The Graphics Pipeline

Prof. Dr. Markus Gross
The Graphics Pipeline

- From geometry, materials, and lighting to pixels
The Graphics Pipeline

• From geometry, materials, and lighting to pixels
The Graphics Pipeline

- Input: Geometry representation
- Triangles, points, lines, other primitives
The Graphics Pipeline

• Input: Materials & Lighting Models
The Graphics Pipeline

• Input: Virtual camera

Camera position

Frustum
The Graphics Pipeline

- Modeling Transform
- Transform
- Viewing Transform
- Primitive Processing
- Projection to Screen Space
- 3D Clipping
- Scan Conversion
- Lighting, Shading, Texturing
- Occlusion Handling
- Display

ETH Zürich
The Graphics Pipeline

Modeling Transform

From object to world space
The Graphics Pipeline

Viewing Transform

From world to camera space
The Graphics Pipeline

Output primitives from transformed vertices
The Graphics Pipeline

3D Clipping

Remove parts of objects (primitives) outside the frustum
Why do we need clipping?
- Avoid unnecessary computations
- Avoid numerical instabilities
The Graphics Pipeline

Projection to Screen Space

Project from 3D to 2D screen space
The Graphics Pipeline

- Discretize continuous primitives
- Triangles, lines, polygons
The Graphics Pipeline

Scan Conversion

- Interpolate attributes at all covered samples (normal, depth, uv)
The Graphics Pipeline

- Compute color based on lighting, shading, texture map
The Graphics Pipeline

- Lighting, Shading, Texturing

• Texture mapping using uv coordinate
The Graphics Pipeline

Occlusion Handling

- Update color buffer using the depth buffer (Z-buffer)
Programmer’s View

- Contemporary pipeline
Programmer’s View

- Contemporary pipeline

CPU → Vertex Processing → Rasterization → Fragment Processing → Display

Per-vertex operations
Transforms & Lighting
Flow control
Programmer’s View

• Contemporary pipeline

CPU ➔ Vertex Processing ➔ Rasterization ➔ Fragment Processing ➔ Display

Per-vertex operations
Transforms & Lighting
Flow control

Per-fragment operations
Shading & Texturing
Blending
Programmer’s View

- Contemporary pipeline

- Historically: Hardwired floating point operations, fixed point
- Now: Programmable, complex floating point operations
- 1 billion vertices / sec. & 50 billion fragments / sec.
Programmer’s View

- Contemporary pipeline

CPU → Vertex Processing → Rasterization → Fragment Processing → Display

Programmable
Programmer’s View

- Programming with “shaders”
Programming with Shaders

- Programming with “shaders”

- Attributes (per-vertex)
- Varying (per-vertex)
- Varying (per-fragment)
- Fragment Color (per-fragment)

- Vertex Shader
- Interpolation
- Fragment Shader
- Uniforms (constants)
Programming with Shaders

Attributes given per vertex

Vertex Shader computes \textit{varying}

Interpolate \textit{varying} values

Fragment Shader computes pixel color
Programmer’s View

- **Vertex Shader**

```cpp
uniform mat4 modelViewMatrix;
uniform vec3 lightPosition;
attribute vec3 pos;
attribute vec3 vertexNormal;

varying vec3 lightDirection, normal;

void main()
{
    vec4 vertexPosition = modelViewMatrix * pos;
    lightDirection = vec3(lightPosition - vertexPosition);
    normal = vertexNormal;
    gl_Position = vertexPosition;
}
```
Programmer’s View

- **Vertex Shader**

```glsl
uniform mat4 modelViewMatrix;
uniform vec3 lightPosition;
attribute vec3 pos;
attribute vec3 vertexNormal;

varying vec3 lightDirection, normal;

void main()
{
    vec4 vertexPosition = modelViewMatrix * pos;
    lightDirection = vec3(lightPosition - vertexPosition);
    normal = vertexNormal;
    gl_Position = vertexPosition;
}
```

Variables to be interpolated and passed to fragment shader
Programmer’s View

• Vertex Shader

uniform mat4 modelViewMatrix;
uniform vec3 lightPosition;
attribute vec3 pos;
attribute vec3 vertexNormal;

varying vec3 lightDirection, normal;

void main()
{
    vec4 vertexPosition = modelViewMatrix * pos;
    lightDirection = vec3(lightPosition - vertexPosition);
    normal = vertexNormal;
    gl_Position = vertexPosition;
}

Computing per-vertex attributes
Programmer’s View

- Fragment Shader

```cpp
varying vec3 lightDirection, normal;

void main()
{
    vec3 lightDirectionNormalized = normalize(lightDirection);
    float intensity = dot(lightDirectionNormalized, normal);
    gl_FragColor = vec4(intensity, intensity, intensity, 1.0);
}
```
Programmer’s View

- Fragment Shader

```cpp
varying vec3 lightDirection, normal;

void main()
{
    vec3 lightDirectionNormalized = normalize(lightDirection);
    float intensity = dot(lightDirectionNormalized, normal);
    gl_FragColor = vec4(intensity, intensity, intensity, 1.0);
}
```

Interpolated variables passed from vertex shader
Programmer’s View

• Fragment Shader

```cpp
varying vec3 lightDirection, normal;

void main()
{
    vec3 lightDirectionNormalized = normalize(lightDirection);
    float intensity = dot(lightDirectionNormalized, normal);
    gl_FragColor = vec4(intensity, intensity, intensity, 1.0);
}
```

Computing the color of this fragment (gl_FragColor)
Programmer’s View

- Fragment ≠ Pixel
- A fragment can store
  - Color
  - Position
  - Depth
  - Texture Coordinates
  - Window ID
  - …
Graphics APIs

- Application Programming Interfaces

- OpenGL
- RenderMan
- WebGL
- OpenGL ES
- Vulkan
- OpenVG
- 3dfx
- DirectX11
Graphics APIs

- Application Programming Interfaces
- Access to the graphics hardware
- Hardware-independent
- Abstract away complex details
Graphics APIs

- Application Programming Interfaces

- Describe the scene
  - Scene graph

- Sequence of drawing commands
  - More direct control
• **Open Graphics Library**
• Initially defined by Silicons Graphics Inc.
• Since 2006: managed by Khronos Group
• OpenGL Architectural Review Board (ARB)
• Open Graphics Library
• Platform-independent
• Available on many platforms

…more
• Updates automatic by GPU drivers
• Language bindings
  – C++ Java Ada Fortran Perl Python
• Versions and variants

OpenGL
OpenGL ES
OpenVG
WebGL
OpenCL