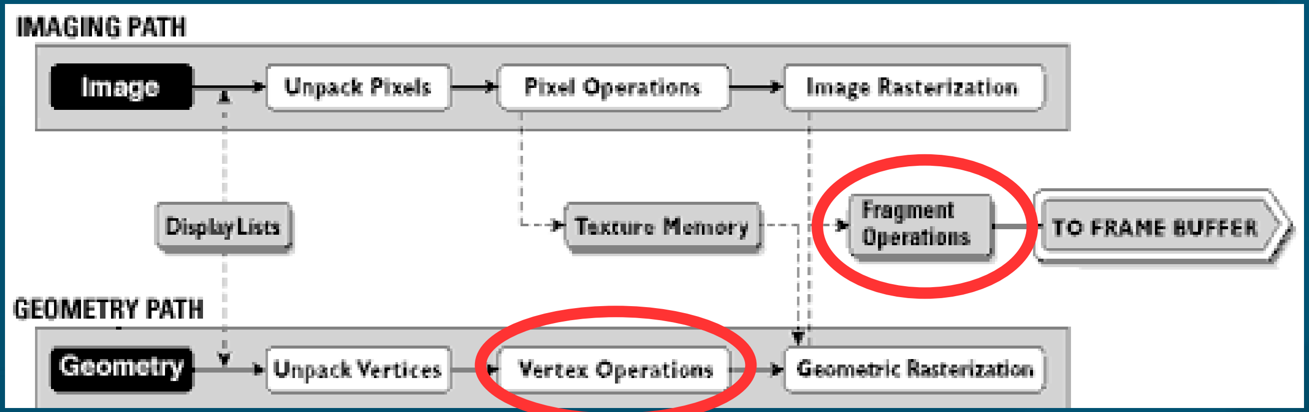
Programmable Shaders

17 March 2009
CMPT370
Dr. Sean Ho
Trinity Western University

Review of rendering pipeline

- OpenGL rendering pipeline:



- Vertex operations:
  - Transform points via model-view matrix
  - Normals, other per-vertex data
- Fragment operations:
  - Shading: colour for each pixel of fragment

Vertex processing: input

- Vertex processing operates per-vertex
  - Mostly geometric operations
  - Vertices may come from program, display list, GL evaluator
- Input:
  - (x,y,z,w) coords of vertex: glVertex
  - Normal vector: glNormal
  - Texture coordinates: glTexCoord
  - RGBA colour, material properties, GL state
  - Other user-defined data via GLSL

Vertex processing: tasks

- Transform vertex location: model-view matrix
- Transform normals, too!
  - What if model-view matrix has scaling?
- Vertex colour if desired
- Auto-generate texture coordinates if needed, and apply texture matrix
  - Maps from texture coords to object coords
- Any other per-vertex calculations desired
  - e.g., calculate other vectors needed for lighting model

Primitive assembly

- The output of the vertex processing is: transformed vertices in camera coordinates
- Next steps in pipeline:
  - Vertices assembled into objects (topology)
  - Transformed into 2D by projection matrix
    - Perspective involves a division
  - Clipped:
    - Against user-defined planes
    - Against the view volume
    - May produce new vertices

Rasterization

- The next step in the pipeline is rasterization
- Produces fragments: partial contributions of each primitive to the final image
  - Each fragment is a "potential pixel"
    - May be occluded or blended
      - Fragment tests come afterward
  - Each fragment has:
    - Colour
    - Depth value (possibly)
    - Texture coordinates (if needed)

Fragment processing

- Fragment processing operates per-fragment
  - More intense than per-pixel!
- Input:
  - Colour/material properties
  - Texture coordinates
  - Any user-defined data via GLSL
  - Vertex values have been interpolated over the primitive by the rasterizer
- Output: final colour of fragment according to shading model

Programmable shaders

- Shaders are programs run by the GPU to implement parts of the graphics pipeline
    - First introduced by NVIDIA's GeForce 3
- We are programming on a dedicated GPU chip
- What language to use?
    - Early models: form of assembly
    - NVIDIA's Cg uses C-like syntax
    - Microsoft DirectX 8, 9, 10, HLSL
    - OpenGL ARB extensions
    - GLSL incorporated as a part of OpenGL2.1

Fixed-function vs. programmable

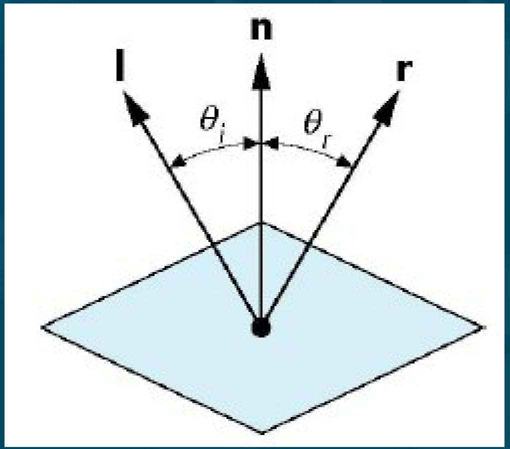
- Fixed-function pipeline:
  - Standard, widely compatible, easy to learn
  - Gouraud shading: lighting model done only per-vertex, not per-pixel
  - Limited number of lights
- Programmable:
  - Enhanced functionality per-vertex/per-frag
  - Parallel on GPU; built-in vector/matrix math
  - Must replace functionality of fixed pipeline
  - Debugging!

e.g.: Phong lighting

- Let's use vertex+fragment shaders to implement per-pixel Phong shading
  - Default is per-vertex Gouraud lighting
- (Shade) = (Ambient) + (Diffuse) + (Specular)
  - $I = k_a I_a + k_d I_d (l * n) + k_s I_s (v * r)^{\alpha}$
- Need vectors l (to light) and r (reflection)
- Use vertex shader to calculate l, n
  - Rasterizer will interpolate these vectors
- Use fragment shader to calc r and do Phong shading

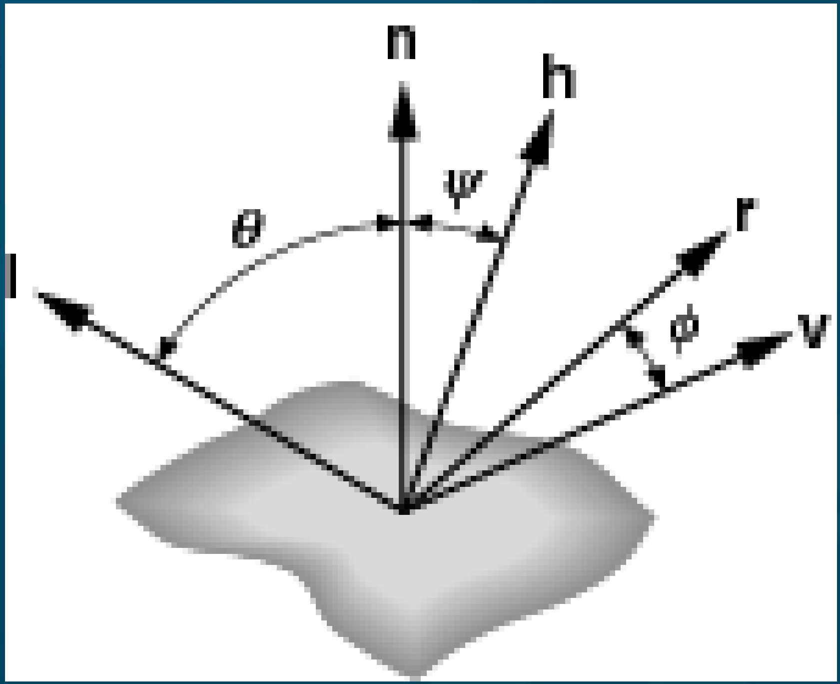
Calculating the reflection vector

- Calculate l vector to light: (light_pos) – (vertex)
- Calculate r (reflection) vector:
  - $\cos(\theta_l) = \cos(\theta_r)$, so l*n = r*n
  - r, n, l are all coplanar, so r = a(l)+b(n)
  - All normal vectors, so r*r = n*n = l*l = 1
  - Solve: r = 2(l*n)n - l



Blinn's halfway vector

- Blinn proposed a simplified model of specularity: instead of $k_s\ I_s\ (v * r)^{\alpha}$, we use $k_s\ I_s\ (n * h)^{\alpha}$:
- Replace v*r with n*h, where h is the unit halfway vector: $h = (l+v) / 2$
- Normalize h: $(l+v) / |l+v|$
- If n, l, and v are coplanar: then psi = phi/2
- The exponent needs to be adjusted



Vertex shader program

- varying vec3 N, L;
- void main() {
-    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
-    N = gl_NormalMatrix * gl_Normal;
-    L = gl_LightSource[0].position.xyz;
-    gl_FrontColor = vec4(0.5, 0.5, 0.8, 1.0);
- }

- Input: gl_Vertex, gl_Normal
- Output: gl_Position (eye coords), N, L (send to fragment shader)