UCL

## Graphics Processing Units (GPUs)

*V1.2 (January 2010)*

Anthony Steed

*Based on slides from Jan Kautz (v1.0)*

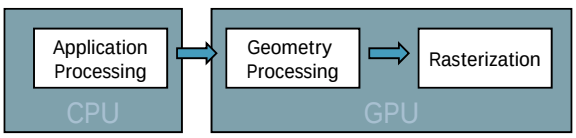
UCL

## Outline

1. GPU Pipeline:
   - Transformations
   - Lighting
   - Rasterization
   - Fragment Operations
2. Vertex Shaders
3. Pixel Shaders
4. Example: Per-Pixel Lighting
5. Geometry Shaders

UCL

## High-Level Pipeline




- What sort of tasks does each stage perform?

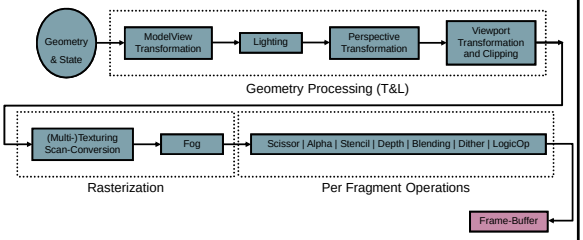
UCL

## High-Level Pipeline

| Application | Geometry (a.k.a. "vertex pipeline") | Rasterization (a.k.a. "pixel pipeline" or "fragment pipeline") |
|---|---|---|
| Handle input | Transform | Rasterize (fill pixels) |
| Simulation & AI | Lighting | Interpolate vertex parameters |
| | | Look up/filter textures |
| Culling | Skinning | Z- and stencil tests |
| LOD selection | Calculate texture coords | Blending |
| Prefetching | | |

UCL

## GPU Pipeline – OpenGL




- Immediate mode API
- State based

UCL

## Transform (OpenGL)

- ModelView Transformation
  - Tranform geometry from world coordinates into eye coords
  - Includes camera positioning and object rotation/translation
  - Single model-view matrix in OpenGL (combined)

UCL

## Transform (OpenGL)

- Perspective Transformation
  - Project eye coordinates into screen-plane (perspective or orthogonal).
  - Simple specification
    - glFrustum(left, right, bottom, top, near, far)
    - gluPerspective( fovy, aspect, near, far )
    - glOrtho(left, right, bottom, top, near, far)
- Viewport Transformation
  - glViewport(x, y, width, height)
- Clipping
  - Done automatically

UCL

## Lighting (Fixed Function Pipeline)

- Lighting
  - If turned off: specified per-vertex color is interpolated
  - Up to 8 light sources
  - Lighting model: physically incorrect Blinn-Phong
    - Ambient: $k_a$ (independent of light source & viewer)
    - Diffuse: $k_d$ * (**L**.**N**)
    - Specular: $k_s$*(**H**.**N**)$^n$, **H** = **L**+**V**/(||**L**||+||**V**||)

UCL

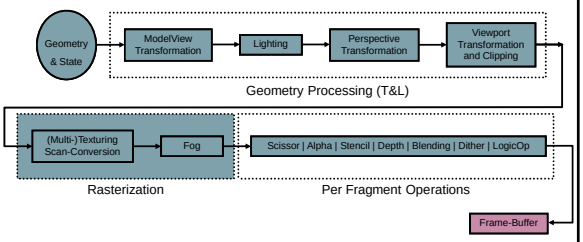
## Lighting (Fixed Function Pipeline)

- Lighting
  - **Only** per-vertex (for every light source, in eye-coord):
    - $k_a$ + I/a(r) * ($k_d$*(**L**.**N**) + $k_s$*(**H**.**N**)$^n$)
    - I: intensity of light source, r: distance to light, a(r) = 1, r, or $r^2$
    - Interpolate result between vertices
    - No per-pixel shading ⇒ coarse mesh: highlight artifacts
    - Extension for per-pixel shading
  - Types of lights:
    - Point lights, spot lights, directional lights

## OpenGL Lighting Example

```
float mat_ambient[] = {0.1, 0.1, 0.1, 1.0};
float mat_shininess[] = {20.0};
float mat_specular[] = {0.1, 0.1, 0.1, 0.0};
float mat_diffuse[] = {0.7, 0.7, 0.7, 0.0};
float light0_ambient[] = {1, 1, 1, 1.0};
float light0_diffuse[] = {1.0, 1.0, 1.0, 0.0};
float light0_position[] = {0, 6, 6, 1.0}; // [3] decides whether directional or positional
float light0_specular[] = {1.0, 1.0, 1.0, 0.0};
glLightModeli( GL_LIGHT_MODEL_TWO_SIDE, GL_FALSE );
// Light definition
glLightfv(GL_LIGHT0, GL_AMBIENT, light0_ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light0_diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, light0_specular);
glLightfv(GL_LIGHT0, GL_POSITION, light0_position);
// Material definition (for subsequent polygons)
glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
// Turn it on
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
```

## GPU Pipeline




## Texturing

- Texturing
  - 1D, 2D, 3D textures
  - Contains:
    - Color (RGB) + Alpha (default: 8888 bits)
    - Luminance (L) + Alpha (default: 88 bits)
    - Latest GPUs: support 16/24/32bit floating point textures
  - Filtering: nearest neighbor, linear, mip-mapping
    - Latest GPUs: floating point filtering on NV, not ATI

UCL

## Texturing

- Texture Coordinates
  - By hand (for every vertex)
  - Generated automatically (by OpenGL)
    - Object coordinates: linear combination of object coords.
    - Eye coordinates: linear combination of eye coords.
    - E.g.: draw height contours: z-coord as texcoord of 1D texmap
  - Texture matrix:
    - 4x4 matrix applied to tex.-coords. before they are used
    - E.g.: project texture onto geometry (slide projector)

UCL

## Lighting+Texturing (OpenGL)

- Lighting and Texturing:
  - GL_REPLACE: replace lighting with texture
  - GL_MODULATE:
    - multiply light with texture
    - ok for diffuse lighting, but specular part should be added later:
      - $L_{a,d}$ * texture + $L_s$
      - Supported via extension
  - GL_BLEND:
    - Blend between fragment, texture and environment color:

$$C = C_f(1-C_t) + C_eC_t$$

UCL

## OpenGL Texturing Example

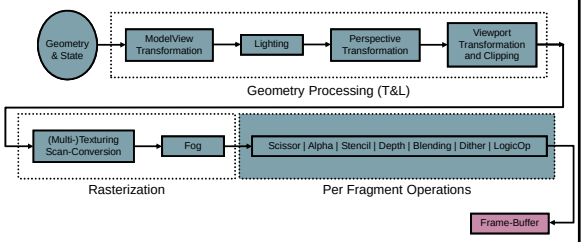
```
// bind texture
glGenTextures( 1, &texname );
glBindTexture( GL_TEXTURE_2D, texname );

// initialize parameters
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP );
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );

// how do we combine texture with lighting/color
glTexEnvi( GL_TEXTURE_2D, GL_TEXTURE_ENV_MODE, GL_MODULATE );

// specify texture
glTexImage2D( GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
              GL_RGB, GL_FLOAT, buffer );
// turn it on
glEnable( GL_TEXTURE_2D );
```

## Fog

- Objects that are further away ⇒ fade to fog
- Affects transformed, lit, and textured objects
- User controls:
  - Fog density, Fog color $C_f$
  - Fog increase (depending on z-value): linear, exp, exp^2
  - Fog start and end
  - Hardware automatically computes fog factor *f*
- Blended together with actual color:

  $C = f * C_i + (1-f) * C_f$, where *f* is fog factor
- Cannot do volumetric fog

## GPU Pipeline




## Fragment Operations

- Fragment: color + alpha + depth
- Scissor-Test
  - Specify screen-rectangle: fragments only drawn in there
- Alpha-Test
  - Test source alpha (incoming fr.) against reference alpha value
  - Depending on result: discard / keep fragment
  - Tests: <, >, <=, >=, =, !=, always, never
  - E.g.: textures for tree leaves

Operations: Scissor | Alpha | Stencil | Depth | Blending | Dither | LogicOp

UCL

## Fragment Operations

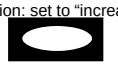
- Stencil-Test: stencil buffer with N bits information
  - Test stencil value against reference value *S* (see above tests)
  - Depending on result
    - discard / keep fragment
  - Depending on result and on depth test result (zfail/zpass):
    - increase / decrease / zero / keep / invert / set stencil value
- Use: e.g. render to certain area of screen only



UCL

## Stencil Example

- Stencil-test example: render to certain area
  - Algorithm:
    - Alpha-Test: only let fragments pass with alpha = 1
    - Stencil-Test: set to “always”, stencil function: set to “increase”
    - Render polygon with alpha-texture



    - Result: Stencil bit is 1, where incoming alphas were 1
    - Stencil-Test: set to “pass if stencil=1”, function to “keep”
    - Render actual things that should appear in mirror
    - Result:

UCL

## Fragment Operations

- Depth-Test
  - Test source depth against destination depth
  - Depending on result: discard / keep fragment
  - Tests: <, >, <=, >=, =, !=, always, never
- Newer hardware: early z-reject
  - Before fragment is shaded (i.e. textured, etc…)
  - If test fails, fragment is killed
  - GPUs keep a separate buffer for that!
  - Helps tremendously for some applications

UCL

## Fragment Operations

- Frame-buffer blending
  - Combine source fragment with destination fragment
  - Blending equation:

$$C = C_s*S + C_d*D$$

    - $C_s$ = source color, $C_d$ = destination color
    - S = source factor, D = destination factor
    - S/D = zero, one, dst_color, src_color, src_alpha, …
  - Extension: change "+" to "-" in equation
  - All standard imaging operators possible
  - E.g.: See through a window with dark glass



UCL

## Fragment Operations

- Dither
  - Dithering is needed if frame buffer has less bits than incoming fragment
  - By default turned on!
  - Not sure, if hardware nowadays even supports this anymore…
  - Enable/disable



UCL

## Fragment Operations

- Logic Operators
  - Legacy feature
  - Logic operations between incoming fragment and destination fragment.
  - Done for every color channel separately
  - Operations:
    - clear (=0), copy (=src), noop (=dst), set (=1)
    - copy invert, invert, and, or, nand, nor, xor, …

UCL

**Frame Buffer**

- Color Buffer: RGB + alpha (32bit)
- Depth Buffer: z-value (24bit)
- Stencil Buffer: stencil value (8bit)
- Standard depths:
  - 32bit color
  - 24bit depth
  - 8bit stencil
- Only very few bits, especially if used for multi-pass rendering

UCL

**Frame Buffer – Latest GPUs**

- Color Buffer: RGB + alpha (128bits, floating point)
- Depth Buffer: z-value (24bit)
- Stencil Buffer: stencil value (8bit)
- Color Buffer = special buffer
  - Usually only for storing intermediate results
  - Use as textures in next pass

UCL

**Outline**

1. GPU Pipeline:
   - Transformations
   - Lighting
   - Rasterization
   - Fragment Operations
2. Vertex Shaders
3. Pixel Shaders
4. Example: Per-Pixel Lighting
5. Geometry Shaders

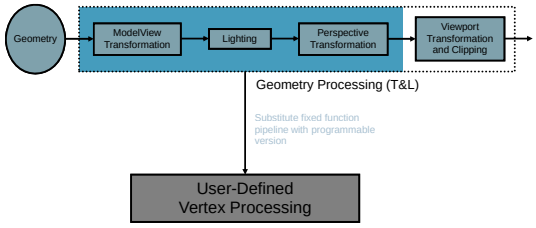
UCL

## Pixel Shaders and Vertex Shaders

- Are different “processors”, using subtly different machine code
- Programmers rarely program the machine code
  - High-Level Shading Language (DirectX)
  - CG (Nvidia – similar to HLSL, works in both DirectX and OpenGL)
  - GL Shading Language

UCL

## Vertex Shaders




- Vertex Shader has to do everything (except clipping)

UCL

## Vertex Shaders: What for?

- Custom transform, lighting, and skinning

UCL

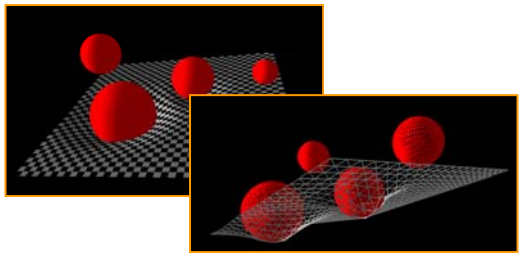
## Vertex Programs: What for?

- Custom cartoon-style lighting

UCL

## Vertex Shaders: What for?

- Dynamic displacements of surfaces by objects



UCL

## Vertex Shaders

- What is a vertex shader?
  - A small program running on GPU for each vertex
  - GPU instruction set to perform all vertex math
  - Reads an untransformed, unlit vertex
  - Creates a transformed vertex
  - Optionally …
    - Lights a vertex
    - Creates texture coordinates
    - Creates fog coordinates
    - Creates point sizes
    - Latest GPUs: can read from texture

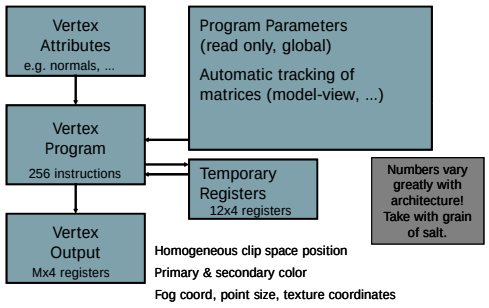
UCL

## Vertex Shaders

- What does it not?
  - Does not create or delete vertices (1 in ⇒ 1 out)
  - No topological information provided
- This has changed with next generation
  - Has an additional geometry shader (after vertex shader)
  - Can create/delete polygons at that stage.

UCL

## Vertex Shaders




UCL

## Vertex Shaders

- Assembly Language
  - Powerful SIMD instruction set
  - Four operations simultaneously
  - Many instructions (even branch & loops now!)
  - Operate on scalar or 4-vector input
  - Result in a vector or replicated scalar output
- Instruction Format:
  Opcode dst, [-]s0 [,[-]s1 [,[-]s2]];#comment

Instruction name | Destination Register | Source0 Register | Source1 Register | Source2 Register

UCL

## Vertex Shaders

- Simple Example:

```
MOV    R1, R2;
```


**before**

| R1 | | R2 | |
|---|---|---|---|
| 0.0 | x | 7.0 | x |
| 0.0 | y | 3.0 | y |
| 0.0 | z | 6.0 | z |
| 0.0 | w | 2.0 | w |

**after**

| R1 | | R2 | |
|---|---|---|---|
| 7.0 | x | 7.0 | x |
| 3.0 | y | 3.0 | y |
| 6.0 | z | 6.0 | z |
| 2.0 | w | 2.0 | w |

UCL

## Vertex Shaders

- Input Mapping:
  - Negation
  - Swizzling
  - Smearing
- Example:

```
MOV    R1, -R2.yzzx;
```

**before**

| R1 | | R2 | |
|---|---|---|---|
| 0.0 | x | 7.0 | x |
| 0.0 | y | 3.0 | y |
| 0.0 | z | 6.0 | z |
| 0.0 | w | 2.0 | w |

**after**

| R1 | | R2 | |
|---|---|---|---|
| -3.0 | x | 7.0 | x |
| -6.0 | y | 3.0 | y |
| -6.0 | z | 6.0 | z |
| -7.0 | w | 2.0 | w |

UCL

## Vertex Shaders: Instruction Set

- **ARL** address relative
- **MOV** move
- **MUL** multiply
- **ADD** add
- **MAD** multiply and add
- **RCP** reciprocal
- **RSQ** recipr. sqrt
- **DP3** dot product 3
- **DP4** dot product 4
- **DST** distance
- **MIN** minimum
- **MAX** maximum
- **SLT** set on less than
- **SGE** set on greater than
- **EXP** exponential (base 2)
- **LOG** logarithm
- **LIT** do lighting
- **...** and many more

UCL

## Vertex Shaders
### Simple Specular and Diffuse Lighting

```
!!VP1.0
#
# c[0-3]  = modelview projection (composite) matrix
# c[4-7]  = modelview inverse transpose
# c[32]   = eye-space light direction
# c[33]   = constant eye-space half-angle vector (infinite viewer)
# c[35].x = pre-multiplied monochromatic diffuse light color & diffuse mat.
# c[35].y = pre-multiplied monochromatic ambient light color & diffuse mat.
# c[36]   = specular color
# c[38].x = specular power
# outputs homogenous position and color
#
DP4   o[HPOS].x, c[0], v[OPOS];      # Compute position.
DP4   o[HPOS].y, c[1], v[OPOS];
DP4   o[HPOS].z, c[2], v[OPOS];
DP4   o[HPOS].w, c[3], v[OPOS];
DP3   R0.x, c[4], v[NRML];           # Compute normal.
DP3   R0.y, c[5], v[NRML];
DP3   R0.z, c[6], v[NRML];           # R0 = N' = transformed normal
DP3   R1.x, c[32], R0;               # R1.x = Ldir DOT N'
DP3   R1.y, c[33], R0;               # R1.y = H DOT N'
MOV   R1.w, c[38].x;                 # R1.w = specular power
LIT   R2, R1;                        # Compute lighting values
MAD   R3, c[35].x, R2.y, c[35].y;    # diffuse + ambient
MAD   o[COL0].xyz, c[36], R2.z, R3;  # + specular
END
```

UCL

## Vertex processor capabilities

- 4-vector FP32 operations
- True data-dependent control flow
  - Conditional branch instruction
  - Subroutine calls, up to 4 deep
  - Jump table (for switch statements)
- Condition codes
- New arithmetic instructions (e.g. COS)
- User clip-plane support

UCL

## Vertex Processor Resource Limits

- 256 instructions per program
  (effectively much higher w/branching)
- 16 temporary 4-vector registers
- 256 “uniform” parameter registers
- 2 address registers (4-vector)

- Changes with every generation though!

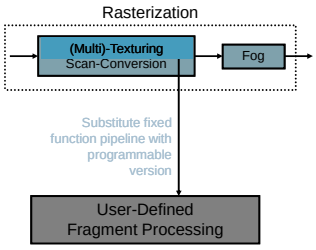
UCL

## Outline

1. GPU Pipeline:
   - Transformations
   - Lighting
   - Rasterization
   - Fragment Operations
2. Vertex Shaders
3. Pixel Shaders
4. Example: Per-Pixel Lighting
5. Geometry Shaders

UCL

## Pixel Shaders




UCL

## Pixel Shaders

- Little assembly program for every pixel
- What for? E.g. per-pixel lighting

UCL

**Latest Pixel Shaders**

- Support 16 textures
- Range: floating point [16/24/32bit]
- Texture lookup is just another command
- Result from texture lookup can be used to compute new tex coordinates
  - dependent texture lookup
- Instruction set identical to vertex shader
  - data-dependent loops
  - dynamic branching (also from inside loops)
- Multiple rendering targets (e.g., 4 color buffers)

UCL

**Fragment Processor / Tex-Mapping**

- Texture reads are just another instruction (TEX, TXP, or TXD)
- Allows computed texture coordinates, nested to arbitrary depth
- Allows multiple uses of a single texture unit
- Optional LOD control – specify filter extent
- Think of it as:
  - A memory-read instruction, with optional user-controlled filtering

UCL

**Additional Fragment Capabilities**

- Read access to window-space position
- Read/write access to fragment Z
- Built-in derivative instructions
  - Partial derivatives w.r.t. screen-space x or y
  - Useful for anti-aliasing
- Conditional fragment-kill instruction
- FP32, FP24, FP16, and fixed-point data

UCL

## Fragment Processor Limitations

- No indexed reads from registers
  - Use texture reads instead
- No memory writes, except to frame-buffer
- Shaders have a cost!!!
  - Especially reading randomly from textures is very expensive!
  - Hardware relies on coherent memory accesses

UCL

## Fragment Processor Resource Limits

- Example Limitations (changes rapidly):
  - 1024 instructions
  - 512 constants or uniform parameters
    - Each constant counts as one instruction
  - 16 texture units
    - Reuse as many times as desired
  - 8 FP32 x 4 perspective-correct inputs
  - 128-bit framebuffer “color” output (use as 4 x FP32, 8 x FP16, etc…)

UCL

## 32-bit IEEE Floating-Point

- 32bit throughout the pipeline:
  - Framebuffer
  - Textures
  - Fragment processor
  - Vertex processor
  - Interpolants

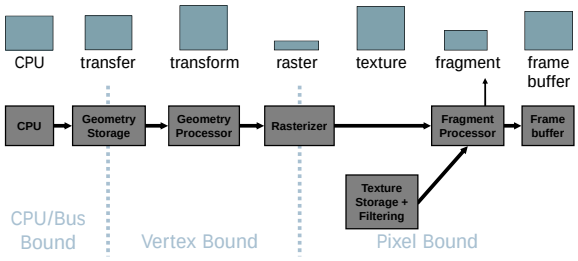
UCL

## Support for Multiple Data Types

- Supports 32-bit IEEE fp throughout pipeline
  - Framebuffer, textures, computations, interpolants
- Fragment processor also supports:
  - 16-bit “half” floating point, 12-bit fixed point
  - These may be faster than 32-bit on some HW
- Framebuffer/textures also support:
  - Large variety of fixed-point formats
    - E.g., classical 8-bit per component RGBA, BGRA, etc.
  - These formats use less memory bandwidth than FP32

UCL

## Possible Bottlenecks




UCL

## Outline

1. GPU Pipeline:
   - Transformations
   - Lighting
   - Rasterization
   - Fragment Operations
2. Vertex Shaders
3. Pixel Shaders
4. Example: Per-Pixel Lighting
5. Geometry Shaders

UCL

## What gets put down the pipe

- *State* which exists and is available throughout pipeline, changes slowly, if at all:

```
GLfloat diffuse0[] = {1.0f, 1.0f, 0.0f, 1.0f};
GLfloat adcolor[] = { 0.3f, 0.3f, 0.3f, 1.0f };
glEnable(GL_LIGHT0);
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse0);
…
glMaterialfv(GL_FRONT, GL_DIFFUSE, adcolor);
```

UCL

## What gets put down the pipe

- Per polygon data, which changes rapidly

```
float size = 1.0f;
float scale = 0.2f;
float delta = 0.1f;

float I[3] = { 1.0f,  0.0f,  0.0f};
float A[3] = { size,  size,  size * scale +
     delta };
float B[3] = { size,  size, -size * scale +
     delta };
float C[3] = { size, -size, -size * scale };
float D[3] = { size, -size,  size * scale };
```

```
glBegin(GL_QUADS);

glNormal3fv(I);
glVertex3fv(D);
glVertex3fv(C);
glVertex3fv(B);
glVertex3fv(A);

glNormal3fv(K);
…
```

UCL

## Vertex Shader

```
vec4 Ambient;
vec4 Diffuse;
vec4 Specular;

void pointLight(in  int i, in vec3 N, in vec3 E, in vec3 vertex)
{
   float NdotL;      // N . light direction
   float NdotH;      // N . light half vector
   float pf;         // power factor
   vec3  L;          // direction from surface to light position
   vec3  H;  // direction of maximum highlights

   // Compute vector from surface to light position
   L = vec3 (gl_LightSource[i].position) - vertex;

   // normalize the vector from surface to light position
   L = normalize(L);
   H = normalize(L + E);
   NdotL = max(0.0, dot(N, L));
   NdotH = max(0.0, dot(N, H));
   if (NdotL == 0.0)
   {
      pf = 0.0;
   }
   else
   {
      pf = pow(NdotH, gl_FrontMaterial.shininess);
   }
   Ambient  += gl_LightSource[i].ambient;
   Diffuse  += gl_LightSource[i].diffuse * NdotL;
   Specular += gl_LightSource[i].specular * pf;
}
```

```
void flight(in vec3 N, in vec3 vertex)
{
   vec4 color;
   vec3 E;

   E = vec3 (0.0, 0.0, 1.0);

   // Clear the light intensity accumulators
   Ambient  = vec4 (0.0);
   Diffuse  = vec4 (0.0);
   Specular = vec4 (0.0);

   pointLight(0, N, E, vertex);
   pointLight(1, N, E, vertex);

   color = Ambient  * gl_FrontMaterial.ambient +
     Diffuse  * gl_FrontMaterial.diffuse +
     Specular * gl_FrontMaterial.specular;
   color = clamp( color, 0.0, 1.0 );
   gl_FrontColor = color;
}

void main (void)
{
   vec3  N;

   // E-coordinate position of vertex, needed in various calculations
   vec3 vertex = vec3(gl_ModelViewMatrix * gl_Vertex);

   // Do fixed functionality vertex transform
   gl_Position = ftransform();
   N = gl_NormalMatrix * gl_Normal;
   N = normalize(N);
   flight(N, vertex);
}
```

## Vertex Shader: Output of Shader

```
color = Ambient  * gl_FrontMaterial.ambient
+ Diffuse  * gl_FrontMaterial.diffuse
+ Specular * gl_FrontMaterial.specular;

color = clamp( color, 0.0, 1.0 );

gl_FrontColor = color;
```

*Note we accumulate all light before applying it to the material*

```
gl_Position = ftransform();
```

```
void flight(in vec3 N, in vec3 vertex)
{
  vec4 color;
  vec3 E;

  E = vec3 (0.0, 0.0, 1.0);

  // Clear the light intensity accumulators
  Ambient  = vec4 (0.0);
  Diffuse  = vec4 (0.0);
  Specular = vec4 (0.0);

  pointLight(0, N, E, vertex);
  pointLight(1, N, E, vertex);

  color = Ambient  * gl_FrontMaterial.ambient +
    Diffuse  * gl_FrontMaterial.diffuse +
    Specular * gl_FrontMaterial.specular;
  color = clamp( color, 0.0, 1.0 );
  gl_FrontColor = color;
}

void main (void)
{
  vec3  N;

  // E-coordinate position of vertex, needed in various calculations
  vec3 vertex = vec3(gl_ModelViewMatrix * gl_Vertex);

  // Do fixed  functionality vertex transform
  gl_Position = ftransform();
  N = gl_NormalMatrix * gl_Normal;
  N = normalize(N);
  flight(N, vertex);
}
```

## Vertex Shader: Iterate over two lights

```
// Clear the light intensity accumulators
  Ambient  = vec4 (0.0);
  Diffuse  = vec4 (0.0);
  Specular = vec4 (0.0);

  pointLight(0, N, E, vertex);
  pointLight(1, N, E, vertex);
```

```
void flight(in vec3 N, in vec3 vertex)
{
  vec4 color;
  vec3 E;

  E = vec3 (0.0, 0.0, 1.0);

  // Clear the light intensity accumulators
  Ambient  = vec4 (0.0);
  Diffuse  = vec4 (0.0);
  Specular = vec4 (0.0);

  pointLight(0, N, E, vertex);
  pointLight(1, N, E, vertex);

  color = Ambient  * gl_FrontMaterial.ambient +
    Diffuse  * gl_FrontMaterial.diffuse +
    Specular * gl_FrontMaterial.specular;
  color = clamp( color, 0.0, 1.0 );
  gl_FrontColor = color;
}

void main (void)
{
  vec3  N;

  // E-coordinate position of vertex, needed in various calculations
  vec3 vertex = vec3(gl_ModelViewMatrix * gl_Vertex);

  // Do fixed  functionality vertex transform
  gl_Position = ftransform();
  N = gl_NormalMatrix * gl_Normal;
  N = normalize(N);
  flight(N, vertex);
}
```

## Vertex Shader

```
vec4 Ambient;
vec4 Diffuse;
vec4 Specular;

void pointLight(in  int i, in vec3 N, in vec3 E, in vec3 vertex)
{
  float NdotL;      // N . light direction
  float NdotH;      // N . light half vector
  float pf;         // power factor
  vec3  L;          // direction from surface to light position
  vec3  H;   // direction of maximum highlights

  // Compute vector from surface to light position
  L = vec3 (gl_LightSource[i].position) - vertex;

  // normalize the vector from surface to light position
  L = normalize(L);
  H = normalize(L + E);
  NdotL = max(0.0, dot(N, L));
  NdotH = max(0.0, dot(N, H));
  if (NdotL == 0.0)
  {
    pf = 0.0;
  }
  else
  {
    pf = pow(NdotH, gl_FrontMaterial.shininess);
  }
  Ambient  += gl_LightSource[i].ambient;
  Diffuse  += gl_LightSource[i].diffuse * NdotL;
  Specular += gl_LightSource[i].specular * pf;
}
```

```
L = vec3 (gl_LightSource[i].position) - vertex;

L = normalize(L);
H = normalize(L + E);
NdotL = max(0.0, dot(N, L));
NdotH = max(0.0, dot(N, H));
if (NdotL == 0.0)
{
  pf = 0.0;
}
else
{
  pf = pow(NdotH, gl_FrontMaterial.shininess);
}
```

UCL

## Vertex Shader

```
vec4 Ambient;
vec4 Diffuse;
vec4 Specular;

void pointLight(in  int i, in vec3 N, in vec3 E, in vec3 vertex)
{
   float NdotL;      // N . light direction
   float NdotH;      // N . light half vector
   float pf;         // power factor
   vec3  L;          // direction from surface to light position
   vec3  H;   // direction of maximum highlights

   // Compute vector from surface to light position
   L = vec3 (gl_LightSource[i].position) - vertex;

   // normalize the vector from surface to light position
   L = normalize(L);
   H = normalize(L + E);
   NdotL = max(0.0, dot(N, L));
   NdotH = max(0.0, dot(N, H));
   if (NdotL == 0.0)
   {
      pf = 0.0;
   }
   else
   {
      pf = pow(NdotH, gl_FrontMaterial.shininess);
   }
   Ambient  += gl_LightSource[i].ambient;
   Diffuse  += gl_LightSource[i].diffuse * NdotL;
   Specular += gl_LightSource[i].specular * pf;
}
```

```
void flight(in vec3 N, in vec3 vertex)
{
   vec4 color;
   vec3 E;

   E = vec3 (0.0, 0.0, 1.0);

   // Clear the light intensity accumulators
   Ambient  = vec4 (0.0);
   Diffuse  = vec4 (0.0);
   Specular = vec4 (0.0);

   pointLight(0, N, E, vertex);
   pointLight(1, N, E, vertex);

   color = Ambient  * gl_FrontMaterial.ambient +
      Diffuse  * gl_FrontMaterial.diffuse +
      Specular * gl_FrontMaterial.specular;
   color = clamp( color, 0.0, 1.0 );
   gl_FrontColor = color;
}

void main (void)
{
   vec3  N;

   // E-coordinate position of vertex, needed in various calculations
   vec3 vertex = vec3(gl_ModelViewMatrix * gl_Vertex);

   // Do fixed functionality vertex transform
   gl_Position = ftransform();
   N = gl_NormalMatrix * gl_Normal;
```

$$I_r = k_a I_a + I_i (k_d (n.l) + k_s (h.n)^m)$$

UCL

## Fragment Shader

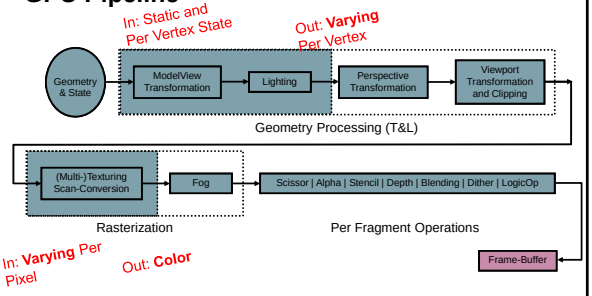
```
void main (void)
{
   vec4 color;
   color = gl_Color;
   color = clamp( color, 0.0, 1.0 );
   gl_FragColor = color;
}
```

UCL

## Clipping and Rasterisation

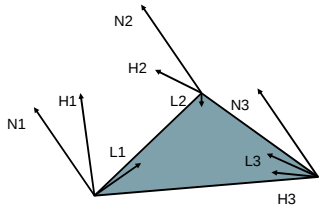
- Vertices (and our calculated colors!) might not be inside the view volume
- Clipped to view volume and resulting polygons rasterised
- Fragments are all the values necessary to calculate a colour of a pixel
- **gl_Color that we receive is *interpolated*, by clipping and rasterisation**

## GPU Pipeline




## Per Pixel Lighting

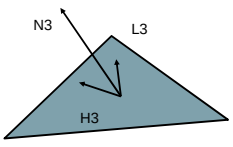
- We want to calculate lighting per pixel – so now we interpolate L, N and H across the polygon instead of just color




## Per Pixel Lighting

- We want to calculate lighting per pixel – so now we interpolate L, N and H across the polygon instead of just color

Note the centre of triangle is brighter than any vertex



UCL

## Vertex Shader 2

```
varying vec4 Ambient;
varying vec3 N;
varying vec3 L0;
varying vec3 H0;
varying vec3 L1;
varying vec3 H1;

void pointLight(in int i, in vec3 N, in vec3 E, in vec3 vertex)
{
   vec3  L;         // direction from surface to light position
   vec3  H;  // direction of maximum highlights

   // Compute vector from surface to light position
   L = vec3 (gl_LightSource[i].position) - vertex;

   L = normalize(L);

   // normalize the vector from surface to light position
   if (i==0) {
            L0 = normalize(L);
            H0 = normalize(L + E);
         }
         else {
            L1 = normalize(L);
            H1 = normalize(L + E);
         }

   // Calculate ambient lighting only
   Ambient  += gl_LightSource[i].ambient;
}
```

```
void flight(in vec3 N, in vec3 vertex)
{
   vec4 color;
   vec3 E;

   E = vec3 (0.0, 0.0, 1.0);

   // Clear the light intensity accumulators
   Ambient  = vec4 (0.0);

   pointLight(0, N, E, vertex);
   pointLight(1, N, E, vertex);
}

void main (void)
{
   // E-coordinate position of vertex, needed in various calculations
   vec3 vertex = vec3(gl_ModelViewMatrix * gl_Vertex);

   // Do fixed functionality vertex transform
   gl_Position = ftransform();
   N = gl_NormalMatrix * gl_Normal;

   flight(N, vertex);
}
```

UCL

## Vertex Shader 2

```
varying vec4 Ambient;
varying vec3 N;
varying vec3 L0;
varying vec3 H0;
varying vec3 L1;
varying vec3 H1;

void pointLight(in int i, in vec3 N, in vec3 E, in vec3 vertex)
{
   vec3  L;         // direction from surface to light position
   vec3  H;  // direction of maximum highlights

   // Compute vector from surface to light position
   L = vec3 (gl_LightSource[i].position) - vertex;

   L = normalize(L);

   // normalize the vector from surface to light position
   if (i==0) {
            L0 = normalize(L);
            H0 = normalize(L + E);
         }
         else {
            L1 = normalize(L);
            H1 = normalize(L + E);
         }

   // Calculate ambient lighting only
   Ambient  += gl_LightSource[i].ambient;
}
```

**varying vec4 Ambient;**
**varying vec3 N;**
**varying vec3 L0;**
**varying vec3 H0;**
**varying vec3 L1;**
**varying vec3 H1;**

UCL

## Vertex Shader 2

```
varying vec4 Ambient;
varying vec3 N;
varying vec3 L0;
varying vec3 H0;
varying vec3 L1;
varying vec3 H1;

void pointLight(in int i, in vec3 N, in vec3 E, in vec3 vertex)
{
   vec3  L;         // direction from surface to light position
   vec3  H;  // direction of maximum highlights

   // Compute vector from surface to light position
   L = vec3 (gl_LightSource[i].position) - vertex;

   L = normalize(L);

   // normalize the vector from surface to light position
   if (i==0) {
            L0 = normalize(L);
            H0 = normalize(L + E);
         }
         else {
            L1 = normalize(L);
            H1 = normalize(L + E);
         }

   // Calculate ambient lighting only
   Ambient  += gl_LightSource[i].ambient;
}
```

// Calculate ambient lighting only
Ambient  += gl_LightSource[i].ambient

*Calculate L and H, for both lights, and send them down the pipeline as varying*

UCL

## Fragment Shader 2

```
varying vec4 Ambient;
varying vec3 N;
varying vec3 L0;
varying vec3 H0;
varying vec3 L1;
varying vec3 H1;

vec4 Specular;
vec4 Diffuse;

void pointLightFrag(in int i, in vec3 N, in vec3 L, in vec3 H)
{
   float NdotL;     // N . light direction
   float NdotH;     // N . light half vector
   float pf;

   NdotL = max(0.0, dot(N, L));
   NdotH = max(0.0, dot(N, H));

   if (NdotL == 0.0)
   {
      pf = 0.0;
   }
   else
   {
      pf = pow(NdotH, gl_FrontMaterial.shininess);

   }
   Diffuse  += gl_LightSource[i].diffuse * NdotL;
   Specular += gl_LightSource[i].specular * pf;
}
```

```
void main (void)
{
        vec4 color;
        vec3 N1;

   Diffuse = vec4 (0.0);
   Specular = vec4 (0.0);

        N1 = normalize(N);

        pointLightFrag(0, N1, L0, H0);
        pointLightFrag(1, N1, L1, H1);

   color = Ambient  * gl_FrontMaterial.ambient +
     Diffuse  * gl_FrontMaterial.diffuse +
     Specular * gl_FrontMaterial.specular;

   color = clamp( color, 0.0, 1.0 );
   gl_FragColor = color;
}
```

UCL

## Fragment Shader 2

```
varying vec4 Ambient;
varying vec3 N;
varying vec3 L0;
varying vec3 H0;
varying vec3 L1;
varying vec3 H1;

vec4 Specular;
vec4 Diffuse;

void pointLightFrag(in int i, in vec3 N, in vec3 L, in vec3 H)
{
   float NdotL;     // N . light direction
   float NdotH;     // N . light half vector
   float pf;

   NdotL = max(0.0, dot(N, L));
   NdotH = max(0.0, dot(N, H));

   if (NdotL == 0.0)
   {
      pf = 0.0;
   }
   else
   {
      pf = pow(NdotH, gl_FrontMaterial.shininess);

   }
   Diffuse  += gl_LightSource[i].diffuse * NdotL;
   Specular += gl_LightSource[i].specular * pf;
}
```

```
NdotL = max(0.0, dot(N, L));
 NdotH = max(0.0, dot(N, H));

 if (NdotL == 0.0)
 {
    pf = 0.0;
 }
 else
 {
    pf = pow(NdotH, gl_FrontMaterial.shininess);

 }
 Diffuse  += gl_LightSource[i].diffuse * NdotL;
 Specular += gl_LightSource[i].specular * pf;
```

UCL

## Outline

1. GPU Pipeline:
   - Transformations
   - Lighting
   - Rasterization
   - Fragment Operations
2. Vertex Shaders
3. Pixel Shaders
4. Example: Per-Pixel Lighting
5. Geometry Shaders

UCL

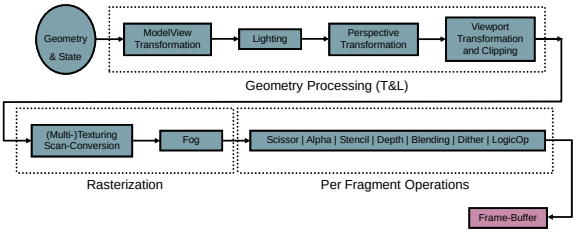
## Geometry Shaders

- New in DirectX10/OpenGL3.2 (or via OpenGL extensions)
- Takes geometry primitives (vertices, primitive state) and produces more (or zero) geometry primitives
- Get over a problem that Vertex Shaders are exactly one vertex in, over vertex out
- Useful for tesselation, shadow volume extrusion, etc.

UCL

## Geometry Shaders

- New in DirectX10/OpenGL3.2 (or via OpenGL extensions)




UCL

## Conclusions

- GPUs are:
  - Fast
  - Powerful
  - Hard to program
  - Many new features each generation, often vendor-specific
  - Hard to keep up with these features
  - Lesson: high-level languages needed for programming
  - Solution: HLSL / CG