CMSC 498M: Chapter 7
GPU Programming and Shaders

Reading:
- Chapter 18 of "Fundamentals of Computer Graphics" by P. Shirley and S. Marschner
- "OpenGL Shading Language" by Randi J. Rost
- Lighthouse GLSL tutorial: http://www.lighthouse3d.com/tutorials/glsl-tutorial/

Overview:
- GPU architecture and shaders
- GLSL

---

GPU Architecture

Graphics Processing Unit:
- Designed for high-performance, real-time rendering of raster graphics
- Highly parallel
- Stream processing

Basic GPU Processing:
- User program: Specifies primitives (points, polygons, textures, ...)
- Geometry processing: Act on vertices of the primitives (2D and 3D transformations, lighting)
- Fragment (pixel) processing: Rasterization, coloring, texturing, stencil/depth tests

User Program → primitives → Vertex Processing → transformed vertices → Fragment Processing → fragments
OpenGL and Shaders

OpenGL provides for certain default processing:

**Vertex:**
- Primitives (triangles, triangle strips, quads) are composed from vertices
- Based on a vertex’s surface material, position, normal vector, local lighting computations are performed to determine its color
- Texture coordinates may also be provided

**Fragment:**
- An abstract pixel that has a fixed (x,y) position on the screen and carries additional information such as color, transparency, depth
- Texture mapping: performed by interpolating vertex texture coordinates
- Coloring: is performed on fragments by interpolating the light-based color of the vertices

**Shaders:** Programmable units that can override the default vertex and fragment processing

**Languages:** Cg (NVIDIA), HLSL (Microsoft), GLSL (OpenGL).

---

GPU Shader Examples

Textured teapot with normal mapping

Toon-shading

Fancy lighting using Blinn-Phong model

Refraction shader and chromatic dispersion

Environment mapping
**GPU Architecture**

**Note:** This presentation assumes GLSL and OpenGL.

**Default Vertex Processing:**
- Vertex transformation (e.g., by modelview matrix)
- Normal transformation and normalization
- Texture coordinate generation/ transformation
- Lighting and color material application

**Vertex Shader:** Things a vertex shader might do:
- Nonstandard transformation (e.g., displacement mapping)
- Nonstandard normal transformation (e.g., bump, normal mapping)
- Nonstandard coloring (e.g., simulated ray tracing)
- Nonstandard lighting (e.g., fancy illumination models)
GPU Architecture

Default Fragment Processing:
- Interpolating color and depth from vertices
- Texture access and texture application
- Fog

GPU Architecture

Fragment Shader: Things a fragment shader might do:
- Nonstandard coloring (e.g., procedural textures, fractals, noise)
- Texture image processing (e.g., blurring, sharpening)
- Depth displacement (e.g., for displacement mapping)
- Texture displacement (e.g., refraction shading)
Vertex Shader

Vertex Shader:
- A programmable unit operating on vertices and associated data
- Typical tasks:
  - Vertex/normal transformations
  - Projection
  - Texture coordinate generation
  - Lighting and distance attenuation
- What it cannot do: (these are done by OpenGL)
  - Viewport mapping
  - Primitive assembly (forming triangle strips, for example)
  - Clipping (outside view frustum) or culling (back/front-face culling)
  - Apply shading mode (flat or smooth)
- If you supply a vertex shader, it must perform all the required OpenGL functionality for vertices.
  (You cannot just do lighting and leave texture mapping to OpenGL)

Vertex Shader Inputs:
- Attributes: (properties that apply to individual vertices)
  - Built-in attributes: gl_Color, gl_Normal, gl_Vertex, gl_TexCoord, ...
  - User-defined attributes: vertex info meaningful to your application
- Uniform variables: (fixed for entire primitive)
  - Built-in uniform variables: gl_ModelViewMatrix, gl_LightSource[i],
    gl_FrontMaterial, gl_Fog, ...
  - User-defined uniform variables: (based on your application)
- Textures: Provided implicitly

Vertex Shader Outputs:
- Special output variables: gl_Position (required output!)
  
  Varying variables: Interpolated between vertices and sent to frag's:
    - Built-in varying variables: gl_FrontColor, gl_BackColor, glTexCoord[i]
      (be sure to write all that your fragment program will need)
    - User-defined varying variables: (up to you)
### Vertex Shader

**Built-in attribute variables**
- `gl_Normal`
- `gl_Vertex`
- `gl_TexCoord` (Tangent)
- `gl_Color`
- `StartColor`

**User-defined attribute variables**
- `Velocity`
- `Elevation`
- `Tangent`, ...

**Special output variables**
- `gl_Position`
- `gl_PointSize`
- `gl_ClipVertex`, ...

**Built-in uniform variables**
- `gl_ModelViewMatrix`, `gl_LightSource[i]`, `gl_FrontMaterial`, `gl_Fog`, ...

**User-defined uniform variables**
- `ModelScaleFactor`, `EyePosition`, `WeightFactor`, ...

**Built-in varying variables**
- `gl_FrontColor`
- `gl_BackColor`
- `gl_FogFragCoord`, ...

**User-defined varying variables**
- `Normal`
- `ModelCoord`
- `RefractionIndex`, ...

**Global state**

---

### Fragment Shader

**Fragment Shader:**
- A programmable unit operating on fragments and associated data

**Typical tasks:**
- Texture application
- Color assignment
- Fog computation (if enabled)

**What it cannot do:** (these are done by OpenGL)
- Specifying which texture(s) to use
- Cannot modify the (x,y) coordinates of the fragment
- Cannot access neighboring fragments
- Depth/stencil test or alpha (transparency) blending

**If you supply a fragment shader, it must performed all the required OpenGL functionality for fragments**
Fragment Shader

Fragment Shader Inputs:
- **Special input variables**: (provided by OpenGL)
  - `gl_FragCoord` (x,y-coords), `gl_FrontFacing` (true of from front face), …
- **Varying variables**: (interpolated from vertex values)
  - Built-in varying variables: `gl_Color`, `gl_TexCoord[i]`, …
  - User-defined varying variables: (based on your application)
- **Uniform variables**: (fixed for entire primitive)
  - Built-in uniform variables: (same as for vertex shader)
  - User-defined uniform variables: (based on your application)
- **Textures**: Provided implicitly (no limit on number of accesses)

Fragment Shader Outputs:
- **Special output var's**: `gl_FragColor` (or `gl_FragData[n]`) `gl_FragDepth`
- **Overlap**: If fragments overlap, **depth-test** determines which is written. If no depth-test, then **last write is final**.
- **Discard**: Rather than writing outputs, fragment may be **discarded**

Chapter 7, Slide 13
Copyright © David Mount and Amitabh Varshney
### Compiling/Linking a GLSL Shader

**Compiling/Linking:**

- `glCreateProgram`: Create a program object
- `glCreateShader`: Create a shader object
- `glShaderSource`: Get source code for shader (given as a character string)
- `glCompileShader`: Compile it
- `glAttachShader`: Attach shader to program object
- `glLinkProgram`: Link program to create an executable
- `glUseProgram`: Install program object executable as part of state

#### Compiling/Linking Example

```c
void setShaders()
{
    GLuint v = glCreateShader(GL_VERTEX_SHADER);  // create shaders; get ids
    GLuint f = glCreateShader(GL_FRAGMENT_SHADER);
    char* vs = textFileRead("mshepherd.vert");    // input shader code (allocate string storage)
    char* fs = textFileRead("mshepherd.frag");
    const char* vv = vs;
    const char* ff = fs;
    glShaderSource(v, 1, &vv, NULL);              // bind source to shader ids
    glShaderSource(f, 1, &ff, NULL);
    free(vs); free(fs);                           // deallocate storage
    glCompileShader(v);                          // compile
    glCompileShader(f);
    p = glCreateProgram();                       // create program; get id
    glAttachShader(p, v);                       // attach shaders to program
    glAttachShader(p, f);
    glLinkProgram(p);                           // link
    glUseProgram(p);                            // make this program active
}
```

Assumes OpenGL 2.0

See `glGetShaderiv()` and `glGetShaderLog()` to determine whether compilation succeeded

Setting p=0 restores system default shader
GLSL Language - Types

GLSL Data Types:
Basic Types: float, int, bool
Vector Types:
- vec2, vec3, vec4: 2D, 3D and 4D floating point vectors
- ivec2, ivec3, ivec4: 2D, 3D and 4D integer vectors
- bvec2, bvec3, bvec4: 2D, 3D and 4D boolean vectors

Constructor:
- Vec3 v = vec3(1.0, 2.0, 3.0);

Accessing components:
Indexing: v[0], …, v[3]
Geometry coordinates: v.x, v.y, v.z, v.w
Color: v.r, v.g, v.b, v.a
Texture: v.s, v.t

Operators:
Vector-vector: u+v, u-v, u*v, …: Applied componentwise
Scalar-vector: s+v, s*v, …: Apply to each component

GLSL Language - Types

GLSL Data Types:
Matrix Types:
- mat2, mat3, mat4: 2x2, 3x3, 4x4 floating point matrices

Constructor: Constructed column-by-column:
- mat2 m = mat2(1.0, 2.0, 3.0, 4.0) = [ 1.0  3.0 ]
  [ 2.0  4.0 ]

Indexing:
- m[i][j]: column i and row j. (Warning: Column-major order)
- m[i]: the i-th column as a vector

Matrix-Vector Operators:
- v * m: column vector × matrix
- m * v: matrix × row vector
- m * v: matrix × matrix
GLSL Language - Types

GLSL Data Types:
- **Sampler**: Used for storing texture maps
  - sampler1D, sampler2D, sampler3D

Structs:
- Same as C/C++:
  ```
  struct light {    // structure for light object
    vec3 position;
    vec3 color;
  };
  ```
  - union, enum, class: Don’t exist

Arrays:
- Similar to C/C++:
  ```
  vec4 points[10]; // array of 10 vec4
  ```
  - Warning: vec4 and float[4] are entirely different types

Type Qualifiers: attribute, uniform, varying, const

Example: Phong Diffuse Shading

**Phong Diffuse Shading**:
- OpenGL computes lighting at vertices and interpolates colors throughout the polygon (Gouraud shading)
- **Phong shading** is more accurate. It interpolates surface normals at each pixel, and computes lighting for each fragment.

**Simple lighting formula**:
- \[ I = C_A + \max(0, n \cdot l) \cdot C_D \]
  where:
  - \( I \) = final intensity
  - \( C_A \) = ambient color (= ambient light \cdot surface color)
  - \( C_D \) = diffuse color (= diffuse light \cdot surface color)
  - \( n \) = surface normal (unit length)
  - \( l \) = vector to light (unit length)

For simplicity, we ignore the specular component.
Example: Phong Diffuse Shading

Vertex Shader:
- Compute the vertex position:
  - Multiply vertex by the built-in matrix \( gl\_ModelViewProjectionMatrix \)
  - Store result in \( gl\_Position \) (this step is required)
- Compute the surface normal and at each vertex
  - Multiply the surface normal by \( gl\_NormalMatrix \)
  - Save as varying variable \( normal \) (by declaring it to be varying, the normals at the vertices will be interpolated throughout the fragments)
- Get the vector from the vertex to the light source
  - Convert vertex to camera frame by multiplying by \( gl\_ModelViewMatrix \)
  - Subtract from light source position and save as varying variable \( toLight \)

Fragment Shader:
- Get the (interpolated) normal and \( toLight \) vectors and normalize
- Compute the intensity (by above equation)
- Save in special output variable \( gl\_FragColor \)

```cpp
// Vertex Shader
varying vec3 normal;
varying vec3 toLight;
void main() {
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex; // transform vertex (required)
  normal = gl_NormalMatrix * gl_Normal; // transform normal
  vec4 viewVertex = gl_ModelViewMatrix * gl_Vertex; // transform to camera frame
  toLight = vec3(gl_LightSource[0].position - viewVertex); // vector to light source
}

// Fragment Shader
const vec4 AmbientColor = vec4(0.1, 0.0, 0.0, 1.0); // define colors
const vec4 DiffuseColor = vec4(1.0, 0.0, 0.0, 1.0);
vec3 uNormal = normalize(normal); // normalize to unit length
vec3 uToLight = normalize(toLight); // vector to light source
float diffuseTerm = clamp(dot(uNormal, uToLight), 0.0, 1.0); // diffuse term
gl_FragColor = AmbientColor + DiffuseColor * DiffuseTerm; // final pixel color
```
Vertex Attributes

**Attributes:** Variables that may vary between individual vertices within a primitive (in contrast to uniform variables, which are constant for all vertices of a primitive).

**Built-in vertex attributes:**
- `gl_Color`: Vertex's color (vec4)
- `gl_Normal`: Surface normal vector (vec3)
- `gl_Vertex`: Vertex's coordinates (vec4)
- `gl_MultiTexCoord0, gl_MultiTexCoord1, ...`: Texture coordinates (vec4)

  - Each is generated by the corresponding OpenGL command:
    - `glColor(...)`: set `gl_Color`
    - `glNormal(...)`: set `gl_Normal`
    - `glTexCoord(…)`: set `gl_MultiTexCoord0`
    - `glVertex(…)`: set `gl_Vertex`

User-Defined Vertex Attributes

**Defining your own attributes:**
- OpenGL provides generic attribute variables, which you can set.
- `glVertexAttrib*(i, v)`: Associate value v with attribute i

  ```c
  GLfloat opacity[] = {1, 7, 3, 2.5}; // vector of 4 floats (4fv)
  glVertexAttrib4fv(13, opacity); // set attribute 13 to vector opacity
  ```

**Accessing user-defined attributes in the shader:**
- `glBindAttribLocation(p, i, "var")`: Associates user-defined attribute i with shader variable "var" in program p

  ```c
  glBindAttribLocation(1, 13, "opacity"); // bind attribute 13 to "opacity"
  vec4 opacity; // this receives the value [1,7,3,2.5]
  main() { ... }
  ```

These names don't need to match, but it makes your program more readable if they do.
Uniform Variables

Uniform Variables: Variables that are constant for all vertices of a primitive.

Built-in vertex uniform variables:
- **gl_ModelViewMatrix**: Model view matrix (mat4)
  - Transforms a vertex into camera coordinates
- **gl_ProjectionMatrix**: Projection matrix (mat4)
  - Transforms camera coordinates to normalized device coordinates
- **gl_ModelViewProjectionMatrix**: Projection • ModelView (mat4)
  - The most common way to obtain the final vertex position either:
    - `gl_Position = glModelViewProjectionMatrix * gl_Vertex;`
    - or-
    - `gl_Position = ftransform();` // recommended for multipass rendering

- **gl_NormalMatrix**: Transforms normal(mat4)
  - You can also get the inverses and transposes of the above

Chapter 7, Slide 25
Copyright © David Mount and Amitabh Varshney

Uniform Variables for Lighting

```c
struct gl_LightSourceParameters {
    vec4 ambient, diffuse, specular; // ambient, diffuse, specular light intensities
    vec4 position; // light position
    vec4 halfVector; // halfway vector (used in specular lighting)
    ...
    float constantAttenuation, linearAttenuation, quadraticAttenuation; // attenuation parameters
};
// light source parameters
uniform gl_LightSourceParameters gl_LightSource[gl_MaxLights];
```

```c
struct gl_MaterialParameters {
    vec4 emission, ambient, diffuse, specular; // surface material colors
    float shininess; // shininess (for specular lighting)
};
// object surface materials
uniform gl_MaterialParameters gl_FrontMaterial;
uniform gl_MaterialParameters gl_BackMaterial;
```

Chapter 7, Slide 26
Copyright © David Mount and Amitabh Varshney
User-Defined Uniform Variables

Defining your own uniform variables:
- Use the keyword "uniform":

```glsl
uniform vec4 objectColor; // a user-defined uniform variable
main() { ... }
```

- `glGetUniformLocation(p, "var")`: Returns index of uniform variable "var" in program p

```c
int i = glGetUniformLocation(1, "objectColor"); // get index i of "objectColor"
```

- `glUniform*(i, v)`: Set value of uniform variable with index i to value v

```c
vec4 objectColor[] = {0.0, 1.0, 1.0, 0.0}; // object color
int i = glGetUniformLocation(1, "objectColor"); // get index i of "objectColor"
glUniform4fv(i, objectColor); // set objectColor value in shader
```

These names don't need to match, but it makes your program more readable if they do.

Samplers

Samplers:
- Texture lookups require indication of which texture unit will do the lookup
- Samplers perform texture lookups
- `sampler1D`, `sampler2D`, `sampler3D`: For 1D, 2D, 3D textures

```glsl
uniform sampler2D colorMap;
uniform sampler2D normalMap;

void main (void)
{
    vec3 texColor = texture2D(colorMap, gl_TexCoord[0].st);
    vec3 bump = texture2D(normalMap, gl_TexCoord[0].st);
    ... 
}
```
Control Flow/Functions

Control Flow:
- if-then-else, while loops, for loops: Syntax same as C/C++
- continue and break statements exist
- There is no switch or goto statement
- discard: This is similar to function return, but appears in a fragment shader to indicate that the fragment is to be discarded

Functions:
- Similar syntax with C/C++
- Call by value-return:
  - in: Copy parameter in, can modify, but don’t copy back on return
  - out: Only copy out (undefined on entry)
  - inout: Copy value in, can modify, copy value out on return
- Recursive calls are not allowed
- Functions cannot return arrays (but can return vec or mat types)

Example: Toon Shader

Toon Shader:
- Produces a cartoonish looking shading, by thresholding color intensities to a discrete set of values

Vertex Shader:
- Essentially the same as the Phong shader given earlier.
- Compute the normal vector and the vector to the light source. Convert both to the camera frame.

Fragment Shader:
- Compute the Phong strength as before as max(0, (n · l)), where n is the normal and l is the toLight vector
- Threshold the Phong strength into four intervals, and assign colors appropriately for each category: dark, medium, light, very light.
**Example: Toon Shader**

**Vertex Shader**

```c
varying vec3 normal; // surface normal
varying vec3 toLight; // vector to light source

void main() {
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex; // transform vertex (required)
  normal = gl_NormalMatrix * gl_Normal; // transform normal
  vec4 viewVertex = gl_ModelViewMatrix * gl_Vertex; // transform to camera frame
  toLight = vec3(gl_LightSource[0].position - viewVertex); // vector to light source
}
```

**Fragment Shader**

```c
varying vec3 normal; // surface normal
varying vec3 toLight; // vector to light source

void main() {
  vec3 uNormal = normalize(normal); // normalize vectors
  vec3 uToLight = normalize(toLight); // normalize vectors
  float phong = clamp(dot(uNormal, uToLight), 0.0, 1.0); // compute diffuse intensity
  vec4 color;
  if (phong > 0.95) color = vec4(1.0, 0.5, 0.5, 1.0); // discretize color to 4 bands
  else if (phong > 0.5) color = vec4(0.6, 0.3, 0.3, 1.0);
  else if (phong > 0.25) color = vec4(0.4, 0.2, 0.2, 1.0);
  else color = vec4(0.2, 0.1, 0.1, 1.0);
  gl_FragColor = color; // return fragment color
}
```

**Example: Normal Mapping**

**Normal Mapping:**

- Lighting is determined by the relationship between **surface normals** and light sources
- By **perturbing** the normals of a flat surface it is possible to create the illusion of **bumpiness** on the surface

**Two textures:**

- **Color texture** (RGB)  
  - Encoded as **RGB**
    - Normal vector: \((x, y, z)\), where \(-1 \leq x, y \leq 1, z = 1\)
    - **Encoded** as: \((R, G, B) = ((x+1)/2, (y+1)/2, 1)\)
    - **Decode normal** as: \((x, y, z) = 2(R, G, B) - 1\)
- **Perturbed surface normals**
  - We apply **normal perturbation** to actual normals and then apply the **standard Phong lighting model**
Review: OpenGL Illumination Model

**Ambient Illumination:** Constant everywhere

**Diffuse Illumination:** (Discussed earlier)

**Specular Illumination:**
- Used to simulate metallic, shiny surfaces
- A function of the relative positions of light, viewer, and normal
- Common methods: reflection vector and halfway vector
- We’ll use the reflection-vector method in our example

Specularity brightest when viewer is aligned with reflection vector.

Let \( r \) be the reflection of the view vector with respect to the normal

Specularity is high if the \( r \) and \( l \) are very close.

---

**OpenGL Phong Lighting:** Is based on the following parameters:

- \( C_a, C_d, C_s \): Surface’s ambient, diffuse, and specular colors
  - Typically, \( C_a = C_d \) and \( C_s = \) light color
- \( L_a, L_d, L_s \): Light source’s ambient, diffuse, and specular intensity
  - Typically, \( L_a = L_d = L_s \)

**Vectors:** (All of unit length)
- \( n \): surface normal at vertex
- \( l \): vector from vertex to light source
- \( v \): vector from vertex to viewer
- \( r \): view reflection vector (reflection of \( v \) about \( n \))

\( \sigma \): shininess - larger values result in a smaller specular point

**\( I \): final intensity**

\[
I = C_a L_a + C_d L_d \max(0, n \cdot l) + C_s L_s \max(0, r \cdot l)^\sigma
\]
Example: Normal Mapping

**Vertex Shader:**
- Construct frame aligned with surface geometry:
  - \( n \) - normal vector
  - \( t, b \) - tangent vectors
  - \( n \) and \( t \) are given, and \( b = n \times t \)
- Express the principal illumination vectors \((l, v)\) with respect to the surface coordinate system:
  - \((\text{light}.x, \text{light}.y, \text{light}.z) = (l \cdot t, l \cdot b, l \cdot n)\)
  - \((\text{view}.x, \text{view}.y, \text{view}.z) = (v \cdot t, v \cdot b, v \cdot n)\)

**Fragment Shader:**
- Access the bump vector as an RGB from the normal map and convert to perturbed normal vector \(pn = 2(\text{RGB}) + 1\)
- Normalize: \(l = \text{normalize}(\text{light}), v = \text{normalize}(\text{view})\)
- Compute reflected view vector \(r = \text{reflect}(\text{view}, pn)\)
- Compute final illumination from equation

---

```cpp
varying vec3 light; // vector to light source
varying vec3 view; // vector to viewer
attribute vec4 glTangent4f; // surface tangent (input from application)

void main(void)
{
    gl_Position = ftransform(); // vertex position (required)
    gl_TexCoord[0] = gl_MultiTexCoord0; // save texture coordinate (used in fragment)
    vec3 n = normalize(gl_NormalMatrix * gl_Normal); // normal (in camera coordinates)
    vec3 t = normalize(gl_NormalMatrix * glTangent4f.xyz); // tangent (in camera coordinates)
    vec3 b = cross(n, t); // second tangent vector
    vec3 vert = vec3(gl_ModelViewMatrix * gl_Vertex); // vertex (in camera coordinates)
    vec3 l = gl_LightSource[0].position.xyz - vert; // vector to light source (in camera coords)
    light.x = dot(l, t); light.y = dot(l, b); light.z = dot(l, n); // vector to light source (in surf coords)
    vec3 v = -vert; // vertex to viewer (remember, origin is eye)
    view.x = dot(v, t); view.y = dot(v, b); view.z = dot(v, n); // vertex to viewer (in surface coords)
    
    // compute final illumination
    
    // continue with fragment shader logic
}
```
Example: Normal Mapping

```glsl
varying vec3 light; // vector to light source
varying vec3 view; // vector to viewer
uniform sampler2D colorMap; // color texture map
uniform sampler2D normalMap; // normal map

void main (void)
{
    vec4 base = texture2D(colorMap, gl_TexCoord[0].st); // base color from color map
    vec4 ambient = vec4(0.2, 0.2, 0.2, 1.0) * base; // ambient color
    vec3 l = normalize(light); // normalize light vector
    vec3 v = normalize(view); // normalize view vector
    vec3 pn = normalize(texture2D(normalMap, gl_TexCoord[0].st).xyz * 2.0 - 1.0); // perturbed normal
    float diffCoef = max(dot(l, pn), 0.0); // diffuse coefficient
    vec4 diffuse = gl_FrontLightProduct[0].diffuse * diffCoef * base; // diffuse color
    float specCoef = pow(clamp(dot(reflect(-v, pn), l), 0.0, 1.0), gl_FrontMaterial.shininess); // specular coefficient
    vec4 specular = gl_FrontLightProduct[0].specular * specCoef; // specular color
    gl_FragColor = ambient + diffuse + specular; // final color
}
```

Summary:

- GPU architecture
- Vertex and Fragment Shaders
- GLSL shading language
- Examples:
  - Phong diffuse shader
  - Toon shader
  - Normal mapping