Real-Time Rendering
(Echtzeitgraphik)

Michael Wimmer
wimmer@cg.tuwien.ac.at
Walking down the graphics pipeline

Application → Geometry → Rasterizer
Understanding the rendering pipeline is the key to real-time rendering!

- Insights into how things work
  - Understanding algorithms
- Insights into how fast things work
  - Performance
Simple Graphics Pipeline

- Often found in text books
- Will take a more detailed look into OpenGL
Nowadays, everything part of the pipeline is hardware accelerated.

- Fragment: “pixel”, but with additional info (alpha, depth, stencil, ...)

Vienna University of Technology
Fixed Function Pipeline – Dataflow View

- Video memory:
  - Geometry
  - Commands
  - Textures
  - Frame buffer

- On-chip cache memory:
  - Pre-TnL cache
  - Vertex shading (T&L)
  - Post-TnL cache
  - Triangle setup
  - Rasterization
  - Texture cache
  - Fragment shading and raster operations

- System memory
- CPU
DirectX10 /OpenGL 3.2 Evolution

CPU
- Application
  - Driver
- Command
- Geometry
- Rasterization
  - Texture
  - Fragment
- Display

Geometry
- Vertex Buffer
  - Index Buffer
- Texture
  - Vertex Shader
  - Geometry Shader
- Buffer
  - Memory
  - Stream Out
- Setup/Rasterization
  - Texture
  - Pixel Shader
  - Output Merger

Vienna University of Technology
OpenGL 3.0

- OpenGL 2.x is not as capable as DirectX 10
  - **But**: New features are vendor specific extensions (geometry shaders, streams...)
  - GLSL a little more restrictive than HLSL (SM 3.0)

- OpenGL 3.0 did not clean up this mess!
  - OpenGL 2.1 + extensions
  - Geometry shaders are only an extension
  - New: depreciation mechanism

- OpenGL 4.x
  - New extensions
  - OpenGL ES compatibility!
DirectX 11/OpenGL 4.0 Evolution

Not the final place in the pipeline!!!
DirectX 11

- Tessellation
  - At unexpected position!
- Compute Shaders
- Multithreading
  - To reduce state change overhead
- Dynamic shader linking
- HDR texture compression
- Many other features...
Direct3D 10 pipeline

*Plus*

Three new stages for Tessellation

*Plus*

Compute Shader
Reduce driver overhead

- Indirect drawing
- Pipeline state objects
- Command lists/bundles
- Partly possible already in OpenGL 4.3+

Other features

- Conservative rasterization (for culling)
- New blend modes
- Order-independent transparency
Application

- Generate database (Scene description)
  - Usually only once
  - Load from disk
  - Build acceleration structures (hierarchy, ...)
- Simulation (Animation, AI, Physics)
- Input event handlers
- Modify data structures
- Database traversal
- Shaders (vertex, geometry, fragment)
Driver

- Maintain graphics API state
- Command interpretation/translation
  - Host commands → GPU commands
- Handle data transfer
- Memory management
- Emulation of missing hardware features

- Usually huge overhead!
  - Significantly reduced in DX10
Geometry Stage

- Command
- Vertex Processing
- Primitive Assembly
- Clipping
- Perspective Division
- Culling

- Tessellation
- Geometry Shading
Command buffering (!)

Command interpretation

Unpack and perform format conversion ("Input Assembler")

```c
glLoadIdentity( );
glMultMatrix( T );
glBegin( GL_TRIANGLE_STRIP );
glColor3f ( 0.0, 0.5, 0.0 );
glVertex3f( 0.0, 0.0, 0.0 );
glColor3f ( 0.5, 0.0, 0.0 );
glVertex3f( 1.0, 0.0, 0.0 );
glColor3f ( 0.0, 0.5, 0.0 );
glVertex3f( 0.0, 1.0, 0.0 );
glColor3f ( 0.5, 0.0, 0.0 );
glVertex3f( 1.0, 1.0, 0.0 );
glEnd( );
```
Vertex Processing

Transformation

Vertex Processing

object → eye → clip

Modelview Matrix → Projection Matrix

Perspective Division → Viewport Transform
Vertex Processing

- Fixed function pipeline:
  - User has to provide matrices, the rest happens automatically

- Programmable pipeline:
  - User has to provide matrices/other data to shader
  - Shader Code transforms vertex explicitly
    - We can do whatever we want with the vertex!
    - Usually a `gl_ModelViewProjectionMatrix` is provided
    - In GLSL-Shader: `gl_Position = ftransform();`
Vertex Processing

- Lighting
- Texture coordinate generation and/or transformation
- Vertex shading for special effects

Object-space triangles

\( T \)

Screen-space lit triangles
Tesselation

- If just triangles, nothing needs to be done, otherwise:
- Evaluation of polynomials for curved surfaces
  - Create vertices (tesselation)
- DirectX11 specifies this in hardware!
  - 3 new shader stages!!
  - Still not trivial (special algorithms required)
DirectX11 Tessellation

**Vertex Shader**
- Animate/skin Control Points

**Hull Shader**
- Transform basis, Determine how much to tessellate

**Control Shader**
- Tess Factors

**Tessellator**
- Tessellate!

**Evaluation Shader**
- Evaluate surface including displacement

- **Patch Control Points**
- **Transformed Control Points**
- **Control Points in Bezier Patch**
- **U V \{W\} Domain Points**
- **Sub-D Patch**
- **Bezier Patch**

Displacement Map
Tessellation Example

Sub-D Modeling  Animation  Displacement Map

Optimally tesslated!
Geometry Shader

- Calculations on a primitive (triangle)
- Access to neighbor triangles
- Limited output (1024 32-bit values) → No general tessellation!

Applications:
- Render to cubemap
- Shadow volume generation
- Triangle extension for ray tracing
- Extrusion operations (fur rendering)
Rest of Geometry Stage

- Primitive assembly
- Geometry shader
- Clipping (in homogeneous coordinates)
- Perspective division, viewport transform
- Culling
- Setup (per-triangle)
- Sampling (triangle = \{fragments\})
- Interpolation (interpolate colors and coordinates)
Rasterization

- Sampling inclusion determination
- In tile order improves cache coherency
- Tile sizes vendor/generation specific
  - Old graphics cards: 16x64
  - New: 4x4
    - Smaller tile size favors conditionals in shaders
    - All tile fragments calculated in parallel on modern hardware
Fragments represent “future” pixels

Pixel center at (2.5, 1.5)!
Rasterization – Rules

- Separate rule for each primitive
- Non-ambiguous!
- Polygons:
  - Pixel center contained in polygon
  - On-edge pixels: only one is rasterized
Texture “transformation” and projection
- E.g., projective textures

Texture address calculation (programmable in shader)

Texture filtering
- Texture operations (combinations, modulations, animations etc.)
Raster Tests

- **Ownership**
  - Is pixel obscured by other window?

- **Scissor test**
  - Only render to scissor rectangle

- **Depth test**
  - Test according to z-buffer

- **Alpha test**
  - Test according to alpha-value

- **Stencil test**
  - Test according to stencil buffer
Raster Operations

- Blending or compositing
- Dithering
- Logical operations

Textured Fragments → Framebuffer Pixels
After fragment color calculation ("Output Merger")

- Pixel Ownership Test
- Scissor Test
- Alpha Test
- Depth Test
- Stencil Test
- Depth Buffer
- Stencil Buffer
- Blending (RGB only)
- Dithering
- Logicop
- Frame Buffer
- Gamma correction
- Digital to analog conversion if necessary
Display

- Frame buffer pixel format: RGBA vs. index (obsolete)
- Bits: 16, 32, 128 bit floating point, ...
- Double buffered vs. single buffered
- Quad-buffered for stereo
- Overlays (extra bit planes) for GUI
- Auxiliary buffers: alpha, stencil
Functionality vs. Frequency

- Geometry processing = per-vertex
  - Transformation and Lighting (T&L)
  - Historically floating point, complex operations
  - Today: fully programmable flow control, texture lookup
  - 20-1500 million vertices per second

- Fragment processing = per-fragment
  - Blending and texture combination
  - Historically fixed point and limited operations
  - Up to 50 billion fragments ("Gigatexel"/sec)
  - Floating point, programmable complex operations
Assume typical non-trivial fixed-function rendering task

- 1 light, texture coordinates, projective texture mapping
- 7 interpolants \((z,r,g,b,s,t,q)\)
- Trilinear filtering, texture-, color blending, depth buffering

Rough estimate:

<table>
<thead>
<tr>
<th></th>
<th>ADD</th>
<th>CMP</th>
<th>MUL</th>
<th>DIV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex</td>
<td>102</td>
<td>30</td>
<td>108</td>
<td>5</td>
</tr>
<tr>
<td>Fragment</td>
<td>66</td>
<td>9</td>
<td>70</td>
<td>1</td>
</tr>
</tbody>
</table>
Communication Requirements

- **Vertex size:**
  - Position x, y, z
  - Normal x, y, z
  - Texture coordinate s, t
  → \(8 \cdot 4 = 32\) bytes

- **Texture:**
  - Color r, g, b, a, 4 bytes

- **Display:**
  - Color r, g, b, 3 bytes
  - Fragment size (in frame buffer):
    - Color r, g, b, a
    - Depth z (assume 32 bit)
  → 8 bytes, but goes both ways (because of blending!)
Communication Requirements

- **Application**: 0.640 GB/s
- **Command**: 4 GB/s
- **Geometry**: 1000 Mpix/s
- **Rasterization**: 120 Mpix/s
- **Texture**: 4 GB/s
- **Fragment**: 150 Gops
- **Display**: 0.36 GB/s
- **Framebuffer**: 16 GB/s
- **Texture Memory**: 16 GB/s

- **Vertex**: 5 Gops
- 20 Mver/s
- 1000 Mpix/s
- 120 Mpix/s

Vienna University of Technology