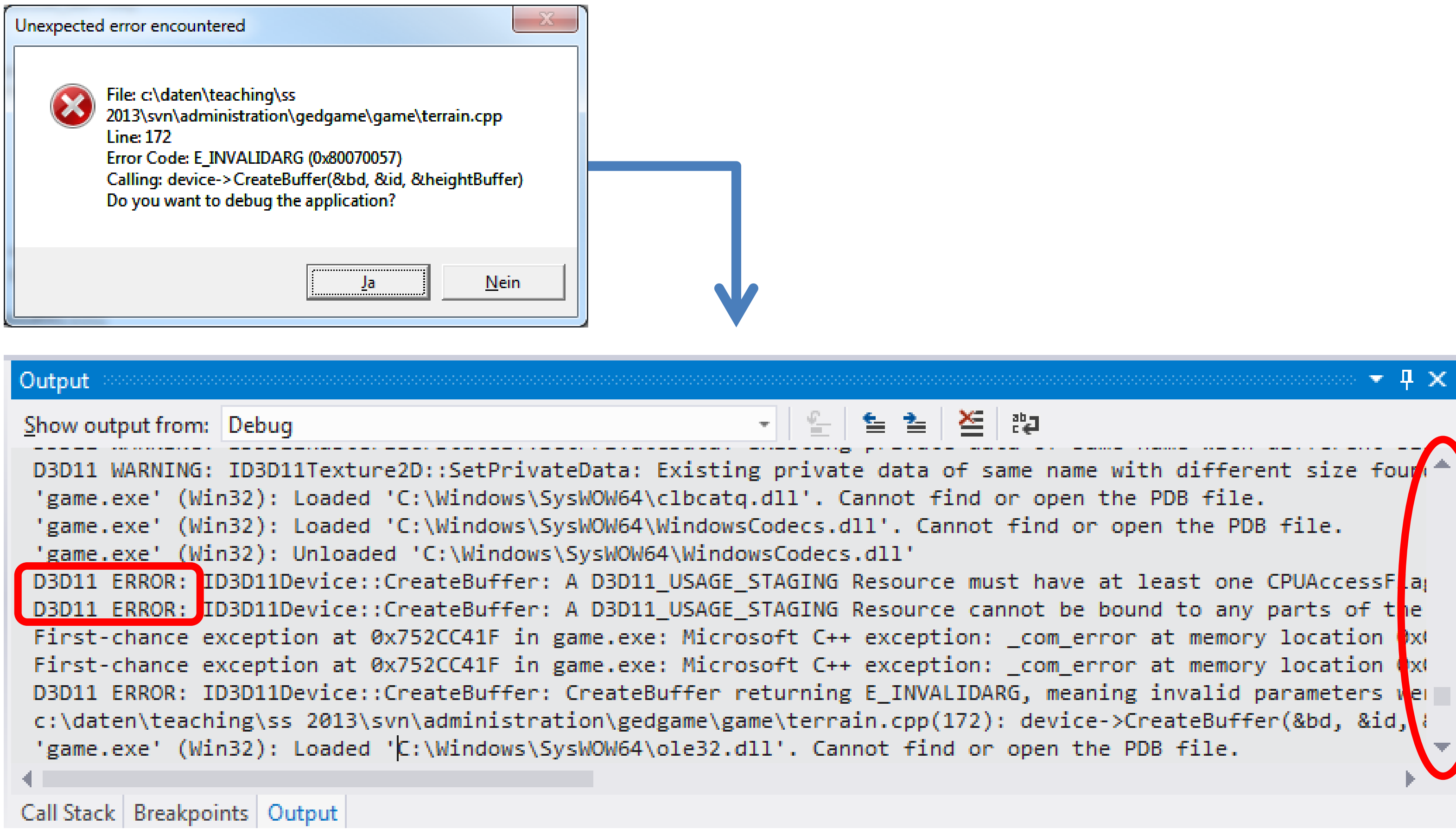
Game Engine Design

computer graphics & visualization

- Reminder: DirectX prints pretty detailed error messages to the output console



Shader Debugging Hints

computer graphics & visualization

- Pixel Shader „Debugging“: You can pass anything you want to SV_TARGET0!
  - To visualize your normals, texture coordinates etc
  - Beware: Your normals will look much brighter, but that‘s fine
    - Keyword: „Gamma Correction“

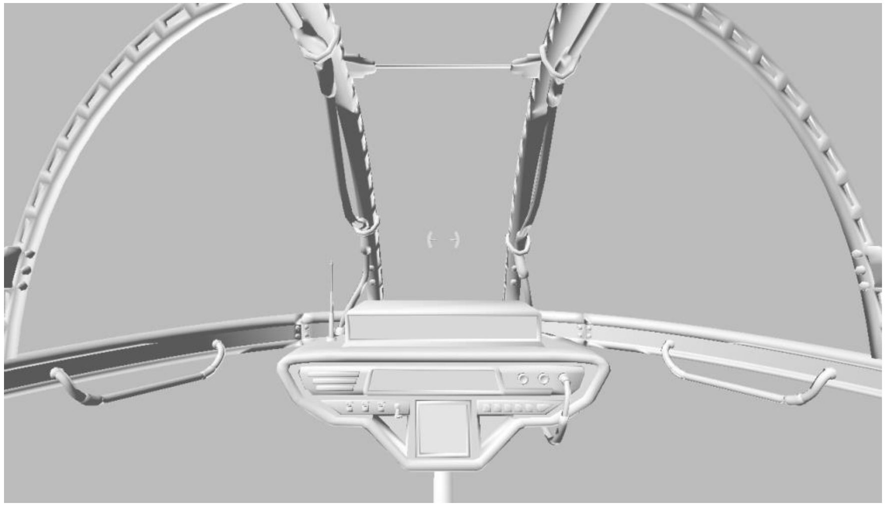
- Vertex Shader „Debugging“: Pass stuff to your pixel shader
  - Of course not very helpful if nothing is shown…

- And remember the VS2012 & VS2013 Graphics Debugger!

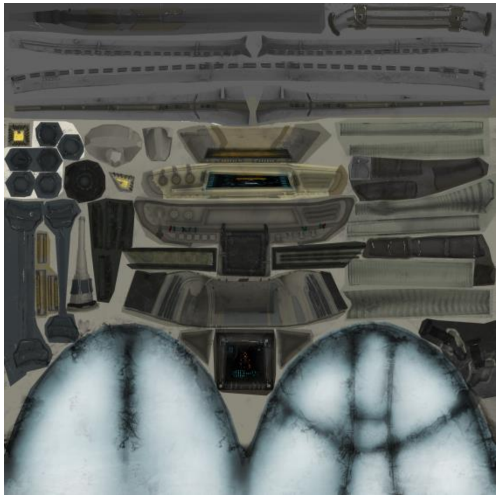
Assignment 6

- This week: Rendering and Lighting a Mesh

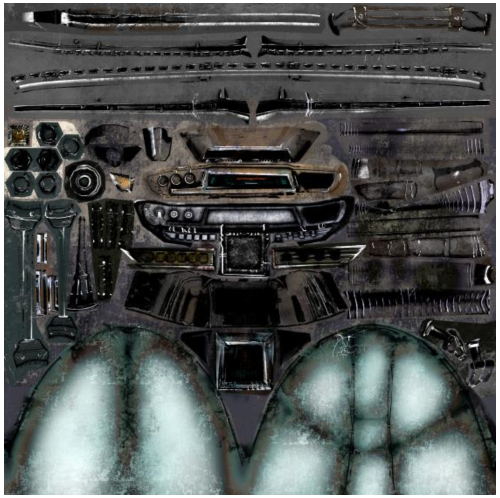
Input Resources for Cockpit Mesh



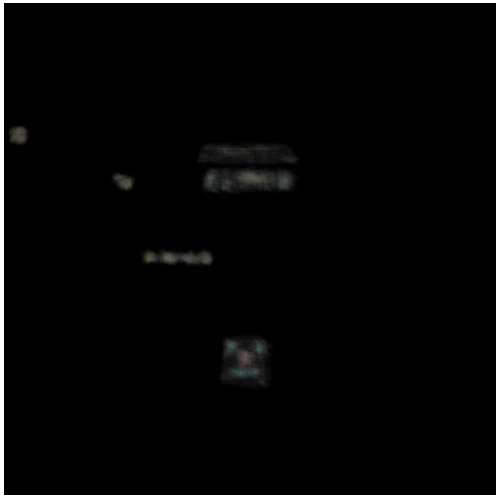
Geometry



Diffuse texture



Specular texture



Glow texture
(self emission of light)

Input Geometry

- A variety of meshes is provided (in external/art/)
- For now we will restrict to one: cockpit_o_low.obj
- Obj-Format:
  - Open and very flexible
  - Problem: hard to parse
  - Solution: we use our own file format (t3d)
- T3d-Format:
  - Generated with tool obj2t3d.exe (called via NMake script)
  - Simply contains the vertex and index buffer
  - A class for loading is provided

Obj-Format

- Example:

```
# List of Vertices, with (x,y,z[,w]) coordinates, w is optional and defaults to 1.0.
v 0.123 0.234 0.345 1.0
v ... ...

# Texture coordinates, in (u[,v][,w]) coordinates, v and w are optional and default to 0.
vt 0.500 -1.352 [0.234]
vt ... ...

# Normals in (x,y,z) form; normals might not be unit.
vn 0.707 0.000 0.707
vn ... ...

# Face Definitions (see below)
f 6/4/1 3/5/3 7/6/5
f ... ...
```

- For more info: http://en.wikipedia.org/wiki/Wavefront_.obj_file

Assignment 6

- Tasks this week:
  - Prepare and load resources for the cockpit mesh
    - Most parts are already implemented for you
  - Create a transformation for the mesh from object into world space (correct mesh placement in first person view)
  - Write a pixel and a vertex shader
    - Vertex shader: Apply your transformation
    - Pixel shader: Phong lighting model (in world space)

Transformations & Spaces

- The usual transformation pipeline:

vertex shader

(Local) Object Space

World Transformation M_{World}

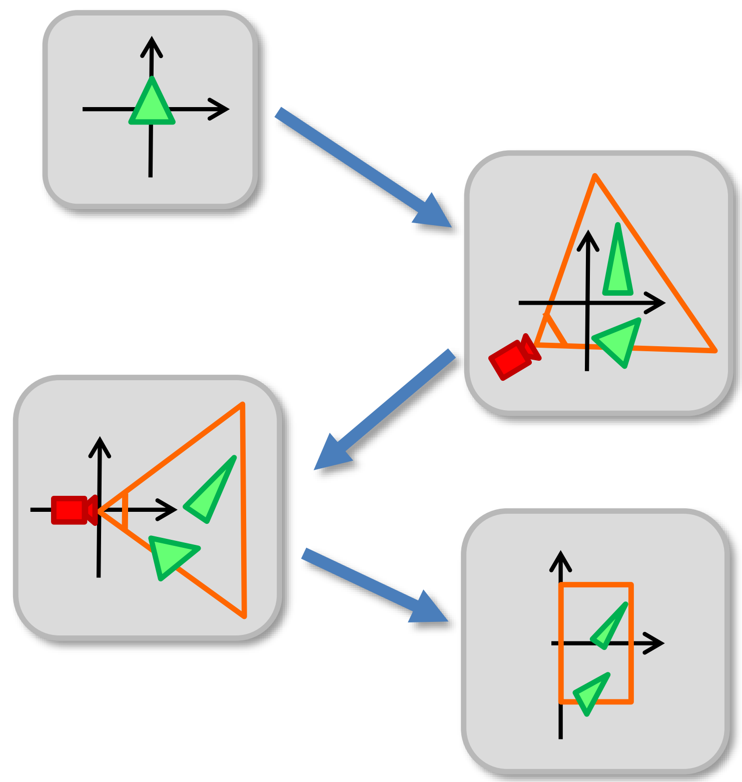
(Global) World Space

View Transformation M_{View}

View Space

Projection Transformation M_{Proj}

Clip coordinates (w!=1)



Cockpit Transformation

- Cockpit should "stick" to Camera
  - Cockpit position and rotation must match the camera's
  - General approach: apply inverse view matrix
  - CFirstPersonCamera specific: GetWorldMatrix() returns the inverse of GetViewMatrix()

- Transformation with view and inverse view matrix „cancel out" when composing the worldViewProjection matrix
  - But: Transformation to world space neccessary due to lighting in world space

Transformations with D3D11

- Usually composed on CPU before every draw call and then sent to GPU (by setting corresponding effect variables)
- **Caution:** In DirectX we combine transformation matrices in reversed order
  - Mathematician friendly style:
  $$\mathrm{p}' = \mathrm{M}_{Proj} \cdot M_{View} \cdot M_{World} \cdot p$$
  - DirectX style (order of transformations = writing order):
  $$\mathrm{p}'^{T} = p^{T} \cdot \mathrm{M}_{World}^{T} \cdot M_{View}^{T} \cdot M_{Proj}^{T}$$
  - Matrices created by the XMMatrix*-functions are already transposed
  - XMVECTOR is automatically treated correctly

→ e.g. `XMVector4Transform(p, M)`
calculates $p'^{T} = p^{T} \cdot M$

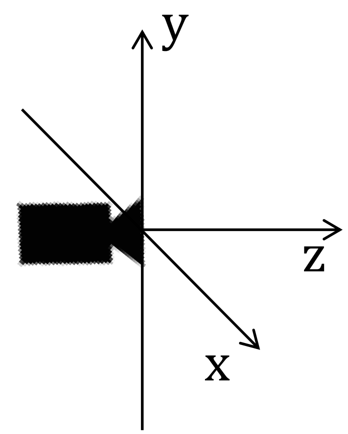
D3D11 Transformation Example

```
//Create transformation matrices
XMMATRIX mTrans, mScale, mRot;
mRot = XMMatrixRotationY(…); //set angleInRadians yourself
mTrans = XMMatrixTranslation(…);
mScale = XMMatrixScaling(…);

//Object to world space for cockpit (for lighting):
//  rotation first, then translation and then scaling, then transform
//  to camera position / rotation
XMMATRIX mWorld = mScale * mRot * mTrans * g_camera.GetWorldMatrix();

//Object to clip space for cockpit (for rendering):
XMMATRIX mWorldViewProj = mWorld * g_Camera.GetViewMatrix() * g_Camera.GetProjMatrix();
// Note: mRot * mTrans * mScale * (*g_Camera.GetProjMatrix()) yields the
// same result since GetWorldMatrix() is the inverse of GetViewMatrix()

//Inverse transposed of world for transformation of normals
XMMATRIX worldNormals;
//....
```

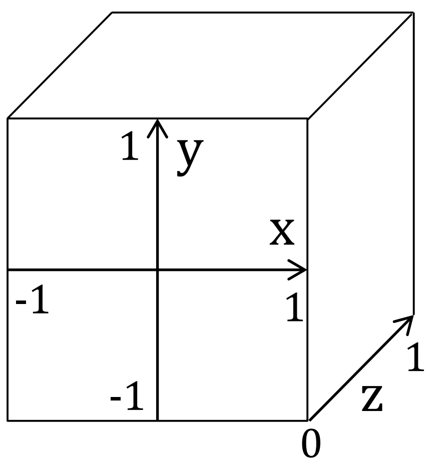
For the cockpit mesh: rotation angle = 180°, **translation = (0, -0.8, 2.1),** scaling = (0.05, 0.05, 0.05).

DirectX Space Conventions

- View space
  - Left-handed coordinate system
  - Camera at (0,0,0), looks into $+z$ direction
  - $+x$ is right, $+y$ is top



- Normalized Device Coordinates (after projection transformation and perspective division)
  - $x \in [-1; 1] \leftrightarrow$ screen from left to right
  - $y \in [-1; 1] \leftrightarrow$ screen from bottom to top
  - $z \in [0; 1] \leftrightarrow$ depth from near to far



Vertex Shader for the Mesh

- **Pseudo**-code for
  `T3dVertexPSIn MeshVS(T3dVertexVSIn in){…}`

  $$
  \begin{aligned}
  out.Pos &\leftarrow (in.Pos, 1) \cdot M_{WorldViewProj} \\
  out.Tex &\leftarrow in.Tex \\
  out.Pos_{World} &\leftarrow dehom_1((in.Pos, 1) \cdot M_{World}) \\
  out.Nor_{World} &\leftarrow normalize(dehom_0\left((in.Nor, 0) \cdot M_{WorldNormals}\right)) \\
  out.Tan_{World} &\leftarrow normalize(dehom_0((in.Tan, 0) \cdot M_{World}))
  \end{aligned}
  $$

- We don't need $out.Tan_{World}$ in this assignment
- $dehom_1(x\ y\ z\ w) \coloneqq \frac{1}{w} \cdot (x\ y\ z)$, $dehom_0(x\ y\ z\ w) \coloneqq (x\ y\ z)$
- It is safe here to leave out $\frac{1}{w}$ in $dehom_1$ (M_{World} contains no projective components, i.e. $w = 1$)
- Transformation of directions/normals: $M_{WorldNormals} = \left(M_{World}^{-1}\right)^T$

Lighting in World Space

- Eye is at camera position
  - Generally: apply inverse view transformation to (0, 0, 0, 1)
  - CFirstPersonCamera: GetEyePt()

- Light direction is given in world space
  - Fixed direction for now

- Two transformations of positions and normals neccessary
  - Object -> World space for lighting
  - Object -> Clip space for rendering
  - Calculate **both** in vertex shader and pass to the pixel shader

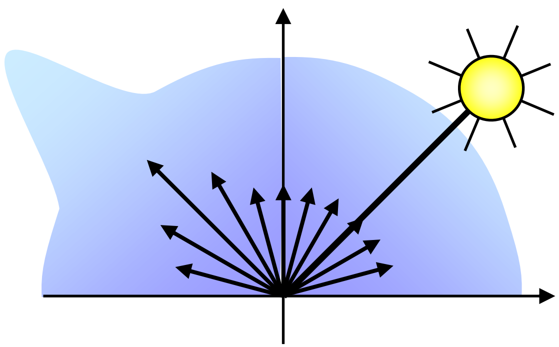
Phong Lighting Model (from Lecture)

- Combines diffuse, specular and ambient terms to model all effects

$$I_r(\mathrm{x}, \omega_v) =$$

$$k_d \cdot \left(\vec{I} \cdot \vec{n}\right) \cdot I_i(x, \omega_l) +$$
$$k_s \cdot (\vec{r} \cdot \vec{v})^s \cdot I_i(x, \omega_l) +$$
$$k_a \cdot I_a$$

  - Ambient term models constant background light



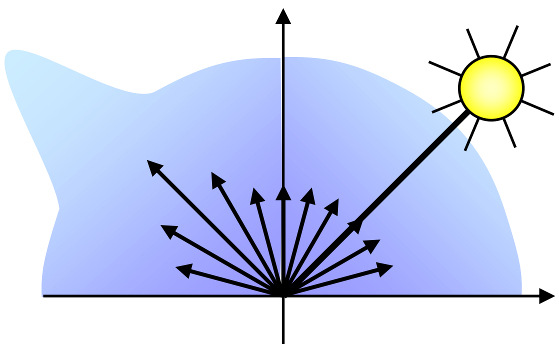
Phong Lighting Model (for Mesh)

- Combines diffuse, specular and ambient terms to model all effects

$$I_r(\mathrm{x}, \omega_v) =$$

$$k_d \cdot (\vec{l} \cdot \vec{n}) \cdot I_i(x, \omega_l) +$$
$$k_s \cdot (\vec{r} \cdot \vec{v})^s \cdot I_i(x, \omega_l) +$$
$$k_a \cdot I_a + k_g$$

  - Ambient term models constant background light
  - Glow term models self emission of light (e.g. backlit displays)



Pixel Shader for the Mesh

- **Pseudo**-code for `float4 MeshPS(T3dVertexPSIn in) : SV_Target0 {…}`

$$
\begin{aligned}
mat_{Diffuse} &\leftarrow DiffuseTexture.Sample(in.Tex)\\
mat_{Specular} &\leftarrow SpecularTexture.Sample(in.Tex)\\
mat_{Glow} &\leftarrow GlowTexture.Sample(in.Tex)\\
col_{Light} &\leftarrow (1,1,1,1)\\
col_{LightAmbient} &\leftarrow (1,1,1,1)
\end{aligned}
$$

$$
\begin{aligned}
n &\leftarrow normalize(in.Nor_{world})\\
I &\leftarrow LightDir_{world}\\
r &\leftarrow reflect(-I,n)\\
v &\leftarrow normalize(cameraPos_{world} - in.Pos_{world})
\end{aligned}
$$

$$
\begin{aligned}
result \leftarrow\; & c_d \cdot mat_{Diffuse} \cdot saturate(dot(n,I)) \cdot col_{Light}\\
&+ c_s \cdot mat_{Specular} \cdot saturate(dot(r,v))^s \cdot col_{Light}\\
&+ c_a \cdot mat_{Diffuse} \cdot col_{LightAmbient}\\
&+ c_g \cdot mat_{Glow}
\end{aligned}
$$

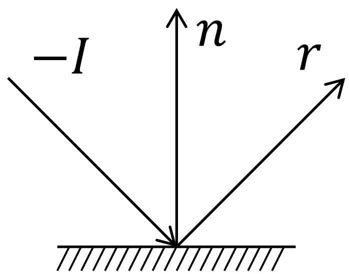
- „·“: scalar multiplication / component wise multiplication
  = *-operator in HLSL

Pixel Shader for the Mesh

- Some remarks:
  - $c_x \cdot mat_x$ corresponds to k_x in the phong model formula
  - c_d, c_s, c_a, c_g control weighting of individual terms
    - In theory $c_d + c_s + c_a + c_g = 1$, but > 1 might produce nicer results
    - Example: $c_d = 0.5, c_s = 0.4, c_a = 0.1, c_g = 0.5$
    - Play around with the c's
  - Try various specular exponents s (eg. 1,10,100,....)
  - No $mat_{ambient}$: for the ambient term we simply use the same color texture as for the diffuse term

HLSL Functions for Shaders

- Useful HLSL intrinsic functions for this assignment (some should already be known):
  - mul, normalize, dot, pow
  - $\mathrm{reflect}(-\mathrm{I},\mathrm{n}) \coloneqq \mathrm{reflect\ I\ at\ surface\ with\ normal\ n}$



  - $\mathrm{saturate}(\mathrm{c}) \coloneqq \begin{cases} 0, & \mathrm{if\ c} < 0 \\ 1, & \mathrm{if\ c} > 1 \\ \mathrm{c}, & \mathrm{else} \end{cases}$
  - $dehom_{0/1}$: not a HLSL-function, implement using subscripts: „`vec.xyz/vec.w`“ or „`vec.xyz`“
- Look up the functions in the DirectX documentation!

Questions?