SwapChainReleasing`:

    Release everything which was *Created* in SwapChainResized.
  - `DXUTSetCallbackD3D11DeviceDestroyed`:

    Release everything which was *Created* in DeviceCreated.
- Be careful with device->*CreateX* / SAFE_*RELEASE(…)*
  - *We hate GPU memory leaks! (… CPU leaks too)*

GPU memory management (DXUT)

- Also on the GPU all allocated memory must also be freed.
- With DXUT this is done via `SAFE_RELEASE (`*`Pointer`*`)`.
- In debug builds the following message is displayed if resources are not yet released when the program is closed:



SimpleSample

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

- We NEVER wanna see this message
  - Whenever your write device->CreateX you should immediately also write SAFE_RELEASE(x)

DXUT

- Callback-functions continued:
  - `DXUTSetCallbackD3D11FrameRender`:

    Render your scene (context->Draw). Unless you control all device state yourself, also place all your context->SetX calls here.
  - `DXUTSetCallbackFrameMove`:

    Called **before** the rendering. Update your scene here.
  - `DXUTSetCallbackMsgProc`:

    Handle window messages (e.g. mouse/UI events).
  - `DXUTSetCallbackKeyboard`:

    Handle keyboard events
  - `DXUTSetCallbackDeviceChanging`:

    Called before CreateDevice. Can change device/frame-buffer options (e.g. anti-aliasing mode, framebuffer format, etc.).

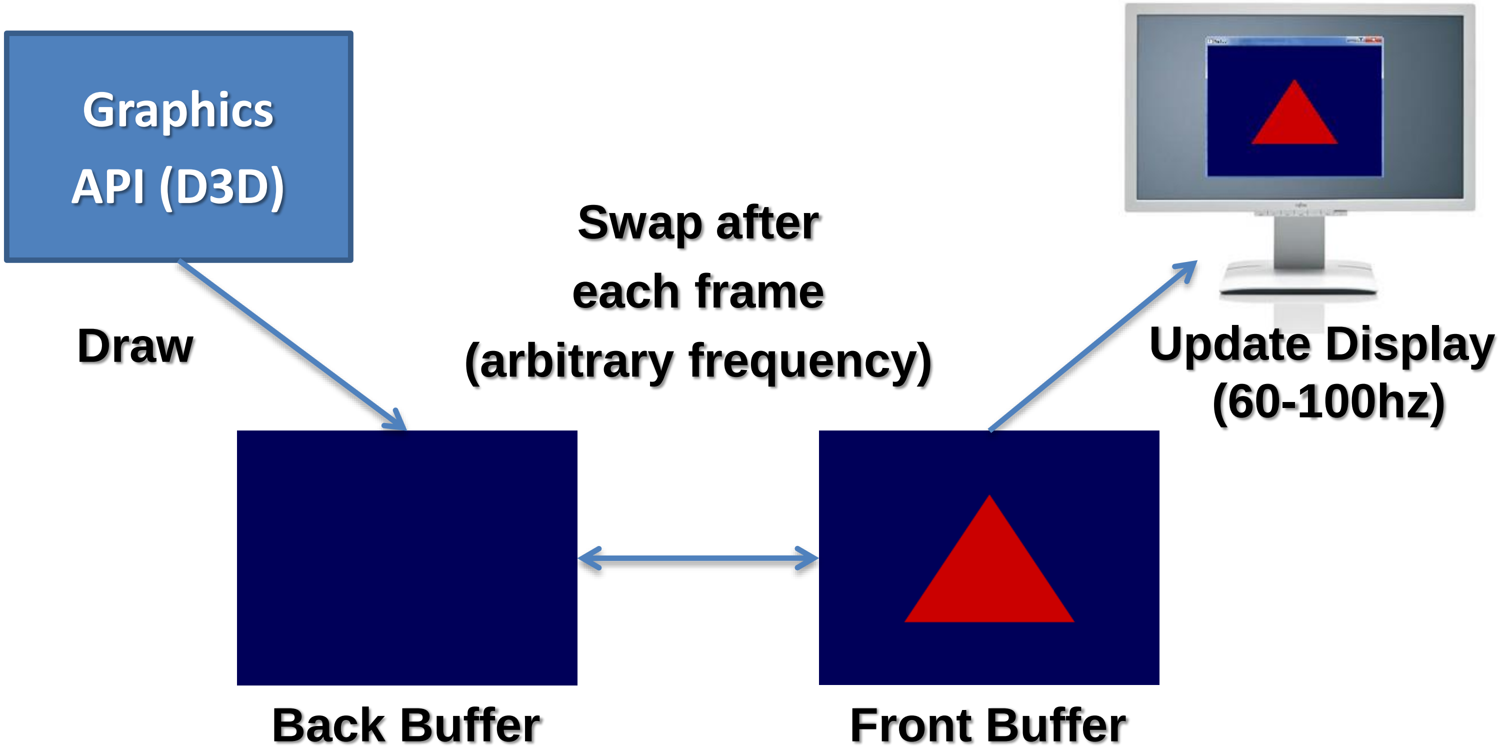
Main Loop

- `DXUTMainLoop()` roughly does the following:
  (a typical event loop for interactive applications)

```
while( WM_QUIT != msg.message ) //Until application is closed
{
    if( GotMsg(msg) ) {
        //Try to forward msg to application defined msg callbacks
        if (!CallbackKeyboard(msg) &&
            !CallbackMouse   (msg) &&
            !CallbackMsgProc (msg))
        {
            DXUTHandleMsg(msg); //default message handler
        }
    } else {
        //Move and render, then update swap chain
        CallbackFrameMove  (...);
        CallbackFrameRender(...);
        SwapChain->Present (...); //Swap back and front buffer
    }
}
```

Swap Chain

- Use (at least) two different textures for rendering and displaying
  - Front buffer: Texture that is currently displayed on the screen
  - Back buffer: Texture everything is currently drawn into



Graphics API (D3D)

Draw

Swap after each frame (arbitrary frequency)

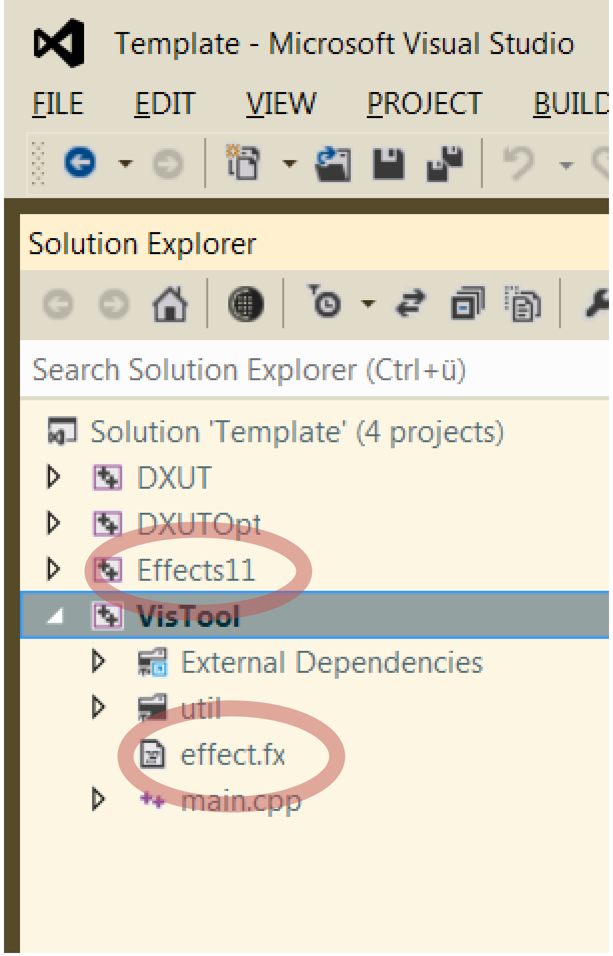
Update Display (60-100hz)

Back Buffer

Front Buffer

Template Project

- Minimal starting point ++
  - Based on DirectX Empty Project sample
  - Everything you need to get started immediately (e.g. a working effect file)
  - Runs out-of-the-box ☺
- VS Solution contains four projects
  - DirectX Utility Library (DXUT)
  - DirectX 11 Effect Framework
  - VisTool (your playground!)

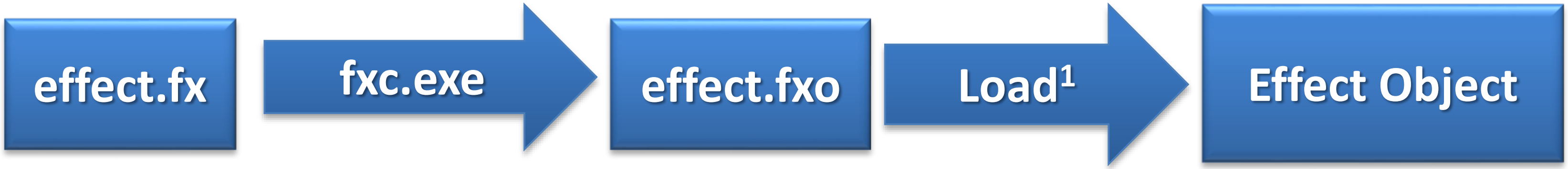


Template - Microsoft Visual Studio
FILE
EDIT
VIEW
PROJECT
BUILD
Solution Explorer
Search Solution Explorer (Ctrl+ü)

Solution 'Template' (4 projects)
- DXUT
- DXUTOpt
- Effects11
- VisTool
  - External Dependencies
  - util
  - effect.fx
  - main.cpp

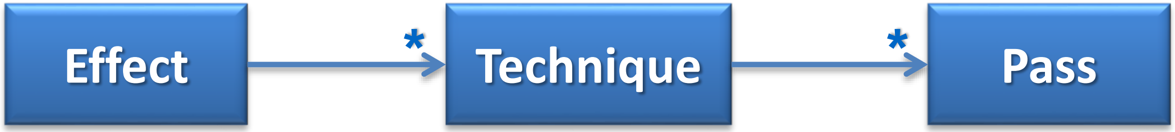
Rendering Effects

- In Direct3D11 we can use the Effect Framework to specify most of the states of the graphics pipeline in a ***Rendering Effect***
- Each rendering effect is stored in a separate effect source code file (.fx)
- The HLSL compiler (fxc.exe) converts an effect source code file into a compiled effect file (.fxo)
- The function `D3DX11CreateEffectFromFile`[1] creates an `ID3DX11Effect` object from a compiled effect in main memory



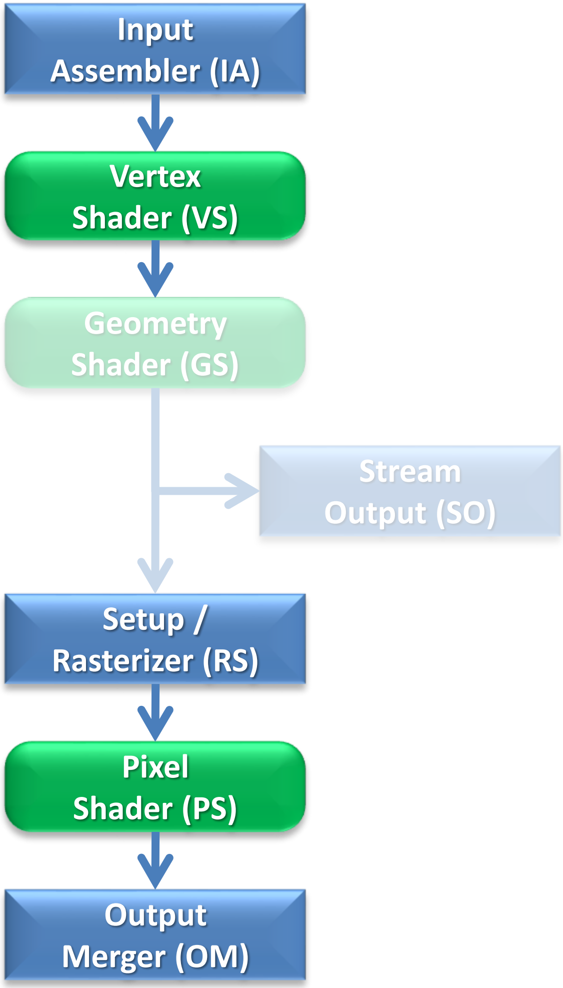
Rendering Effects

- A rendering effect can contain multiple ***Effect Techniques***
  - The effect member function `GetTechniqueByName` retrieves a handle to an effect technique
- An effect technique can contain multiple ***Render Passes***
  - 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 can be defined inside the effect file using HLSL



Reminder: Graphics Pipeline

- Here: cutout of the D3D11 pipeline (which is basically the D3D10 pipeline)
- The pipeline consists of:
  - Programmable stages
  - Fixed-function stages
- GS + SO are optional (and not important for now)



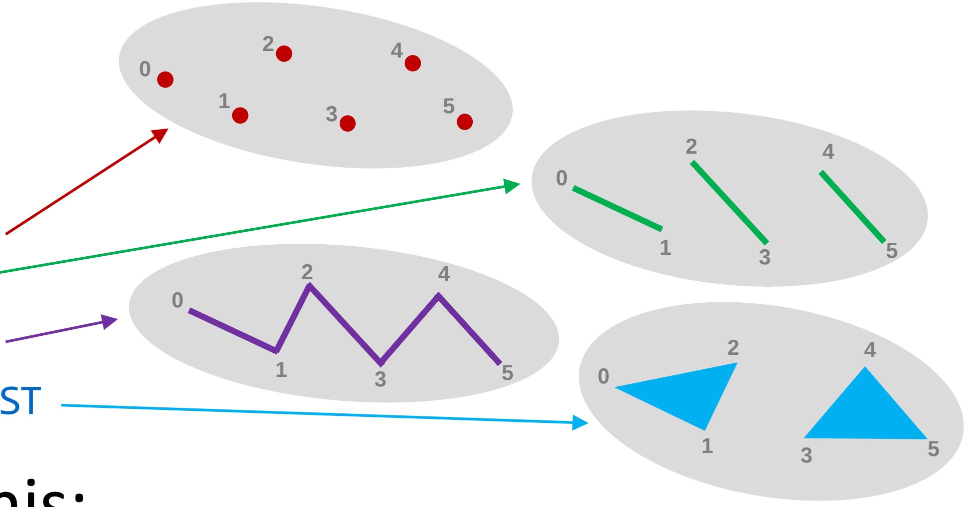
Execute the Rendering (Draw)

- After the pipeline state is completely set, we „draw“ our data, i.e. we input vertices into the pipeline
- In `OnD3D11FrameRender()`, e.g.

```
// Setup input
pd3dImmediateContext->IASetIndexBuffer(...);
pd3dImmediateContext->IASetVertexBuffers(...);
pd3dImmediateContext->IASetInputLayout(...);
pd3dImmediateContext->IASetPrimitiveTopology(...);

// Setup pipeline by applying pass
g_MyPass->Apply(0, pd3dImmediateContext);

// Draw n vertices starting with ID 0
pd3dImmediateContext->Draw(n,0);
```

Primitive Topology

- Vertices in (pre-rasterizer) vertex stream are associated with a primitive topology
- Tells the rasterizer which kind of geometric primitive the stream describes
- Examples:
  - `D3D11_PRIMITIVE_TOPOLOGY_POINTLIST`
  - `D3D11_PRIMITIVE_TOPOLOGY_LINELIST`
  - `D3D11_PRIMITIVE_TOPOLOGY_LINESTRIP`
  - `D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST`



- We set the topology like this:

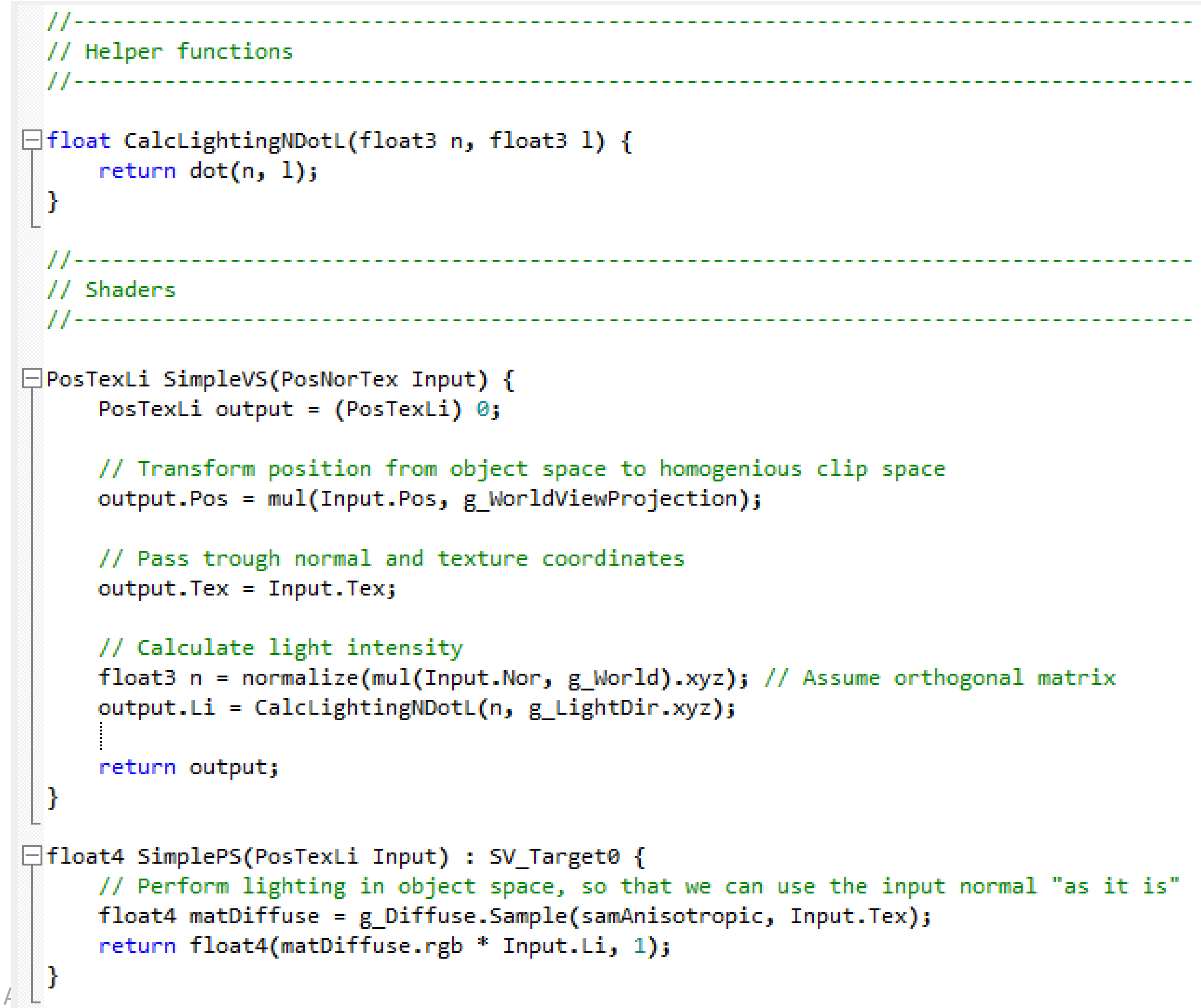
```
pd3dImmediateContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_LINESTRIP);
```

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 Direct3D versions correspond to major HLSL versions
  (Direct3D 10 -> HLSL 4, Direct3D 11 -> HLSL 5)

Shaders

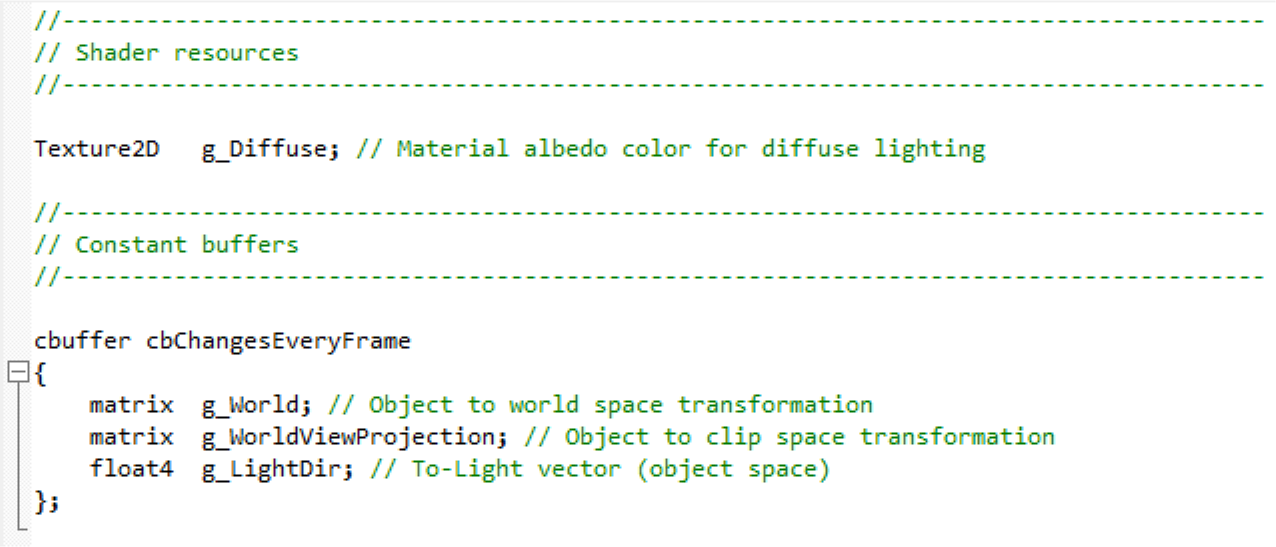
- In HLSL, shaders are defined very similar to functions



```
//--------------------------------------------------------------------------------------
// 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);
}
```

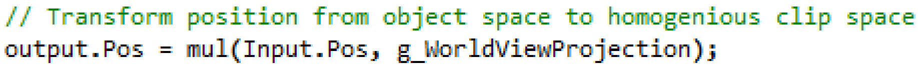
Effect variables

- 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)
};
```

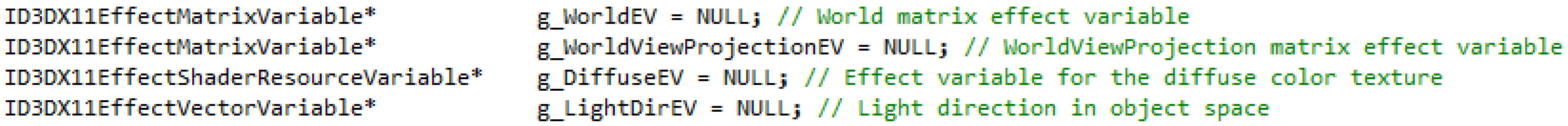
- In the shaders, both types can be accessed like global variables though



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

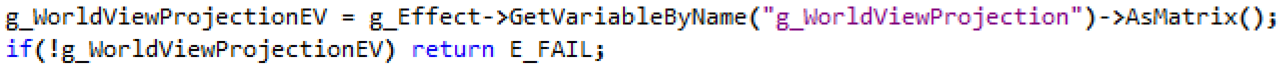
Effect variables

- In the .cpp file on C++ side, declare effect variables like this:



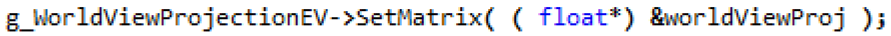
```
ID3DX11EffectMatrixVariable*            g_WorldEV = NULL; // World matrix effect variable
ID3DX11EffectMatrixVariable*            g_WorldViewProjectionEV = NULL; // WorldViewProjection matrix effect variable
ID3DX11EffectShaderResourceVariable*    g_DiffuseEV = NULL; // Effect variable for the diffuse 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



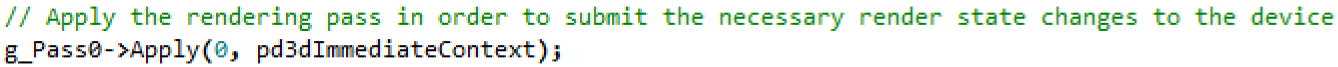
```
g_WorldViewProjectionEV = g_Effect->GetVariableByName("g_WorldViewProjection")->AsMatrix();
if(!g_WorldViewProjectionEV) return E_FAIL;
```

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



```
g_WorldViewProjectionEV->SetMatrix( ( float*) &worldViewProj );
```

- The values on the GPU are updated **when the rendering pass is applied** (Apply() should always be called immediately before a draw call!)



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

Overview - Today

- Organizational Issues
- Template Project & Assignment 1
- DXUT
- Effects & HLSL
- **Transformations**

Transformation Pipeline

- We use a lot of different coordinates systems in computer graphics:

vertex shader

(Local) Object coordinates

↓ World Transformation

(Global) World coordinates

↓ View Transformation

View/Camera coordinates

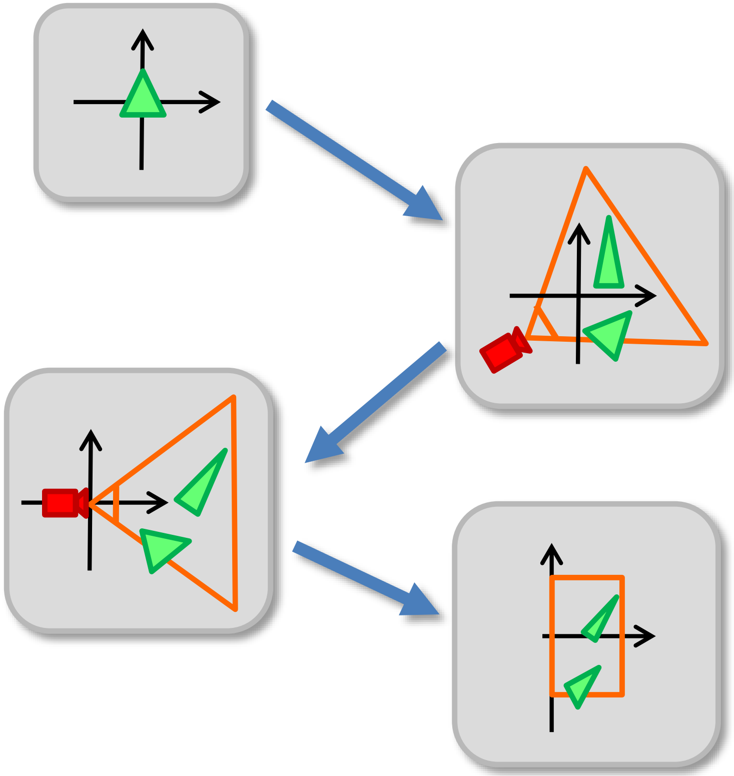
↓ Projection Transformation

Clip coordinates ($w \in R \backslash \{0\}$)

rasterizer

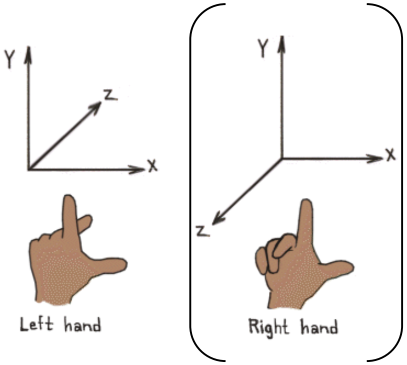
Normalized device coordinates ($w = 1$)

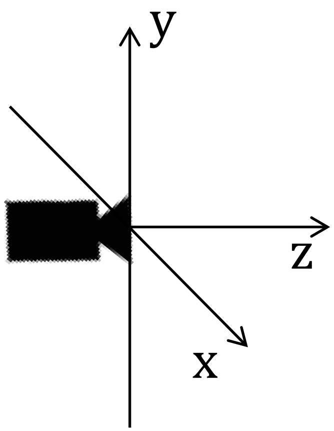
Pixel coordinates



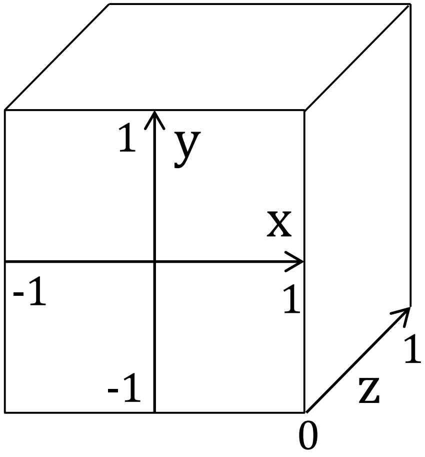
D3D Space Conventions

- View space
  - Left-handed coordinate system (in our case)
  - Camera at origin, looks into $+z$ direction
  - $+x$ is right, $+y$ is top





- NDC: Normalized Device Coordinates (= clip coordinates after perspective division)
  - $x \in [-1; 1]$ ↔ screen from left to right
  - $y \in [-1; 1]$ ↔ screen from bottom to top
  - $z \in [0; 1]$ ↔ depth from near to far



Transformations in D3D

- In D3D, points and vectors are represented as row-vectors in homogenous coordinates:
  - Point (e.g. position): (x, y, z, 1)
  - Vector (e.g. normal): (x, y, z, 0)
- Transformations are written as 4x4-matrices:

Linear transformation (rotate / scale)

Projective Part

$$\begin{bmatrix} x & y & z & w \end{bmatrix} \begin{bmatrix} L_{11} & L_{21} & L_{31} & 0 \\ L_{12} & L_{22} & L_{32} & 0 \\ L_{13} & L_{23} & L_{33} & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix} = \begin{bmatrix} x' & y' & z' & w \end{bmatrix}$$

Translation

$$\begin{bmatrix} x & y & z & w \end{bmatrix} \cdot \underbrace{\begin{bmatrix} World \\ Transform \end{bmatrix} \cdot \begin{bmatrix} View \\ Transform \end{bmatrix} \cdot \begin{bmatrix} Projection \\ Transform \end{bmatrix}}_{\text{Results in a single matrix}} = \begin{bmatrix} x' & y' & z' & w' \end{bmatrix}$$

Transformations in D3D

- Caution: In D3D we perform calculations in a „transposed world“
  - Remember linear transformations from Linear Algebra?
$$p' = M \cdot p$$
  - Transposing yields
$$p'^T = \mathrm{p}^\mathrm{T} \cdot M^T$$
    Same effect on p!
  - In the „transposed world“, writing order corresponds to the order of transformations:
$$p' = M_{proj} \cdot M_{view} \cdot M_{world} \cdot p$$
vs.
$$p'^T = p^T \cdot M_{world}^T \cdot M_{view}^T \cdot M_{proj}^T$$

Homogeneous Coordinates

- Homogenization and dehomogenization
  - $h_1 : (x, y, z) \rightarrow (x, y, z, 1)$
  - $h_0 : (x, y, z) \rightarrow (x, y, z, 0)$
  - $d_1 : (x, y, z, w) \rightarrow \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right)$
  - $d_0 : (x, y, z, w) \rightarrow normalize((x, y, z))$
- Transformation recipes for $M \in R^{4\times4}$
  - Points $p \in R^3$

$$\mathrm{p}' = \mathrm{d}_1(\mathrm{h}_1(p) \cdot M)$$

  - Normals/directions $n \in R^3$ with $\|n\| = 1$

$$n' = d_0(h_0(n) \cdot (M^{-1})^T)$$

(only works if M is an affine transformation, i.e. has no projective part)

Transformation Examples

- C++/D3D Example

```
XMVECTOR p = ...;

XMMATRIX scale = XMMatrixScaling(2,2,2);
XMMATRIX trans = XMMatrixTranslation(1,2,3);
XMMATRIX M = scale * trans; // scale first, then trans
p = XMVector3TransformCoord(p, M); // apply M
```

- HLSL Example

```
void MyVertexShader (in  float3 pos    : POSITION,
                     out float4 svPos : SV_Position)
{
    // Rasterizer expects clip coordinates,
    // no manual dehomogenization!!!
    svPos = mul(float4(pos,1), g_WorldViewProj);
}
```

Support

- Assignments and slides are not self-contained
- See references: docs / samples (next slide)
- **Seriously, you will need them!**
- Search the web

- If you‘re stuck, ask us:
  - Email: ferstlf@in.tum.de
    treib@tum.de
  - Come to our office:
    02.13.056 / 02.13.061

References / Docs

- **C++ / Windows API References**
  - http://www.cplusplus.com/reference/
  - http://msdn.microsoft.com/library
  - **Much faster:** In Visual Studio place cursor at keyword and press F1
- **DirectX / HLSL / DirectXMath Documentation**
  - http://msdn.microsoft.com/en-us/library/ee663274%28v=vs.85%29.aspx
- **DirectX Sample Projects**
  - http://code.msdn.microsoft.com
  - Many DirectX Samples based on DXUT (e.g. try “Effects 11 Samples”, “DXUT Tutorial Win32 Sample”, “Basic DXUT Win32 Samples”)
  - Caution: Not all DirectX samples use the Effect Framework!

Q&A

Questions ?

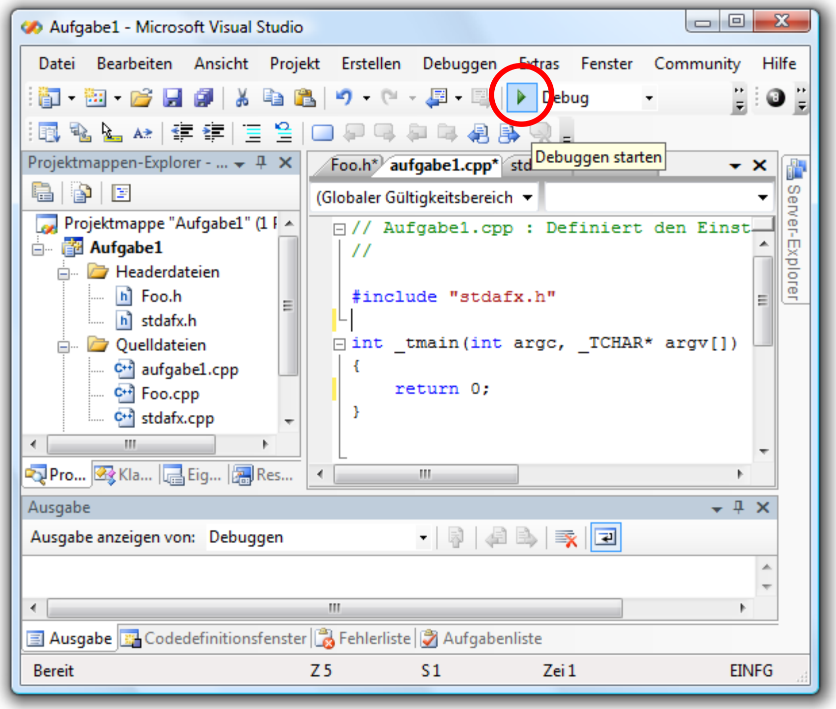
(the remaining slides are for self-study)

Appendix

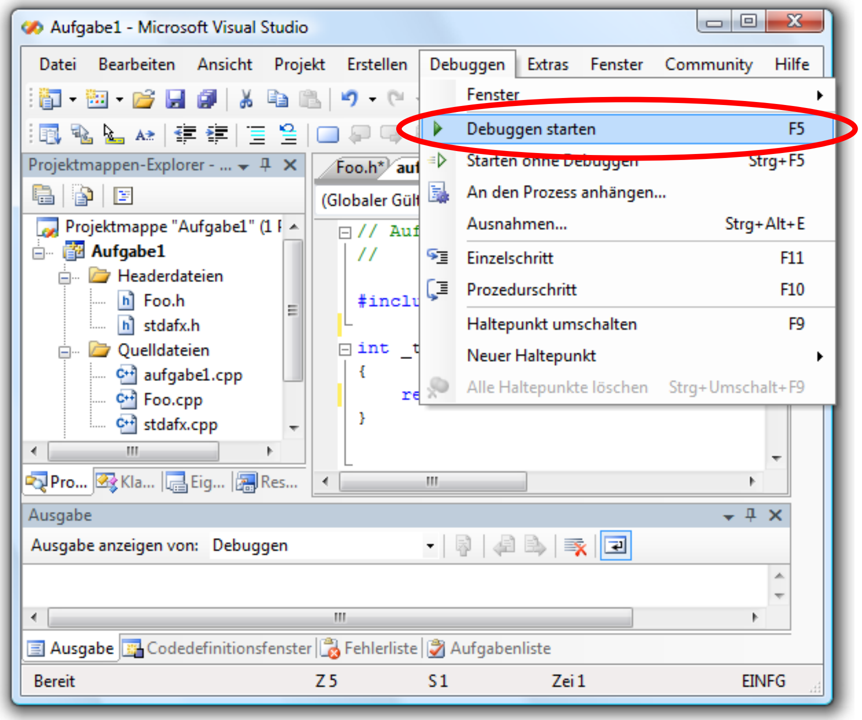
- **Visual Studio Tips & Tricks**
- Overview: Graphics Pipeline

Debugging in Visual Studio

- Start your program in Debug mode

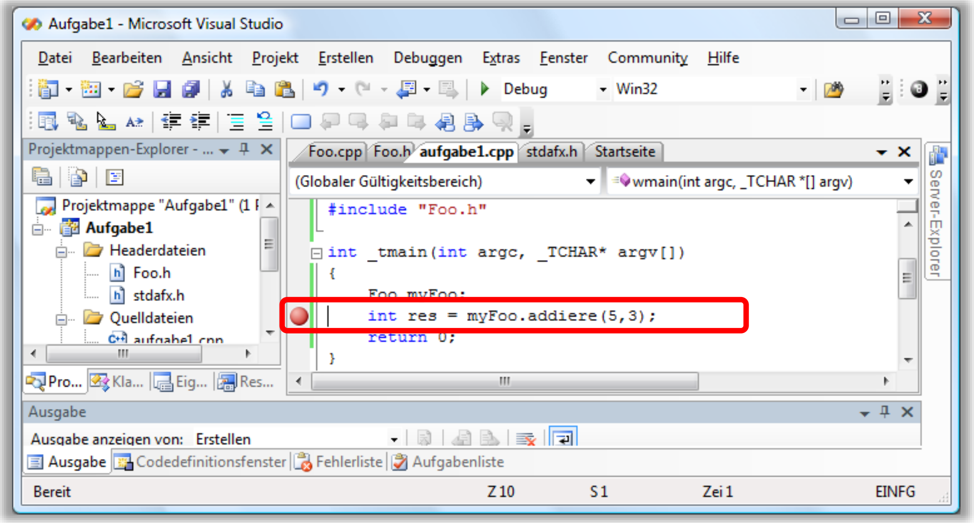

Aufgabe1 - Microsoft Visual Studio
Datei Bearbeiten Ansicht Projekt Erstellen Debuggen Extras Fenster Community Hilfe
Debug
Debuggen starten
Projektmappen-Explorer - ...
Projektmappe "Aufgabe1" (1 P
Aufgabe1
Headerdateien
Foo.h
stdafx.h
Quelldateien
aufgabe1.cpp
Foo.cpp
stdafx.cpp
Foo.h* aufgabe1.cpp* std
(Globaler Gültigkeitsbereich
Server-Explorer
Pro... Kla... Eig... Res...
Ausgabe
Ausgabe anzeigen von: Debuggen
Ausgabe Codedefinitionsfenster Fehlerliste Aufgabenliste
Bereit Z 5 S 1 Zei 1 EINFG

```
// Aufgabe1.cpp : Definiert den Einst
//

#include "stdafx.h"

int _tmain(int argc, _TCHAR* argv[])
{
    return 0;
}
```


Aufgabe1 - Microsoft Visual Studio
Datei Bearbeiten Ansicht Projekt Erstellen Debuggen Extras Fenster Community Hilfe
Fenster
Debuggen starten F5
Starten ohne Debuggen Strg+F5
An den Prozess anhängen...
Ausnahmen... Strg+Alt+E
Einzelschritt F11
Prozedurschritt F10
Haltepunkt umschalten F9
Neuer Haltepunkt
Alle Haltepunkte löschen Strg+Umschalt+F9
Projektmappen-Explorer - ...
Projektmappe "Aufgabe1" (1 P
Aufgabe1
Headerdateien
Foo.h
stdafx.h
Quelldateien
aufgabe1.cpp
Foo.cpp
stdafx.cpp
Pro... Kla... Eig... Res...
Ausgabe
Ausgabe anzeigen von: Debuggen
Ausgabe Codedefinitionsfenster Fehlerliste Aufgabenliste
Bereit Z 5 S 1 Zei 1 EINFG

Debugging: Breakpoints

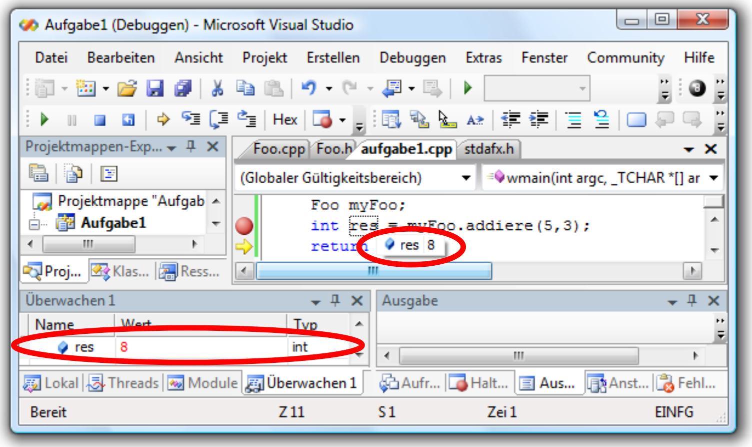
- At breakpoints the program is paused **right before** the execution of the marked line
- Breakpoints can be created through
  - context menu (right-click)
  - grey bar left of the source code (left-click)


Aufgabe1 - Microsoft Visual Studio
Datei Bearbeiten Ansicht Projekt Erstellen Debuggen Extras Fenster Community Hilfe
Debug
Win32
Projektmappen-Explorer - ...
Projektmappe "Aufgabe1" (1 P
Aufgabe1
Headerdateien
Foo.h
stdafx.h
Quelldateien
aufgabe1.cpp
Pro... Kla... Eig... Res...
Foo.cpp Foo.h aufgabe1.cpp stdafx.h Startseite
(Globaler Gültigkeitsbereich)
wmain(int argc, _TCHAR *[] argv)
Server-Explorer
Ausgabe
Ausgabe anzeigen von: Erstellen
Ausgabe Codedefinitionsfenster Fehlerliste Aufgabenliste
Bereit Z 10 S 1 Zei 1 EINFG

```
#include "Foo.h"

int _tmain(int argc, _TCHAR* argv[])
{
    Foo myFoo;
    int res = myFoo.addiere(5,3);
    return 0;
}
```

Debugging: Values of Variables

- If the program is paused, the current value of variables can be inspected:
  a) Hover a variable with the cursor
  b) Add permantent watches through right click menu


Aufgabe1 (Debuggen) - Microsoft Visual Studio
Datei Bearbeiten Ansicht Projekt Erstellen Debuggen Extras Fenster Community Hilfe
Hex
Projektmappen-Exp...
Projektmappe "Aufgab
Aufgabe1
Proj... Klas... Ress...
Foo.cpp Foo.h aufgabe1.cpp stdafx.h
(Globaler Gültigkeitsbereich)
wmain(int argc, _TCHAR *[] ar
Überwachen 1
Ausgabe
Lokal Threads Module Überwachen 1
Aufr... Halt... Aus... Anst... Fehl...
Bereit Z 11 S 1 Zei 1 EINFG

```
Foo myFoo;
int res = myFoo.addiere(5,3);
retur
```

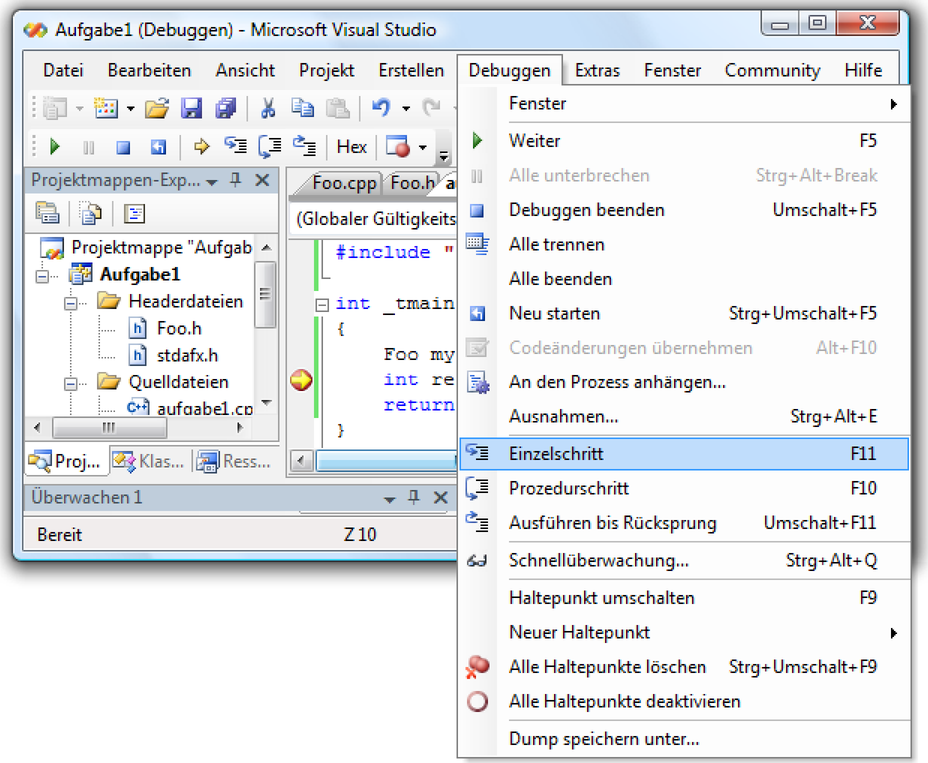
res 8

Name	Wert	Typ
res	8	int

- Visual Studio knows std - try inspecting a `std::vector` or `std::map` (one good reason to use as much std as possible, e.g. `std::vector` instead of raw C++ arrays)

Debugging: Step-by-Step Execution

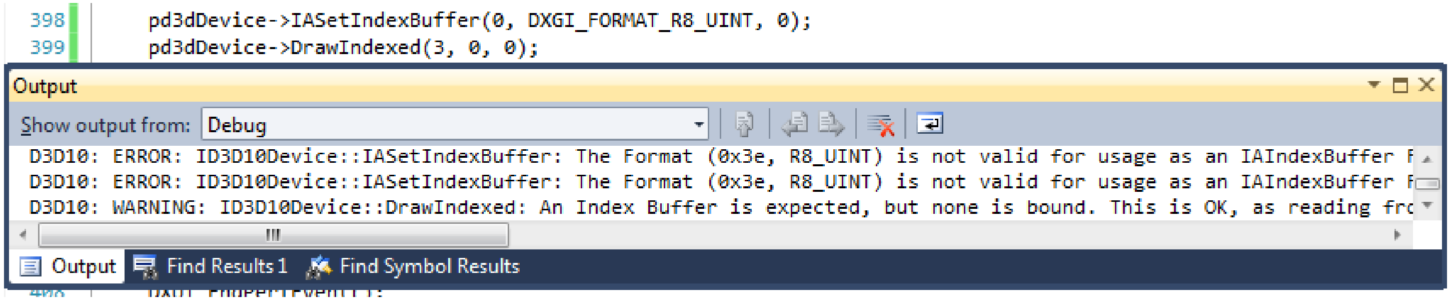
- If the program is paused, the next line to be executed is marked by a yellow arrow
- Through the menu or corresponding hotkeys the program can be executed step-by-step



```
#include "
int _tmain
{
    Foo my
    int re
    return
}
```

Direct3D Debugging Tipps

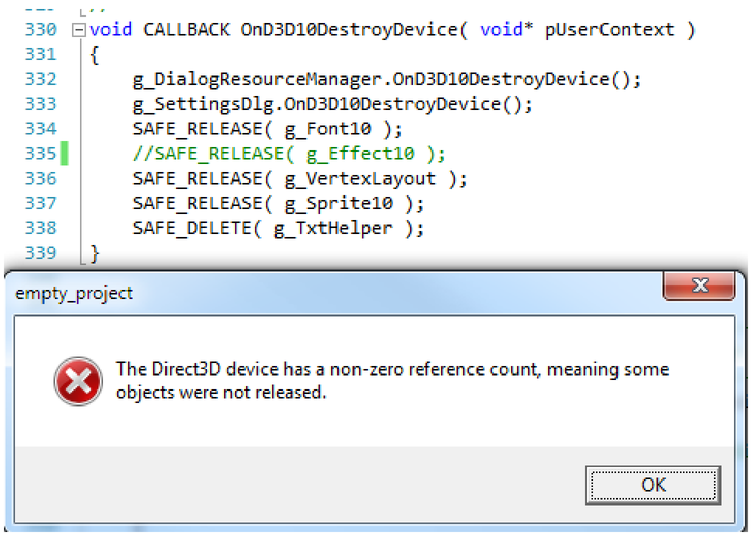
- At Runtime (in debug builds):
  - Direct3D emits warnings and error messages
    - When you see them popping up every frame, your code is almost certainly doing something wrong



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

```
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
```

  - DXUT reports GPU memory leaks when the process terminates



```
void CALLBACK OnD3D10DestroyDevice( void* pUserContext )
{
    g_DialogResourceManager.OnD3D10DestroyDevice();
    g_SettingsDlg.OnD3D10DestroyDevice();
    SAFE_RELEASE( g_Font10 );
    //SAFE_RELEASE( g_Effect10 );
    SAFE_RELEASE( g_VertexLayout );
    SAFE_RELEASE( g_Sprite10 );
    SAFE_DELETE( g_TxtHelper );
}
```

empty_project

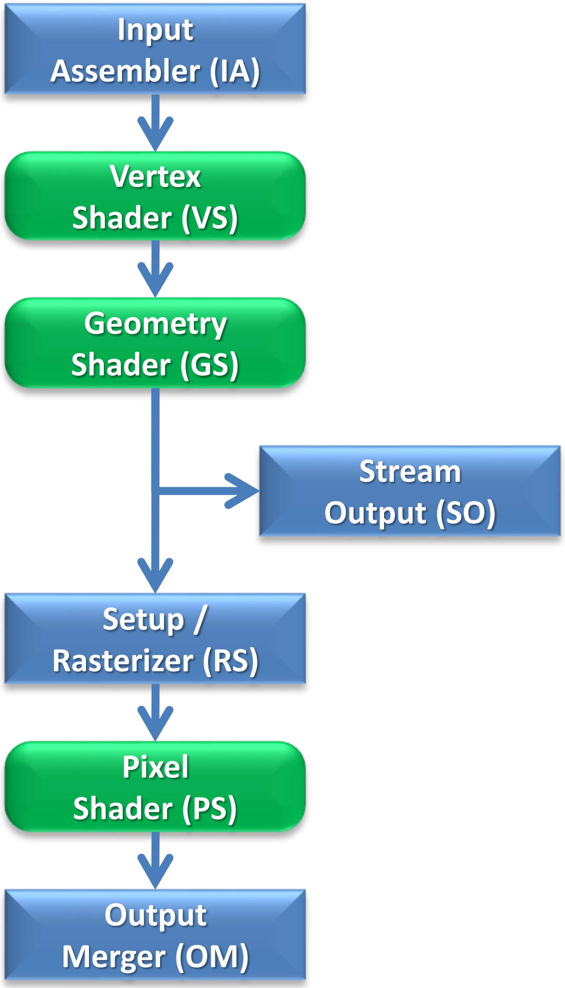
The Direct3D device has a non-zero reference count, meaning some objects were not released.

Appendix

- Visual Studio Tips & Tricks
- **Overview: Graphics Pipeline**

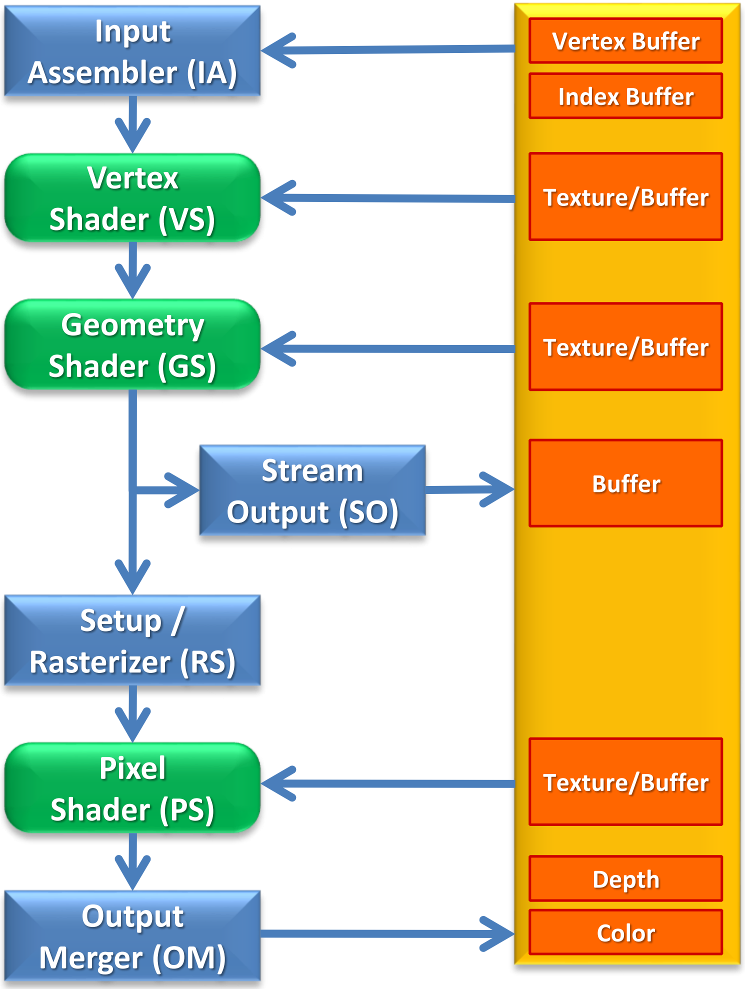
Graphics Pipeline

- Here: cutout of the D3D11 pipeline (which is basically the D3D10 pipeline)
- The pipeline consists of:
  - Programmable stages
    - Vertex Shader
    - Geometry Shader
    - Pixel Shader
  - Fixed-function stages
    - Input Assembler
    - Stream Output
    - Setup / Rasterizer
    - Output Merger



Input Assembler

- Fixed function
- Purpose:
  - Generate vertex data from input
- Input:
  - Vertex Buffers + Index Buffer
- Output:
  - Vertices with attributes
  - VertexID, PrimitiveID, InstanceID
- Controllable through:
  - IASetVertexBuffers/SetIndexBuffer
  - IASetInputLayout
  - IASetPrimitiveTopology



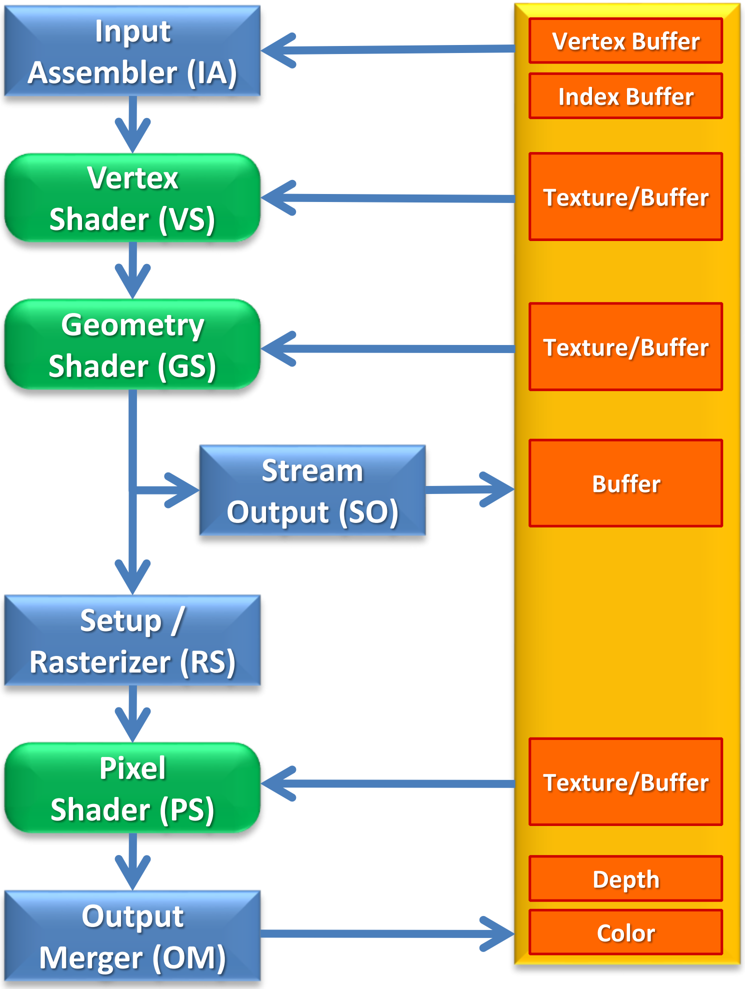
Input Assembler (IA)

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 the *Input Layout* to interpret the data
  - Generates a set of geometric primitives controlled by `D3D11_PRIMITIVE_TOPOLOGY`
  - Supplies the assebled primitives to the rest of the pipeline
- The elemental unit thereby is the edge point (vertex), which can carry various attributes (e.g. position, normal, color, …)
- Additionally it provides system generated values to the pipliene: `SV_VertexID`, `SV_PrimitiveID`, `SV_InstanceID`

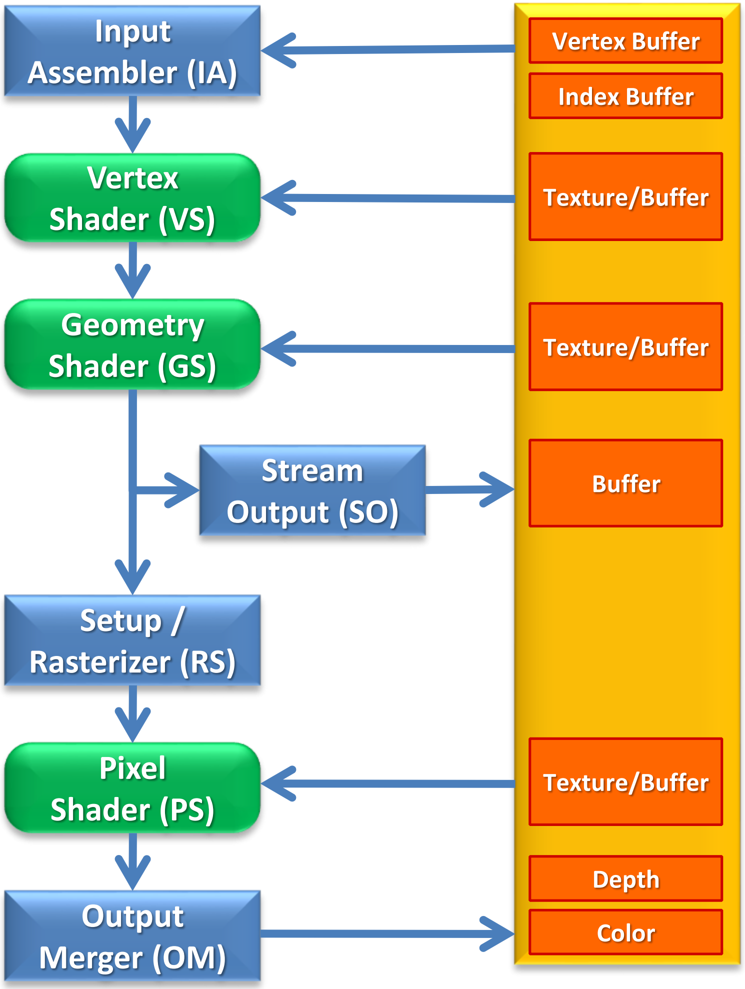
Vertex Shader

- Programmable
- Only necessary calculation:
  - Transformation
    (from object to clip coordinates)
- Input:
  - Vertex
- Output:
  - Vertex
- Read from GPU memory possible



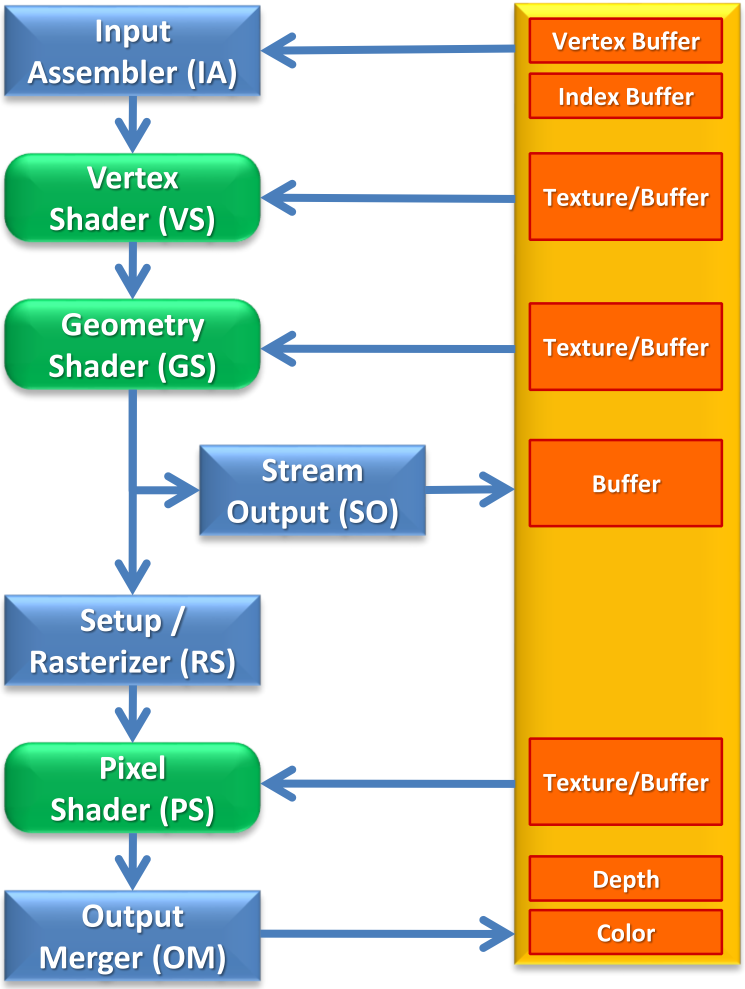
Geometry Shader

- Programmable
- Optional
- Calculations per primitive:
  - Create / Delete primitives
  - Change primitives (per-vertex data)
- Input:
  - 1 primitive
  - Optionally: adjacent primitives
- Output:
  - k primitives
- Read from memory possible
- Write to stream-out possible



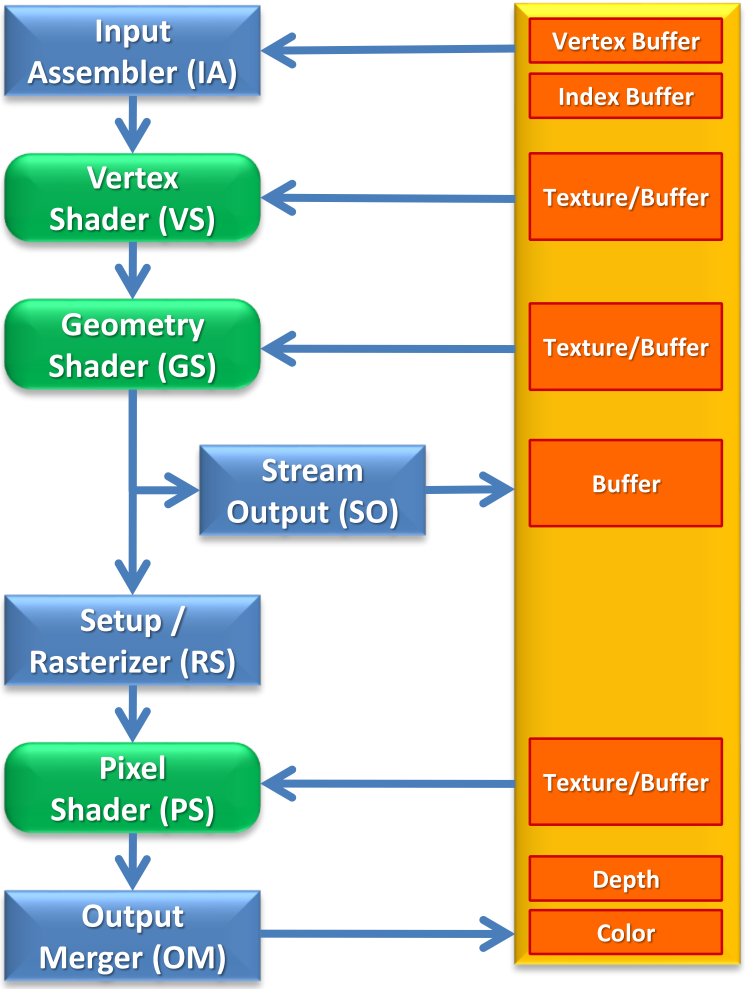
Stream Out

- Fixed function
- Optional
- Task:
  - Redirect primitive output to a buffer
  - Additionally to, or instead of actual rendering
- Controllable through:
  - `SOSetTargets`



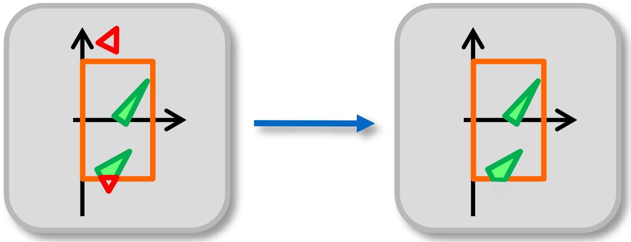
Setup / Rasterizer

- Task:
  - Clipping + Culling
  - Fragment generation
  - Dehomogenization
- Input:
  - 1 primitive
- Output:
  - n fragments
- Controlable trough:
  - RSSetState
  - RSSetViewports/ScissorRects



Clipping + Culling (Setup)

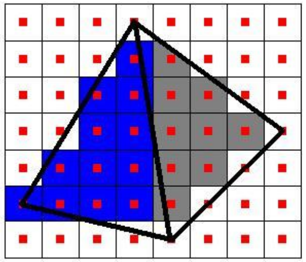
- After the VS-/GS-transformations, all visible content lies within a half-cube
- Everything outside **must not** be rendered (otherwise artifacts are possible) :
  - Discard all primitives which are completely outside („*Frustum Culling*“)
  - Cut all primitives which are partially outside („*Clipping*“)



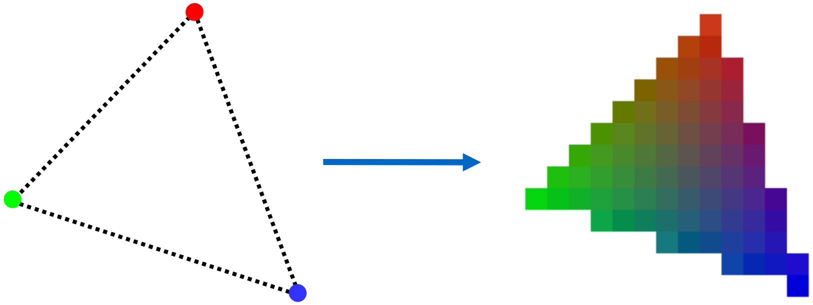
  - Optionally we can also discard primitives which face away from the view („Back face culling“)

Fragment generation (Rasterizer)

- The rasterizer creates one fragment for each covered center point in the pixel raster:

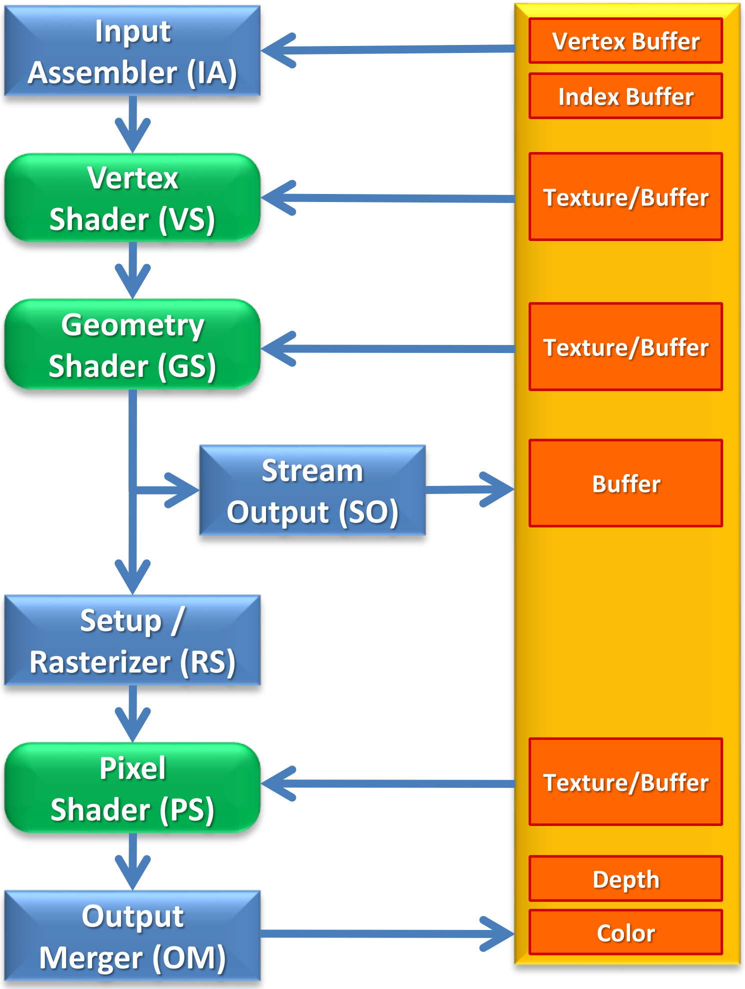


- For each fragment it linearly interpolates the data (texture coordinates, normals, etc.) from the edge vertices (barycentric interpolation):



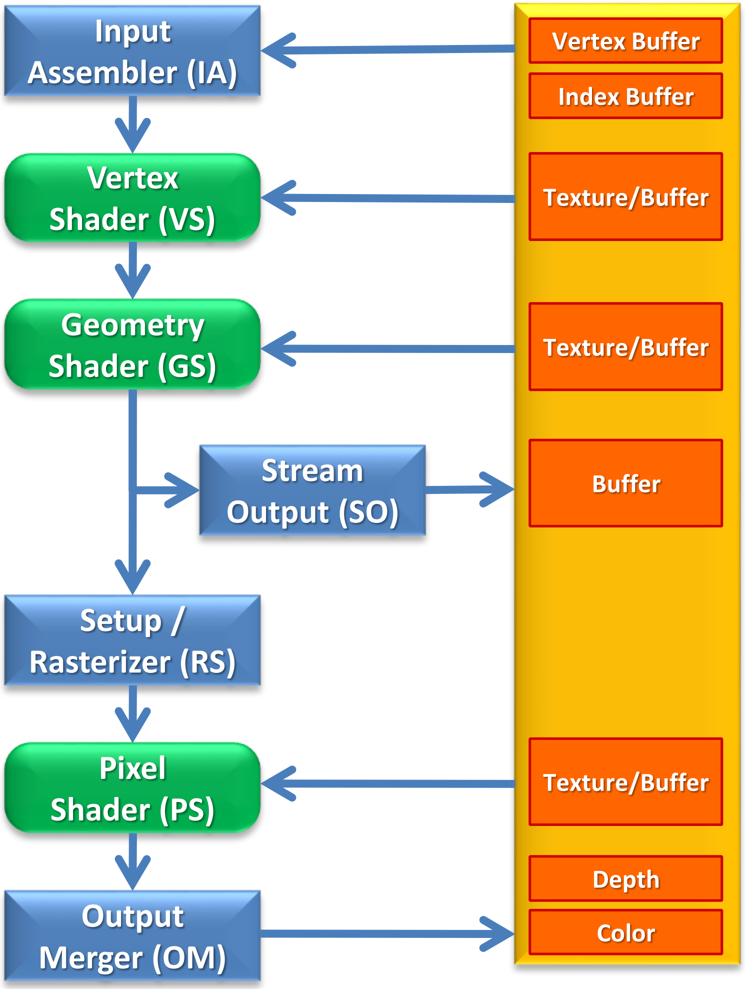
Pixel / Fragment Shader

- Programmable
- Calculations per fragment:
  - Lighting
  - Texturing
  - Simulation of surface effects
- Input:
  - 1 Fragment
    (with interpolated vertex attributes)
- Output:
  - 0 or 1 fragment
- Read from memory possible (textures!)



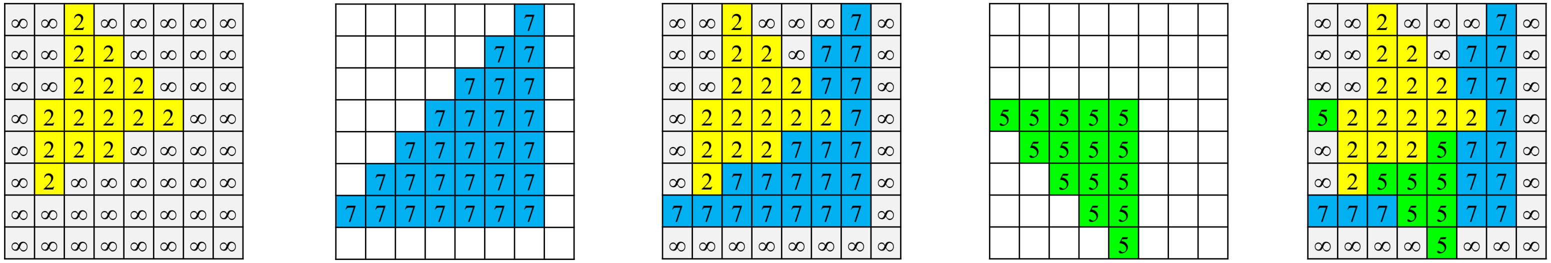
Output Merger

- Fixed function
- Task:
  - Depth- / Stencil tests
  - Color buffer blending
- Input:
  - 1 Fragment
- Output:
  - Possible changes of color (frame-buffer) and depth (depth buffer) values in the rendered image
- Controllable through:
  - `OMSetRenderTargets`
  - `OMSetBlendState`
  - `OMSetDepthStencilState`



Depth test (part of OM)

- Problem:
  - Calculating the correct depth order for the fragments is too expensive
- How do we decide what is visible if primitives are drawn in arbitrary order?
  - During rendering a depth value is stored additionally to the color
  - If a fragment overwrites the values of a pixel is decided in the depth test



∞	∞	2	∞	∞	∞	∞	∞
∞	∞	2	2	∞	∞	∞	∞
∞	∞	2	2	2	∞	∞	∞
∞	2	2	2	2	2	∞	∞
∞	2	2	2	∞	∞	∞	∞
∞	2	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞

Frame Buffer +

						7	
					7	7	
				7	7	7	
			7	7	7	7	
		7	7	7	7	7	
	7	7	7	7	7	7	
7	7	7	7	7	7	7	

New Triangle →

∞	∞	2	∞	∞	∞	7	∞
∞	∞	2	2	∞	7	7	∞
∞	∞	2	2	2	7	7	∞
∞	2	2	2	2	2	7	∞
∞	2	2	2	7	7	7	∞
∞	2	7	7	7	7	7	∞
7	7	7	7	7	7	7	∞
∞	∞	∞	∞	∞	∞	∞	∞

Frame Buffer +

5	5	5	5	5			
	5	5	5	5			
		5	5	5			
			5	5			
				5			

New Triangle →

∞	∞	2	∞	∞	∞	7	∞
∞	∞	2	2	∞	7	7	∞
∞	∞	2	2	2	7	7	∞
5	2	2	2	2	2	7	∞
∞	2	2	2	5	7	7	∞
∞	2	5	5	5	7	7	∞
7	7	7	5	5	7	7	∞
∞	∞	∞	∞	5	∞	∞	∞

Frame Buffer

Note: Color and depth are stored in separate buffers (called frame-buffer and z-buffer)