Math Review

CS 432 Interactive Computer Graphics
Prof. David E. Breen
Department of Computer Science
Geometric Preliminaries

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

• **Euclidian 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:**

  - Vector addition
  - Point subtraction
  - Point-vector addition

<table>
<thead>
<tr>
<th>Operation</th>
<th>Expression</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector ← scalar * vector</td>
<td>$\alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n$</td>
<td>where $v_1, v_2, \ldots, v_n$ are vectors and $\sum_i \alpha_i = 1$</td>
</tr>
<tr>
<td>Vector ← vector + vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector ← vector - vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector ← point - point</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Point ← point + vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Point ← point - vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scalar-vector multiplication</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector-vector addition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Point-point difference</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Point-vector addition</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Example:**

$$R = (1 - \alpha)P + \alpha Q$$
Mathematical Preliminaries

- Vector: an \( n \)-tuple of real numbers
- Vector Operations
  - Vector addition: \( u + v = w \)
    - Commutative, associative, identity element (0)
  - Scalar multiplication: \( cv \)
- Note: Vectors and Points are different
  - Can not add points
  - Can find the vector between two points
Linear Combinations & Dot Products

• 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 \((0, 0)\)

\[
\sqrt{x^2 + y^2} \quad \text{in general:} \quad \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}
\]

which is just:

\[
\sqrt{\vec{x} \cdot \vec{x}}
\]

- This is also the length of vector \(\vec{v}\):

\[
\|\vec{v}\| \quad \text{or} \quad |\vec{v}|
\]

- **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:
  \[
  \begin{bmatrix}
  x_1 \\
  \vdots \\
  \vdots \\
  x_n 
  \end{bmatrix}
  \]

- 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

\[
c_{ij} = \sum_{s=1}^{m} a_{is} \cdot b_{sj}
\]
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, A and B
• $A \times B$ calculates third vector $C$ that is orthogonal to $A$ and $B$
• $A \times B = (a_y b_z - a_z b_y, a_z b_x - a_x b_z, a_x b_y - a_y b_x)$

$$A \times B = \begin{vmatrix} \vec{x} & \vec{y} & \vec{z} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix}$$
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

  $AB = BA = I \quad B \rightarrow A^{-1}$

  (only defined for square matrices)
Derivatives of Polynomials

\[ f(x) = \alpha x^n \]
\[ \frac{df(x)}{dx} = \alpha nx^{n-1} \]
\[ f(x) = 5x^3 \]
\[ \frac{df(x)}{dx} = 15x^2 \]
Partial Derivatives of Polynomials

\[ f(x, y) = \alpha x^n y^m \]
\[ \frac{\partial f(x, y)}{\partial x} = \alpha n x^{n-1} y^m \]
\[ f(x, y) = 5x^3 y^4 \]
\[ \frac{\partial f(x, y)}{\partial x} = 15x^2 y^4 \]
Programming with OpenGL
Part 3: Shaders

CS 432 Interactive Computer Graphics
Prof. David E. Breen
Department of Computer Science
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)
Fragment Shader Applications

Texture mapping

- Procedural textures
- Environment mapping
- Bump mapping
• 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 ES 1.0 and up
• High level C-like language
• New data types
  - Matrices
  - Vectors
  - Samplers
• As of OpenGL 3.1, application **must** provide shaders
Execution Model

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

Vertex Data
Uniform Variables
Shader Program

Application Program → GPU

Vertex Shader

Primitive Assembly

glDrawArrays

Vertex
Simple (Old) Vertex Shader

```glsl
attribute vec4 vPosition;

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

Use “in vec4 vPosition” for GLSL 1.5

---

input from application (GLSL 1.4)

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

must link to variable in application

built in variable
Simple (New) Vertex Shader

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

- **input from application (GLSL ES 3.0)**
- **in vec4 aPosition;**
- **void main(void) {**
  - **gl_Position = aPosition;**
  - **Simple pass-through**
- **}**
- **must link to variable in application**
- **built in variable**
Execution Model

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

Application

Uniform Variables

Shader Program

Rasterizer ➔ Fragment Shader ➔ Frame Buffer

2D Geom ➔ Fragment ➔ Fragment Color

Triangles to Fragments

Clipping Window

Viewport

3D

2D

(1,1)

(-1,-1)

(w,h) pixels

(0,0)
Triangles to Fragments (Rasterization)
Triangles to Fragments (Rasterization)

Each red square is a fragment!
What is a Fragment?

- An enhanced pixel
- Has an \((i,j)\) location in viewport coordinates \((\text{gl\_FragCoord})\)
- Associated interpolated varying data
  - Computed by the Rasterizer
    - Depth
  - Interpolated by the Rasterizer
    - Color
    - Texture coordinates
    - Normal
    - Etc.
Simple (Old) Fragment Program

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

Every fragment simply colored red
Simple (New) Fragment Program

```c
out vec4 fragcolor;
void main(void)
{
    fragcolor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Every fragment simply colored red
Declaring Variables

WARNING!!!!!

• Only declare variables that you actually use in your shader programs!

• In other words, if you change your shader programs and stop using a variable, REMOVE ITS DECLARATION

• Unused declared variables will generate incomprehensible, app-killing errors
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.

  mat3 func(mat3 a)

• variables passed by copying
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 primitive
  - Once per vertex
  - Once per fragment
  - At any time in the application
- Vertex attributes are interpolated by the rasterizer into fragment attributes
Attribute Interpolation

- Vertex attributes are interpolated by the rasterizer into fragment attributes.
- For example, a color associated with a vertex will be interpolated over the fragments/pixels generated from the associated triangle.
(Old) 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)
  - attribute float temperature
  - attribute vec3 velocity
• Current versions of GLSL use in qualifier to get attribute data to the vertex shader
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 time, or a color or bounding box of a primitive
(Old) Varying Qualifier

• Variables that are passed from vertex shader to fragment shader
• Automatically interpolated by the rasterizer

• With WebGL 1.0, GLSL uses the varying qualifier in both shaders
  
  \begin{verbatim}
  varying vec4 color;
  \end{verbatim}

• Current versions of WebGL use \texttt{out} in vertex shader and \texttt{in} in the fragment shader
  
  \begin{verbatim}
  out vec4 vcolor;  // vertex shader
  in vec4 vcolor;   // fragment shader
  \end{verbatim}
Our Naming Convention

- Attributes passed to vertex shader have names beginning with ‘a’ (aPosition, aColor) in both the application and the shader
  - Note that these are different entities with the same name
- Varying variables begin with ‘v’ (vColor) in both shaders
  - must have same name and type
- Uniform variables are can have any/the same name in application and shaders
in vec4 aPosition, aColor;
out vec4 vColor;
void main()
{
    gl_Position = aPosition;
    vColor = aColor;
}
Corresponding Fragment Shader

```glsl
precision mediump float;

in vec4 vColor;
out vec4 fColor
void main()
{
    fColor = vColor;
}
```
Precision Declaration

• In GLSL for WebGL we must specify desired precision in fragment shaders
  - artifact inherited from OpenGL ES
  - ES must run on very simple embedded devices that may not support 32-bit floating point
  - All implementations must support mediump
  - No default for float in fragment shader

• Can use preprocessor directives (#ifdef) to check if highp supported and, if not, default to mediump
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
in vec4 aPosition;
out vec4 vColor;
void main(void)
{
    gl_Position = aPosition;
    vColor = aPosition.x * red;
}
precision highp float;

in vec4 vColor;
out vec4 fColor;
void main(void)
{
    fColor = vColor;
}
var cBuffer = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );
gl.bufferData( gl.ARRAY_BUFFER, flatten(colors),
    gl.STATIC_DRAW );

var aColor = gl.getAttribLocation( program, "aColor" );
gl.vertexAttribPointer( aColor, 3, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( aColor );
Sending a Uniform Variable

// in application
vec4 color = vec4(1.0, 0.0, 0.0, 1.0);
colorLoc = gl.getUniformLocation(program, "color");
gl.uniform4fv(colorLoc, color);

// in fragment shader (similar in vertex shader)
uniform vec4 color;
out fColor;
void main()
{
    fColor = color;
}
Consistent Declaration of Variables

• Data is being passed between multiple programs (application, vertex & fragment shaders)

• Variables storing the same data in different programs must be declared consistently!
Consistent Declaration of Variables (Example)

- **attribute** and **varying** variables

- In application

```javascript
var aColor = gl.getAttribLocation(program, "aColor");
gl.vertexAttribPointer(aColor, 3, gl.FLOAT, false, 0, 0);
```

- In vertex shader

```glsl
in vec3 aColor;
out vec3 vColor;
vColor = aColor;
```

- In fragment

```glsl
in vec3 vColor;
out vec4 fColor;
fColor = vec4(vColor, 1.0);
```
Consistent Declaration of Variables (Example)

• *uniform* variables

• In application

\[
\text{vec4 color} = \text{vec4}(1.0, 0.0, 0.0, 1.0);
\]
\[
\text{colorLoc} = \text{gl.getUniformLocation( program, "color" )};
\]
\[
\text{gl.uniform4fv( colorLoc, color);}
\]

• In fragment shader  // Same in vertex shader

\[
\text{uniform vec4 color;}
\]
\[
\text{Out vec4 fColor;}
\]
\[
\text{fColor} = \text{color;}
\]

• Also note *gl.uniform4fv*
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
  - No qualifier $\to$ in
Operators and Functions

• Standard C functions
  - Trigonometric
  - Arithmetic
  - normalize, reflect, length

• Overloading of vector and matrix types
  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

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

• Expanding primitive set
• Adding color
• Vertex attributes
WebGL Primitives

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

- WebGL 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
Triangularization

- 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
OpenGL 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 WebGL 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
Indexed Color

• Colors are indices into tables of RGB values
• Requires less memory
  - indices usually 8 bits
  - not as important now
    • Memory inexpensive
    • Need more colors for shading
Smooth Color

• Default is *smooth* shading
  - Rasterizer interpolates vertex colors across visible polygons
• Alternative is *flat shading*
  - Color of last vertex determines fill color
  - Handled 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