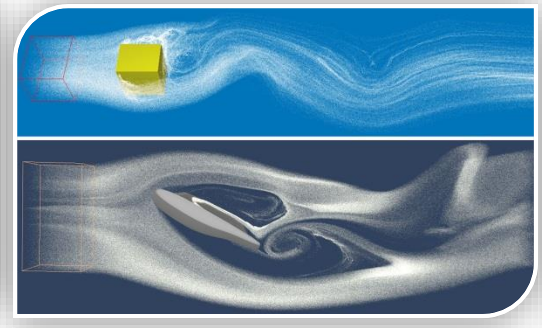
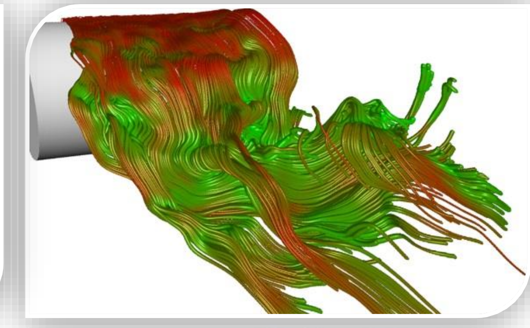
Master Practical Course

Interactive Visual Data Analysis





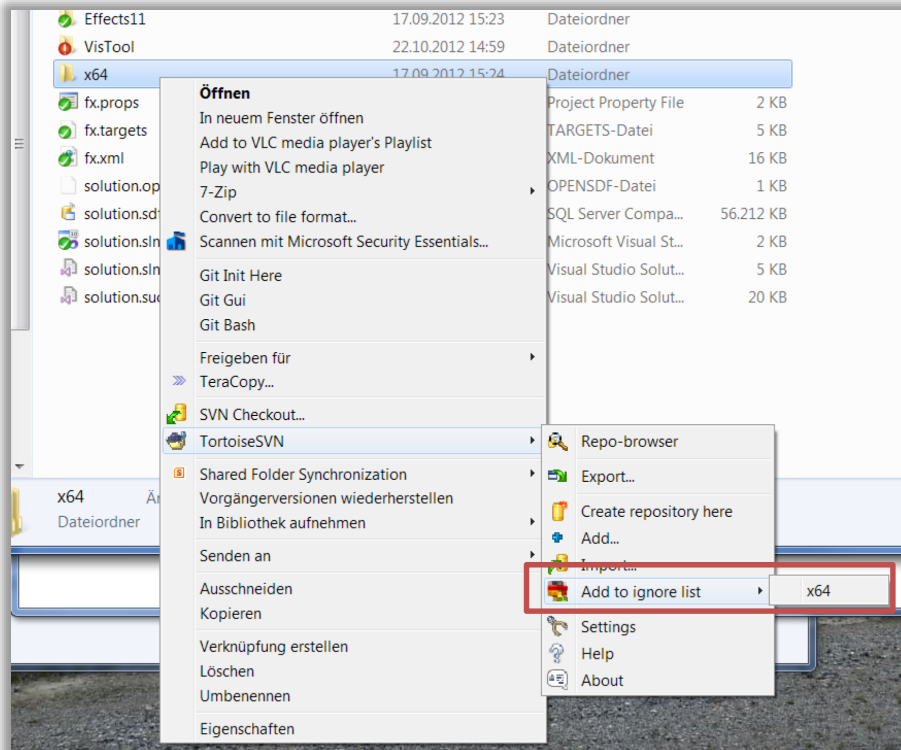
tum.3D

computer graphics & visualization

About last week

tum.3D computer graphics & visualization

- Out-of-the-box ✓
- Keep your repository clean ✓
  - Hint: Use `svn:ignore`



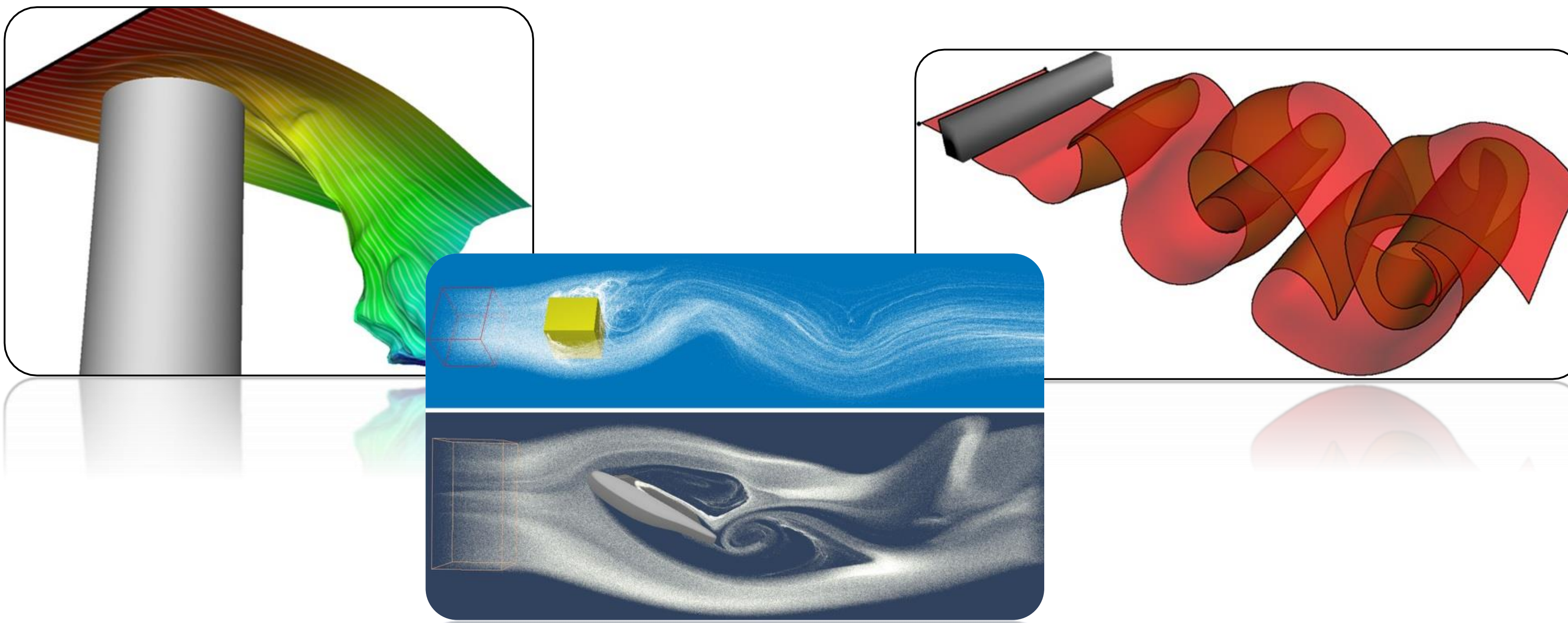
- Group3?
- There will be no minuses this week!

Today

- Assignment 2
  - Load & render a mesh

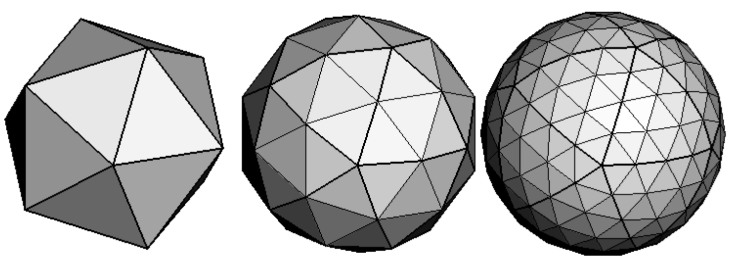
Meshes

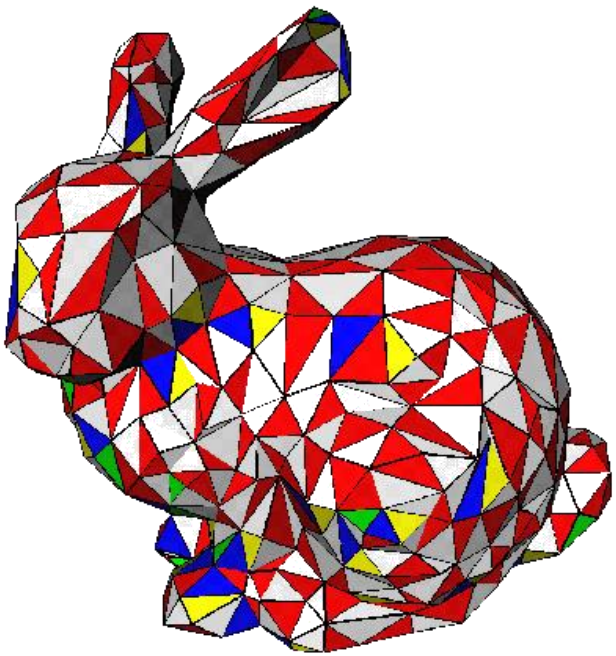
- Volumes and FlowVis – why meshes?
  - Good Direct3D exercise
  - Learn to use the graphics pipeline in the traditional way before „abusing“ it for volume rendering
  - We'll need meshes later for obstacles in FlowVis



Geometry Representation

- A triangular mesh is an approximation of a continuous object by a set of triangles
- Vertices can carry various attributes
  - Position
  - [Normal]
  - [Color]
  - [...]





Scene Description

tum.3D computer graphics & visualization

- `.dat` file contains metadata:

```
MeshFileName:     bunny.ply
SliceThickness:   0.2 0.2 0.15
```

Path to .ply file

BBox size (for now!)

- Will be extended in later assignments!

- `.ply` contains mesh data in PLY format
  - Polygon File Format / Stanford Triangle Format

Mesh Rendering

- Meshes are provided as *.ply files
  - PLY: Polygon File Format / Stanford Triangle Format
- You need to
  - **Parse PLY file & create a CPU vertex / index buffer**
  - Create a GPU vertex / index buffer
  - Upload vertex and index buffer from CPU to GPU
  - Input data from buffers into the pipeline during draw call
  - Stream data through pipeline and shaders (Semantics)
  - Apply Phong lighting model in pixel shader

PLY Example

```
ply
format ascii 1.0
comment VCGLIB generated
element vertex 642
property float x
property float y
property float z
property float nx
property float ny
property float nz
element face 1280
property list uchar int vertex_indices
end_header
0.723607 -0.44722 0.525725 0.723606 -0.447219 0.525728
-0.276388 -0.44722 0.850649 -0.276388 -0.447221 0.850649
[…]
3 0 12 32
3 1 18 46
3 0 32 60
[…]
```

position

normal

list length

vertex indices

header

vertices

triangles

Parsing PLY

- Use RPly
  - RPly: *„ANSI C Library for PLY file format input and output"*
  - Sources are in external ("Add Existing Item" to your VS Project, or simply drag & drop into VS solution explorer)
- Generate
  - Array of vertices (size = numVertices)
    - Use a struct to represent your vertex type in C++, e.g.

```
struct SimpleVertex {
    XMFLOAT3 Pos;      // Position
    XMFLOAT3 Nor;      // Normal
};
```

  - Array of indices (size = 3*numTriangles, 32bit unsigned integers)
    - `#include <cstdint>` → `uint32_t`

Mesh Rendering

- Meshes are provided as *.ply files
  – PLY: Polygon File Format / Stanford Triangle Format
- You need to
  – Parse PLY file & create a CPU vertex / index buffer
  – **Create a GPU vertex / index buffer**
  – **Upload vertex and index buffer from CPU to GPU**
  – Input data from buffers into the pipeline during draw call
  – Stream data through pipeline and shaders (Semantics)
  – Apply Phong lighting model in pixel shader

GPU Vertex & Index Buffer

- Input assembler needs input data in GPU memory
- Therefore we create a ***Vertex Buffer*** and an ***Index Buffer*** on the GPU and fill them with our data
- We need two structures:
  - `D3D11_BUFFER_DESC` describes the buffer
  - `D3D11_SUBRESOURCE_DATA` to deliver the data
- And the function to create the buffers:
  - `ID3D11Device::CreateBuffer`

Vertex Buffer Creation

- Example:
  In `OnCreateDevice()`

```
// Create and fill the description
D3D11_BUFFER_DESC bd = {}; // init to binary zero = default values
bd.Usage = D3D11_USAGE_DEFAULT;
bd.ByteWidth = sizeof(SimpleVertex) * numVertices; // size in bytes
bd.BindFlags = D3D11_BIND_VERTEX_BUFFER;

// Define initial data
D3D11_SUBRESOURCE_DATA initData = {};
initData.pSysMem = pVertices; // pointer to the array

// Create vertex buffer
ID3D11Buffer* pVertexBuffer = nullptr;
V_RETURN(pd3dDevice->CreateBuffer(&bd, &initData, &pVertexBuffer));
```

- Buffers need to be released!
  → `SAFE_RELEASE()` in `OnDestroyDevice()`

Index Buffer Creation

- Example:
  In OnCreateDevice()

```
// Create and fill the description
D3D11_BUFFER_DESC bd = {}; // init to binary zero = default values
bd.Usage = D3D11_USAGE_DEFAULT;
bd.ByteWidth = sizeof(uint32_t) * numTriangles * 3; // size in bytes
bd.BindFlags = D3D11_BIND_INDEX_BUFFER;

// Define initial data
D3D11_SUBRESOURCE_DATA initData = {};
initData.pSysMem = pIndices; // pointer to the array

// Create index buffer
ID3D11Buffer* pIndexBuffer = nullptr;
V_RETURN(pd3dDevice->CreateBuffer(&bd, &initData, &pIndexBuffer));
```

- Buffers need to be released!
  → SAFE_RELEASE() in OnDestroyDevice()

Mesh Rendering

- Meshes are provided as *.ply files
  - PLY: Polygon File Format / Stanford Triangle Format
- You need to
  - Parse PLY file & create a CPU vertex / index buffer
  - Create a GPU vertex / index buffer
  - Upload vertex and index buffer from CPU to GPU
  - **Input data from buffers into the pipeline during draw call**
  - Stream data through pipeline and shaders (Semantics)
  - Apply Phong lighting model in pixel shader

Input Layout

- The GPU gets the raw data in a buffer
- Additionally the ***Input Assembler (IA)*** needs an ***Input Layout*** to interpret the data
- A `D3D11_INPUT_ELEMENT_DESC` thereby describes each vertex attribute
- Must be consistent with the vertex in C++ (struct) and HLSL (vertex shader input) sides!

- D3D11_INPUT_ELEMENT_DESC has the following fields:

Felder	Bedeutung
SemanticName	Name of the element (must equal the semantic name in the shader)
SemanticIndex	Necessary if a semantic name shall be used more than once
Format	Data type of the element (DXGI_FORMAT)
InputSlot	Integer, describes from which input slot the GPU reads the data. In D3D11 the input assembler can read data from multiple vertex buffers at once.
AlignedByteOffset	Offset from the start of the vertex to this element. D3D11_APPEND_ALIGNED_ELEMENT can be used to determine the offset automatically (if the order within *Vertex Layout* and *Input Layout* is the same).
InputSlotClass	D3D11_INPUT_PER_VERTEX_DATA or D3D11_INPUT_PER_INSTANCE_DATA
InstanceDataStepRate	Number of instances which shall use the same data. 0 for D3D11_INPUT_PER_VERTEX_DATA.

Input Layout Creation

- Example:
  In `OnCreateDevice()`

```
auto AAE = D3D11_APPEND_ALIGNED_ELEMENT;
auto IPVD = D3D11_INPUT_PER_VERTEX_DATA;

// Array of descriptions for each vertex attribute
D3D11_INPUT_ELEMENT_DESC layout[] = {
    { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, AAE, IPVD, 0 },
    { "NORMAL"  , 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, AAE, IPVD, 0 },
};
UINT numElements = sizeof(layout) / sizeof(layout[0]);

// Get input signature of pass using this layout
D3DX11_PASS_DESC passDesc;
pTechnique->GetPassByName("MyPass")->GetDesc(&passDesc);

// Create the input layout
V_RETURN(pd3dDevice->CreateInputLayout(layout, numElements,
    passDesc.pIAInputSignature, passDesc.IAInputSignatureSize, &pInputLayout));
```

Input Layout Creation - Notes

- Semantic name and index are needed to access the vertex attributes in HLSL in the vertex shader, e.g.
  `"POSITION", 0` → `float3 pos : POSITION0`
  HLSL
- `passDesc`
  - Input Signature of **any pass that uses the input layout** we want to create
  - Used to verify that Input Layout and Vertex Shader match
- Input Layouts need to be released
  → `SAFE_RELEASE()` in `OnDestroyDevice()`

Drawing from Buffer Data

- To draw using an index buffer we have to
  - Setup input assembler correctly
  - Bind shader pass
  - Use `DrawIndexed()`
- In `OnFrameRender()`, e.g.

```
// Set vertex and index buffer
UINT stride = sizeof(SimpleVertex);
UINT offset = 0;
pd3dContext->IASetVertexBuffers(0, 1, &pVertexBuffer, &stride, &offset);
pd3dContext->IASetIndexBuffer(pIndexBuffer, DXGI_FORMAT_R32_UINT, 0);
pd3dContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
pd3dContext->IASetInputLayout(pInputLayout);

// Apply pass (shader)
pMyPass->Apply(0,  pd3dContext);

// Draw using index buffer
pd3dContext->DrawIndexed(numTriangles * 3, 0, 0);
```

Mesh Rendering

- Meshes are provided as *.ply files
  – PLY: Polygon File Format / Stanford Triangle Format
- You need to
  – Parse PLY file & create a CPU vertex / index buffer
  – Create a GPU vertex / index buffer
  – Upload vertex and index buffer from CPU to GPU
  – Input data from buffers into the pipeline during draw call
  – **Stream data through pipeline and shaders**
  – Apply Phong lighting model in pixel shader

Semantics

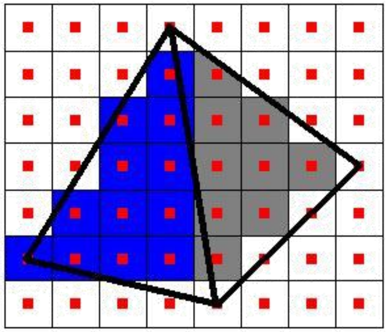
- Semantics are identifiers used for passing information from stage to stage
  - Input Assembler to Vertex Shader
  - Vertex Shader to Rasterizer to Pixel Shader
  - …
- Semantics have a name and an index, e.g. `POSITION5`, `COLOR1`, `SV_Target` (= `SV_Target0`), …
  - Default index 0
  - System value semantics start with SV_
- Every shader input and output has to have a semantic associated with it

Semantics Example

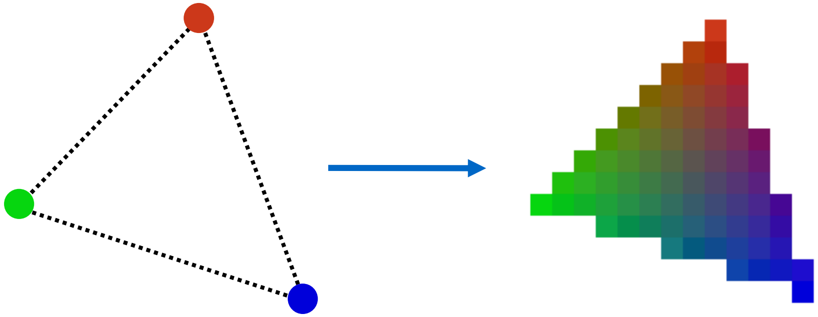
```
struct MyVertex {
    float3 pos : POSITION; // semantics matched
    float3 nor : NORMAL;   //   to input layout
};

struct PSIn {
    float4 svpos : SV_Position; // mandatory output from VS/GS
    float3 nor   : FOOBAR;      // semantic names can be anything
};

// No need to specify semantics here, they are already defined in the struct
PSIn SimpleVS(MyVertex v) {
    PSIn result;
    result.svpos = mul(float4(v.pos, 1.0), g_WorldViewProj);
    result.nor   = ...;
    return result;
}

// Between VS and PS, rasterization magic happens to members of PSIn!!!

float4 SimplePS(float4 pos : SV_Position, float3 n : FOOBAR) : SV_Target {
    n = normalize(n);  // re-normalize interpolated normal
    return ...;
}
```

Rasterization

- Fragment generation: for each covered pixel, one fragment is generated
(based on coordinates given by `SV_Position`)



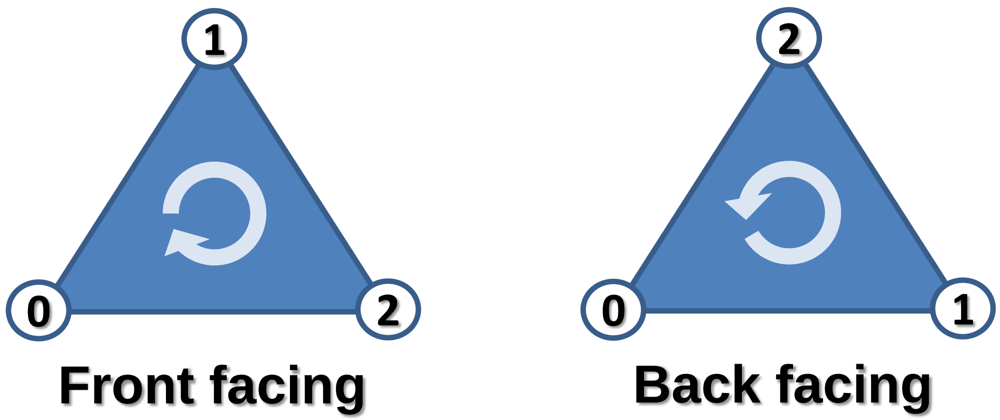
- For each fragment: per-vertex attributes (color, normal, z-value, texture coordinates,…) are interpolated



  – Caution: For SV_* semantics, magic may happen!

Culling

- Triangles have an orientation depending on vertex order (in screen space)



- We can instruct the rasterizer not to generate any fragments for back or front facing triangles (e.g. for performance)

HLSL

```
// don't forget to "SetRasterizerState" in your effect pass
RasterizerState rsCullFront {
    CullMode = Front; // can be Front | Back | None
};
```

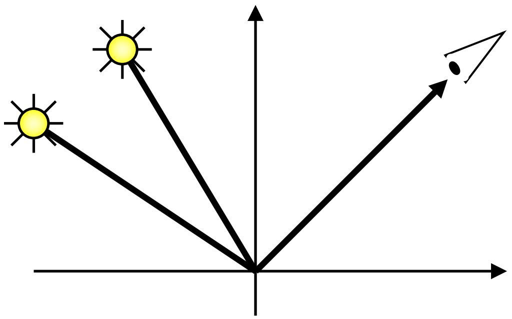
Mesh Rendering

- Meshes are provided as *.ply files
  - PLY: Polygon File Format / Stanford Triangle Format
- You need to
  - Parse PLY file & create a CPU vertex / index buffer
  - Create a GPU vertex / index buffer
  - Upload vertex and index buffer from CPU to GPU
  - Input data from buffers into the pipeline during draw call
  - Stream data through pipeline and shaders
  - **Apply Phong lighting model in pixel shader**

Local Illumination

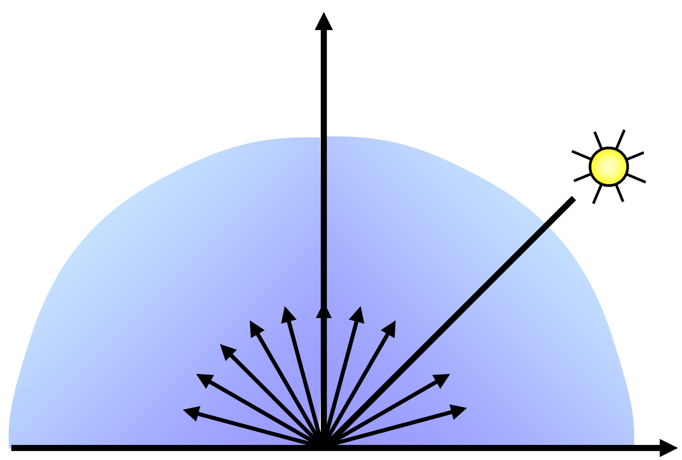
General Idea

- Consider only (non-area) light sources that are directly visible from the point on the object's surface without reflections.
- Each point is illuminated independently of its "global" surroundings
- Phong Lighting: Approximate illumination by three additive components, representing ***diffuse***, ***specular*** and ***ambient*** lighting



Diffuse Lighting

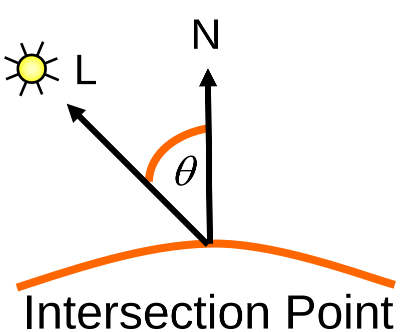
- Rough material
- Brightness ~ incoming Energy (Lambertian reflection)
- Object scatters light into all directions equally
- Heuristic reflection model but plausible for certain materials

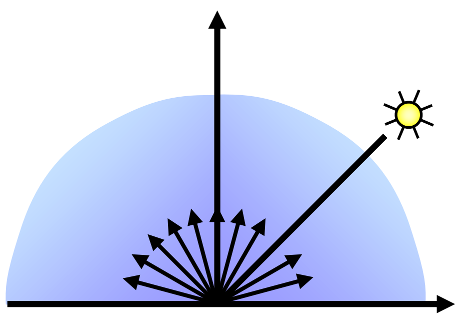


Diffuse Lighting

- Rough material
- Brightness ~ incoming Energy (Lambertian reflection)
- Object scatters light into all directions equally
- Heuristic reflection model but physically plausible for certain k_d

$$\boxed{I_{diff}(x) = k_d \cdot I_{in}(x) \cdot \cos(\angle(N,L)) = k_d \cdot I_{in}(x) \cdot (N \circ L)}$$





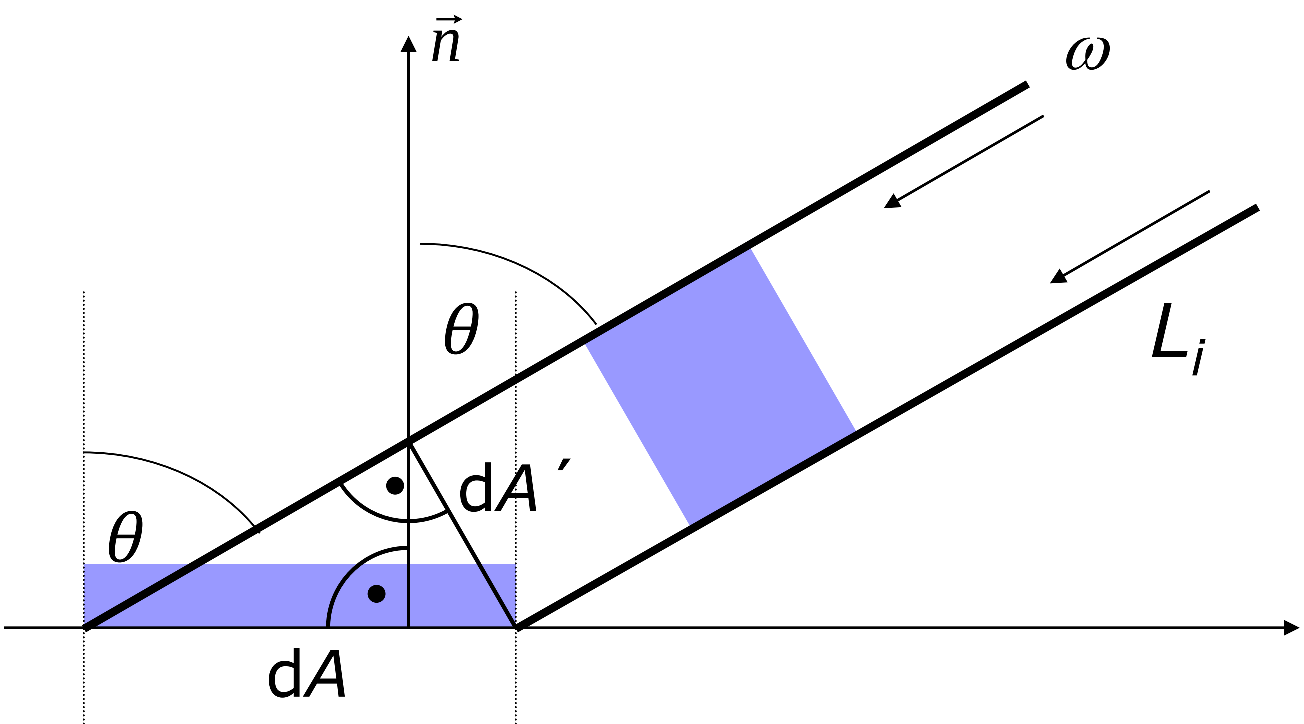
Lambertian Law

Johann Friedrich Lambert (1783):

Power per unit area arriving at some object point x also depends on the angle of the surface to the light direction

Effectively lit area: dA

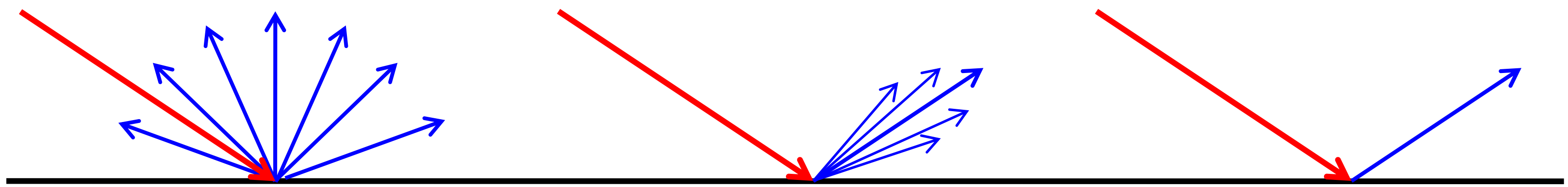
dA´= dA cosθ



Local Illumination

Specular Lighting

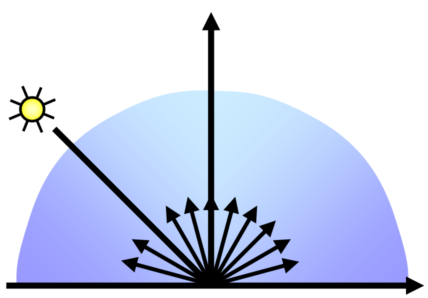
- Glossy/smooth material
- Light is mostly reflected into the directions around the mirror direction R_L of L

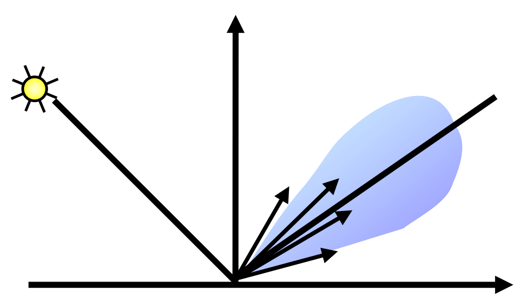


Diffuse

Glossy

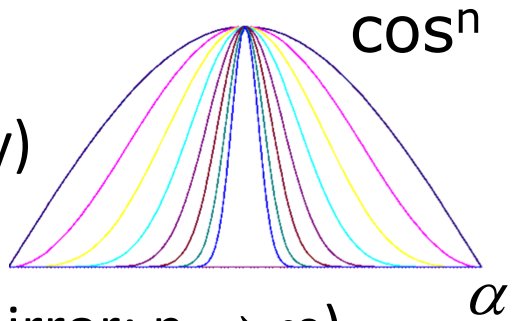
(Perfectly) Specular



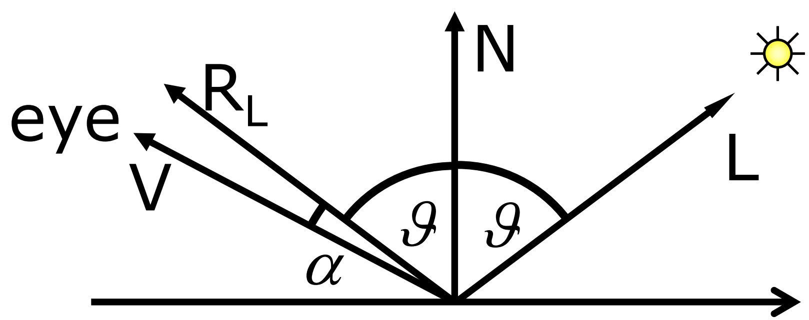


Specular Lighting: The Phong model

- Light is mostly reflected into the directions around the mirror direction R_L of L (Rapid decay)
- Use cosine power as heuristic
- Perfect mirroring only in direction R_L (perfect mirror: $n \to \infty$)



$$\boxed{I_{spec}(x) = k_s \cdot I_{in}(x) \cdot \cos(\alpha)^n = k_s \cdot I_{in}(x) \cdot (V \circ R_L)^n}$$

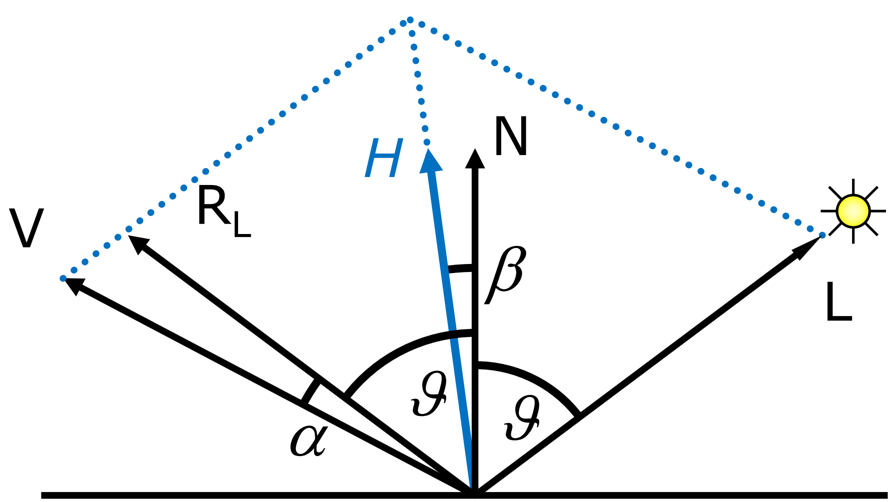


Specular Lighting: The Blinn-Phong model

- Calculation of R_L used to be expensive
- Approximate α using the halfway vector *H*

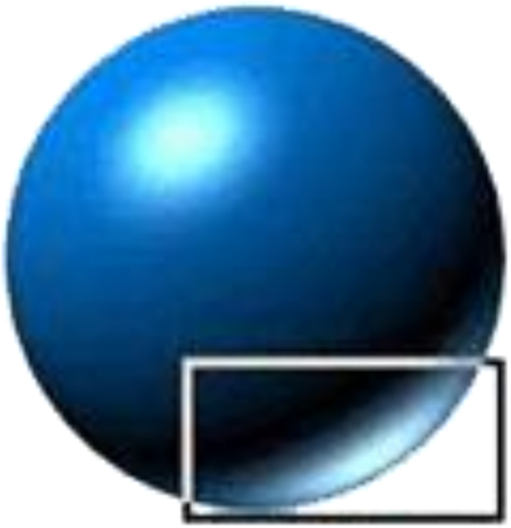
$$H = \frac{L+V}{|L+V|}$$

$$\boxed{I_{spec}(x) = k_s \cdot I_{in}(x) \cdot \cos(\beta)^n = k_s \cdot I_{in}(x) \cdot (H \circ N)^n}$$

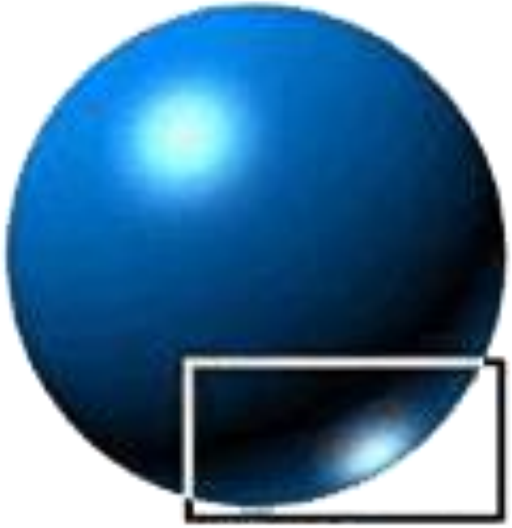


$$\Rightarrow \alpha \approx \beta$$

Phong vs. Blinn-Phong



Blinn-Phong



Phong

The angle between R and V is (roughly) twice the angle between N and H

→ Highlights appear sharper in the Phong model

Local Illumination

Ambient Lighting

- Hack for replacing true global illumination (i.e. light bouncing off from other objects)
- No direction
- Incoming light component that is identical everywhere in the whole scene

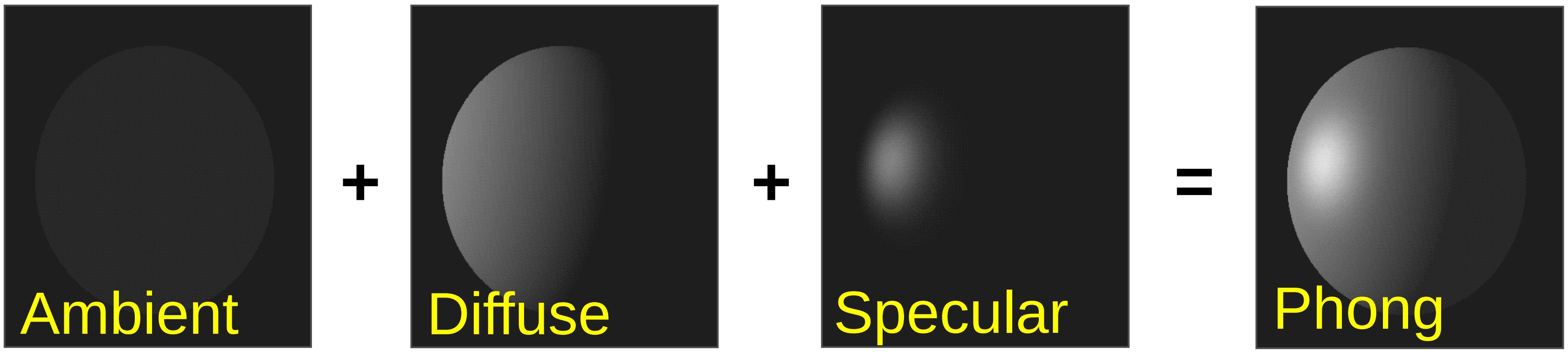
$$\boxed{I_{amb} = k_a \cdot I_a}$$

where

- k_a is the ambient material coefficient of reflection with $0.0 <= k_a <= 1.0$ and
- I_a is the intensity of the ambient light

Local Illumination

$$I_{amb} = k_a \cdot I_a, \quad I_{diff}(x) = k_d \cdot I_{in}(x) \cdot (N \circ L), \quad I_{spec}(x) = k_s \cdot I_{in}(x) \cdot (V \circ R_L)^n$$



Careful!

If a light is behind the object (α > 90°) then cos(α) < 0.

→ Discard negative intensity values by clamping the dot products to the range [0,1], for instance, use `saturate(dot(N, L));`

Incorporating Color

- So far we have only dealt with *Intensity*:

$$I_{local} = k_a \cdot I_a + \sum_{x=0}^{numLights} (k_d \cdot I_{in}(x) \cdot (N \circ L) + k_s \cdot I_{in}(x) \cdot (V \circ R_L)^n)$$

- To incorporate color:
  - Diffusely reflected light results from the reflection via multiple scattering events in the micro-scale geometry $\Rightarrow$ *reflected light is colored by selective absorption by the **surface***, i.e. a green surface absorbs all wavelengths except green
  - Specularly reflected light interacts once with the surface and is *not colored by the surface,* i.e. the reflection of a light source takes on the color of the ***source***

Color of Material and Light

Usually we define color as a 3-component vector C(r,g,b)

Therefore, the following variables become vectors:

k_a ambient material color
k_d diffuse material color

Often the same!

I_a global ambient light color, defined once in the whole scene

I_{in} light color

$$\boxed{k_a} \cdot \boxed{I_a} + \sum_{x=0}^{numLights} \left(\boxed{k_d} \cdot \boxed{I_{in}}(x) \cdot (N \circ L) + k_s \cdot \boxed{I_{in}}(x) \cdot (V \circ R_L)^n \right)$$

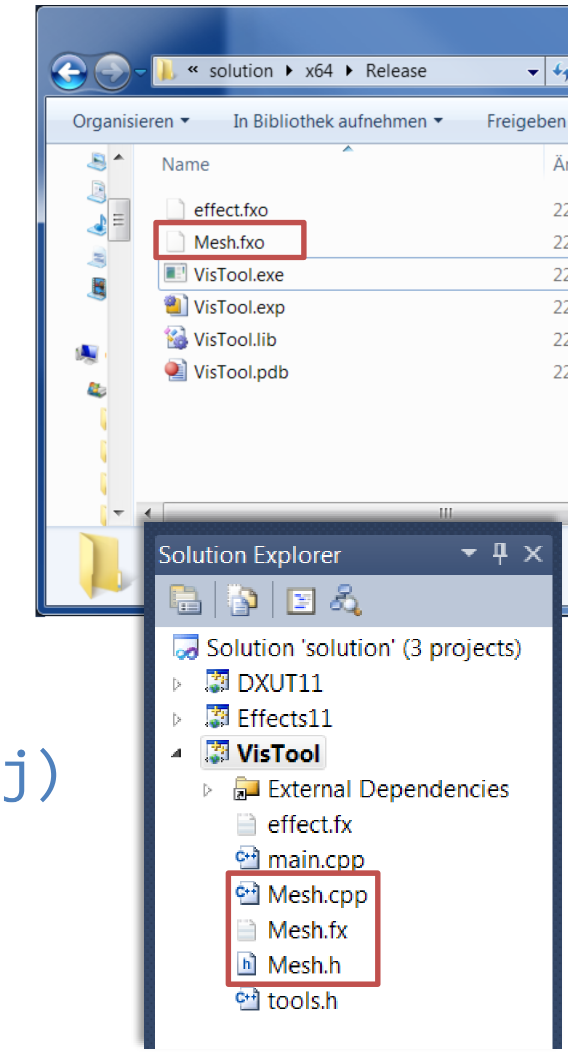
k_s is still a scalar!

Light Direction

- Where do we get L from?
  - We need it in the pixel shader for every fragment
- Simple directional light
  - $L = const$ (given e.g. in world-space)
  - Global effect variable for light direction
- „Head light“ ($pos_{camera} = pos_{light}$)
  - $L = normalize(pos_{light} - pos_{fragment})$
  - In vertex shader, calculate world space position of vertex and hand it to pixel shader (in addition to normal)
  - In pixel shader, use this (now interpolated) position as $pos_{fragment}$
  - Global effect variable for light position

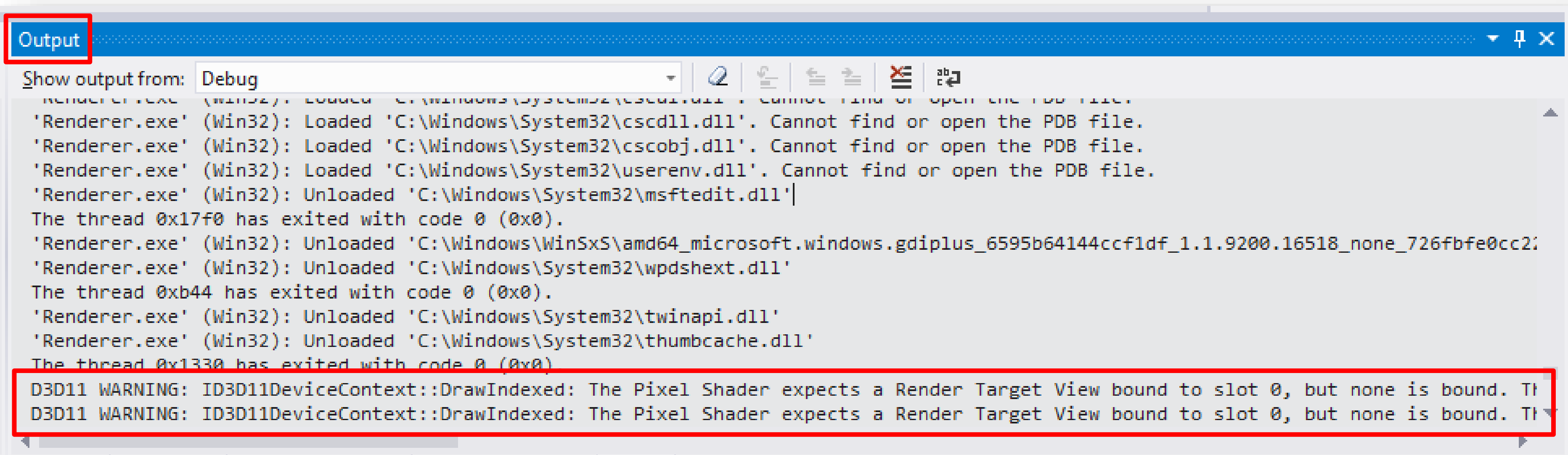
C++ Classes and Direct3D/Effects

- Good coding style: structure your project, use classes!
- If a class has GPU functionality
  - Create a separate effect file
  - Effect, effect variables, input layouts etc. as static member variables (loaded once on program start)
  - Create methods that correspond to the DXUT callback functions and are called from there, e.g.
    - `Mesh::Create()` called from `OnCreateDevice()`
    - `Mesh::Draw(Matrix worldViewProj)` called from `OnFrameRender()`
    - …



D3D Debugging

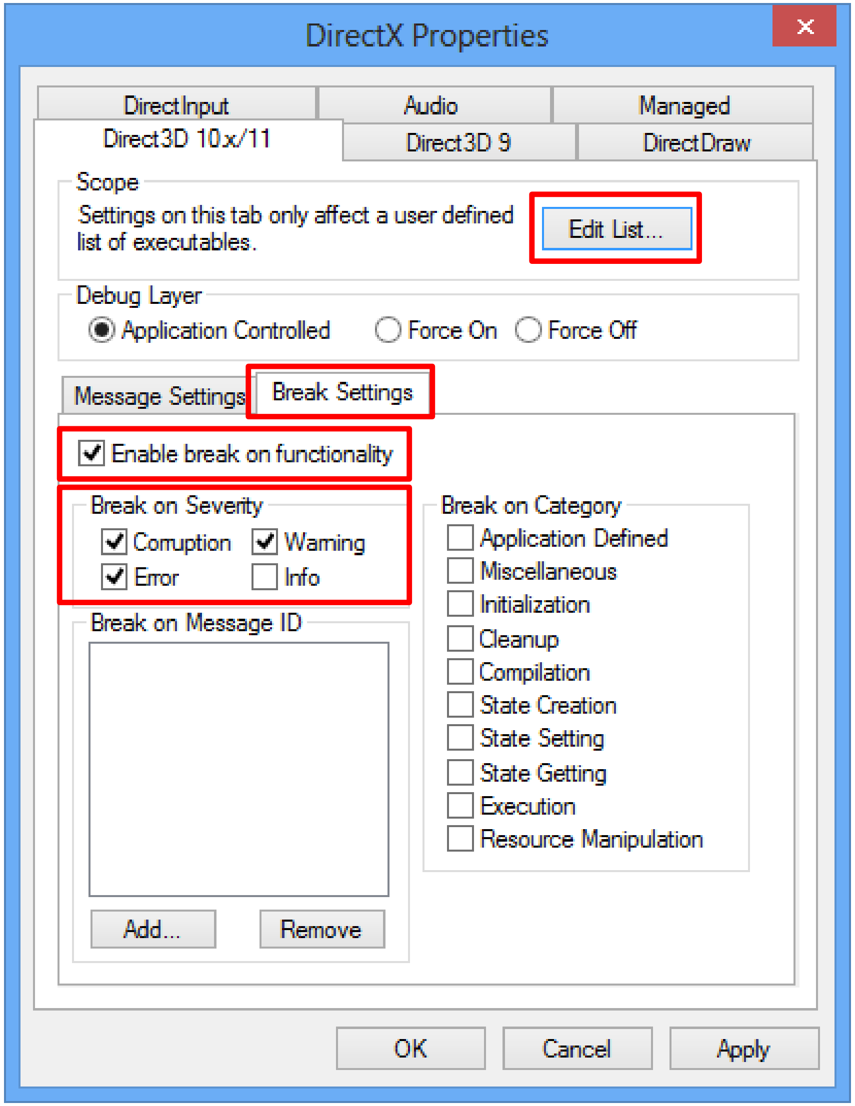
- Most common error: „I don‘t see anything!“
- In debug mode, VS Output window often shows helpful hints (warnings/errors)



- Fix them!!!

D3D Debugging

- To make D3D warnings/errors easier to track down:
  - Automatically fire a breakpoint when warning/error occurs
  - Available from the DirectX Control Panel (64-Bit)



Shader Debugging

- Ensure that your effect compiles without warnings!
- Vertex Shader Debugging: „I can‘t see anything“
  - Use a trivial (constant color) PS, e.g.
    ```
    float4 RedPS() : SV_Target { return float4(1,0,0,1); }
    ```
  - Disable culling and blending
  - Build your shader incrementally, start by returning constant NDC positions, e.g.
    - $(0,0,0,1)^T$: screen center (front)
    - $(-1,-1,1,1)^T$: screen bottom left (back)
- Pixel Shader Debugging:
  - „printf-debugging in HLSL“: map variables you want to inspect to color values $\in [0; 1]^3$, e.g.
    ```
    return float4((float3)0.5 + 0.5 * normal, 1.0);
    ```

References

- RPly
  - http://w3.impa.br/~diego/software/rply/

Q&A

Questions ?