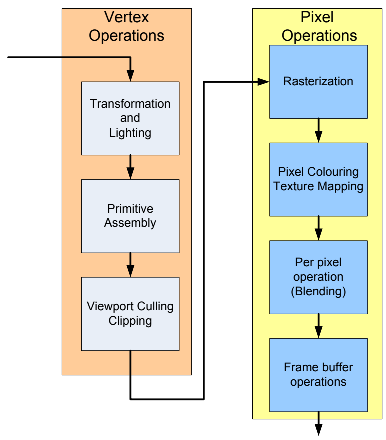
Shaders

(some slides taken from David M. course)

Doron Nussbaum

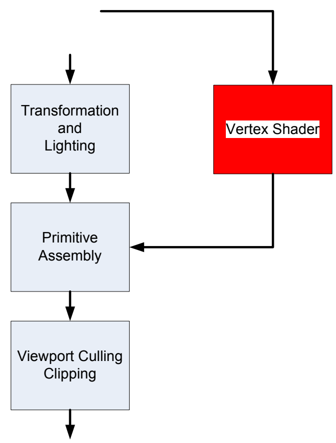
Traditional Rendering Pipeline

- Traditional pipeline (older graphics cards) restricts developer to texture and the 3-term lighting model
- Pixel information was interpolated to vertices
- Shaders
  - Lift the restrictions
  - Allow developers to manipulate outcome
- Result
  - vertex shaders
  - Pixel/fragment shaders



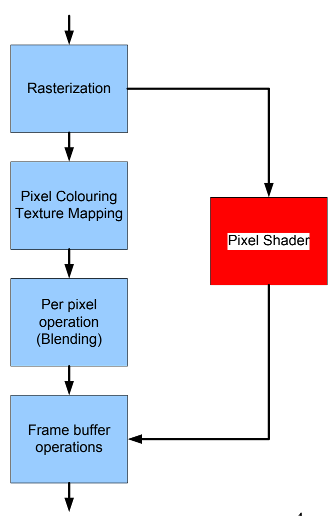
Vertex Shader

- Act in parallel across all vertices
- Responsible for transforming
  - Initial lighting of vertex
  - Vertex transformation
  - Normal transformation & normalization
  - Texture coordinate generation & transformation
- Compute temporary variables to be consumed by fragment shader
- Per-vertex processing
  - only information from one vertex available



Fragment (pixel) Shader

- Act in parallel across all fragments
- Responsible for computing depth and color information
- Only information from current fragment available
  - Operations on interpolated values
  - Texture access (multipass shaders)
- Produces Colour

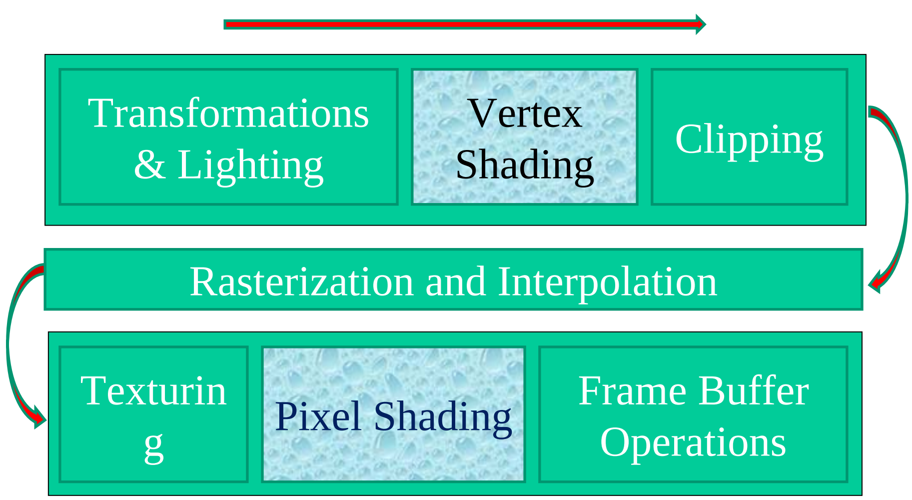


Shading languages (last)

- High level shading languages available
- Three main ones:
  - Cg (NVIDIA's "C for graphics")
  - HLSL (MS's "high level shading language")
  - GLSL (OpenGL's shading language)

- All quite similar, differ in details of API
- DirectX – shader code is added during runtime
  - Shader code is written in a separate file
  - Requires “transfer” of data to the shader code

The Graphics Pipeline



Vertex Shader(s)

- Replaces fixed functionality of vertex processor
- Normally get:
  - vertex transformation
  - normal transformation
  - illumination
- Now, can write programs that do anything
  - same program on all vertices

Vertex Shaders

- Input:
  - Built-in attributes
    - color
    - normal
    - position
    - texture coordinate
  - User-defined attributes
  - Texture maps

Vertex Shaders

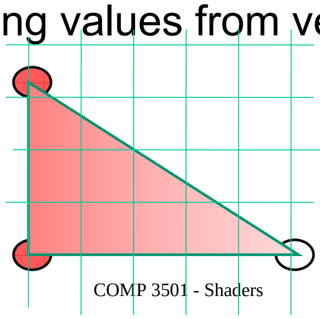
- Transform and light a single vertex
- Output:
  – position
  – color
  – user-defined variables (for fragment shader)

Vertex Shader

- Changing vertex position in a shader
  – create simple animations
    - Use time as input, compute $x(t)$, $\theta(t)$
  – perform displacement mapping (to mesh resolution)
  – Generally, modify the geometry at runtime

Rasterization and Interpolation

- Turns the space between vertices into fragments
- Computes input values for fragments by interpolating values from vertices



Pixel Shader

- Operates on fragments
- Input:
  - color
  - texture coordinates
  - user-defined variables (from vertex shader)
  - texture maps
- Cannot access other pixels
  - although, texture can be exploited for this

Pixel Shader

- Output:
  - color (RGBA)
  - depth
- Note, cannot change position at this point
- But, can compute color in sophisticated way, in parallel at each pixel

Shader input

- Previously mentioned inputs available per-vertex (per-pixel)
- Also have shader parameters that can be set (like global variables)
  - E.g., direction of sun, location of viewer (same for all primitives)
  - E.g., specific parameters (material glossiness, what texture to use)

Phong and Gouraud Shading

- Traditional Gouraud shading: per-vertex lighting, interpolated to pixels
- Various problems:
  - Visible mesh boundaries
  - Strange effects at low mesh resolution
- Phong shading: normals interpolated, per-pixel lighting: done in pixel shader

Render to Texture

- Shaders are "memoryless"
- Also, limit to how many instructions in shader (varies with card)
- Can render display buffer to texture, then use texture as input to later (or same) shader
  - Or, just display texture on object
    - eg, mirror

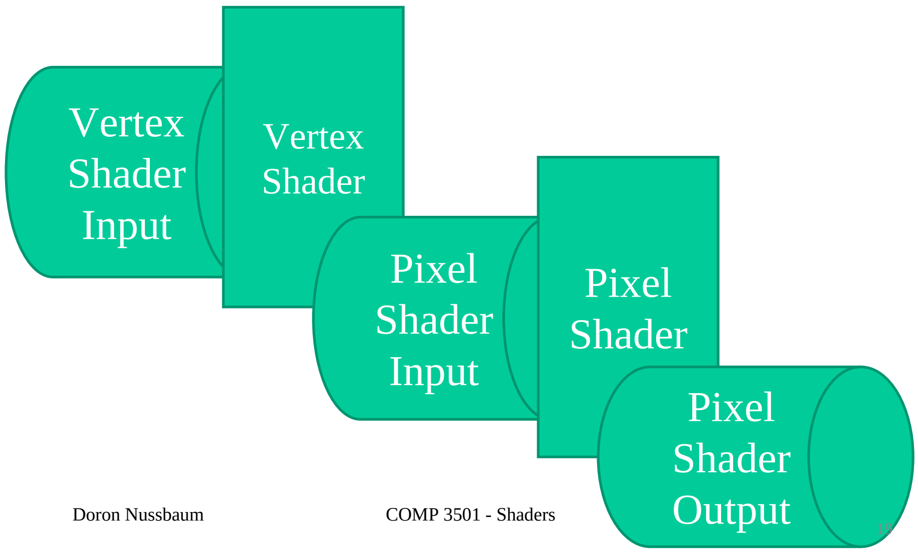
Render to Texture

- Simple postprocessing effects:
  - render to texture
  - apply pixel shader involving texture
  - draw texture to screen
    - Create quad covering screen, apply texture
- Color modification, darken/brighten, fade/dissolve

Shaders in DirectX

- Write a single .fx file with:
  - list of shader parameters
  - structure definitions for shader I/O
  - vertex shader
  - pixel shader
  - technique and pass definition

Data flow



Shader I/O Semantics

- Markers that suggest how data is passed into and out of the shaders
  - e.g., put "position" data in POSITION semantic
- Example vertex shader input:

```
struct vs_in {
  float4 pos : POSITION0;
  float4 col : COLOR0;
};
```

Available semantics

- Differ depending on what the structure is
  - position semantic not available for pixel shader output, for example
- TEXCOORD semantic for user-defined variables
- Can be used multiple times: TEXCOORD0 for first, then TEXCOORD1, TEXCOORD2...

Vertex Shader Input Semantics

- POSITION – vertex location in space
- COLOR – for the vertex color
- NORMAL – surface normal
- TEXCOORD – texture coordinates, also "generic"
- PSIZE – point size (used with point sprites, chiefly for particle system effects)

Vertex Shader Output Semantics

- subset of the inputs

- POSITION
- COLOR
- TEXCOORD
- PSIZE

Pixel Shader Input Semantics

- Only COLOR and TEXCOORD are allowed
- Note: considered good practice to use vertex shader output as pixel shader input – but permitted semantics different!
  - notably, POSITION required for vs_out, prohibited by ps_in
- "Illegal" semantics ignored by compiler in definition, but DO NOT USE in ps code

Pixel Shader Output Semantics

- COLOR
- DEPTH – depth value for depth test

- In routine operation, one color value (ultimate pixel color) and one depth value
- Can have more outputs if multipass shader, unconventional operation

Techniques and Passes

```
technique mytechnique {
  // one or more passes, ordinarily one
  pass mypass {
    vertexshader = compile vs_1_1 myvs();
    pixelshader = compile ps_2_0 myps();
  }
}
```

Shader Code

- Written in C-like HLSL
  - if, for, = for assignment...
- scalar, vector, and matrix types
  - int, float
  - float2, float3, float4
  - float4x4
- Texture type (2D texture)

Intrinsic Functions

- Some functions are built in
- Various mathematical functions
  - math (exp, pow, log)
  - trigonometric (sin, cos, tan, asin...)
  - vector (dot, cross)
  - "housekeeping" (clamp, lerp, abs, max)
- Functions for dealing with texture
- Partial list (common ones) in textbook

Shader Code

- Take control of the pipeline
- Very abstract right now, but examples to come in the following weeks
  - texture
  - procedural texture
  - basic lighting (Phong shading)
  - specialized lighting
  - particle systems

Connecting your Shader

- May need a custom vertex format, if your vertex shader input is different from all existing vertex formats
- Set all your parameters
- Link to project, configure

Recap

- Can program parts of the graphics pipeline
  - vertex shader
    - output is position, color, texture coordinates
  - pixel shader
    - output is color
- for DirectX, use HLSL, a C-like language with many built-in functions

Recap

- Things to know:
  - graphics pipeline
    - tasks of vertex and pixel shaders
    - rasterization and interpolation
  - vertex and pixel shader input and output semantics
- To learn by practice: writing your own shaders