Programming with OpenGL
Part 3: Shaders

CS 537 Interactive Computer Graphics
Prof. David E. Breen
Department of Computer Science
Geometric Preliminaries

• **Affine Geometry**
  - Scalars + Points + Vectors and their ops

• **Euclidean Geometry**
  - Affine Geometry lacks angles, distance
  - New op: Inner/Dot product, which gives
    - Length, distance, normalization
    - Angle, Orthogonality, Orthogonal projection

• **Projective Geometry**
Affine Geometry

**Affine Operations:**

- **Affine Combinations:** \( \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n \)
  where \( v_1, v_2, \ldots, v_n \) are vectors and \( \sum_i \alpha_i = 1 \)

**Example:** \( R = (1 - \alpha)P + \alpha Q \)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector ( \leftarrow ) scalar ( \cdot ) vector</td>
<td>scalar-vector multiplication</td>
</tr>
<tr>
<td>vector ( \leftarrow ) vector + vector</td>
<td>vector-vector addition</td>
</tr>
<tr>
<td>vector ( \leftarrow ) vector - vector</td>
<td>point-point difference</td>
</tr>
<tr>
<td>point ( \leftarrow ) point + vector</td>
<td>point-vector addition</td>
</tr>
<tr>
<td>point ( \leftarrow ) point - vector</td>
<td></td>
</tr>
</tbody>
</table>

Diagram:

- Vector addition
- Point subtraction
- Point-vector addition
• Vector: an \( n \)-tuple of real numbers
• Vector Operations
  - Vector addition: \( u + v = w \)
    - Commutative, associative, identity element (0)
  - Scalar multiplication: \( c \mathbf{v} \)

• Note: Vectors and Points are different
  - Can not add points
  - Can find the vector between two points
A linear combination of the vectors $v_1, v_2, \ldots, v_n$ is any vector of the form

$$\alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n$$

where $\alpha_i$ is a real number (i.e. a scalar).

Dot Product:

$$u \cdot v = \sum_{k=1}^{n} u_k v_k$$

a real value $u_1 v_1 + u_2 v_2 + \ldots + u_n v_n$ written as $u \cdot v$
Fun with Dot Products

- **Euclidian Distance from** \((x,y)\) to \((0,0)\) in general:
  \[
  \sqrt{x^2 + y^2}
  \]
  which is just:
  \[
  \sqrt{\langle x \rangle \cdot \langle x \rangle}
  \]

- This is also the length of vector \(\vec{v}\):
  \[
  ||\vec{v}|| \text{ or } |\vec{v}|
  \]

- **Normalization** of a vector:
  \[
  \hat{\vec{v}} = \frac{\vec{v}}{||\vec{v}||}.
  \]

- **Orthogonal vectors**:
  \[
  \vec{u} \cdot \vec{v} = 0
  \]
Projections & Angles

- **Angle between vectors**,  \( \theta \)
  \[
  \vec{u} \cdot \vec{v} = |\vec{u}| |\vec{v}| \cos(\theta)
  \]

  \[
  \theta = \text{ang}(\vec{u}, \vec{v}) = \cos^{-1}
  \left( \frac{\vec{u} \cdot \vec{v}}{|\vec{u}||\vec{v}|} \right)
  = \cos^{-1}(\hat{u} \cdot \hat{v}).
  \]

- **Projection of vectors**
  \[
  \vec{u}_1 = \frac{(\vec{u} \cdot \vec{v})}{(\vec{v} \cdot \vec{v})} \vec{v}
  \]
  \[
  \vec{u}_2 = \vec{u} - \vec{u}_1.
  \]
Matrices and Matrix Operators

• A \( n \)-dimensional vector:

\[
\begin{bmatrix}
  x_1 \\
  . \\
  . \\
  . \\
  x_n 
\end{bmatrix}
\]

• Matrix Operations:
  - Addition/Subtraction
  - Identity
  - Multiplication
    - Scalar
    - Matrix Multiplication

• Implementation issue: Where does the index start? (0 or 1, it’s up to you…)

\[
\begin{align*}
A + B &= B + A \\
A + (B + C) &= (A + B) + C \\
(cd)A &= c(dA) \\
1A &= A \\
c(A + B) &= cA + cB \\
(c + d)A &= cA + dA
\end{align*}
\]
Matrix Multiplication

- \([C] = [A][B]\)
- Sum over rows & columns
- Recall: matrix multiplication is *not* commutative

- **Identity Matrix:**
  1s on diagonal
  0s everywhere else

\[
c_{ij} = \sum_{s=1}^{m} a_{is}b_{sj}
\]

\[
\begin{bmatrix}
a_{11} & a_{12} & a_{13} & a_{14} \\
a_{21} & a_{22} & a_{23} & a_{24} \\
a_{31} & a_{32} & a_{33} & a_{34} \\
a_{41} & a_{42} & a_{43} & a_{44}
\end{bmatrix}
\begin{bmatrix}
b_{11} & b_{12} & b_{13} & b_{14} \\
b_{21} & b_{22} & b_{23} & b_{24} \\
b_{31} & b_{32} & b_{33} & b_{34} \\
b_{41} & b_{42} & b_{43} & b_{44}
\end{bmatrix}
= 
\begin{bmatrix}
c_{11} & c_{12} & c_{13} & c_{14} \\
c_{21} & c_{22} & c_{23} & c_{24} \\
c_{31} & c_{32} & c_{33} & c_{34} \\
c_{41} & c_{42} & c_{43} & c_{44}
\end{bmatrix}
\]
Matrix Determinants

• A single real number
• Computed recursively
  \[ \text{det}(A) = \sum_{j=1}^{n} A_{i,j} (-1)^{i+j} M_{i,j} \]
• Example:
  \[ \text{det}\begin{bmatrix} a & c \\ b & d \end{bmatrix} = ad - bc \]
• Uses:
  - Find vector ortho to two other vectors
  - Determine the plane of a polygon
Cross Product

• Given two non-parallel vectors, \( \mathbf{A} \) and \( \mathbf{B} \)
• \( \mathbf{A} \times \mathbf{B} \) calculates third vector \( \mathbf{C} \) that is orthogonal to \( \mathbf{A} \) and \( \mathbf{B} \)
• \( \mathbf{A} \times \mathbf{B} = (a_y b_z - a_z b_y, a_z b_x - a_x b_z, a_x b_y - a_y b_x) \)

\[
\mathbf{A} \times \mathbf{B} = \begin{vmatrix} \hat{x} & \hat{y} & \hat{z} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix}
\]
Matrix Transpose & Inverse

- **Matrix Transpose:** Swap rows and cols:
  
  \[ A = \begin{bmatrix} 2 \\ 8 \end{bmatrix}, \quad A^T = \begin{bmatrix} 2 & 8 \end{bmatrix} \]

- **Facts about the transpose:**
  
  \[(A^T)^T = A\]
  
  \[(A + B)^T = A^T + B^T\]
  
  \[(cA)^T = c(A^T)\]
  
  \[(AB)^T = B^T A^T\]

- **Matrix Inverse:** Given \( A \), find \( B \) such that
  
  \[ AB = BA = I \]

  \( B \rightarrow A^{-1} \)

  (only defined for square matrices)
Objectives

- Simple Shaders
  - Vertex shader
  - Fragment shaders
- Programming shaders with GLSL
- Finish first program
Vertex Shader Applications

- Moving vertices
  - Transformations
    - Modeling
    - Projection
  - Morphing
  - Wave motion
  - Fractals
  - Particle systems

- Lighting
  - More realistic shading models
  - Cartoon shaders
Fragment Shader Applications

Per fragment lighting calculations

per vertex lighting (Gouraud shading)  per fragment lighting (Phong shading)

Texture mapping

- Procedural textures
- Environment mapping
- Bump mapping
Writing Shaders

• First programmable shaders were programmed in an assembly-like manner
• OpenGL extensions added vertex and fragment shaders
• Cg (C for graphics) C-like language for programming shaders
  - Works with both OpenGL and DirectX
  - Interface to OpenGL complex
• OpenGL Shading Language (GLSL)
GLSL

- OpenGL Shading Language
- Part of OpenGL 2.0 and up
- High level C-like language
- New data types
  - Matrices
  - Vectors
  - Samplers
- As of OpenGL 3.1, application **must** provide shaders
Execution Model

![Diagram of GPU processing pipeline]

Vertex data
Shader Program

Application Program

Vertex Shader

Primitive Assembly

glDrawArrays

Simple Vertex Shader

input from application (GLSL 1.5)

in vec4 vPosition;
void main(void)
{
    gl_Position = vPosition;  // Simple pass-through
}

Use “attribute vec4 vPosition” for GLSL 1.4

must link to variable in application

built in variable
Execution Model

Vertices → Vertex Processor → Clipper and Primitive Assembler → Rasterizer → Fragment Processor → Pixels

Application

Shader Program

Rasterizer → Fragment Shader → Frame Buffer

Fragment → Fragment Color

void
main()
{
    gl_FragColor = vec4( 1.0, 0.0, 0.0, 1.0 );
}

Every fragment simply colored red
out vec4 fragcolor;
void main(void)
{
    fragcolor = vec4(1.0, 0.0, 0.0, 1.0);
}

Every fragment simply colored red
Data Types

- **C types:** int, float, bool, uint, double
- **Vectors:**
  - float vec2, vec3, vec4
  - Also int (ivec), boolean (bvec), uvec, dvec
- **Matrices:** mat2, mat3, mat4
  - Stored by columns
  - Standard referencing m[row][column]
- **C++ style constructors**
  - vec3 a = vec3(1.0, 2.0, 3.0)
  - vec2 b = vec2(a)
Pointers

• There are no pointers in GLSL
• We can use C structs which can be copied back from functions
• Because matrices and vectors are basic types they can be passed into and out from GLSL functions, e.g.

```c
mat3 func(mat3 a)
```
Qualifiers

• GLSL has many of the same qualifiers such as `const` as C/C++

• Need others due to the nature of the execution model

• Variables can change
  - Once per vertex
  - Once per primitive
  - Once per fragment
  - At any time in the application

• Vertex attributes are interpolated by the rasterizer into fragment attributes
Attribute Qualifier

- Attribute-qualified variables can change at most once per vertex
- There are a few built in variables such as gl_Position but most have been deprecated
- User defined (in application program)
  - Use ‘in’ qualifier to get to shader
    - in float temperature
    - in vec3 velocity
Uniform Qualifier

- Variables that are constant for an entire primitive
- Can be changed in application and sent to shaders
- Cannot be changed in shader
- Used to pass information to shader such as the bounding box of a primitive
Varying Qualifier

• Variables that are passed from vertex shader to fragment shader
• Automatically interpolated by the rasterizer
• Old style used the varying qualifier
  
  ```glsl
  varying vec4 color;
  ```
• Now use `out` in vertex shader and `in` in the fragment shader
  
  ```glsl
  out vec4 color;
  ```
Example: Vertex Shader

```
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
in vec4 vPosition;
out vec4 color_out;
void main(void)
{
    gl_Position = vPosition;
    color_out = vPosition.x * red;
}
```
in vec4 color_out;
void main(void)
{
    // Now deprecated
    gl_FragColor = color_out;
}
in vec4 color_out;
out vec4 fragcolor;
void main(void)
{
    fragcolor = color_out;
}
User-defined functions

• Similar to C/C++ functions
• Except
  - Cannot be recursive
  - Specification of parameters

```c
returnType MyFunction(in float inputValue,
                      out int outputValue,
                      inout float inAndOutValue);
```
Passing values

• call by `value-return`
• Variables are copied in
• Returned values are copied back
• Three possibilities
  - in
  - out
  - inout
Operators and Functions

- Standard C functions
  - Trigonometric
  - Arithmetic
  - 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
  - a[2], a.b, a.z, a.p are the same

• **Swizzling** operator lets us manipulate components

```cpp
vec4 a, b;
a.yz = vec2(1.0, 2.0);
a.xw = b.yy;
```
Programming with OpenGL
Part 4: Color and Attributes
Objectives

• Expanding primitive set
• Adding color
• Vertex attributes
• Uniform variables
OpenGL Primitives

- GL_POINTS
- GL_LINES
- GL_LINE_STRIP
- GL_LINE_LOOP
- GL_TRIANGLES
- GL_TRIANGLE_STRIP
- GL_TRIANGLE_FAN
Polygon Issues

- OpenGL will only display triangles
  - Simple: edges cannot cross
  - Convex: All points on line segment between two points in a polygon are also in the polygon
  - Flat: all vertices are in the same plane

- Application program must tessellate a polygon into triangles (triangulation)

- OpenGL 4.1 contains a tessellator
Polygon Testing

• Conceptually simple to test for simplicity and convexity
• Time consuming
• Earlier versions assumed both and left testing to the application
• Present version only renders triangles
• Need algorithm to triangulate an arbitrary polygon
Good and Bad Triangles

• Long thin triangles render badly

• Equilateral triangles render well
• Maximize minimum angle
• Delaunay triangulation for unstructured points
• Convex polygon

• Start with abc, remove b, then acd, ....
Non-convex (concave)
Recursive Division

• There are a variety of recursive algorithms for subdividing concave polygons
Attributes

• Attributes determine the appearance of objects
  - Color (points, lines, polygons)
  - Size and width (points, lines)
  - Stipple pattern (lines, polygons)
  - Polygon mode
    • Display as filled: solid color or stipple pattern
    • Display edges
    • Display vertices

• Only a few (glPointSize) are supported by OpenGL functions
RGB color

• Each color component is stored separately in the frame buffer
• Usually 8 bits per component in buffer
• Color values can range from 0.0 (none) to 1.0 (all) using floats or over the range from 0 to 255 using unsigned bytes
Smooth Color

• Default is *smooth* shading
  - OpenGL interpolates vertex colors across visible polygons

• Alternative is *flat shading*
  - Color of first vertex determines fill color
  - Handle in shader
Setting Colors

• Colors are ultimately set in the fragment shader but can be determined in either shader or in the application
  • Application color: pass to vertex shader as a uniform variable (next lecture) or as a vertex attribute
  • Vertex shader color: pass to fragment shader as varying variable (next lecture)
  • Fragment color: can alter via shader code