Introduction to 3D Graphics

Processing Flow

Lan-Da Van (范倫達), Ph. D.
Department of Computer Science
National Chiao Tung University
Hsinchu, Taiwan

Fall, 2016
Outline

- 2D Game
- What is GPU? What can it do?
- GPU pipeline
  - Vertex shader
  - Geometry shader
  - Fragment shader
- Summery
Super Mario World (2D)

Graphics Processing Unit (GPU)

- Designed for real-time graphics
- Present in almost every PC
- Increasing realism and complexity

Logical Representation of Visual Information

Output Signal
Counter Strike (3D)

https://youtu.be/o6bPwxdKUnY

Motivation

- Real Time: 15 – 60 fps
- High Resolution
Motivation

• High CPU load
  – Physics, AI, sound, network, ...

• Graphics demand:
  – Fast memory access
    • Many lookups [vertices, normal, textures, ...]
  – High bandwidth usage
    • A few GB/s needed in regular cases!
  – Large number of flops
    • Flops = Floating Point Operations [ADD, MUL, SUB, ...]
    • Illustration: matrix-vector products
      \[
      (16 \text{ MUL} + 12 \text{ ADD}) \times (#\text{vertices} + #\text{normals}) \times \text{fps} =
      \]
      \[
      (28 \text{ Flops}) \times (6.000.000) \times 30 \approx 5\text{GFlops}
      \]

• **Conclusion: Real time graphics needs supporting hardware!**
Graphics Pipeline (1/2)

Application
- LOD selection
- Frustum Culling
- Portal Culling
- ...

Geometry Processing
- Modelview/Projection tr.
- Division by w
- Viewport transform
- Backface culling
- Clipping
- Primitive Assembly

Rasterization
- Scan Conversion
- Fragment Shading [Color and Texture interpol.]
- Frame Buffer Ops [Z-buffer, Alpha Blending, …]

Output
- Output to Device
Graphics Pipeline (2/2)

- Application
  - LOD selection
  - Frustum Culling
  - Portal Culling
  - ...

- Geometry Processing
  - VERTEX SHADER
    - Division by w
    - Viewport transform
    - Backface culling

- Rasterization
  - Scan Conversion
  - FRAGMENT SHADER

- Output
  - Output to Device
Vertex and Fragment Shaders

\[(x, y, z, w)\]
\[(nx, ny, nz)\]
\[(s, t, r, q)\]
\[(r, g, b, a)\]

VERTEX SHADER

\[(x', y', z', w')\]
\[(nx', ny', nz')\]
\[(s', t', r', q')\]
\[(r', g', b', a')\]

FRAGMENT SHADER

\[(x, y)\]
\[(r', g', b', a')\]
\[(depth')\]

\[(x, y)\]
\[(r, g, b, a)\]
\[(depth)\]
GPU Pipeline

Software

- Program/API
  - Driver

Hardware

- Vertex Processing
  - Vertex Shader
  - Geometry Engine (Front-End)

- Primitive Assembly
  - Geometry Shader

- Triangle Setup & Scan Conversion
- Depth Test
  - Fragment Shader
  - Fragment Processing
  - Rasterization Engine (Back-End)

- Framebuffer
Vertex Shaderv (1/4)
Vertex Shaders (2/4)

- Vertex Processing
  - Transform from “world space” to “image space”
  - Compute per-vertex lighting

From http://www.hourences.com/tutorials-vtx-lighting/
Vertex Shader (3/4)

Data for Rasterization

POSITION (x, y)
COLOR (R, G, B, Alpha)

Parameters for Pre-lighting

POSITION (x, y, z, w)
COLOR (R, G, B, Alpha)
Vertex Shader (4/4)

- One element in / one element out
- No communication
- Can select fragment address

**Input:**
- Vertex data (position, normal, color, …)
- Shader constants

**Output:**
- Required: Transformed clip-space position
- Optional: Colors, normals (data you want passed on to the pixel shader)

**Restrictions:**
- Can’t create new vertices
Geometry Shader (1/4)
Geometry Shader (2/4)

- Primitive Assembly
  - How the vertices connect to form a primitive
  - Per-primitive operations
Geometry Shader (3/4)

- One element in / 0 to ~ 100 out
  - Limited by hardware buffer sizes
- Like vertex:
  - No communication
  - Can select fragment address
- Input:
  - Entire primitive (point, line, or triangle)
  - Optional: Adjacency
- Output:
  - Zero or more primitives (a homogenous list of points/lines or triangles)
- Restrictions:
  - Allow parallel processing but preserve serial order
Applications

- Fur/fins, procedural geometry/detailing,
- Data visualization techniques,
- Wide lines and strokes, …
• Rasterizer

  – Convert geometric rep. (vertex) to image rep. (fragment)
    • Fragment = image fragment
      – Pixel + associated data: color, depth, stencil, etc.

  – Interpolate per-vertex quantities across pixels
Fragment Shader (1/5)

Program/API → Driver

Software

- Vertex Processing
  - Vertex Shader
- Primitive Assembly
  - Geometry Shader

Geometry Engine (Front-End)

Hardware

- Triangle Setup & Scan Conversion
- Depth Test
- Fragment Shader
- Fragment Processing
- Rasterization Engine (Back-End)
- Framebuffer
Bump/Displacement mapping

- Diffuse light without bump
- Height map
- Diffuse light with bump

Per-pixel displacement mapping
Fragment Shader (3/5)

Vertex Data
- DEPTH
- COLOR (R, G, B, alpha)
- POSITION (x, y)
- NORMAL
- TEXTURE CORD.

Fragment Shader

Pixel Data
- COLOR (R, G, B, alpha)
- DEPTH

TEXURES
Biggest computational resource

Cannot change destination address

No communication

Input:
- Interpolated data from vertex shader
- Shader constants, Texture data

Output:
- Required: Pixel color (with alpha)
- Optional: Can write additional colors to multiple render targets

Restrictions:
- Can’t read and write the same texture simultaneously
• Fragment processors (multiple in parallel)
  – Compute a color for each pixel
  – Optionally read colors from textures (images)
Summary

- GPUs convert 3D models into pixels.
- How GPUs work? => GPU pipeline
  - Vertex shaders transform the 3D position of each vertex in virtual space to the screen (2D) coordinate.
  - Geometry shaders do primitive assembly. It can also add and remove vertices from a mesh.
  - Fragment shaders generate output pixels.
Reference

● GPU Architecture & CG, Mark Colbert, 2006
● Introduction to Graphics Hardware and GPUs, Yannick Francken, Tom Mertens
● GPU Tutorial, Yiyunjin, 2007
● Evolution of GPU and Graphics Pipelining, Weijun Xiao
● Referencing SIGGRAPH 2005 Course Notes from David Luebke
● Adapted from: David Luebke (University of Virginia) and NVIDIA

● Acknowledgement:
  ● Thanks for TA’s help for preparing the material.