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:
  - Affine Combinations:
    \[ a_1v_1 + a_2v_2 + \ldots + a_nv_n \]
    where \( v_1, v_2, \ldots, v_n \) are vectors and \( \sum a_i = 1 \)
    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: \( c \cdot v \)
• 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
  \[ a_1v_1 + a_2v_2 + \ldots + a_nv_n \]
  where \( a_i \) is a real number (i.e. a scalar)
• Dot Product: \( u \cdot v = \sum_{k=1}^{n} u_kv_k \)
  a real value \( u_1v_1 + u_2v_2 + \ldots + u_nv_n \) written as \( u \cdot v \)

Fun with Dot Products

• Euclidian Distance from \((x,y)\) to \((0,0)\)
  \[ \sqrt{x^2 + y^2} \]
  in general: \[ \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2} \]
  which is just: \[ \sqrt{v \cdot v} \]
• This is also the length of vector \( v \):
  \[ |v| \] or \[ |\mathbf{v}| \]
• Normalization of a vector:
  \[ \hat{v} = \frac{v}{|v|} \]
• Orthogonal vectors: \( u \cdot v = 0 \)
Projections & Angles

• Angle between vectors. \( \theta \)
  \[ \bar{u} \cdot \bar{v} = |\bar{u}| |\bar{v}| \cos(\theta) \]
  \[ \theta = \text{ang}(\bar{u}, \bar{v}) = \cos^{-1} \left( \frac{\bar{u} \cdot \bar{v}}{|\bar{u}| |\bar{v}|} \right) = \cos^{-1} (\bar{u} \cdot \bar{v}). \]

• Projection of vectors
  \[ \bar{u}_1 = \frac{(\bar{u} \cdot \bar{v}) \bar{v}}{\bar{v} \cdot \bar{v}} \]
  \[ \bar{u}_2 = \bar{u} - \bar{u}_1. \]

Matrices and Matrix Operators

• A \( n \)-dimensional vector:
  \[ \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \]

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

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

Matrix Multiplication

• \([C] = [A][B]\)
• Sum over rows & columns
• Recall: matrix multiplication is **not** commutative
• Identity Matrix:
  - 1s on diagonal
  - 0s everywhere else
  \[
  C = \sum_{j=1}^{n} a_{ij} b_{j} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix}
  \]

Matrix Determinants

• A single real number
• Computed recursively
  \[ \det(A) = \sum_{j=1}^{n} A_{ij} (-1)^{i+j} M_{ij} \]
• Example:
  \[ \det \begin{bmatrix} a & c \\ b & d \end{bmatrix} = ad - bc \]
• 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 \( B \)
• \( A \times B = (a_y b_z - a_z b_y, a_z b_x - a_x b_z, a_x b_y - a_y b_x) \)

  \[
  A \times B = \begin{bmatrix} \hat{x} \\ \hat{y} \\ \hat{z} \end{bmatrix} = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} \times \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix}
  \]

Matrix Transpose & Inverse

• Matrix Transpose:
  Swap rows and cols:
  \[ A = \begin{bmatrix} 2 \\ 8 \end{bmatrix} \]
  \[ A^T = \begin{bmatrix} 2 & 8 \end{bmatrix} \]
• Facts about the transpose:
  \( (A^T)^T = A \)
  \( (A^T + B^T)^T = A^T + B^T \)
  \( (cA)^T = c(A^T) \)
  \( (AB)^T = B^T A^T \)
• Matrix Inverse: Given \( A \), find \( B \) such that
  \[ AB = BA = I \]
  \( B \neq A^T \)
  (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**

\[ f(x,y) = \alpha x^n y^m \]
\[ \frac{df(x,y)}{dx} = \alpha nx^{n-1} y^m \]
\[ f(x,y) = 5x^3 y^4 \]
\[ \frac{df(x,y)}{dx} = 15x^2 y^4 \]

**Objectives**

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

**Programing with OpenGL**

**Part 3: Shaders**

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

**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

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 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

Simple Vertex Shader

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

Use "in vec4 vPosition" for GLSL 1.5
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()
{
    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. 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 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`
- recent versions of GLSL use in and out qualifiers to get to and from shaders
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 bounding box of a primitive

Varying Qualifier

- Variables that are passed from vertex shader to fragment shader
- Automatically interpolated by the rasterizer
- With WebGL, GLSL uses the varying qualifier in both shaders
  varying vec4 color;
- More recent versions of WebGL use `out` in vertex shader and `in` in the fragment shader
  out vec4 color; // vertex shader
  in vec4 color; // fragment shader

Our Naming Convention

- Attributes passed to vertex shader have names beginning with `v` (vPosition, vColor) in both the application and the shader
  - Note that these are different entities with the same name
- Varying variables begin with `f` (fColor) in both shaders
  - must have same name
- Uniform variables are can have any/the same name in application and shaders

Example: Vertex Shader

```glsl
attribute vec4 vPosition, vColor;
varying vec4 fColor;
void main()
{
  gl_Position = vPosition;
  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

```glsl
precision mediump float;

varying vec4 fColor;
void main()
{
  gl_FragColor = fColor;
}
```
Pass Through Fragment Shader

```glsl
#ifdef GL_FRAGMENT_SHADER_PRECISION_HIGH
  precision highp float;
#else
  precision mediump float;
#endif

varying vec4 fcolor;

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

Another (New) Example: Vertex Shader

```glsl
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;
}
```

Required Fragment Shader (New)

```glsl
precision highp float;

in vec4 color_out;
out vec4 fragcolor;
void main(void)
{
  fragcolor = color_out;
}
```

Sending Colors from Application

```glsl
var cBuffer = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );
gl.bufferData( gl.ARRAY_BUFFER, flatten(colors), gl.STATIC_DRAW );
var vColor = gl.getAttribLocation( program, "vColor" );
gl.vertexAttribPointer( vColor, 3, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vColor );
```

Sending a Uniform Variable

```glsl```

User-defined functions

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

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 → 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.yyxw;

Objectives

• Expanding primitive set
• Adding color
• Vertex attributes

OpenGL 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

Non-simple polygon
Non-convex polygon

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, ...

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 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 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