Real-Time Rendering (Echtzeitgraphik)

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

Application ➔ Geometry ➔ Rasterizer
What for?

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

Application → Geometry → Rasterizer → Display
Nowadays, everything part of the pipeline is hardware accelerated.

Fragment: “pixel”, but with additional info (alpha, depth, stencil, …)
Fixed Function Pipeline – Dataflow View

- On-chip cache memory:
  - pre-TnL cache
  - post-TnL cache
  - triangle setup
  - rasterization
  - fragment shading and raster operations

- Video memory:
  - geometry
  - commands
  - textures
  - frame buffer

- System memory:
  - CPU
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

- Tessellation
  - At unexpected position!
- Compute Shaders
- Multithreading
  - To reduce state change overhead
- Dynamic shader linking
- HDR texture compression
- Many other features...
DirectX 11 Pipeline

Direct3D 10 pipeline
*Plus*
Three new stages for Tessellation
*Plus*
Compute Shader
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
- Primitive generation
- Shaders (vertex, geometry, fragment)
Driver

- Maintain graphics API state
- Command interpretation/translation
  - Host commands $\rightarrow$ 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

Tesselation

Geometry Shading
Command buffering (!)

Command interpretation

Unpack and perform format conversion (“Input Assembler”)

```
glLoadIdentity( );
glMultMatrix( T );
glBegin( GL_TRIANGLES ) ;
glColor3f( 0.0, 0.5, 0.0 );
glVertex3f( 0.0, 0.0, 0.0 );
glColor3f( 0.5, 0.0, 0.0 );
glVertex3f( 0.0, 0.0, 0.0 );
glColor3f( 0.0, 0.5, 0.0 );
glVertex3f( 1.0, 0.0, 0.0 );
glColor3f( 0.5, 0.0, 0.0 );
glVertex3f( 0.0, 1.0, 0.0 );
glColor3f( 0.0, 0.5, 0.0 );
glVertex3f( 0.5, 0.0, 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 \texttt{gl\_Model\_View\_Projection\_Matrix} is provided
    - In GLSL-Shader: \texttt{gl\_Position = ftransform();}
Vertex Processing

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

Object-space triangles  \rightarrow  \begin{pmatrix} T \end{pmatrix}  \rightarrow  \text{Screen-space lit triangles}
Tessellation

- If just triangles, nothing needs to be done, otherwise:
- Evaluation of polynomials for curved surfaces
  - Create vertices (tessellation)
- 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

**tessellator**
- Tessellate!

**domain shader**
- Evaluate surface including displacement

- patch control points
- transformed control points
- control points in Bezier patch
- U V \{W\} domain points

Sub-D Patch

Beziers 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
Rasterization Stage

- Triangle Setup
- Rasterization
- Fragment Processing
- Texture Processing
- Raster Operations
Rasterization

- Setup (per-triangle)
- Sampling (triangle = \{fragments\})
- Interpolation (interpolate colors and coordinates)
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
Rasterization – Coordinates

- Fragments represent “future” pixels

Pixel center at (2.5, 1.5)!

Lower left corner of the window
Rasterization – Rules

- Separate rule for each primitive
- Non-ambiguous!
- Polygons:
  - Pixel center contained in polygon
  - On-edge pixels: only one is rasterized
Texture

- Texture “transformation” and projection
  - E.g., projective textures
- Texture address calculation (programmable in shader)
- Texture filtering

Fragments  ➔  Texture Fragments
Texture operations (combinations, modulations, animations etc.)

Texture Fragments → Textured Fragments
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
Raster Operations

- After fragment color calculation ("Output Merger")

Fragment and associated data

- Pixel Ownership Test
- Scissor Test
- Alpha Test
- Depth Test
- Stencil Test
- Depth Buffer
- Stencil Buffer
- Blending (RGBA only)
- Dithering
- Logicop

Frame Buffer
Display

- Gamma correction
- Digital to analog conversion if necessary
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
Computational Requirements

- 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>Vertex</th>
<th>ADD</th>
<th>CMP</th>
<th>MUL</th>
<th>DIV</th>
</tr>
</thead>
<tbody>
<tr>
<td>102</td>
<td>30</td>
<td>108</td>
<td>5</td>
<td></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
    \[ 8 \cdot 4 = 32 \text{ bytes} \]

- **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)
    \[ 8 \text{ bytes}, \text{ but goes both ways (because of blending!)} \]
Communication Requirements

- Application: 0.640 GB/s
- Command
- Geometry
- Rasterization
- Texture
- Fragment
- Display: 0.36 GB/s
- Texture Memory
- Framebuffer: 16 GB/s

- Vertex: 5 Gops
- Fragment: 150 Gops

- 20 Mvert/s
- 1000 Mpix/s
- 120 Mpix/s