Objectives

- Introduce the elements of geometry
  - Scalars
  - Vectors
  - Points
- Develop mathematical operations among them in a coordinate-free manner
- Define basic primitives
  - Line segments
  - Polygons

Basic Elements

- Geometry is the study of the relationships among objects in an n-dimensional space
  - In computer graphics, we are interested in objects that exist in three dimensions
- Want a minimum set of primitives from which we can build more sophisticated objects
- We will need three basic elements
  - Scalars
  - Vectors
  - Points

Coordinate-Free Geometry

- When we learned simple geometry, most of us started with a Cartesian approach
  - Points were at locations in space \( p=(x,y,z) \)
  - We derived results by algebraic manipulations involving these coordinates
- This approach was nonphysical
  - Physically, points exist regardless of the location of an arbitrary coordinate system
  - Most geometric results are independent of the coordinate system
  - Example Euclidean geometry: two triangles are identical if two corresponding sides and the angle between them are identical

Scalars

- Need three basic elements in geometry
  - Scalars, Vectors, Points
- Scalars can be defined as members of sets which can be combined by two operations (addition and multiplication) obeying some fundamental axioms (associativity, commutivity, inverses)
- Examples include the real and complex number systems under the ordinary rules with which we are familiar
- Scalars alone have no geometric properties

Vectors

- Physical definition: a vector is a quantity with two attributes
  - Direction
  - Magnitude
- Examples include
  - Force
  - Velocity
  - Directed line segments
    - Most important example for graphics
    - Can map to other types
Vector Operations

• Every vector has an inverse
  - Same magnitude but points in opposite direction
• Every vector can be multiplied by a scalar
• There is a zero vector
  - Zero magnitude, undefined orientation
• The sum of any two vectors is a vector
  - Use head-to-tail axiom

Linear Vector Spaces

• Mathematical system for manipulating vectors
• Operations
  - Scalar-vector multiplication \( u = \alpha v \)
  - Vector-vector addition: \( w = u + v \)
• Expressions such as
  \[ v = \alpha u + 2v - 3r \]
Make sense in a vector space

Vectors Lack Position

• These vectors are identical
  - Same direction and magnitude (length)
• Vectors spaces insufficient for geometry
  - Need points

Points

• Location in space
• Operations allowed between points and vectors
  - Point-point subtraction yields a vector
  - Equivalent to point-vector addition

Affine Spaces

• Point + a vector space
• Operations
  - Vector-vector addition
  - Scalar-vector multiplication
  - Point-vector addition
  - Scalar-scalar operations
• For any point define
  - \( 1 \cdot P = P \)
  - \( 0 \cdot P = \mathbf{0} \) (zero vector)

Lines

• Consider all points of the form
  - \( P(\alpha) = P_0 + \alpha \mathbf{d} \)
  - Set of all points that pass through \( P_0 \) in the direction of the vector \( \mathbf{d} \)
**Parametric Form**

- This form is known as the parametric form of the line
  - More robust and general than other forms
  - Extends to curves and surfaces

- Two-dimensional forms
  - Explicit: $y = mx + b$
  - Implicit: $ax + by + c = 0$
  - Parametric:
    
    \[
    x(\alpha) = ax_0 + (1-\alpha)x_1 \\
    y(\alpha) = ay_0 + (1-\alpha)y_1
    \]

**Rays and Line Segments**

- If $\alpha \geq 0$, then $P(\alpha)$ is the ray leaving $P_0$ in the direction $d$
  - If we use two points to define $v$, then
    
    \[
    P(\alpha) = Q + \alpha (R-Q) = Q + \alpha v \\
    = \alpha R + (1-\alpha)Q
    \]
  - For $0 \leq \alpha \leq 1$ we get all the points on the line segment joining $R$ and $Q$

**Convexity**

- An object is convex iff for any two points in the object all points on the line segment between these points are also in the object

**Affine Sums**

- Consider the “sum” $P = \alpha_1 P_1 + \alpha_2 P_2 + \ldots + \alpha_n P_n$
  - If $\alpha_1 + \alpha_2 + \ldots + \alpha_n = 1$
    - in which case we have the affine sum of the points $P_1, P_2, \ldots, P_n$
  - If, in addition, $\alpha_i \geq 0$, we have the convex hull of $P_1, P_2, \ldots, P_n$

**Convex Hull**

- Smallest convex object containing $P_1, P_2, \ldots, P_n$
- Formed by “shrink wrapping” points

**Curves and Surfaces**

- Curves are one parameter entities of the form $P(\alpha)$ where the function is nonlinear
- Surfaces are formed from two-parameter functions $P(\alpha, \beta)$
  - Linear functions give planes and polygons

**E. Angel and D. Shreiner: Interactive Computer Graphics 6E © Addison-Wesley 2012.**
Planes

A plane can be defined by a point and two vectors or by three points.

\[ P(a, b) = R + au + bv \]

\[ P(a, b) = R + \alpha(Q - R) + \beta(P - R) \]

Triangles

Convex sum of \( P \) and \( Q \)

Convex sum of \( S(\alpha) \) and \( R \)

for \( 0 \leq \alpha, \beta \leq 1 \), we get all points in triangle

Barycentric Coordinates

Triangle is convex so any point inside can be represented as an affine sum.

\[ P(\alpha_1, \alpha_2, \alpha_3) = \alpha_1P + \alpha_2Q + \alpha_3R \]

where

\[ \alpha_1 + \alpha_2 + \alpha_3 = 1 \]

\[ \alpha_i \geq 0 \]

The representation is called the barycentric coordinate representation of \( P \).

Normals

Every plane has a vector \( n \) normal (perpendicular, orthogonal) to it.

From point-two vector form \( P(a, b) = R + au + bv \), we know we can use the cross product to find \( n = u \times v \) and the equivalent form

\[ (P(\alpha)-P) \cdot n = 0 \]

Objectives

- Introduce concepts such as dimension and basis
- Introduce coordinate systems for representing vectors spaces and frames for representing affine spaces
- Discuss change of frames and bases
- Introduce homogeneous coordinates
Linear Independence

- A set of vectors \( v_1, v_2, \ldots, v_n \) is **linearly independent** if
  \[ a_1 v_1 + a_2 v_2 + \ldots + a_n v_n = 0 \iff a_1 = a_2 = \ldots = 0 \]
- If a set of vectors is linearly independent, we cannot represent one in terms of the others.
- If a set of vectors is linearly dependent, at least one can be written in terms of the others.

Dimension

- In a vector space, the maximum number of linearly independent vectors is fixed and is called the **dimension** of the space.
- In an \( n \)-dimensional space, any set of \( n \) linearly independent vectors form a **basis** for the space.
- Given a basis \( v_1, v_2, \ldots, v_n \), any vector \( v \) can be written as
  \[ v = a_1 v_1 + a_2 v_2 + \ldots + a_n v_n \]
  where the \( \{a_i\} \) are unique.

Representation

- Until now we have been able to work with geometric entities without using any frame of reference, such as a coordinate system.
- Need a frame of reference to relate points and objects to our physical world.
  - For example, where is a point? Can’t answer without a reference system
  - World coordinates
  - Camera coordinates

Coordinate Systems

- Consider a basis \( v_1, v_2, \ldots, v_n \).
- A vector is written \( v = a_1 v_1 + a_2 v_2 + \ldots + a_n v_n \).
- The list of scalars \( \{a_1, a_2, \ldots, a_n\} \) is the **representation** of \( v \) with respect to the given basis.
- We can write the representation as a row or column array of scalars
  \[ a = [a_1, a_2, \ldots, a_n]^T = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \]

Example

- \( v = 2v_1 + 3v_2 - 4v_3 \)
- \( a = [2, 3, -4]^T \)
- Note that this representation is with respect to a particular basis.
- For example, in WebGL we start by representing vectors using the object basis but later the system needs a representation in terms of the camera or eye basis.

Coordinate Systems

- Which is correct?
  \[ \begin{align*}
  &v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \\
  &\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}
  \end{align*} \]
  - Both are because vectors have no fixed location.
Frames

• A coordinate system is insufficient to represent points
• If we work in an affine space we can add a single point, the origin, to the basis vectors to form a frame

\[
P_0 \quad v_1 \quad v_2 \quad v_3
\]

Representation in a Frame

• Frame determined by \((P_0, v_1, v_2, v_3)\)
• Within this frame, every vector can be written as

\[
v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n
\]

• Every point can be written as

\[
P = P_0 + b_1 v_1 + b_2 v_2 + \ldots + b_n v_n
\]

Confusing Points and Vectors

Consider the point and the vector

\[
P = P_0 + b_1 v_1 + b_2 v_2 + \ldots + b_n v_n
\]

\[
v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n
\]

They appear to have the similar representations

\[
p = [b_1 \ b_2 \ b_3] \quad v = [\alpha_1 \ \alpha_2 \ \alpha_3]
\]

which confuses the point with the vector.

A vector has no position

Point: fixed

Vector can be placed anywhere

Homogeneous Coordinates

The homogeneous coordinates form for a three dimensional point \([x \ y \ z]\) is given as

\[
p = [x' \ y' \ z' \ w]\]

We return to a three dimensional point (for \(w \neq 0\)) by

\[
x = x' / w
\]

\[
y = y' / w
\]

\[
z = z' / w
\]

If \(w = 0\), the representation is that of a vector

Note that homogeneous coordinates replaces points in three dimensions by lines through the origin in four dimensions

For \(w = 1\), the representation of a point is \([x \ y \ z \ 1]\)
**Homogeneous Coordinates and Computer Graphics**

- Homogeneous coordinates are key to all computer graphics systems
  - All standard transformations (rotation, translation, scaling) can be implemented with matrix multiplications using 4 x 4 matrices
  - Hardware pipeline works with 4 dimensional representations
  - For orthographic viewing, we can maintain \( w = 0 \) for vectors and \( w = 1 \) for points
  - For perspective we need a perspective division

**Change of Coordinate Systems**

- Consider two representations of the same vector with respect to two different bases. The representations are

\[
\mathbf{a} = [a_1 \ a_2 \ a_3] \\
\mathbf{b} = [b_1 \ b_2 \ b_3]
\]

where

\[
\mathbf{v} = a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + a_3\mathbf{v}_3 = [a_1 \ a_2 \ a_3] \ [v_1 \ v_2 \ v_3]^T
\]

\[
= b_1\mathbf{v}_1 + b_2\mathbf{v}_2 + b_3\mathbf{v}_3 = [b_1 \ b_2 \ b_3] \ [u_1 \ u_2 \ u_3]^T
\]

**Representing second basis in terms of first**

Each of the basis vectors, \( \mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3 \), are vectors that can be represented in terms of the first basis

\[
\mathbf{u}_1 = \gamma_1 \mathbf{v}_1 + \gamma_2 \mathbf{v}_2 + \gamma_3 \mathbf{v}_3 \\
\mathbf{u}_2 = \gamma_2 \mathbf{v}_1 + \gamma_3 \mathbf{v}_2 + \gamma_1 \mathbf{v}_3 \\
\mathbf{u}_3 = \gamma_3 \mathbf{v}_1 + \gamma_1 \mathbf{v}_2 + \gamma_2 \mathbf{v}_3
\]

**Change of Frames**

- We can apply a similar process in homogeneous coordinates to the representations of both points and vectors

Consider two frames:

\( (P_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3) \) \\
(\( Q_0, \mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3 \))

- Any point or vector can be represented in either frame
- We can represent \( Q_0, \mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3 \) in terms of \( P_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3 \)

**Matrix Form**

The coefficients define a 3 x 3 matrix

\[
M = \begin{bmatrix}
\gamma_{11} & \gamma_{12} & \gamma_{13} \\
\gamma_{21} & \gamma_{22} & \gamma_{23} \\
\gamma_{31} & \gamma_{32} & \gamma_{33}
\end{bmatrix}
\]

and the bases can be related by

\[
\mathbf{a} = M^T \mathbf{b}
\]

see text for numerical examples

**Representing One Frame in Terms of the Other**

Extending what we did with change of bases

\[
\mathbf{u}_1 = \gamma_1 \mathbf{v}_1 + \gamma_2 \mathbf{v}_2 + \gamma_3 \mathbf{v}_3 \\
\mathbf{u}_2 = \gamma_2 \mathbf{v}_1 + \gamma_3 \mathbf{v}_2 + \gamma_1 \mathbf{v}_3 \\
\mathbf{u}_3 = \gamma_3 \mathbf{v}_1 + \gamma_1 \mathbf{v}_2 + \gamma_2 \mathbf{v}_3
\]

defining a 4 x 4 matrix

\[
M = \begin{bmatrix}
\gamma_{11} & \gamma_{12} & \gamma_{13} & 0 \\
\gamma_{21} & \gamma_{22} & \gamma_{23} & 0 \\
\gamma_{31} & \gamma_{32} & \gamma_{33} & 0 \\
\gamma_{41} & \gamma_{42} & \gamma_{43} & 1
\end{bmatrix}
\]
Working with Representations

Within the two frames any point or vector has a representation of the same form
\[ \mathbf{a} = [a_1, a_2, a_3, a_4] \] in the first frame
\[ \mathbf{b} = [b_1, b_2, b_3, b_4] \] in the second frame

where \( a_4 = b_4 = 1 \) for points and \( a_4 = b_4 = 0 \) for vectors and
\[ \mathbf{a} = M^T \mathbf{b} \]
The matrix \( M \) is 4 x 4 and specifies an affine transformation in homogeneous coordinates.

Transformations to Change Coordinate Systems

- 4 coordinate systems
- 1 point \( P \)

\[ M_{x \rightarrow z} = T(4, 2) \]
\[ M_{z \rightarrow z} = T(2, 3) \times S(0.5, 0.5) \]
\[ M_{x \rightarrow z} = T(6.7, 1.8) \times R(45°) \]

Moving the Camera

If objects are on both sides of \( z=0 \), we must move the camera frame

\[ M = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -d \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

Objectives

- Develop a more sophisticated three-dimensional example
  - Sierpinski gasket: a fractal
- Introduce hidden-surface removal
Three-dimensional Applications

- In Open/WebGL, two-dimensional applications are a special case of three-dimensional graphics.
- Going to 3D:
  - Not much changes
  - Use vec3, gl.uniform3f
  - Have to worry about the order in which primitives are rendered or use hidden-surface removal

Sierpinski Gasket (2D)

- Start with a triangle
- Connect bisectors of sides and remove central triangle
- Repeat

Example

- Five subdivisions

The gasket as a fractal

- Consider the filled area (black) and the perimeter (the length of all the lines around the filled triangles)
- As we continue subdividing
  - the area goes to zero
  - but the perimeter goes to infinity
- This is not an ordinary geometric object
  - It is neither two- nor three-dimensional
- It is a fractal (fractional dimension) object

Gasket Program

- HTML file
  - Same as in other examples
  - Pass through vertex shader
  - Fragment shader sets color
  - Read in JS file

```javascript
var points = [];
var NumTimesToSubdivide = 5;

/* initial triangle */
var vertices = [
  vec2(-1, -1),
  vec2( 0,  1),
  vec2( 1, -1 )
];
divideTriangle( vertices[0], vertices[1], vertices[2], NumTimesToSubdivide );
```
Draw one triangle

```javascript
/* display one triangle */
function triangle( a, b, c ){
    points.push( a, b, c );
}
```

Triangle Subdivision

```javascript
function divideTriangle( a, b, c, count ){
    // check for end of recursion
    if ( count === 0 ) {
        triangle( a, b, c );
    } else {
        // bisect the sides
        var ab = mix( a, b, 0.5 );
        var ac = mix( a, c, 0.5 );
        var bc = mix( b, c, 0.5 );
        --count;
        // three new triangles
        divideTriangle( a, ab, ac, count - 1 );
        divideTriangle( c, ac, bc, count - 1 );
        divideTriangle( b, bc, ab, count - 1 );
    }
}
```

init()

```javascript
var program = initShaders(gl, "vertex-shader", "fragment-shader");
gl.useProgram( program );
var bufferId = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, bufferId );
gl.bufferData( gl.ARRAY_BUFFER, flatten(points), gl.STATIC_DRAW );
var aPosition = gl.getAttribLocation( program, "aPosition" );
gl.vertexAttribPointer( aPosition, 2, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( aPosition );
render();
```

Render Function

```javascript
function render(){
    gl.clear( gl.COLOR_BUFFER_BIT );
    gl.drawArrays( gl.TRIANGLES, 0, points.length );
}
```

Example

- Five subdivisions

Moving to 3D

- We can easily make the program three-dimensional by using `point3 v[3]`
  and we start with a tetrahedron
3D Gasket

- We can subdivide each of the four faces
- Appears as if we remove a solid tetrahedron from the center leaving four smaller tetrahedra
- Code almost identical to 2D example

Almost Correct

- Because the triangles are drawn in the order they are specified in the program, the front triangles are not always rendered in front of triangles behind them
- We want to see only those surfaces in front of other surfaces
- Open/WebGL uses a hidden-surface method called the z-buffer algorithm that saves depth information as objects are rendered so that only the front objects appear in the image

Hidden-Surface Removal

- We want to see only those surfaces in front of other surfaces
- Open/WebGL uses a hidden-surface method called the z-buffer algorithm that saves depth information as objects are rendered so that only the front objects appear in the image

Z-buffering

- Z-buffering (depth-buffering) is a visible surface detection algorithm
- Implementable in hardware and software
- Requires data structure (z-buffer) in addition to frame buffer.
- Z-buffer stores values [0 .. ZMAX] corresponding to depth of each point.
- If the point is closer than one in the buffers, it will replace the buffered values

Z-buffering w/ front/back clipping

```plaintext
for (y = 0; y < YMAX; y++)
for (x = 0; x < XMAX; x++) {
  F[x][y] = BACKGROUND_VALUE;
  Z[x][y] = -1; /* Back value in NPC */
}

for (each polygon)
  for (each pixel in polygon’s projection) {
    pz = polygon’s z-value at pixel coordinates (x,y)
    if (pz < FRONT && pz > Z[x][y]) { /* New point is behind front
                                        plane & closer than previous point */
      Z[x][y] = pz;
      F[x][y] = polygon’s color at pixel coordinates (x,y)
    }
  }
```
Using the z-buffer algorithm

- The algorithm uses an extra buffer, the z-buffer, to store depth information as geometry travels down the pipeline
- Depth buffer is required to be available in WebGL
- It must be enabled:
  - `gl.enable(gl.DEPTH_TEST)`
- Cleared in for each render:
  - `gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT)`

Surface vs Volume Subdivision

- In our example, we divided the surface of each face
- We could also divide the volume using the same midpoints
- The midpoints define four smaller tetrahedrons, one for each vertex
- Keeping only these tetrahedrons removes a volume in the middle
- See text for code

Volume Subdivision

- In our example, we divided the surface of each face
- We could also divide the volume using the same midpoints
- The midpoints define four smaller tetrahedrons, one for each vertex
- Keeping only these tetrahedrons removes a volume in the middle
- See text for code

Swapping Shader Programs

- May need different shader programs for different objects or situations in a graphics application
- Could pass a flag to one set of shader programs to execute different code for different objects or situations
- This strategy might produce performance degradation
- Alternative: Dynamically change shader programs

Swapping Shader Programs

- Create and fill buffers
  - `positionBuffer1 = gl.createBuffer();`
  - `gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer1);`
  - `gl.bufferData(gl.ARRAY_BUFFER, flatten(positions1), gl.STATIC_DRAW);`
  - `positionBuffer2 = gl.createBuffer();`
  - `gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer2);`
  - `gl.bufferData(gl.ARRAY_BUFFER, flatten(positions2), gl.STATIC_DRAW);`
Swapping Shader Programs

In the render() function
• Use first shader program
  
  ```cpp
  gl.useProgram(program1_ID);
  
  • Set uniform value
  ```
  
  ```cpp
  gl.uniform1f(thetaLoc1, rotateAngle1);
  
  • Bind a buffer and define its attributes
  ```
  
  ```cpp
  gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer1);
  gl.vertexAttribPointer(positionLoc1, 2, gl.FLOAT, false, 0, 0);
  gl.enableVertexAttribArray(positionLoc1);
  
  • Draw data in the buffer
  ```
  
  ```cpp
  gl.drawArrays(gl.TRIANGLE_FAN, 0, numOfVertices1);
  ```

In the render() function
• Use second shader program
  
  ```cpp
  gl.useProgram(program2_ID);
  
  • Set uniform value
  ```
  
  ```cpp
  gl.uniform1f(thetaLoc2, rotateAngle2);
  
  • Bind a buffer and define its attributes
  ```
  
  ```cpp
  gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer2);
  gl.vertexAttribPointer(positionLoc2, 2, gl.FLOAT, false, 0, 0);
  gl.enableVertexAttribArray(positionLoc2);
  
  • Draw data in the buffer
  ```
  
  ```cpp
  gl.drawArrays(gl.TRIANGLE_FAN, 0, numOfVertices2);
  ```