GL Shading Language (GLSL)

- GLSL: high level C-like language
- Main program (e.g. example1.cpp) program written in C/C++
- Vertex and Fragment shaders written in GLSL
- From OpenGL 3.1, application must use shaders

What does keyword out mean?

```c
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
out vec3 color_out;

void main(void){
    gl_Position = vPosition;
    color_out = red;
}
```

gl_Position not declared

Built-in types (already declared, just use)
Passing values

- Variable declared `out` in vertex shader can be declared as `in` in fragment shader and used
- Why? To pass result of vertex shader calculation to fragment shader

```cpp
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
out vec3 color_out;

void main(void){
    gl_Position = vPosition;
    color_out = red;
}

in vec3 color_out;

void main(void){
    // can use color_out here.
}
```
Data Types

- **C types**: `int`, `float`, `bool`
- **GLSL types**:
  - `float vec2`: e.g. `(x, y)` // vector of 2 floats
  - `float vec3`: e.g. `(x, y, z)` or `(R, G, B)` // vector of 3 floats
  - `float vec4`: e.g. `(x, y, z, w)` // vector of 4 floats

```cpp
const float vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
out float vec3 color_out;

void main(void)
{
  gl_Position = vPosition;
  color_out = red;
}
```

- Also:
  - `int (ivec2, ivec3, ivec4)` and
  - `boolean (bvec2, bvec3, bvec4)`
Data Types

- Matrices: mat2, mat3, mat4
  - Stored by columns
  - Standard referencing m[row][column]
- Matrices and vectors are basic types
  - can be passed in and out from GLSL functions
- E.g
  
    mat3 func(mat3 a)
- **No pointers** in GLSL
- Can use C structs that are copied back from functions
Qualifiers

- GLSL has many C/C++ qualifiers such as `const`.
- Supports additional ones.
- Variables can change:
  - Once per vertex.
  - Once per fragment.
  - Once per primitive (e.g. triangle).
  - At any time in the application.
- Example: variable vPosition may be assigned once per vertex.

```cpp
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
out vec3 color_out;

void main(void){
    gl_Position = vPosition;
    color_out = red;
}
```
Operators and Functions

- **Standard C functions**
  - **Trigonometric**: cos, sin, tan, etc
  - **Arithmetic**: log, min, max, abs, etc
  - Normalize, reflect, length

- **Overloading of vector and matrix types**

```c
mat4 a;
vec4 b, c, d;
c = b*a;       // a column vector stored as a 1d array
d = a*b;       // a row vector stored as a 1d array
```
Swizzling and Selection

- Can refer to array elements by element using [] or selection (.) operator with
  - x, y, z, w
  - r, g, b, a
  - s, t, p, q
  - `vec4 a;`
  - `a[2], a.b, a.z, a.p` are the same

- **Swizzling** operator lets us manipulate components
  `a.yz = vec2(1.0, 2.0);`
3D Applications

- **2D points**: $(x, y)$ coordinates
- **3D points**: have $(x, y, z)$ coordinates
Setting up 3D Applications: Main Steps

- Programming 3D similar to 2D
  1. Load representation of 3D object into data structure
     - Each vertex has (x,y,z) coordinates.
     - Store as `vec3` NOT `vec2`
  2. Draw 3D object
  3. **Set up Hidden surface removal:** Correctly determine order in which primitives (triangles, faces) are rendered (e.g. Blocked faces **NOT** drawn)
3D Coordinate Systems

- Vertex \((x,y,z)\) positions specified on coordinate system
- OpenGL uses **right hand coordinate system**

---

**Right hand coordinate system**
- Tip: sweep fingers \(x\)-\(y\): thumb is \(z\)

**Left hand coordinate system**
- Not used in OpenGL
Generating 3D Models: GLUT Models

- Make GLUT 3D calls in **OpenGL program** to generate vertices describing different shapes (Restrictive?)

- Two types of GLUT models:
  - Wireframe Models
  - Solid Models
3D Modeling: GLUT Models

- Basic Shapes
  - **Cone**: `glutWireCone()`, `glutSolidCone()`
  - **Sphere**: `glutWireSphere()`, `glutSolidSphere()`
  - **Cube**: `glutWireCube()`, `glutSolidCube()`

- More advanced shapes:
  - Newell Teapot: (symbolic)
  - Dodecahedron, Torus
3D Modeling: GLUT Models

- Glut functions under the hood
  - generate sequence of points that define a shape
  - Generated vertices and faces passed to OpenGL for rendering

- **Example**: `glutWireCone` generates sequence of vertices, and faces defining cone and connectivity
Polygonal Meshes

- Modeling with GLUT shapes (cube, sphere, etc) too restrictive
- Difficult to approach realism. E.g. model a horse
- Preferred way is using polygonal meshes:
  - Collection of polygons, or faces, that form “skin” of object
  - More flexible, represents complex surfaces better
- Examples:
  - Human face
  - Animal structures
  - Furniture, etc

Each face of mesh is a polygon
Polygonal Mesh Example

Smoothed Out with Shading (later)

Mesh (wireframe)
Polygonal Meshes

- Meshes now standard in graphics
- OpenGL Good at drawing polygons, triangles
- Mesh = sequence of polygons forming thin skin around object
- Simple meshes exact. (e.g. barn)
- Complex meshes approximate (e.g. human face)
Different Resolutions of Same Mesh

Original: 424,000 triangles
60,000 triangles (14%).
1000 triangles (0.2%)

(courtesy of Michael Garland and Data courtesy of Iris Development.)
Representing a Mesh

- Consider a mesh

- There are 8 vertices and 12 edges
  - 5 interior polygons
  - 6 interior (shared) edges (shown in orange)

- Each vertex has a location $v_i = (x_i, y_i, z_i)$
Simple Representation

- Define each polygon by \((x,y,z)\) locations of its vertices
- OpenGL code

```cpp
vertex[i] = vec3(x1, y1, z1);
vertex[i+1] = vec3(x6, y6, z6);
vertex[i+2] = vec3(x7, y7, z7);
i+=3;
```
Issues with Simple Representation

- Declaring face f1
  
  ```
  vertex[i] = vec3(x1, y1, z1);
  vertex[i+1] = vec3(x7, y7, z7);
  vertex[i+2] = vec3(x8, y8, z8);
  vertex[i+3] = vec3(x6, y6, z6);
  ```

- Declaring face f2
  
  ```
  vertex[i] = vec3(x1, y1, z1);
  vertex[i+1] = vec3(x2, y2, z2);
  vertex[i+2] = vec3(x7, y7, z7);
  ```

- Inefficient and unstructured
  
  - **Repeats**: vertices **v1 and v7 repeated** while declaring f1 and f2
  - Shared vertices shared declared multiple times
  - Delete vertex? Move vertex? Search for all occurrences of vertex
Geometry vs Topology

- Better data structures separate **geometry** from **topology**
  - **Geometry**: \((x,y,z)\) locations of the vertices
  - **Topology**: How vertices and edges are connected
- **Example**:
  - A polygon is **ordered list** of vertices
  - An edge connects successive pairs of vertices
  - Topology holds even if geometry changes (vertex moves)

Example: even if we move \((x,y,z)\) location of \(v_1\), \(v_1\) still connected to \(v_6, v_7\) and \(v_2\)
Polygon Traversal Convention

- Use the *right-hand rule* = *counter-clockwise* encirclement of outward-pointing normal
- Focus on direction of traversal
  - Orders \(\{v_1, v_0, v_3\}\) and \(\{v_3, v_2, v_1\}\) are same (*ccw*)
  - Order \(\{v_1, v_2, v_3\}\) is different (*clockwise*)
- What is outward-pointing normal?
Normal Vector

- **Normal vector**: Direction each polygon is facing
- Each mesh polygon has a **normal vector**
- Normal vector used in shading
**Vertex Lists**

- **Vertex list**: $(x,y,z)$ of vertices (its geometry) are put in array
- Use pointers from vertices into vertex list
- **Polygon list**: vertices connected to each polygon (face)

**Topology example**: Polygon P1 of mesh is connected to vertices $(v1,v7,v6)$

**Geometry example**: Vertex $v7$ coordinates are $(x7,y7,z7)$. Note: If $v7$ moves, changed once in vertex list
Vertex List Issue: Shared Edges

- Vertex lists draw filled polygons correctly
- If each polygon is drawn by its edges, shared edges are drawn twice

- **Alternatively:** Can store mesh by *edge list*
Edge List

Simply draw each edge once:
E.g. e1 connects v1 and v6

Note: Polygons are not represented.
Modeling a Cube

- In 3D, declare vertices as \((x,y,z)\) using \texttt{point3 v[3]}
- Define \texttt{global arrays} for vertices and colors

```c
typedef vec3 point3;
point3 vertices[] = {point3(-1.0,-1.0,-1.0),
                    point3(1.0,-1.0,-1.0), point3(1.0,1.0,-1.0),
                    point3(-1.0,1.0,-1.0), point3(-1.0,-1.0,1.0),
                    point3(1.0,-1.0,1.0), point3(1.0,1.0,1.0),
                    point3(-1.0,1.0,1.0)};

typedef vec3 color3;
color3 colors[] = {color3(0.0,0.0,0.0),
                  color3(1.0,0.0,0.0), color3(1.0,1.0,0.0),
                  color(0.0,1.0,0.0), color3(0.0,0.0,1.0),
                  color3(1.0,0.0,1.0), color3(1.0,1.0,1.0),
                  color3(0.0,1.0,1.0)};
```
Drawing a triangle from list of indices

Draw a triangle from a list of indices into the array `vertices` and assign a color to each index

```cpp
void triangle(int a, int b, int c, int d)
{
    vcolors[i] = colors[d];
    position[i] = vertices[a];
    vcolors[i+1] = colors[d]);
    position[i+1] = vertices[b];
    vcolors[i+2] = colors[d];
    position[i+2] = vertices[c];
    i+=3;
}
```

Variables `a`, `b`, `c` are indices into vertex array
Variable `d` is index into color array
Note: Same face, so all three vertices have same color
void colorcube( )
{
    quad(0,3,2,1);
    quad(2,3,7,6);
    quad(0,4,7,3);
    quad(1,2,6,5);
    quad(4,5,6,7);
    quad(0,1,5,4);
}

Note: vertices ordered (counterclockwise) so that we obtain correct outward facing normals
References