Interactive Computer Graphics: Lecture 4

Graphics Pipeline and APIs

Some slides adopted from Markus Steinberger and Dieter Schmalstieg
The Graphics Pipeline: High-level view

• Declarative (What, not How)
  – For example virtual camera with scene description, e.g. scene graphs
  – Every object may know about every other object
  – Renderman, Inventor, OpenSceneGraph,...

• Imperiative (How, not What)
  – Emit a sequence of drawing commands
  – For example: draw a point (vertex) at position (x,y,z)
  – Objects can be drawn independant from each other
  – OpenGL, PostScript, etc.

• You can always build a declarative pipeline on top of imperative model
The Graphics Pipeline

Modelling Transformations

Illumination (Shading)

Viewing Transformation (Perspective / Orthographic)

Clipping

Projection (to Screen Space)

Scan Conversion (Rasterization)

Visibility / Display

Input:
- geometric model
- illumination model
- camera model
- viewport

drawing commands

imperative pipeline!

Output: 2D image for framebuffer display
<table>
<thead>
<tr>
<th>The Graphics Pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modelling Transformations</td>
</tr>
<tr>
<td>Illumination (Shading)</td>
</tr>
<tr>
<td>Viewing Transformation (Perspective / Orthographic)</td>
</tr>
<tr>
<td>Clipping</td>
</tr>
<tr>
<td>Projection (to Screen Space)</td>
</tr>
<tr>
<td>Scan Conversion (Rasterization)</td>
</tr>
<tr>
<td>Visibility / Display</td>
</tr>
</tbody>
</table>

- 3D models are defined in their own coordinate system
- Modeling transformations orient the models within a common coordinate frame (world coordinates)

![Diagram of the graphics pipeline](image)
## The Graphics Pipeline

<table>
<thead>
<tr>
<th>Modelling Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Illumination (Shading)</td>
</tr>
<tr>
<td>Viewing Transformation (Perspective / Orthographic)</td>
</tr>
<tr>
<td>Clipping</td>
</tr>
<tr>
<td>Projection (to Screen Space)</td>
</tr>
<tr>
<td>Scan Conversion (Rasterization)</td>
</tr>
<tr>
<td>Visibility / Display</td>
</tr>
</tbody>
</table>

- Vertices are lit (shaded) according to material properties, surface properties and light sources
- Uses a local lighting model

Graphics Lecture 4: Slide 5
### The Graphics Pipeline

<table>
<thead>
<tr>
<th>Modelling Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Illumination (Shading)</td>
</tr>
<tr>
<td>Viewing Transformation (Perspective / Orthographic)</td>
</tr>
<tr>
<td>Clipping</td>
</tr>
<tr>
<td>Projection (to Screen Space)</td>
</tr>
<tr>
<td>Scan Conversion (Rasterization)</td>
</tr>
<tr>
<td>Visibility / Display</td>
</tr>
</tbody>
</table>

- Maps world space to eye (camera) space (matrix evaluation)
- Viewing position is transformed to origin and viewing direction is oriented along some axis (typically $z$)
## The Graphics Pipeline

<table>
<thead>
<tr>
<th>Modelling Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Illumination (Shading)</td>
</tr>
<tr>
<td>Viewing Transformation (Perspective / Orthographic)</td>
</tr>
<tr>
<td>Clipping</td>
</tr>
<tr>
<td>Projection (to Screen Space)</td>
</tr>
<tr>
<td>Scan Conversion (Rasterization)</td>
</tr>
<tr>
<td>Visibility / Display</td>
</tr>
</tbody>
</table>

- Portions of the scene outside the viewing volume (view frustum) are removed (clipped)
- Transform to Normalized Device Coordinates

---

Graphics Lecture 4: Slide 7
The Graphics Pipeline

- Modelling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

- The objects are projected to the 2D imaging plane (screen space)
The Graphics Pipeline

- Modelling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

- Rasterizes objects into pixels
- Interpolate values inside objects (color, depth, etc.)
The Graphics Pipeline

<table>
<thead>
<tr>
<th>Modelling Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Illumination (Shading)</td>
</tr>
<tr>
<td>Viewing Transformation</td>
</tr>
<tr>
<td>(Perspective / Orthographic)</td>
</tr>
<tr>
<td>Clipping</td>
</tr>
<tr>
<td>Projection (to Screen Space)</td>
</tr>
<tr>
<td>Scan Conversion (Rasterization)</td>
</tr>
<tr>
<td>Visibility / Display</td>
</tr>
</tbody>
</table>

- Handles occlusions and transparency blending
- Determines which objects are closest and therefore visible
- Depth buffer

Graphics Lecture 4: Slide 10
What do we want to do?

- Computer-generated imagery (CGI) in **real-time**
- Very computationally demanding:
  - full HD at 60hz:
    - 1920 x 1080 x 60hz = 124 Mpx/s
  - and that’s just the output data

→ use specialized hardware for immediate mode (real-time) graphics
Solution

Most of real-time graphics is based on

- rasterization of graphic *primitives*
  - points
  - lines
  - triangles
  - ...

- Implemented in hardware
  - *graphics processing unit* (GPU)
  - controlled through an API such as OpenGL
  - certain parts of graphics pipeline are programmable, e.g. using GLSL
    → shaders
The Graphics Pipeline different view

• High-level view:

• “Vertex”
  – a point in space defining geometry

• “Fragment”:
  – Sample produced during rasterization
  – Multiple fragments are *merged* into pixels.
Application Stage

• Generate database
  – Usually only once
  – Load from disk
  – Build acceleration structures (hierarchy, …)

• Simulation
• Input event handlers
• Modify data structures
• Database traversal
• Utility functions
Application Stage

- Generate render area in OS
- Generate database
  - Usually only once
  - Load from disk
  - Build acceleration structures (hierarchy, …)
- Simulation
- Input event handlers
- Modify data structures
- Database traversal
- Utility functions
Application Stage

solid TEATEST
facet normal 0.986544E+00 0.100166E+00 0.129220E+00
outer loop
vertex 0.167500E+02 0.505000E+02 0.000000E+00
vertex 0.164599E+02 0.505000E+02 0.221480E+01
vertex 0.166819E+02 0.483135E+02 0.221480E+01
endloop
endfacet
facet normal 0.986495E+00 0.100374E+00 0.129434E+00
outer loop
vertex 0.166819E+02 0.483134E+02 0.221470E+01
vertex 0.169653E+02 0.483840E+02 0.000000E+00
vertex 0.167500E+02 0.505000E+02 0.000000E+00
endloop
Endfacet
...
Application Stage

solid TEATEST
facet normal 0.986544E+00 0.100166E+00 0.129220E+00
  outer loop
    vertex 0.167500E+02 0.505000E+02 0.000000E+00
    vertex 0.164599E+02 0.505000E+02 0.221480E+01
    vertex 0.166819E+02 0.483135E+02 0.221480E+01
  endloop
endfacet
facet normal 0.986495E+00 0.100374E+00 0.129434E+00
  outer loop
    vertex 0.166819E+02 0.483134E+02 0.221470E+01
    vertex 0.169653E+02 0.483840E+02 0.000000E+00
    vertex 0.167500E+02 0.505000E+02 0.000000E+00
  endloop
Endfacet
....
The Graphics Pipeline: OpenGL 3.2 and later

Source: www.lighthouse3d.com
The Graphics Pipeline: OpenGL 3.2 and later

- **Geometry Stage**
  - Vertex Shader
  - Primitive/Patch Assembly
  - Geometry Shader
  - Rasterization & Interpolation
  - Fragment Shader
  - Raster Operations

- **Rasterization Stage**
  - Tessellation Control
  - Tessellation Primitive Generation
  - Tessellation Evaluation
  - Buffers
The Graphics Pipeline: OpenGL 3.2 and later

- **Application**
  - vertices → **Vertex Shader** → transformed vertices → **Buffers**
  - transformed vertices → **Primitive/Patch Assembly**
  - primitive/patch connectivity → **Tessellation Control**
  - patches → **Tessellation Primitive Generation**
  - vertices and uv coordinates → **Tessellation Evaluation**
  - patch vertices and primitives → **Geometry Shader**
  - primitives → **Rasterization & Interpolation**
  - primitives → **Fragment Shader**
  - shaded fragments → **Raster Operations**
  - pixels → screen

- **Primitives (e.g. triangles)**
- **vertices + connectivity = primitives**
- **fragments = pixel candidates**
- **pixels**
Geometry Stage

Vertex Processing → Clipping → Projection → Viewport Transform

programmable

fixed function
Geometry Stage: Vertex Processing

• The input vertex stream
  – composed of arbitrary vertex attributes (position, color, …).

• is transformed into stream of vertices mapped onto the screen
  – composed of their clip space coordinates and additional user-defined attributes (color, texture coordinates, …).
  – clip space: homogeneous coordinates

• by the **vertex shader**
  – GPU program that implements this mapping.

• Historically, “Shaders” were small programs performing lighting calculations, hence the name.
Geometry Stage: Vertex Post-Processing

- Uses a common transformation model in rasterization-based 3D graphics:

![Diagram showing the geometry stage process](image)
Graphics Lecture 4: Slide 25

Geometry Stage: Vertex Post-Processing

- **Clipping**
  - Primitives not entirely in view are clipped to avoid projection errors

- **Projection**
  - Projects clip space coordinates to the image plane
  - Primitives in normalized device coordinates

- **Viewport Transform:**
  - Maps resolution-independent normalized device coordinates to a rectangular window in the frame buffer, the viewport.
  - Primitives in window (pixel) coordinates
Geometry Shader

- Optional stage between vertex and fragment shader
- In contrast to the vertex shader, the geometry shader has full knowledge of the primitive it is working on
  - For each input primitive, the geometry shader has access to all the vertices that make up the primitive, including adjacency information.
- Can generate primitives dynamically
  - Procedural geometry, e.g. growing plants
Rasterization Stage

Primitive Assembly → Primitive Traversal → Fragment Shading → Fragment Merging

- Fixed function
- Programmable
- Fixed function

Graphics Lecture 4: Slide 27
**Rasterization Stage**

- **Primitive assembly**
  - Backface culling
  - Setup primitive for traversal
- **Primitive traversal ("scan conversion")**
  - Sampling (triangle $\rightarrow$ fragments)
  - Interpolation of vertex attributes (depth, color, ...)
- **Fragment shading**
  - Compute fragment colors
- **Fragment merging**
  - Compute pixel colors from fragments
  - Depth test, blending, ...
**Rasterization – Coordinates**

Sample location ("pixel center") at (2.5, 1.5)!

Lower left corner of the window

Graphics Lecture 4: Slide 29
**Rasterization – Rules**

- Different rules for each primitive type
  - “fill convention”
- Non-ambiguous!
  - artifacts…
- Polygons:
  - Pixel center contained in polygon
  - Pixels on edge: only one is rasterized
Fragment Shading

• “Fragment”:
  – Sample produced during rasterization
  – Multiple fragments are *merged* into pixels.

• Given the interpolated vertex attributes,
  – output by the Vertex Shader

• the *Fragment Shader* computes color values for each fragment.
  – Apply textures
  – Lighting calculations
  – …

[Diagram: Fragments → Shaded fragments]
Fragment Merging

- Multiple primitives can cover the same pixel.
- Their Fragments need to be composed to form the final pixel values.
  - Blending
  - Resolve Visibility
    - Depth buffering

![Diagram of shaded fragments merging into frame buffer](image)
Fragment Merging

- Shaded Fragment
  - Pixel Ownership Test
  - Scissor Test
  - Stencil Test
  - Depth Test

- Blending
  - Frame Buffer

- Stencil Buffer
- Depth Buffer
Display Stage

- Gamma correction
- Historically: Digital to Analog conversion
- Today: Digital scan-out, HDMI encryption, etc.
Display Format

- Frame buffer pixel format: RGBA vs. index (obsolete)
- Bits: 16, 32, 64, 128 bit floating point, ...
- Double buffer vs. single buffer
- Quad-buffered stereo
- Overlays (extra bitplanes)
- Auxiliary buffers: alpha, stencil
Functionality vs. Frequency

• Geometry processing = per-vertex
  – Transformation and Lighting (T&L)
  – Historically floating point, complex operations
  – **Millions** of vertices per second
  – Today: Vertex Shader

• Fragment processing = per-fragment
  – Blending, texture combination
  – Historically fixed point and limited operations
  – **Billions** of fragments ("Gigapixel" per second)
  – Today: Fragment Shader
Architectural Overview

• Graphics Hardware is a shared resource
• User Mode Driver (UMD)
  – Prepares Command Buffers for the hardware
• Graphics Kernel Subsystem
  – Schedules access to the hardware
• Kernel Mode Driver (KMD)
  – Submits Command Buffers to the hardware
Unified Shader Model

- Since shader Model 4.0
- Unified Arithmetic and Logic Unit (ALU)
- Same instruction set and capabilities for all Shader types
- Dynamic load balancing geometry/fragment
- Floating point or integer everywhere
- IEEE-754 compliant
- Geometry Shader can write to memory
  - “Stream Output“
  - Enables multi-pass for geometry
Graphics APIs

Low-level 3D API

- OpenGL
  - Open Graphics Library (OpenGL) is a cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics.

- OpenGL ES
  - OpenGL for Embedded Systems is a subset of OpenGL

- DirectX, Direct3D
  - a graphics API for Microsoft Windows
Graphics APIs cont.

- Vulcan
  - OpenGL successor
  - targets high-performance realtime 3D graphics applications across all platforms
  - offers higher performance and lower CPU usage than older APIs.
- Mantle
  - low level graphics API by AMD. AMD will move to Vulcan
- Metal
  - low-level, low-overhead hardware-accelerated graphics and compute API by Apple (since IOS 8)
Graphics APIs cont.

- **RenderMan**
  - Interface Specification by Pixar Animation Studios
  - open API
  - describe three-dimensional scenes and turn them into digital photorealistic images.
  - It includes the RenderMan Shading Language.

- **WebGL**
  - JavaScript API for rendering interactive 3D computer graphics and 2D graphics within any compatible web browser without the use of plug-ins.
Graphics APIs cont.

High-level 3D API – declarative models
a lot! Java, SceneGraphs, performer, Irrlicht, mobile SDKs

e.g. SceneGraph APIs (openSG, openInventor, etc.)