Shaders and Visual Realism
Very Brief History

- The state of the art in computer graphics was in offline rendering in the 80s and early 90s.
- A sophisticated and flexible rendering pipeline was being built in software – geared towards ultimate realism.

*The Genesis Effect from Star Trek II - The Wrath of Khan © Pixar*
Very Brief History

• The development was mostly driven by research and commercialization of CGI (Computer Generated Imagery in films and commercials).

• This lead to special shading languages being invented, of which the RenderMan language has been the most successful (used by Pixar).
“Toy Story” - Pixar, 1995 – Using the Pixar RenderMan Language
(Programmable Photorealistic Off-line Rendering)
Photorealistic rendering in Pixie, an Open Source RenderMan
Very Brief History

• When OpenGL 1.0 was announced in 1992, by SGI, DEC, IBM, Intel and Microsoft, they decided to keep the rendering pipeline fixed function and NOT programmable.
• They said: “..programmability would conflict with keeping the API close to the hardware and reduce optimum performance.”
Very Brief History

• OpenGL nevertheless took off, and showed many ground breaking applications, including games (e.g. Quake).

• Although OpenGL was fixed function, it was open to extensions.

• Around this time, new powerful graphics hardware was popping up on regular PCs.

1996 – Quake by id
Part I

VISUAL REALISM IN GAMES
“Alan Wake”
“Alan Wake”
“Alan Wake”
“Alan Wake”
“Unreal Tournament 2007”
“Crysis”
“Test Drive Unlimited”
“Test Drive Unlimited”
Part II

SHADER PROGRAMMING

Most of the following images were taken from
“Shaders for Game Programmers and Artists” by Sebastien St-Laurent,
Copyright 2004 by Course Technology
3D Object (Shaded and colored)
Polygons (Triangles)
Vertex Transformation

$(x_1, y_1, z_1)$

$(0, 0, 0)$
Translating and Scaling

Translation:
\[
\begin{pmatrix}
1 & 0 & 0 & Tz \\
0 & 1 & 0 & Tx \\
0 & 0 & 1 & Ty \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

Scale:
\[
\begin{pmatrix}
Sx & 0 & 0 & 0 \\
0 & Sy & 0 & 0 \\
0 & 0 & Sz & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]
Rotating

\[\begin{align*}
\begin{pmatrix}
\cos \theta & 0 & \sin \theta & 0 \\
0 & 1 & 0 & 0 \\
-\sin \theta & 0 & \cos \theta & 0 \\
0 & 0 & 0 & 1
\end{pmatrix} & \quad \begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & \cos \theta & -\sin \theta & 0 \\
0 & \sin \theta & \cos \theta & 0 \\
0 & 0 & 0 & 1
\end{pmatrix} \\
\begin{pmatrix}
\cos \theta & -\sin \theta & 0 & 0 \\
\sin \theta & \cos \theta & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix} & \quad \begin{pmatrix}
\cos \theta & 0 & \sin \theta & 0 \\
0 & 1 & 0 & 0 \\
-\sin \theta & 0 & \cos \theta & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\end{align*}\]
Vertex Transformation

Object vertices in Object coordinates:
$\text{T}_{\text{obj\,local}}$
World Coordinates

Object vertices in World Space coordinates:
\[ \text{Tobj}_{\text{world}} \cdot \text{Tobj}_{\text{local}} \]
Camera Coordinates

Object vertices in Camera Space coordinates:

\[ T_{\text{cam world}}^{-1} \cdot T_{\text{obj world}} \cdot T_{\text{obj local}} \]
Face Culling and Clipping

Clipped if outside frustum

Culled if facing away

Clipped if outside frustum
View Projection

Object vertices in Screen Space coordinates:

$$T_{\text{viewproj}} \bullet T_{\text{cam}}_{\text{world}}^{-1} \bullet T_{\text{obj}}_{\text{world}} \bullet T_{\text{obj}}_{\text{local}}$$
Rasterization

Colored Vertices After Vertex Transformation → Primitive Assembly → Rasterization

fragment
Fragment Coloring and Blending

Coloring of fragments based on interpolated information from nearby vertices (e.g. Vertex colors, Vertex UV coordinates, Vertex Normals)

Alpha Blending and Depth Tests

Z-buffer keeps track of the front most fragments
Pixels (what we see on the screen)
## PC 3D Rendering in Hardware

<table>
<thead>
<tr>
<th>Year</th>
<th>Graphics Card</th>
<th>Milestone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1987</td>
<td>IBM VGA</td>
<td>Provides a pixel frame buffer that the CPU has to fill</td>
</tr>
<tr>
<td>1996</td>
<td>3dfx Voodoo</td>
<td>Rasterizes and textures pre-transformed vertices (triangles)</td>
</tr>
<tr>
<td>1999</td>
<td>nVidia GeForce 256</td>
<td>Applies both transformation and lighting to vertices (T&amp;L) – fixed pipeline</td>
</tr>
<tr>
<td>2001</td>
<td>nVidia GeForce 3</td>
<td>Configurable pixel shader and programmable vertex shader</td>
</tr>
<tr>
<td>2003</td>
<td>nVidia GeForce FX</td>
<td>Fully programmable pixel and vertex shaders</td>
</tr>
</tbody>
</table>
// Simplest Vertex Shader

// input vertex
struct VertIn {
    float4 pos   : POSITION;
    float4 color : COLOR0;
};

// output vertex
struct VertOut {
    float4 pos   : POSITION;
    float4 color : COLOR0;
};

// vertex shader main entry
VertOut main(VertIn IN, uniform float4x4 modelViewProj) {
    VertOut OUT;
    OUT.pos = mul(modelViewProj, IN.pos); // calculate output coords
    OUT.color = IN.color; // copy input color to output
    return OUT;
}
// Small Pixel Shader (Grayscale Converter)
// input pixel
struct PixIn {
    float3 color : COLOR0;
    float3 texcoord : TEXCOORD0;
};

// output pixel
struct PixOut {
    float3 color : COLOR0;
};

// vertex shader main entry
PixOut main(PixIn IN, uniform sampler2D texture : TEXUNIT0) {
    PixOut OUT;
    float3 color = tex2D(texture, IN.texcoord).rgb;
    OUT.color = dot(color, float3(0.299, 0.587, 0.184)).xxx
    return OUT;
}
Some Categories of Shaders

- Vertex Skinning
- Vertex Displacement Mapping
- Screen Effects
- Light and Surface Models
- Non-photorealistic Rendering
Vertex Skinning

- The vertices on a surface, like the human body, get moved around based on an underlying skeletal structure. An additional deformation may also simulate the dynamic shape of a muscle.
Vertex Displacement

- Vertices can be displaced, for example vertically, based on an algorithm or an existing height map.
Scen Effects

- Pixel shader renders to a temporary texture that it then processes with filters before returning the color values.
Scene Effects: Glow
Scene Effects: Depth of Field
Scene Effects: Distortion
Scene Effects: High Dynamic Range + Bloom

Half-life 2 (Valve) – HDR on Right
Lighting Models

• Shaders calculate new color values by applying various lighting models, involving parameters such as surface normals (N), light angle (L), reflected light angle (R) and view angle (V).
Lighting Models: Per-Pixel Lighting
Lighting Models: Normal Mapping
Lighting Models: Environment Reflection
Lighting Models: Shadows
Non-Photorealistic Rendering

- Light models do not have to imitate the "real world", but can instead assign color values according to imaginary worlds, such as the world of cartoons or oil paintings.
- In fact, any of the aforementioned effects could be taken into the realm of the imaginary or expressionistic art.
Non-Photorealistic Rendering
Non-Photorealistic Rendering