OpenGL Shading Language (GLSL)

- Platform independent
  - Compared to Cg (nVidia) or HLSL (Microsoft)
- A C-like language and incorporated into OpenGL 2.0
- Used to write
  - Vertex/fragment shader programs
  - Geometry/tessellation shader programs
Let's look at a very simple shader example:
- do_nothing.vert, do_nothing.frag

How to import the shader program and link to your OpenGL program:
- SDcubeSimple.C; shaderSetup.C

How to pass the vertex attributes from VBOs to a shader program:
- SDcubeSimple.C
do_nothing.vert

- A very simple shader program
- Replace vertex processing in the fixed function pipeline
- Does nothing except passing the vertex position and color to the fragment shader
- This program does not even perform local to clip space transformation. It assumes that the input vertex position is already in clip space
  - This is not a norm; typically you will at least transform the vertex position to the clip space
do_nothing.vert

attribute vec4 position;  // the vertex position and color are passed from an opengl program
attribute vec4 color;

varying vec4 pcolor;  // this is the output variable to the fragment shader

// this shader just pass the vertex position and color along, doesn't actually do anything
// Note that this means the vertex position is assumed to be already in clip space
//
void main(){
    gl_Position = position;
    pcolor = color;
}

do_nothing.frag

- Replace fragment processing in the fixed function pipeline
- The input to the fragment program are interpolated attributes from the vertices for the fragment
- This fragment program essentially does nothing except passing the fragment color to the frame buffer
do_nothing.frag

varying vec4 pcolor; // Need to match the name with the desired variable from 
// the vertex program

//
// This fragment shader just passes the already interpolated fragment color 
//
void main() {
    gl_FragColor = pcolor; // note that gl_FragColor is a default name for 
    // the final fragment color
}

Why do you want me to look at these simple shader programs??

- I want you to learn how to import and link the shader program with an OpenGL program
- I want you to learn how to pass the vertex attributes (position and color in this example) to the shader
Setup of Shader Programs

- A shader is defined as an array of strings
- Steps to use shaders
  1. Create a shader program
  2. Create **shader objects** (vertex and fragment)
  3. Send **source code** to the shader objects
  4. **Compile** the shader
  1. Create **program object** by linking compiled shaders together
  2. Use the linked program object
Create Shader Program and Shader Objects

• Create a shader program
• Later a vertex and a fragment shader will be attached to this shader program

```c
GLuint programObject;
programObject = glCreateProgram(); // create an overall shader program

if (programObject == 0) { // error checking
    printf(" Error creating shader program object.\n");
    exit(1);
}
else printf(" Succeeded creating shader program object.\n");
```
Create Shader Program and Shader Objects (cont’d)

• Create vertex and fragment shader objects

    GLuint vertexShaderObject;
    GLuint fragmentShaderObject;

    vertexShaderObject = glCreateShader(GL_VERTEX_SHADER);
    if (vertexShaderObject == 0) {  // error checking
        printf(" Error creating vertex shader object.\n");
        exit(1);
    } else printf(" Succeeded creating vertex shader object.\n");

    fragmentShaderObject = glCreateShader(GL_FRAGMENT_SHADER);
    if (fragmentShaderObject == 0) {  // error checking
        printf(" Error creating fragment shader object.\n");
        exit(1);
    } else printf(" Succeeded creating fragment shader object.\n");
Send the source to the shader and compile

- You need to read the source (.vert and .frag files in ascii format) into strings first
  - For example, into the following char arrays
    ```
    GLchar *vertexShaderSource;
    GLchar *fragmentShaderSource;
    ```
  - Remember how to use C fopen and fread? something like
    ```
    fh = fopen(name, "r");
    if (fh==NULL) return -1;
    //
    // Get the shader from a file.
    fseek(fh, 0, SEEK_SET);
    count = fread(shaderText, 1, size, fh);
    shaderText[count] = '\0';
    if (ferror(fh)) count = 0;
    fclose(fh);
    ```
  - Check shaderSetup.C in the example
Send the source to the shader and compile (con’td)

● Now compile the shaders

```c
// After reading the shader files, send the source to the shader objects
glShaderSource(vertexShaderObject, 1, (const GLchar**) &vertexShaderSource, NULL);
glShaderSource(fragmentShaderObject, 1, (const GLchar**) &fragmentShaderSource, NULL);

// now compile the shader code; vertex shader first, followed by fragment shader
glCompileShader(vertexShaderObject);
glCompileShader(fragmentShaderObject);
```

● Remember you need to do some error checking, check shaderSetup.C to see how to do it
Finally, attach and link the shader program

- Attach the vertex and fragment shader objects to the shader program and link together

```cpp
glAttachShader(programObject, vertexShaderObject);
glAttachShader(programObject, fragmentShaderObject);
glLinkProgram(programObject);
```

- Later you can use the shader program any time before you render your geometry (See SDcubeSimple.C)

```cpp
...  
glUseProgram(programObject);  
...  // OpenGL drawing
```
Shaders Setup Summary

GLhandle glCreateProgramObject();
GLhandle glCreateShaderObject(GL_VERTEX_SHADER);
GLhandle glCreateShaderObjectARB(GL_FRAGMENT_SHADER);
void glShaderSource(GLhandle shader, GLsizei nstrings, const GLchar **strings, const GLint *lengths)
    //if lengths==NULL, assumed to be null-terminated
void glCompileShader(GLhandle shader);
void glAttachObjectARB(GLhandle program, GLhandle shader);
    //twice, once for vertex shader & once for fragment shader

void glLinkProgram(GLhandle program);
    //program now ready to use

void glUseProgramObject(GLhandle program);
    //switches on shader, bypasses FFP
    //if program==0, shaders turned off, returns to FFP
Shaders Setup Summary
Let’s go back to look at the shaders

- How to pass the vertex attributes to the vertex shader?

```cpp
attribute vec4 position;  // input: the vertex position and color, passed from an opengl program
attribute vec4 color;
varying vec4 pcolor;  // this is the output to the fragment shader

void main(){
    gl_Position = position;
    pcolor = color;
}
```
Passing vertex attributes to the (vertex) shader

• In your OpenGL program, do the following:
  1. ID = Query the location of the shader variable
  2. Enable the attribute array
  3. Map the attribute array to ID

```c
GLuint c0 = glGetAttribLocation(programObject, "position");
GLuint c1 = glGetAttribLocation(programObject, "color")
...
glEnableVertexAttribArray(c0);
glEnableVertexAttribArray(c1);
...
glVertexAttribPointer(c0, 4, GL_FLOAT, GL_FALSE, sizeof(Vertex), (char*) NULL + 0);
glVertexAttribPointer(c1, 4, GL_FLOAT, GL_FALSE, sizeof(Vertex), (char*) NULL + 16);
...
glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_BYTE, (char*) NULL + 0);
```
# Setup of Attribute Variables

<table>
<thead>
<tr>
<th>Return type</th>
<th>Function Name</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLint</td>
<td>glGetAttribLocation</td>
<td>GLuint program, const GLchar * name</td>
</tr>
<tr>
<td>void</td>
<td>glVertexAttribPointer</td>
<td>GLuint index, GLint size, GLenum type, Glboolean normalized, GLsizei stride, const GLvoid * pointer</td>
</tr>
<tr>
<td>void</td>
<td>glEnableVertexAttribArray</td>
<td>GLuint index</td>
</tr>
</tbody>
</table>

1. Get ID of the attribute
2. Map the pointer to the array of per-vertex attribute to this ID
3. Enable this attribute
How do vertex and fragment shaders communicate?

- Through the **varying** variable

```cpp
attribute vec4 position;
attribute vec4 color;

**varying vec4 pcolor**

void main()
{
    gl_Position = position;
    pcolor = color;
}
```

```cpp
varying vec4 pcolor;

void main()
{
    gl_FragColor = pcolor;
}
```

Vertex Program

Fragment Program
Let’s make the vertex shader do more

- Transform the vertex position from local space to clip space
  
  - Any vertex program is expected to perform the transformation

- Assume you use the OpenGL fixed function pipeline (like in Lab 2) and set up GL_MODELVIEW and GL_PROJECTION

```glsl
attribute vec4 position;
attribute vec4 color;
varying vec4 pcolor;

void main(){
    pcolor = color;
    gl_Position = gl_ModelViewProjectionMatrix * position;
}
```
Use your own transformation matrix

- Need to pass the modelview projection matrix (projection*modelview) to the vertex shader
- Let’s look at the vertex program first

```glsl
attribute vec4 position;
attribute vec4 color;
uniform mat4 local2clip; // this is the concatenated modlview projection matrix passed from your OpenGL program

varying vec4 pcolor;

void main(){
  pcolor = color1;
  gl_Position = local2clip * position;
}
```
How to pass the matrices to the vertex shader

- You can pass any matrices to the shader as desired
  - For example, modeling matrix, viewing matrix, projection matrix, modelview matrix, or modelviewprojection matrix
- These matrices are *uniform* variable
- Uniform variables remain the same values for all vertices/fragments
  - i.e., you cannot change their values between vertices (between glBegin/glEnd or within VBOs)
Set the values of uniform variables to shaders (a glm example)

```cpp
glUseProgram(programObject);
...
// get the location of the uniform variable in the shader
GLuint m1 = glGetUniformLocation(programObject, "local2clip");
...
glm::mat4 projection = glm::perspective(60.0f, 1.0f, 1.0f, 100.0f);
glm::mat4 view = glm::lookAt(glm::vec3(0.0, 0.0, 5.0),
    glm::vec3(0.0, 0.0, 0.0), glm::vec3(0.0, 1.0, 0.0));
glm::mat4 model = glm::mat4(1.0f);
model = glm::rotate(model, x_angle, glm::vec3(0.0f, 1.0f, 0.0f));
model = glm::rotate(model, y_angle, glm::vec3(1.0f, 0.0f, 0.0f));
model = glm::scale(model, scale_size, scale_size, scale_size);

// construct the modelview and modelview projection matrices
glm::mat4 modelview = view * model;
glm::mat4 modelview_projection = projection * modelview;

// pass the modelview_projection matrix to the shader as a uniform
glUniformMatrix4fv(m1, 1, GL_FALSE, &modelview_projection[0][0]);
```
Setup of *Uniform* Variable

Uniform Variable: The Same Content for all Vertices/Fragments

→*attribute*:      → varying

- Vertex Shader
- Transform And Lighting

→ varying:  → varying:

- Fragment Shader
- Clipping
- Primitive Assembly And Rasterization
- Texture Stages
- Fragment Testing

<table>
<thead>
<tr>
<th>Return type</th>
<th>Function Name</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLint</td>
<td>glGetUniformLocation</td>
<td>GLuint program const GLchar * name</td>
</tr>
<tr>
<td></td>
<td>glUniformXX</td>
<td>GLint location Depends on XX</td>
</tr>
</tbody>
</table>

**Notes:**
- The diagram shows the pipeline of vertex and fragment shaders processing stages.
- The function `glGetUniformLocation` retrieves the location of a uniform variable.
- The function `glUniformXX` sets the value of a uniform variable.

**Keywords:**
- Uniform Variable
- attribute
- varying
- Vertex Shader
- Transform And Lighting
- Fragment Shader
- Clipping
- Primitive Assembly And Rasterization
- Texture Stages
- Fragment Testing
- Vertices
- Primitives
- OpenGL Functions

**Function Descriptions:**
- `glGetUniformLocation(program, name)`: Returns the location of the uniform variable.
- `glUniformXX(location)`: Sets the value of the uniform variable at the given location.

**Important Points:**
- Uniform variables have the same value for all vertices/fragments in a single draw call.
- They can be used to control aspects of rendering, such as material properties or transformations.
- The pipeline stages are critical for understanding the flow of data and how uniforms are processed.
Vertex Shader:
From attribute to varying

→ **attribute**: 
  - input to the vertex shader;
  - Defined per vertex;

→ **varying**: 
  - Output of vertex shader
  - Defined per vertex

→ **varying**: 
  - Input of fragment shader per fragment

→ **varying** variables will be *interpolated* in a perspective-correct fashion across the primitives

http://iloveshaders.blogspot.com/2011/05/how-rasterization-process-works.html
Lighting Example
Vertex Shader

```glsl
varying vec4 diffuseColor;
varying vec3 fragNormal;
varying vec3 lightVector;

uniform vec3 eyeSpaceLightVector;

void main(){
    vec3 eyeSpaceVertex= vec3(gl_ModelViewMatrix * gl_Vertex);
    lightVector= vec3(normalize(eyeSpaceLightVector - eyeSpaceVertex));
    fragNormal = normalize(gl_NormalMatrix * gl_Normal);

    diffuseColor = gl_Color;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```
varying vec4 diffuseColor;
varying vec3 lightVector;
varying vec3 fragNormal;

void main(){

    float perFragmentLighting=max(dot(lightVector,fragNormal),0.0);

    gl_FragColor = diffuseColor * lightingFactor;
}

Toon Shading Example

- Toon Shading
  - Characterized by abrupt change of colors
  - Vertex Shader computes the vertex intensity (declared as varying)
  - Fragment Shader computes colors for the fragment based on the interpolated intensity
Vertex Shader

uniform vec3 lightDir;

varying float intensity;

void main() {
    vec3 ld;
    intensity = dot(lightDir, gl_Normal);
    gl_Position = ftransform();
}

Fragment Shader

varying float intensity;

void main() {
    vec4 color;
    if (intensity > 0.95) color = vec4(1.0, 0.5, 0.5, 1.0);
    else if (intensity > 0.5) color = vec4(0.6, 0.3, 0.3, 1.0);
    else if (intensity > 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;
}
Varying Variable Example

Determine color based on x y z coordinates
Vertex Shader

varying float xpos;
varying float ypos;
varying float zpos;

void main(void) {
    xpos = clamp(gl_Vertex.x, 0.0, 1.0);
    ypos = clamp(gl_Vertex.y, 0.0, 1.0);
    zpos = clamp(gl_Vertex.z, 0.0, 1.0);
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
Fragment Shader

varying float xpos;
varying float ypos;
varying float zpos;

void main (void) {
    gl_FragColor = vec4 (xpos, ypos, zpos, 1.0);
}
Color Key Example

- Set a certain color (say FF00FF as transparent)
Vertex Shader

```c
void main(void) {

    gl_TexCoord[0] = gl_MultiTexCoord0;

    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```
Fragment Shader

uniform sampler2D myTexture;

#define epsilon 0.0001

void main (void) {
    vec4 value = texture2D(myTexture, vec2(gl_TexCoord[0]));
    if (value[0] > 1.0-epsilon) && (value[2] > 1.0-epsilon))
        discard;
    gl_FragColor = value;
}
Color Map Example

- Suppose you want to render an object such that its surface is colored by the temperature.
  - You have the temperatures at the vertices.
  - You want the color to be interpolated between the coolest and the hottest colors.
- Previously, you would calculate the colors of the vertices in your program, and say `glColor()`.
- Now, let's do it in the vertex and pixel shaders...
// uniform qualified variables are changed at most once
// per primitive
uniform float CoolestTemp;
uniform float TempRange;

// attribute qualified variables are typically changed per vertex
attribute float VertexTemp;

// varying qualified variables communicate from the vertex
// shader to the fragment shader
varying float Temperature;
void main()
{
    // compute a temperature to be interpolated per fragment,
    // in the range [0.0, 1.0]
    Temperature = (VertexTemp - CoolestTemp) / TempRange;
    /*
    The vertex position written in the application using glVertex() can
    be read from the built-in variable gl_Vertex. Use this value and
    the current model view transformation matrix to tell the rasterizer
    where this vertex is. Could use ftransform(). */
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
Fragment Shader

// uniform qualified variables are changed at most
// once per primitive by the application, and vec3
// declares a vector of three floating-point numbers
uniform vec3 CoolestColor;
uniform vec3 HottestColor;

// Temperature contains the now interpolated
// per-fragment value of temperature set by the
// vertex shader
varying float Temperature;
void main()
{
    // get a color between coolest and hottest colors, using
    // the mix() built-in function
    vec3 color = mix(CoolestColor, HottestColor, Temperature);
    // make a vector of 4 floating-point numbers by appending an
    // alpha of 1.0, and set this fragment’s color
    gl_FragColor = vec4(color, 1.0);
}
Additional GLSL Info

- Built-in names for accessing OpenGL states and for communicating with OpenGL fixed functionality
  - `gl_Position`
  - `gl_FragCoord/gl_FrontFacing/gl_ClipDistance[]/gl_PointCoord/gl_PrimitiveID/gl_SampleID/gl_SamplePosition/gl_SampleMaskIn[]`;

- Type qualifiers `attribute`, `uniform`, and `varying`
  - Or `in/out` in OpenGL 3.X/4.X
Types

- Vector types are supported for floats, integers, and booleans
  - Can be 2-, 3-, or 4- components
  - float, vec2, vec3, vec4
  - int, ivec2, ivec3, ivec4
  - bool, bvec2, bvec3, bvec4

- Matrix types
  - mat2, mat3, mat4

- Texture access
  - sampler1D, sampler2D, sampler3D

- The complete list of GLSL data types functions are in Section 4 of GLSL Spec 4.2.
# Built-in Functions

<table>
<thead>
<tr>
<th>Category</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trigonometry/angle</td>
<td>radians, degrees, sin, cos, tan, asin, acos, atan</td>
</tr>
<tr>
<td>Exponential</td>
<td>pow, exp2, log2, sqrt, inversesqrt</td>
</tr>
<tr>
<td>Geometric and matrix</td>
<td>length, distance, dot, cross, normalize, ftransform, faceforward, reflect, matrixCompMult</td>
</tr>
<tr>
<td>Misc</td>
<td>abs, sign, floor, ceil, fract, mod, min, max, clamp, mix, step, smoothstep</td>
</tr>
</tbody>
</table>

- The complete list of build-in functions are in Section 8 of GLSL Spec 4.2.
Vertex Program Capabilities

- Vertex program can do general processing, including things like:
  - Vertex transformation
  - Normal transformation, normalization and rescaling
  - Lighting
  - Color material application
  - Clamping of colors
  - Texture coordinate generation
  - Texture coordinate transformation
**Variable Qualifies: From **attribute** to **varying** under uniform**

- **attribute:**
  - Input to the vertex shader;
  - Defined per vertex;
  - E.g. vertex color/normal/coordinates

- **varying**
  - Output of vertex shader
  - Defined per vertex

- **Uniform:** The same information used by all vertices/fragment.
  - E.g. ModelView transformation matrices
void main()
{
    /*
    compute a temperature to be interpolated per fragment,
    in the range [0.0, 1.0]
    */
    Temperature = (VertexTemp - CoolestTemp) / TempRange;
    gl_Position = ftransform();
}
Intervening Fixed Functionality

- Results from vertex processing undergo:
  - Perspective division on clip coordinates
  - Viewport mapping
  - Clipping, including user clipping
  - Color clamping or masking (for built-in varying variables that deal with color, but not user-defined varying variables)
  - Depth range
  - Front face determination and culling
  - Interpolate colors, texture coordinate, and user-defined varying variables
  - Etc.
Intervening Fixed Functionality: Rasterization

Rasterization of Lines\(^1\) \hspace{2cm} \text{Rasterization of Lines/Polygons}\(^2\)

1. OpenGL 4.2 SPEC, 08/22/2011.

\(^1\) Rasterization of Lines
\(^2\) Rasterization of Lines/Polygons
Fragment Program Capabilities

Fragment shader can do general processing, like:
- Operations on interpolated values
- Texture access
- Texture application
- Fog
- Color sum
- Color matrix
- Discard fragment
- etc
Fragment Program

- Output of vertex shader is the input to the fragment shader
  - Compatibility is checked when linking occurs
  - Compatibility between the two is based on *varying variables* that are defined in both shaders and that match in type and name

- Fragment shader is executed for each fragment produced by rasterization

- For each fragment, fragment shader has access to the interpolated value for each varying variable
  - Color, normal, texture coordinates, arbitrary values
Fragment Processor Output

- In OpenGL 2.X
  - Output of the fragment processor goes on to the fixed function fragment operations and frame buffer operations using built-in variables
    - `gl_FragColor` – computed R, G, B, A for the fragment
    - `gl_FragDepth` – computed depth value for the fragment
    - `gl_FragData[n]` – arbitrary data per fragment, stored in multiple render targets
Fragment Shader: Example

// uniform qualified variables are changed at most
// once per primitive by the application, and vec3
// declares a vector of three floating-point numbers
uniform vec3 CoolestColor;
uniform vec3 HottestColor;

// Temperature contains the now interpolated
// per-fragment value of temperature set by the
// vertex shader
varying float Temperature;

void main()
{
    // get a color between coolest and hottest colors, using
    // the mix() built-in function
    vec3 color = mix(CoolestColor, HottestColor, Temperature);
    // make a vector of 4 floating-point numbers by appending an
    // alpha of 1.0, and set this fragment's color
    gl_FragColor = vec4(color, 1.0);
}
Fragment Program Capabilities

- Pass the fragment shader output to framebuffers for the following test
  - Scissor test/Alpha test/Depth test/Stencil test/Blending, etc.
  - Values are destined for writing into the frame buffer if all back end tests (stencil, depth etc.) pass

- Clamping or format conversion to the target buffer is done automatically outside of the fragment shader
GLSL for OpenGL 3.X/4.X

- Two profiles for shader languages
  - Compatibility (1.X/2.X)
  - Core (3.X/4.X)
- Main changes
  - Reduction of built-in states
    - No transformation matrix,
    - No lighting,
    - No built-in attributes for vertex/colors/normals, etc
- Change of type qualifiers
- Additional programmable stages
  - Geometry/Tessellation shader
- Much more …

- The features for compatibility profile are listed in GLSL Spec 4.2.
OpenGL 3.X: Type Qualifiers

- **in**
  - for function parameters copied into a function, but not copied out

- **out**
  - for function parameters copied out of a function, but not copied in

- Can be combined with auxiliary storage qualifiers `centroid/sample/patch`

The complete list of GLSL data types functions are in Section 4.3 of GLSL Spec 4.2.