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**

Mathematical Preliminaries

- Vector: an n-tuple of real numbers
- Vector Operations
  - Vector addition: $u + v = w$
    - Commutative, associative, identity element (0)
  - Scalar multiplication: $c v$
- 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
  $c_1 v_1 + c_2 v_2 + \ldots + c_n v_n$
  where $c_i$ 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 $\mathbf{u} \cdot \mathbf{v}$

Fun with Dot Products

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

- Angle between vectors, $\theta$
  \[ \vec{a} \cdot \vec{b} = |\vec{a}| |\vec{b}| \cos(\theta) \]
  
- Projection of vectors
  \[ \vec{a}_p = \frac{(\vec{a} \cdot \vec{b})}{(|\vec{b}|^2)} \vec{b} \]

Matrices and Matrix Operators

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

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

- Matrix Multiplication
  - 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:
  
Matrix Determinants

- A single real number
- Computed recursively
  \[ \det(A) = \sum_{j=1}^{n} A_{i,j} (-1)^{i+j} M_{i,j} \]

- 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
- $A \times B = (a_y b_z - a_z b_y, a_z b_x - a_x b_z, a_x b_y - a_y b_x)$

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:
  \[ (A+B)^T = A^T + B^T \]
  \[ (cA)^T = cA^T \]
  \[ (AB)^T = B^T A^T \]

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

- **Vertex Shader**
  - in vec4 vPosition;
  - void main(void)
    - gl_Position = vPosition;
  - Use "attribute vec4 vPosition" for GLSL 1.4

- **Fragment Shader**
  - void main()
    - gl_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)

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
  varying vec4 color;
- Now use out in vertex shader and in in the fragment shader
  out vec4 color;

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 (old)

```glsl
in vec4 color_out;
void main(void)
{
    // Now deprecated
    gl_FragColor = color_out;
}
```

Required Fragment Shader (new)

```glsl
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

```glsl```
```
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
  ```
  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;
  ```

```
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
Triangularization

- Convex polygon
- Start with abc, remove b, then acd, ...
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