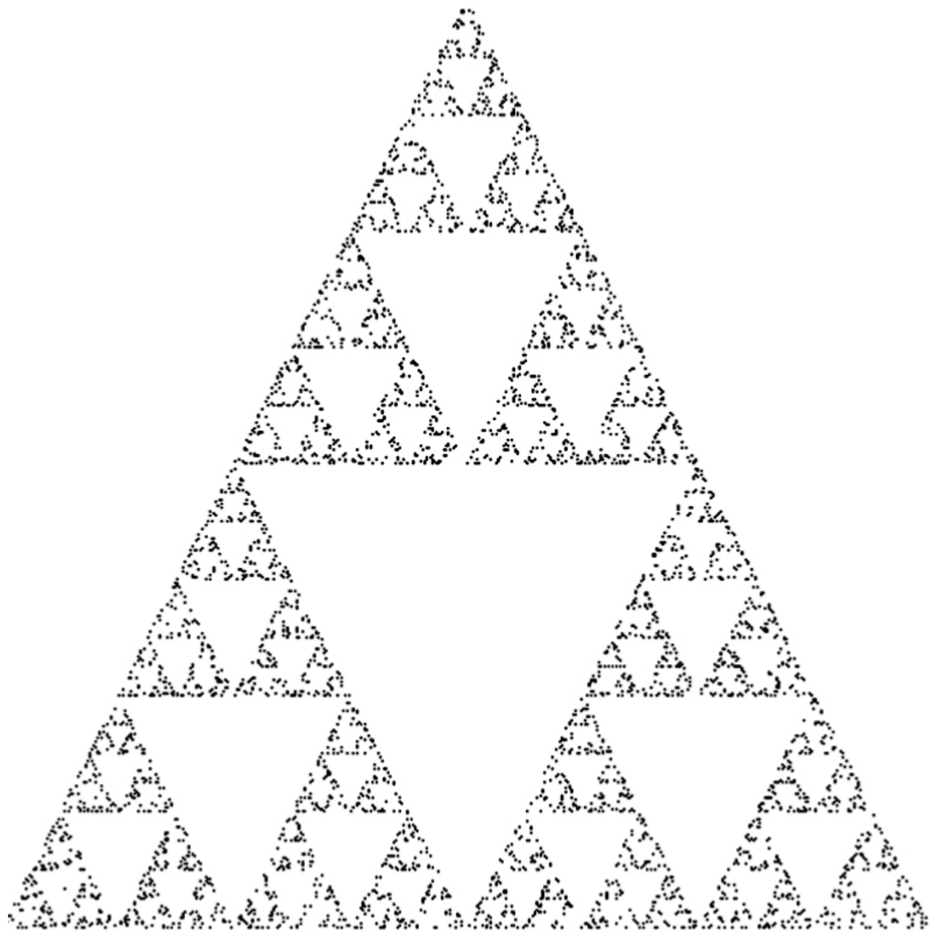
CSE 5542 - Real Time Rendering
Week 4

Slides(Mostly) Courtesy –
E. Angel and D. Shreiner

Recap from Recent Past

The Sierpinski Gasket



Sierpinski Vertex Shader

```
…
// Load shaders and use the resulting shader program
    GLuint program = InitShader( "vshader21.glsl", "fshader21.glsl" );
    glUseProgram( program );
..
```

```
attribute vec4 vPosition;
void
main()
{
    gl_Position = vPosition;
}
```

Sierpinski Fragment Shader

```
…
// Load shaders and use the resulting shader program
    GLuint program = InitShader( "vshader21.glsl", "fshader21.glsl" );
    glUseProgram( program );
..
```

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

Fragment vs Vertex Shader

per vertex lighting

per fragment lighting

OpenGL and GLSL

- Shader based OpenGL is based less on a state machine model than a data flow model
- Most state variables, attributes and related pre 3.1 OpenGL functions have been deprecated
- Action happens in shaders
- Job is application is to get data to GPU

GLSL

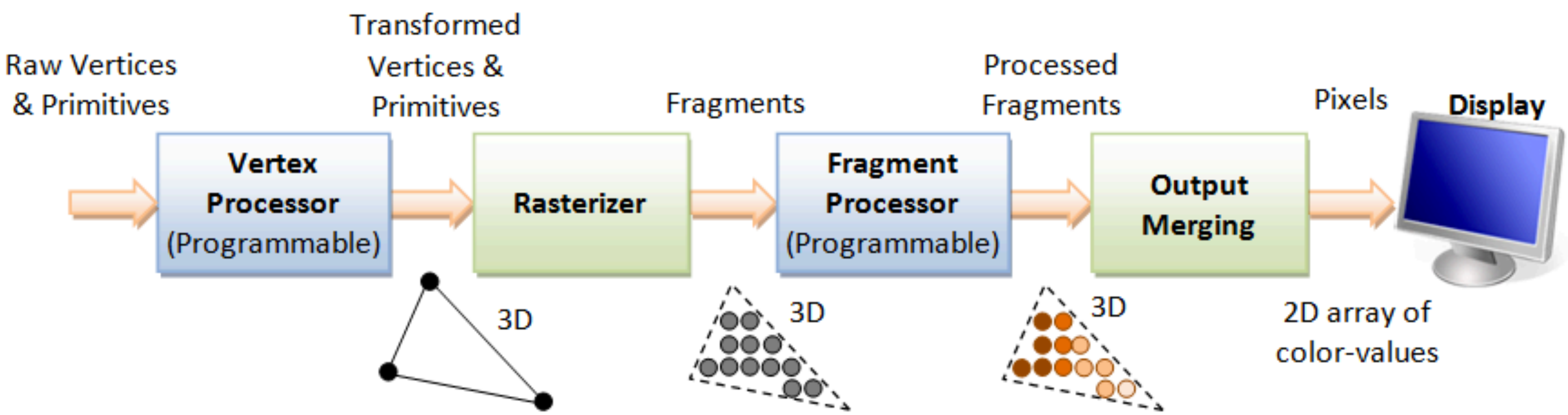
- C-like with
  - Matrix and vector types (2, 3, 4 dimensional)
  - Overloaded operators
  - C++ like constructors
- Similar to Nvidia's Cg and Microsoft HLSL
- Code sent to shaders as source code

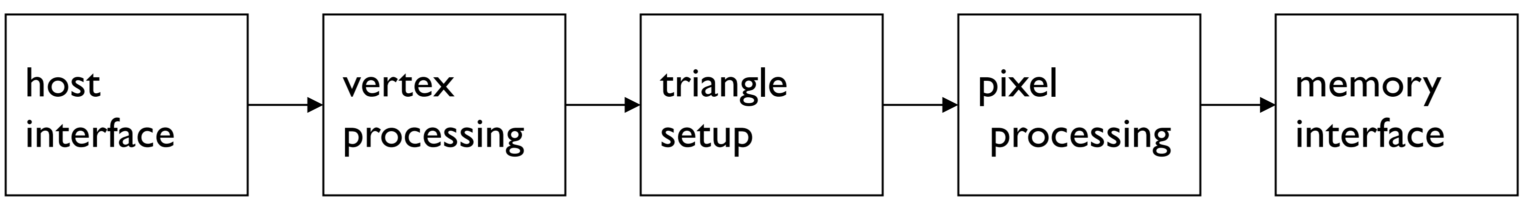
Still Maximal Portability

- Display device independent
- Window system independent
- Operating system independent

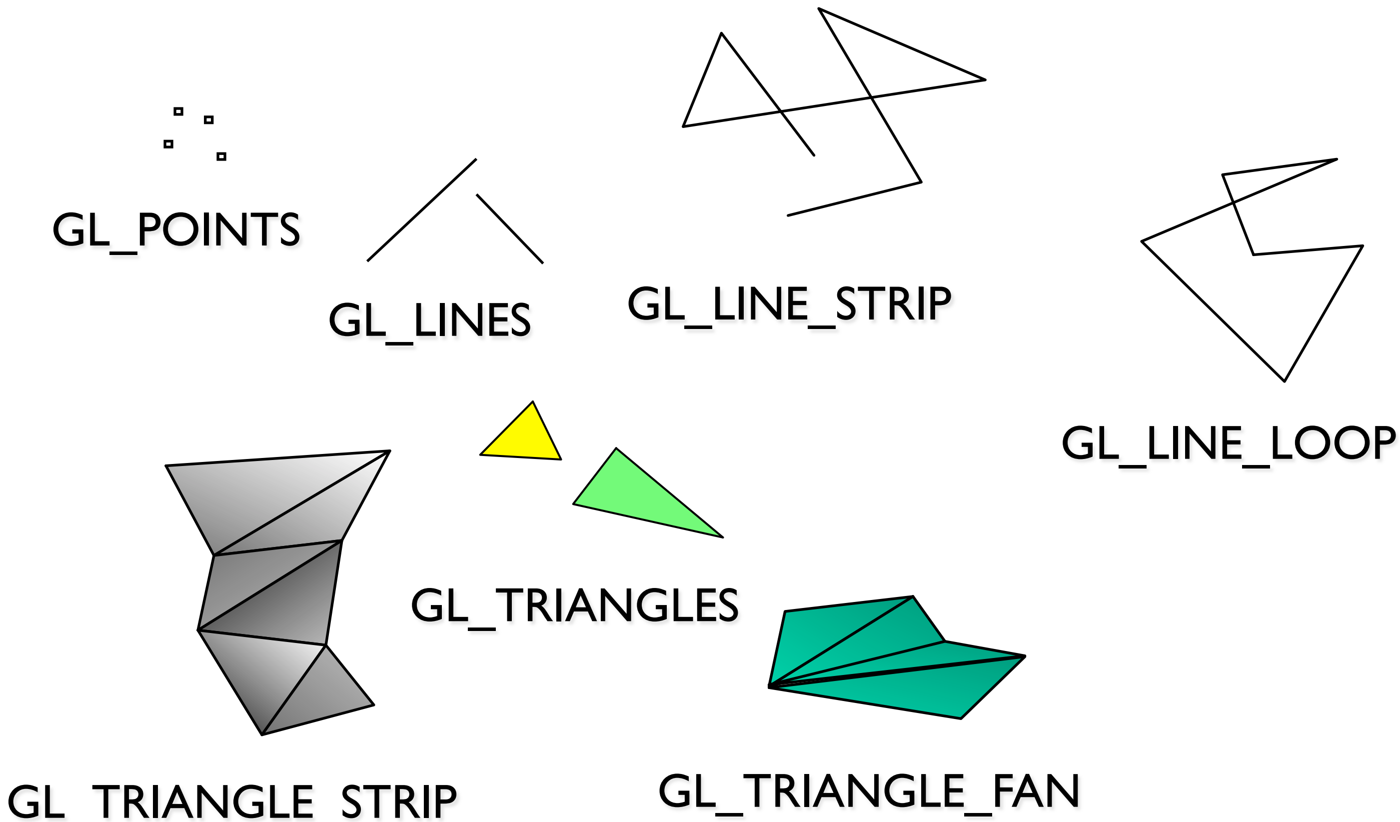
A Few More Things

Hardware Rendering Pipeline





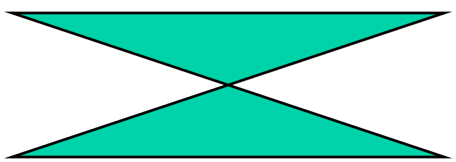
OpenGL Primitives



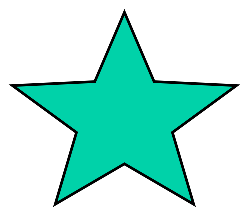
Triangles

- Triangles must be
  - 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
- User must create triangles (triangulation)
- OpenGL contains a tessellator

nonsimple polygon



nonconvex polygon



Space ?

```
point2 vertices[3] = {point2(0.0, 0.0),
    point2( 0.0, 1.0), point2(1.0, 1.0)};
```

Transform Spaces

Object Space → Screen Space

Coordinate Systems

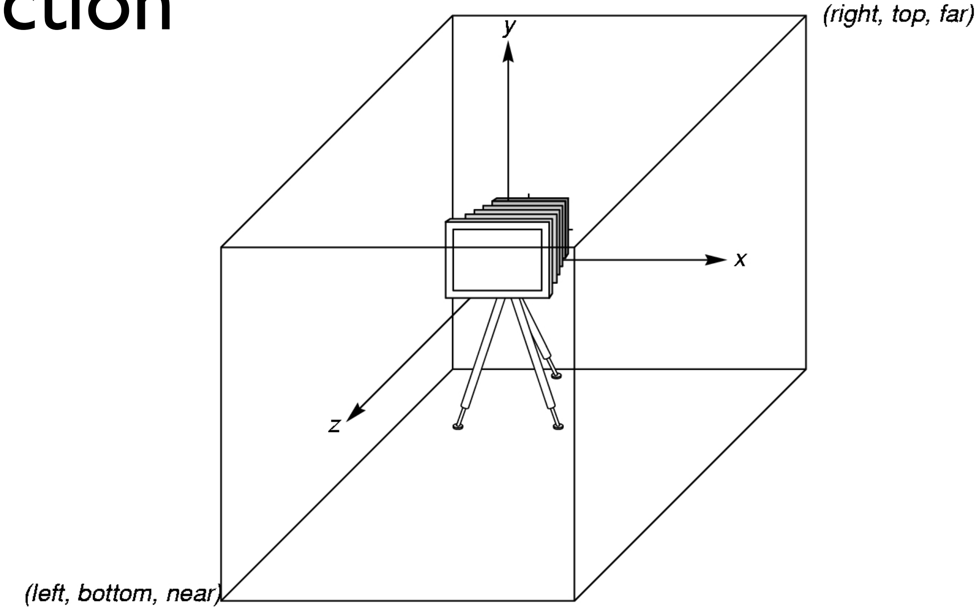
- The units in **points** can be *object, world, model* or *problem coordinates*
- Viewing specifications are also in object coordinates
- Same for lights
- Eventually pixels will be produced in *window coordinates*

Default Camera

- Camera at origin in object space pointing in -z direction

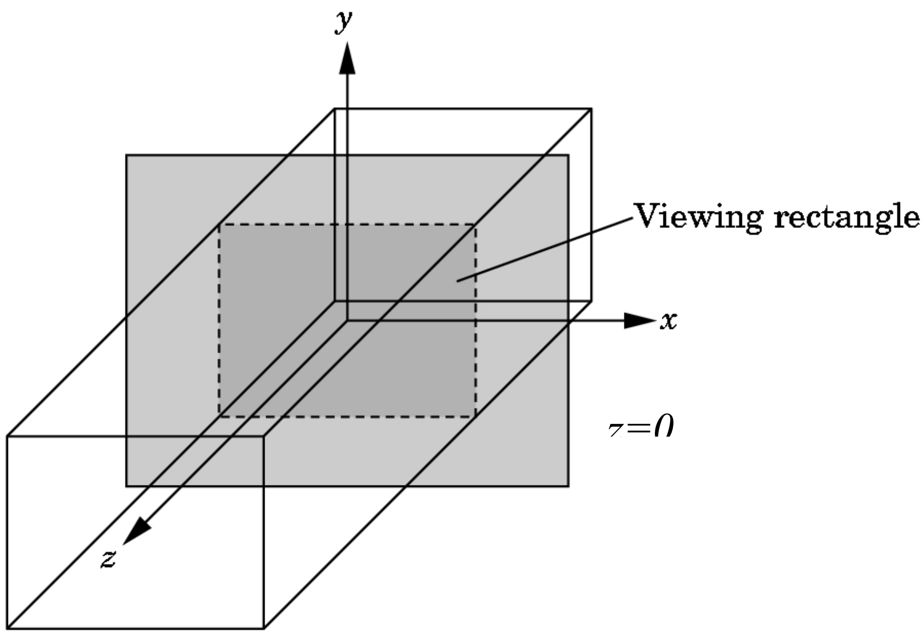
- Default viewing volume

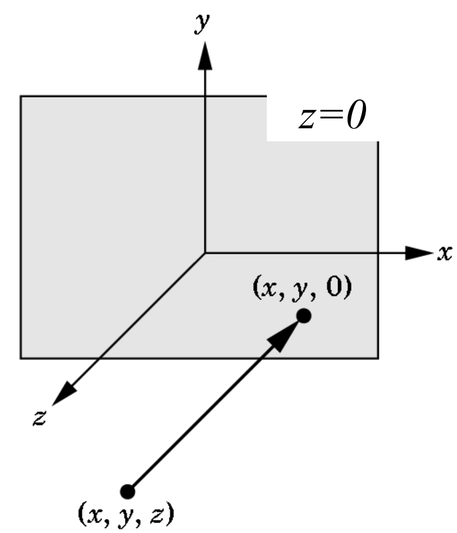
 - box centered at origin with sides of length 2



Orthographic Viewing

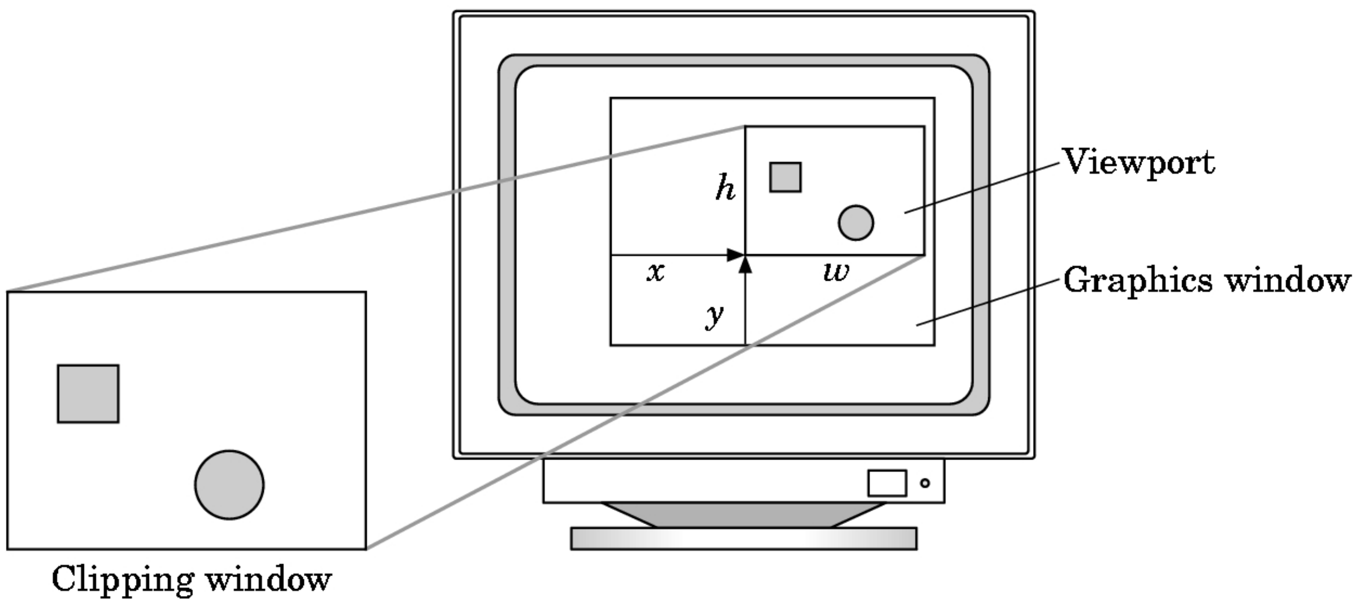
Points projected forward along z axis onto plane $z=0$





Viewports

- Use partial window for image: glViewport(x,y,w,h)
- w, h – pixel coordinates
- x,y – lower corner



Writing Shaders

Simple Vertex Shader

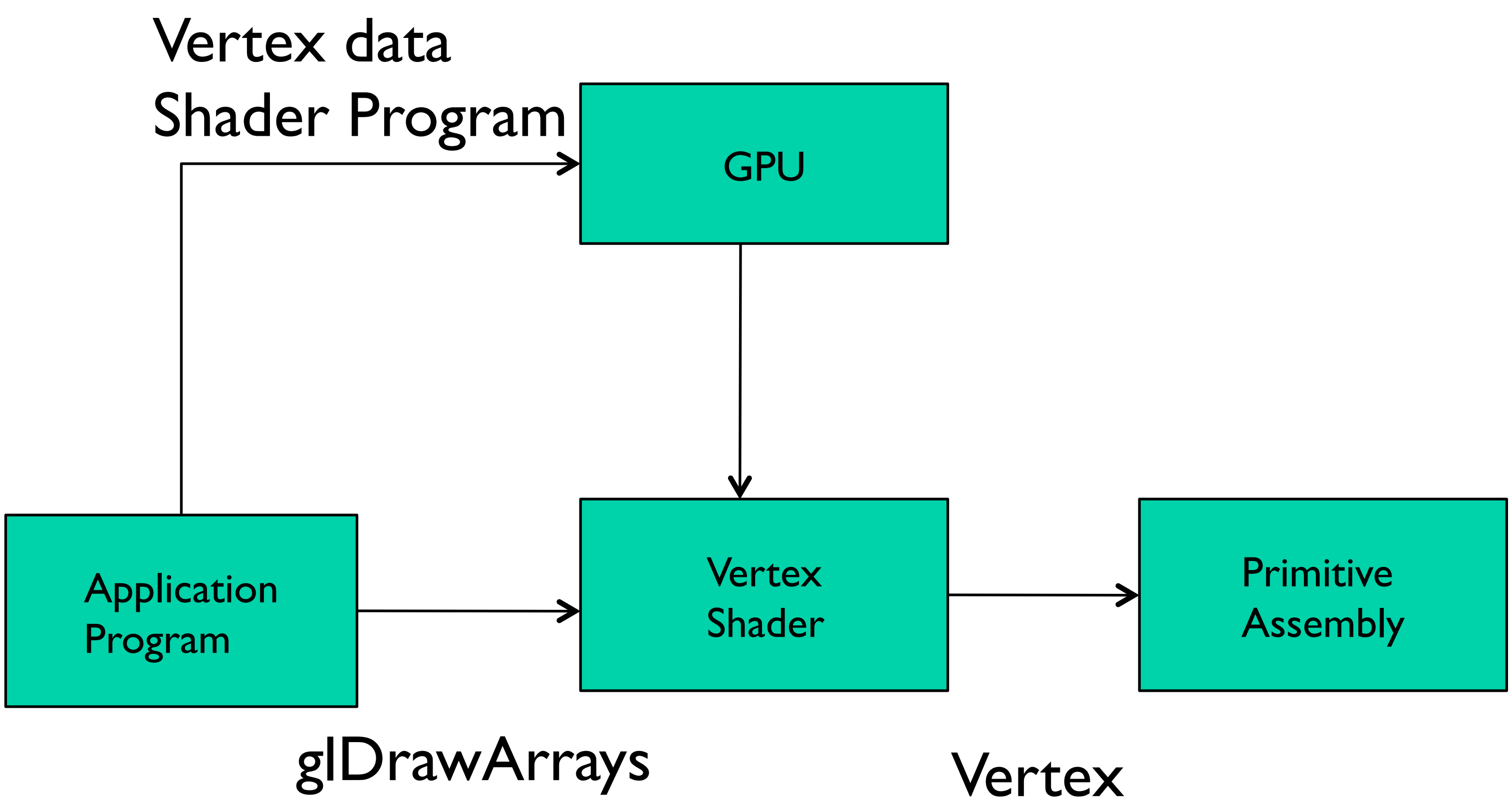
input from application

must link to variable in application

```
in vec4 vPosition;
void main(void)
{
    gl_Position = vPosition;
}
```

built in variable

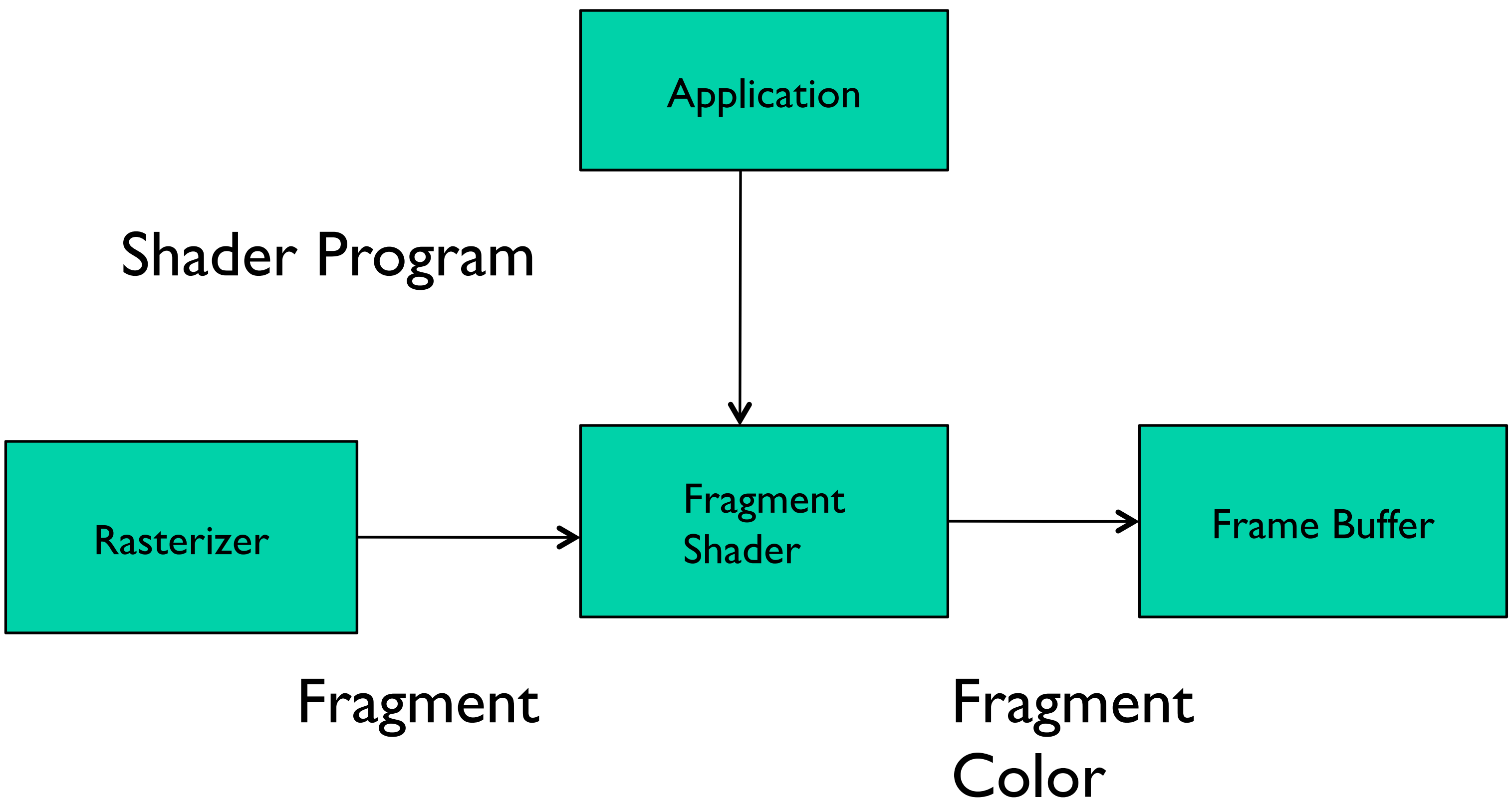
Execution Model



Simple Fragment Program

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

Execution Model



Data Types

- C types: int, float, bool
- Vectors:
  - float vec2, vec3, vec4
  - Also int (ivec) and boolean (bvec)
- 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
- C structs which can be copied back from functions
- Matrices and vectors can be passed to and fro GLSL functions, e.g. mat3 func(mat3 a)

THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Selection and Swizzling

- Access 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 to manipulate components

```
vec4 a;
a.yz = vec2(1.0, 2.0);
```

Example: Vertex Shader

```
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
out vec3 color_out;
void main(void)
{
  gl_Position = vPosition;
  color_out = red;
}
```

Fragment Shader

```
in vec3 color_out;
void main(void)
{
  gl_FragColor = color_out;
}
// in latest version use form
// out vec4 fragcolor;
// fragcolor = color_out;
```

Qualifiers

- GLSL has many qualifiers like **const** as C/C++
- 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

Passing values

- Call by value-return
- Variables are copied in
- Returned values are copied back
- Two possibilities
  - in
  - out

Attribute Qualifier

- Attribute-qualified variables can change at most once per vertex
- User defined (in application program)
  - Use in qualifier to get to shader
  - in float temperature
  - in vec3 velocity

Uniform Qualified

- 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

THE OHIO STATE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Example

```
GLint aParam;
aParam = glGetUniformLocation(myProgObj,
    "angle");
/* angle defined in shader */

/* my_angle set in application */
GLfloat my_angle;
my_angle = 5.0 /* or some other value */

glUniform1f(aParam, my_angle);
```

Varying Qualified

- Variables passed from vertex to fragment shader
- Automatically interpolated by the rasterizer
- Old style - varying vec4 color
- Use out in vertex shader and in in fragment shader
  out vec4 color;

Wave Motion Vertex Shader

```
in vec4 vPosition;
uniform float xs, zs, // frequencies
uniform float h; // height scale
void main()
{
  vec4 t = vPosition;
  t.y = vPosition.y
     + h*sin(time + xs*vPosition.x)
     + h*sin(time + zs*vPosition.z);
  gl_Position = t;
}
```

Particle System

```
in vec3 vPosition;
uniform mat4 ModelViewProjectionMatrix;
uniform vec3 init_vel;
uniform float g, m, t;
void main()
{ vec3 object_pos;
object_pos.x = vPosition.x + vel.x*t;
object_pos.y = vPosition.y + vel.y*t
      + g/(2.0*m)*t*t;
object_pos.z = vPosition.z + vel.z*t;
gl_Position =
  ModelViewProjectionMatrix*vec4(object_pos,1);
}
```

Fragment Shader

```
/* pass-through fragment shader */

in vec4 color;
void main(void)
{
    gl_FragColor = color;
}
```

Vertex Shader Applications

- Moving vertices
  - Morphing
  - Wave motion
  - Fractals
- Lighting
  - More realistic models
  - Cartoon shaders

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
```

Adding Color

- Send color to the shaders as a vertex attribute or as a uniform variable
- Choice depends on frequency of change
- Associate a color with each vertex
- Set up an array of same size as positions
- Send to GPU as a vertex buffer object

Setting Colors

```
typedef  vec3 color3;
color3 base_colors[4] = {color3(1.0, 0.0. 0.0), ….
color3 colors[NumVertices];
vec3 points[NumVertices];

//in loop setting positions

colors[i] = basecolors[color_index]
position[i] = …….
```

Setting Up Buffer Object

```
//need larger buffer

glBufferData(GL_ARRAY_BUFFER, sizeof(points) +
   sizeof(colors), NULL, GL_STATIC_DRAW);

//load data separately

glBufferSubData(GL_ARRAY_BUFFER, 0,
   sizeof(points), points);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(points),
   sizeof(colors), colors);
```

Second Vertex Array

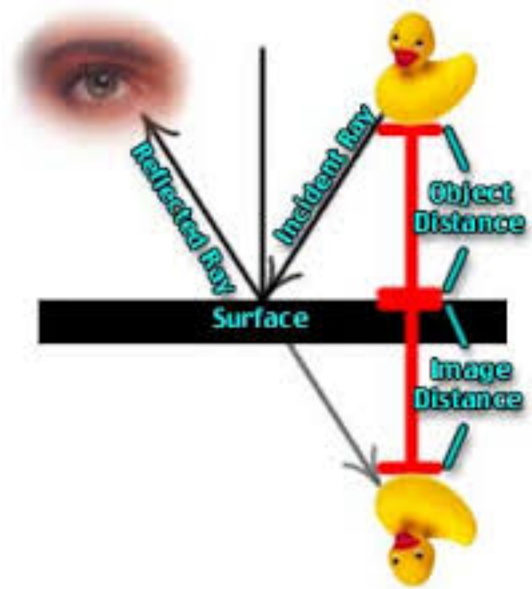
```
// vPosition and vColor identifiers in vertex shader


loc = glGetAttribLocation(program, “vPosition”);
glEnableVertexAttribArray(loc);
glVertexAttribPointer(loc, 3, GL_FLOAT, GL_FALSE, 0,
    BUFFER_OFFSET(0));

loc2 = glGetAttribLocation(program, “vColor”);
glEnableVertexAttribArray(loc2);
glVertexAttribPointer(loc2, 3, GL_FLOAT, GL_FALSE, 0,
    BUFFER_OFFSET(sizeofpoints));
```


THE OHIO STATE UNIVERSITY DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Next Topic – Linear Algebra

Vectors

- Physical definition:
  - Direction
  - Magnitude
- Examples
  - Light Direction
  - View Direction
  - Normal

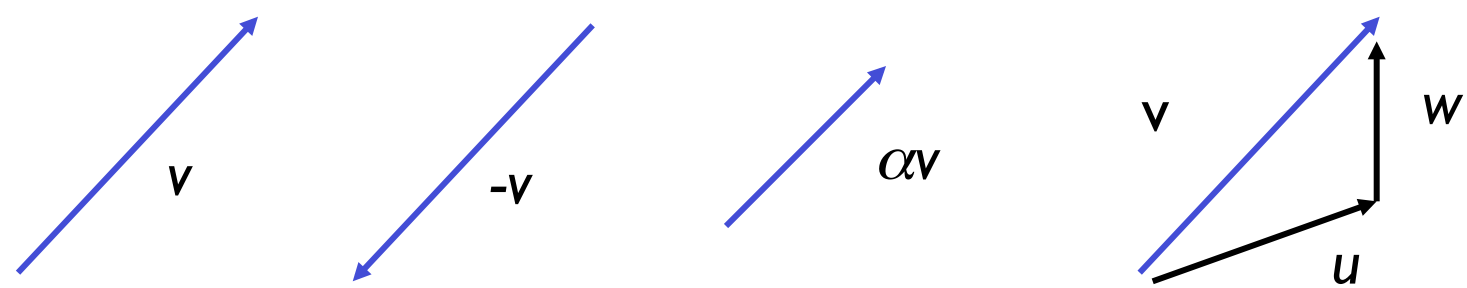


Abstract Spaces

- Scalars
- (Linear) Vector Space
  - Scalars and vectors
- Affine Space
  - Scalars, vectors, and points
- Euclidean Space
  - Scalars, vectors, points
  - Concept of distance
- Projections

Vectors – Linear Space

- Every vector
  - has an inverse
  - can be multiplied by a scalar
- There exists a zero vector
  - Zero magnitude, undefined orientation
- The sum of any two vectors is a vector - closure



Vector Spaces

- Vectors = *n*-tuples

  - Vector-vector addition

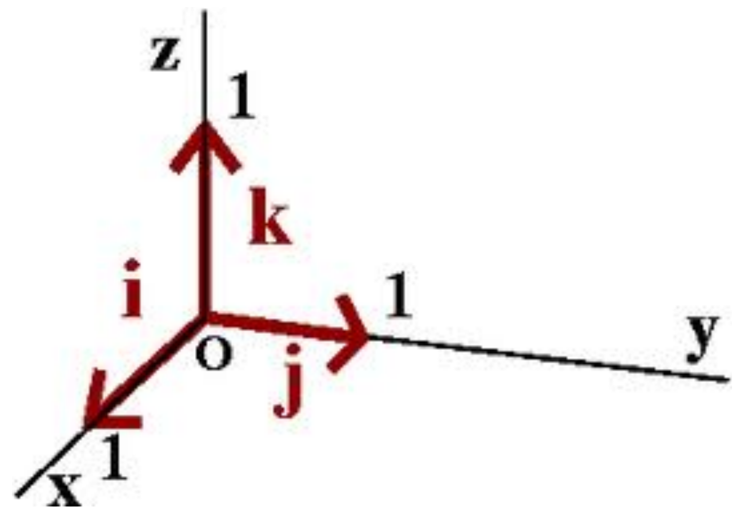
  - Scalar-vector multiplication

  - Vector space:

Linear Independence

$$\alpha_1\vec{u}_1 + \alpha_2\vec{u}_2 + \cdots + \alpha_n\vec{u}_n = 0 \text{ iff}$$

$$\alpha_1 = \alpha_2 = \cdots = \alpha_n = 0$$



$$p = (x, y, z) = x\vec{i} + y\vec{j} + z\vec{k}$$

Vector Spaces

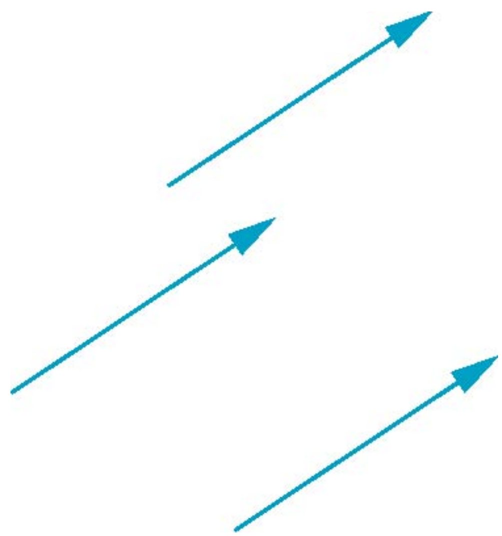
- *Dimension*
  - The greatest number of linearly independent vectors
- *Basis* $\{\beta_i\}$
  - *n* linearly independent vectors (*n*: dimension)
- Representation $\vec{v} = \beta_1 \vec{v}_1 + \beta_2 \vec{v}_2 + \cdots + \beta_n \vec{v}_n$
  - Unique expression in terms of the basis vectors
- Change of Basis: Matrix M
  - Other basis $\vec{v}_1', \vec{v}_2', \ldots, \vec{v}_n'$

$$\vec{v} = \beta_1' \vec{v}_1' + \beta_2' \vec{v}_2' + \cdots + \beta_n' \vec{v}_n'$$

$$\begin{bmatrix} \beta_1' \\ \beta_2' \\ \vdots \\ \beta_n' \end{bmatrix} = \mathbf{M} \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{bmatrix}$$

Vectors

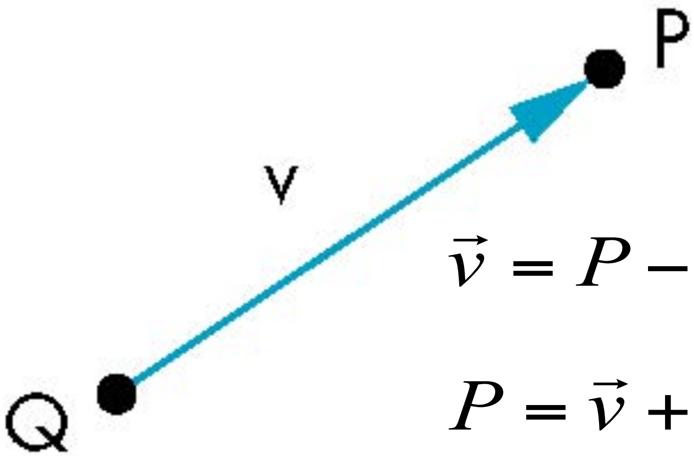
- These vectors are identical
  - Same length and magnitude



- Vectors spaces insufficient for geometry
  - Need points

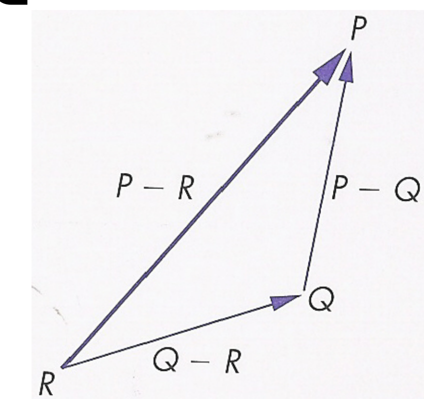
Points

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



$$\vec{v} = P - Q$$

$$P = \vec{v} + Q$$



$$(P-Q)+(Q-R)=(P-R)$$

Affine Spaces

Frame: a Point P_0 and a Set of Vectors $\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n$

Representations of the vector and point: *n* scalars

Vector $v = \alpha_1 v_1 + \alpha_2 v_2 + \cdots + \alpha_n v_n$

Point $P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \cdots + \beta_n v_n$

Affine Spaces

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

THE OHIO STATE UNIVERSITY DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Question

How Far Apart Are Two Points in Affine Spaces ?

Operation: ***Inner (dot) Product***

Euclidean (Metric) Spaces

– ***Magnitude*** (***length***) of a vector

$$|v| = \sqrt{v \cdot v}$$

– ***Distance*** between two points

$$|P - Q| = \sqrt{(P - Q) \cdot (P - Q)}$$

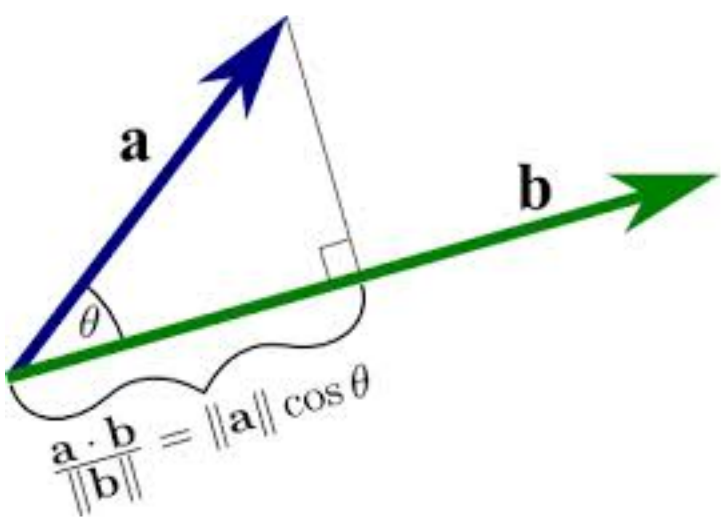
– Measure of the angle between two vectors

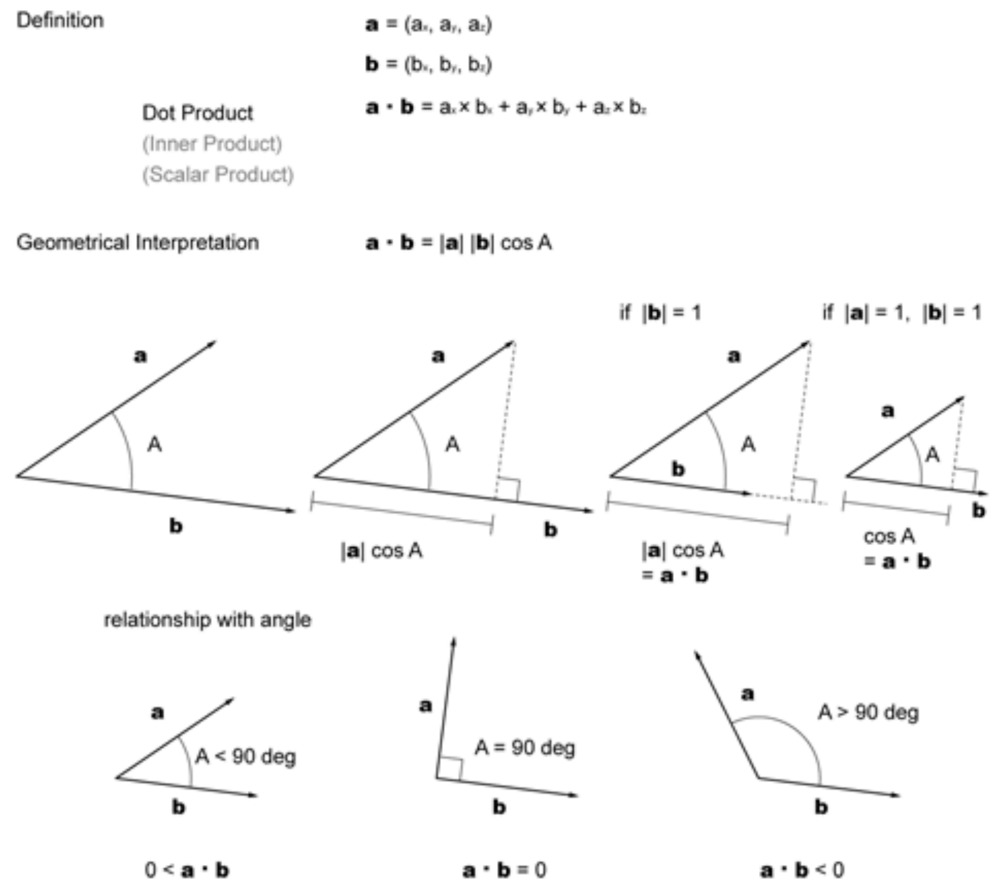
$$u \cdot v = |u||v| \cos\theta$$

- cos θ = 0 ➔ orthogonal
- cos θ = 1 ➔ parallel

In Pictures

$$\begin{pmatrix}a_x\\a_y\\a_z\end{pmatrix}\bullet\begin{pmatrix}b_x\\b_y\\b_z\end{pmatrix}=a_x.b_x+a_y.b_y+a_z.b_z$$





Euclidean Spaces

– Combine two vectors to form a real

– α , β , γ , …: scalars, *u*, *v*, *w*, …:vectors

$$u \cdot v = v \cdot u$$

$$(\alpha u + \beta v) \cdot w = \alpha u \cdot w + \beta v \cdot w$$

$$v \cdot v > 0 \text{ if } v \neq \mathbf{0}$$

$$\mathbf{0} \cdot \mathbf{0} = 0$$

Orthogonal: $u \cdot v = 0$

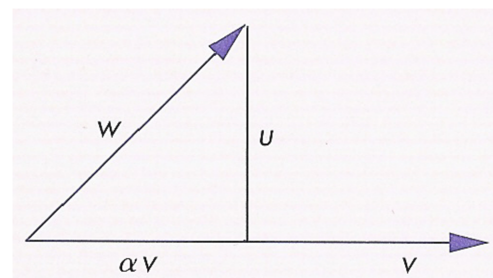
Projections

- Problem: Find shortest distance from a point to a line on a plane
- Given Two Vectors $\quad w = \alpha v + u$
  - Divide into two parts: one parallel and one orthogonal

$$w \cdot v = \alpha v \cdot v + u \cdot v = \alpha v \cdot v$$

$$\therefore \alpha = \frac{w \cdot v}{v \cdot v}$$

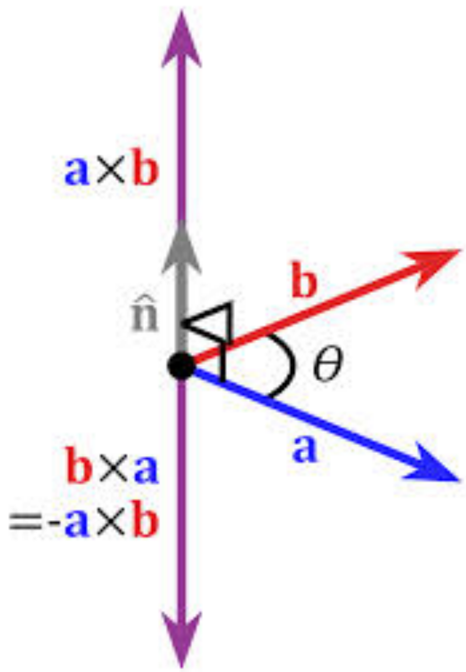
$$\therefore u = w - \alpha v = w - \frac{w \cdot v}{v \cdot v} v$$

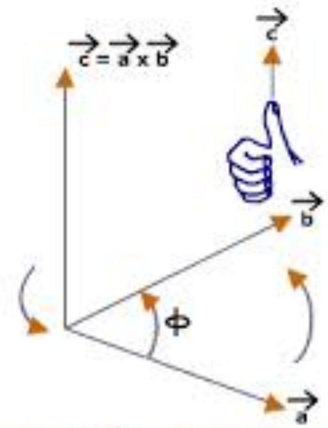


Projection of one vector onto another

Making New Vectors

Cross Product

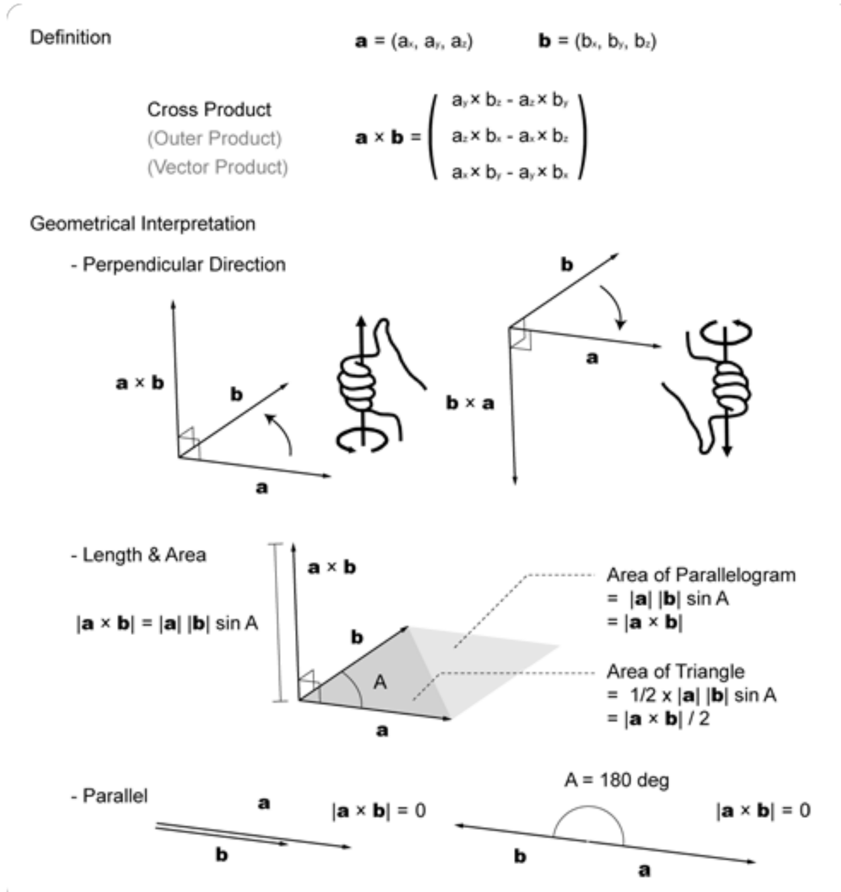




$$\mathbf{c} = \mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} =$$

$$= (a_2 b_3 - a_3 b_2)\mathbf{i} + (a_3 b_1 - a_1 b_3)\mathbf{j} + (a_1 b_2 - a_2 b_1)\mathbf{k}$$

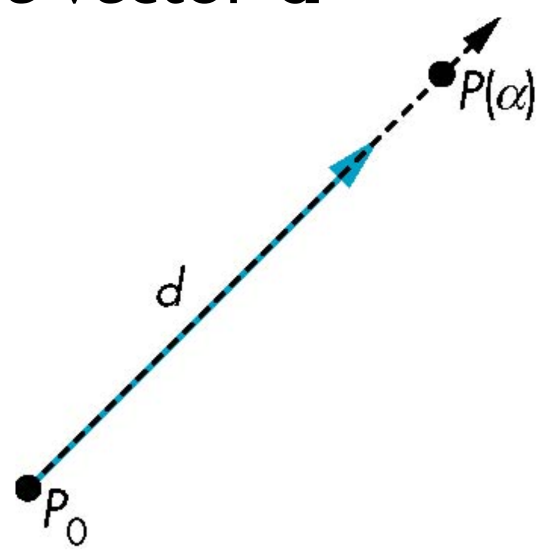
Cross Product



Parametric Forms

Lines, Rays

- 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 **d**



2D Forms for lines

- Two-dimensional forms
  - Explicit: $y = mx + h$
  - Implicit: $ax + by + c = 0$
  - Parametric:

$$x(a) = ax_0 + (1-a)x_1$$
$$y(a) = ay_0 + (1-a)y_1$$

Rays, Line Segments

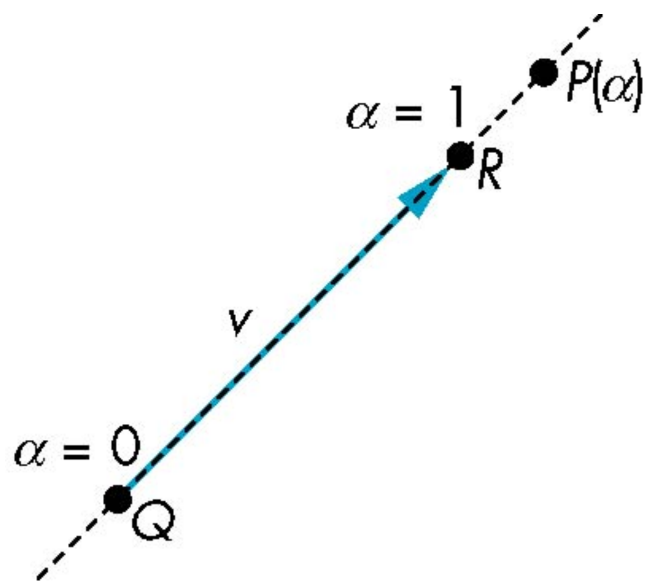
If a >= 0, then P(a) is the *ray* leaving P_0 in the direction **d**

If we use two points to define v, then

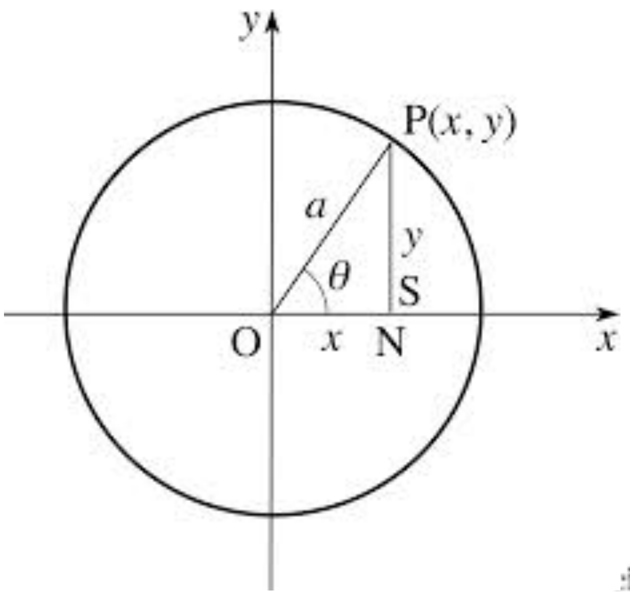
P(a) = Q + a (R-Q)=Q+av

=aR + (1-a)Q

For 0<=a<=1 we get all the points on the *line segment* joining R and Q



Curves

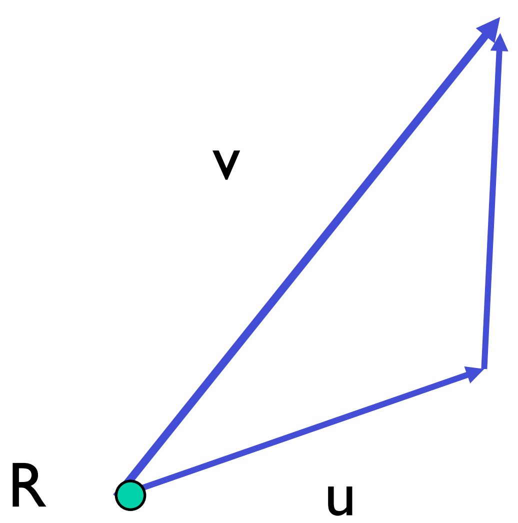


$$\begin{cases} x = R\cos t \\ y = R\sin t \end{cases}, \qquad 0 \le t \le 2\pi,$$

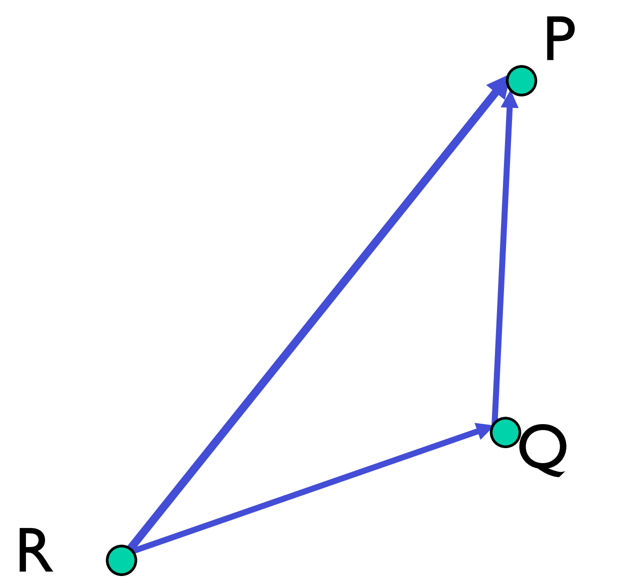
	implicit form	parametric form	
ircle	$x^2 + y^2 - r^2 = 0$	$x(t) = r\dfrac{1-t^2}{1+t^2}$	$y(t) = r\dfrac{2t}{1+t^2}$
ellipse	$\dfrac{x^2}{a^2} + \dfrac{y^2}{b^2} - 1 = 0$	$x(t) = a\dfrac{1-t^2}{1+t^2}$	$y(t) = b\dfrac{2t}{1+t^2}$
hyperbola	$\dfrac{x^2}{a^2} - \dfrac{y^2}{b^2} - 1 = 0$	$x(t) = a\dfrac{1+t^2}{1-t^2}$	$y(t) = b\dfrac{2t}{1-t^2}$
parabola	$y^2 - 2px = 0$	$x(t) = \dfrac{t^2}{2p}$	$y(t) = t$

Planes

Defined by a point and two vectors or by three points

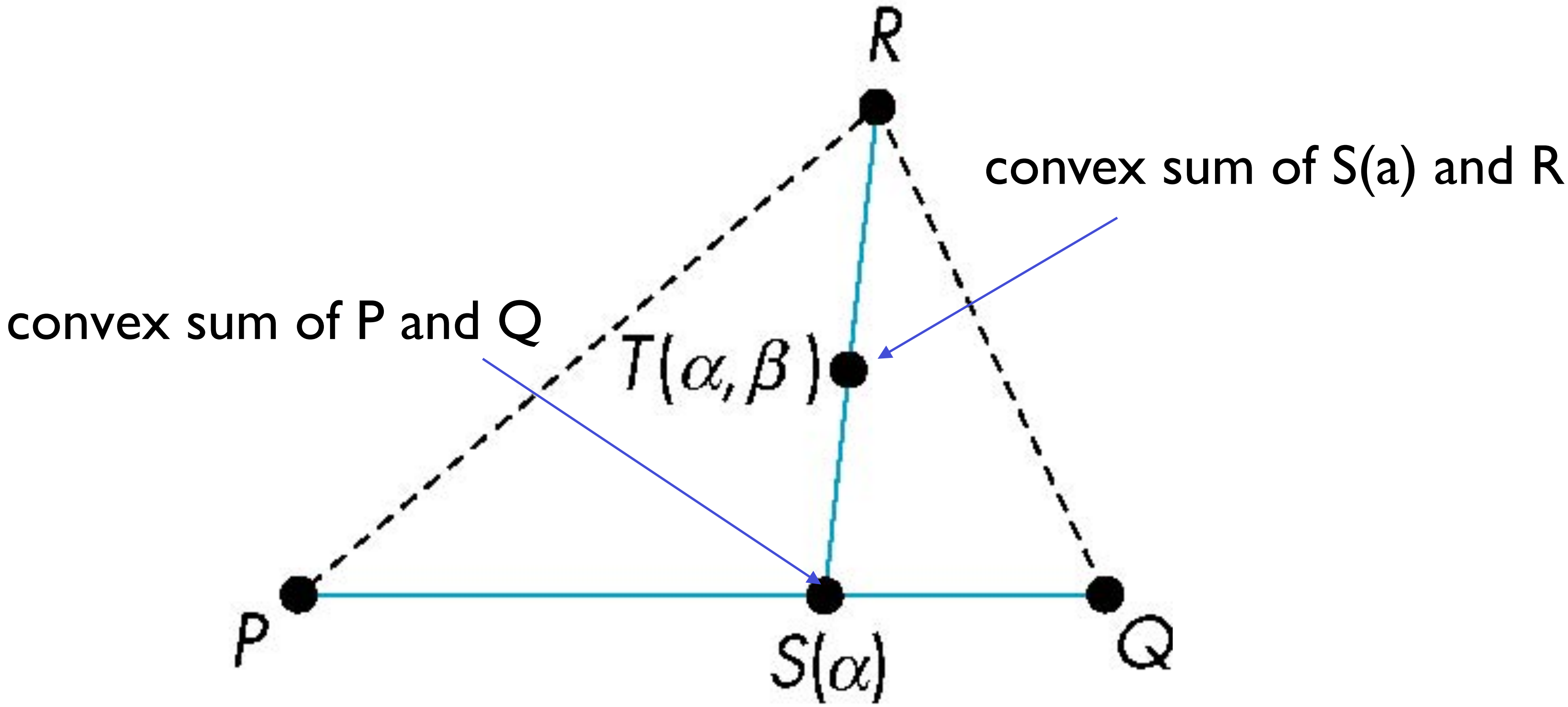


P(a,b)=R+au+bv



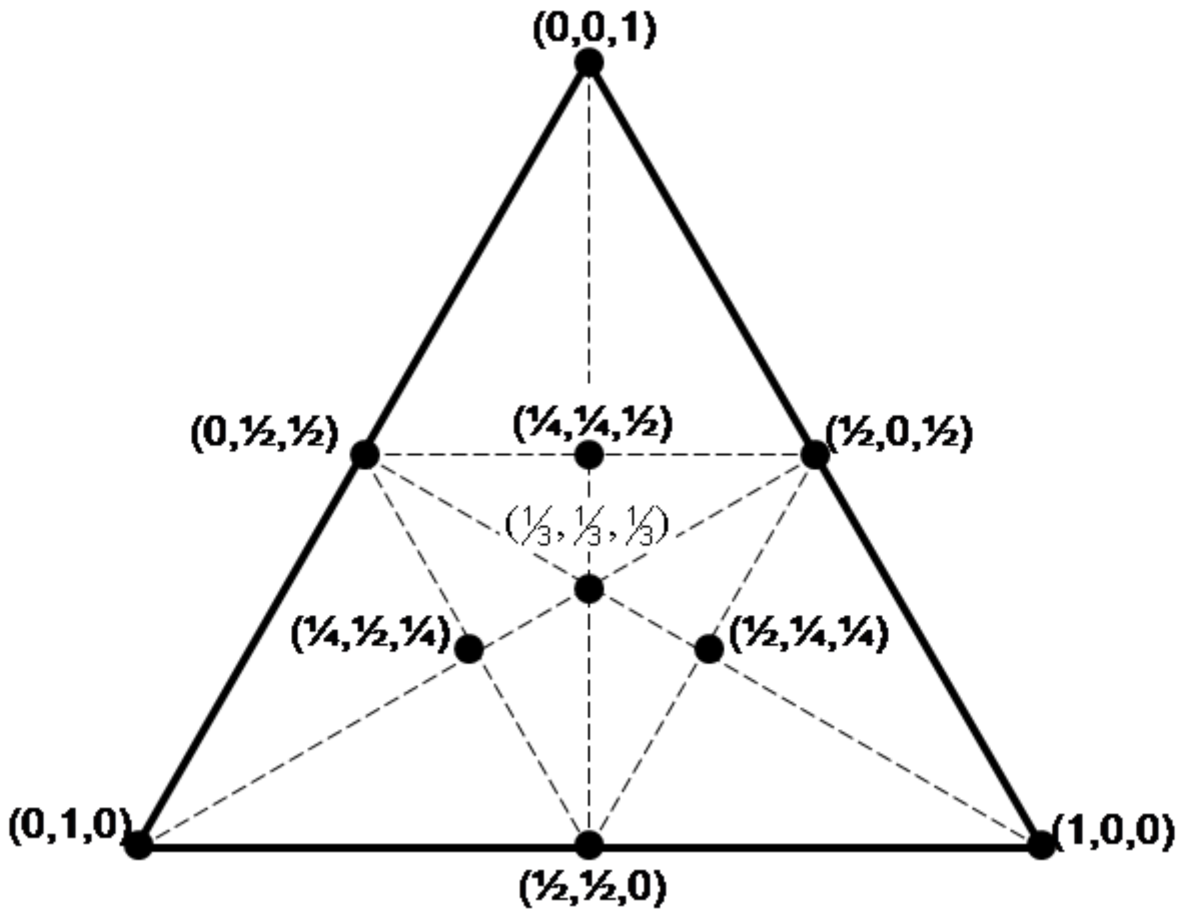
P(a,b)=R+a(Q-R)+b(P-Q)

Triangles



for 0<=α,β<=1, we get all points in triangle

Barycentric Coordinates



Barycentric Coordinates

Triangle is convex

Any point inside can be represented as an affine sum

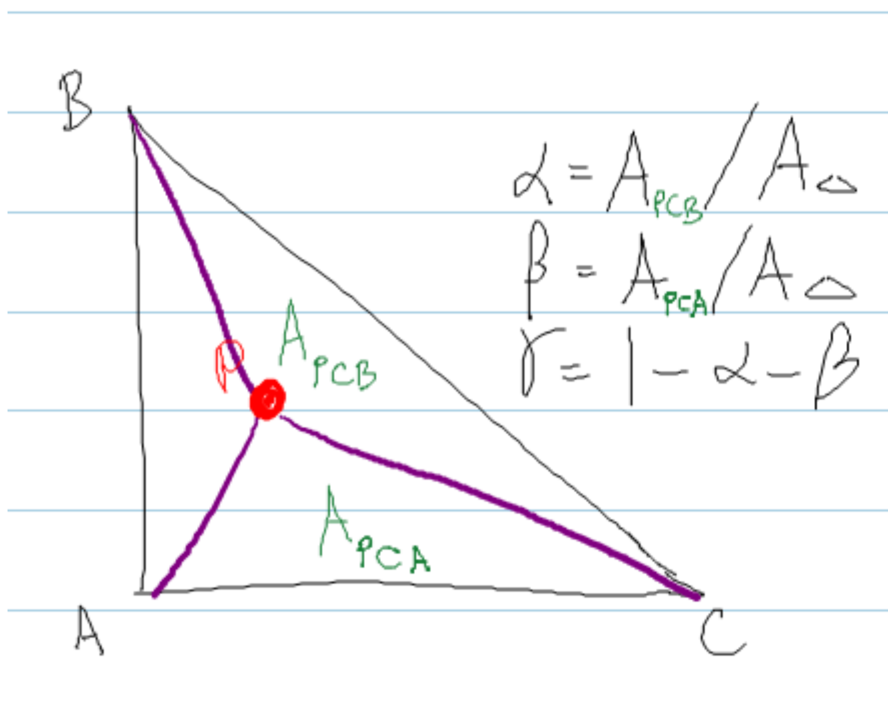
$P(\alpha_{1,}\ \alpha_{2,}\ \alpha_3) = \alpha A + \beta B + \gamma C$

where

$\alpha + \beta + \gamma = 1$

$\alpha_{,}\ \beta, \gamma >= 0$

Barycentric Coordinates



Calculating Areas ?

Matrices

Matrices

- Definitions
- Matrix Operations
- Row and Column Matrices
- Rank
- Change of Representation
- Cross Product

What is a Matrix?

Elements, organized into rows and columns

$$\begin{array}{cc} & \text{rows} \\ \text{columns} & \begin{bmatrix} a & b \\ c & d \end{bmatrix} \end{array}$$

Definitions $\mathbf{A} = \lfloor a_{ij} \rfloor$

n x *m* Array of Scalars (*n* Rows and *m* Columns)

– *n*: row *dimension* of a matrix, *m*: column *dimension*

– *m* = *n*: *square matrix* of dimension *n*

– Element

$\{a_{ij}\}, \; i = 1, \ldots, n, \;\; j = 1, \ldots, m$

– *Transpose*: interchanging the rows and columns of a matrix $\mathbf{A}^T = \lfloor a_{ji} \rfloor$

- Column Matrices and Row Matrices

– *Column matrix* (*n* x 1 matrix): $\mathbf{b} = [b_i] = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$

– *Row matrix* (1 x *n* matrix): $\mathbf{b}^T$

Basic Operations

Addition, Subtraction, Multiplication

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} a+e & b+f \\ c+g & d+h \end{bmatrix}$$

add elements

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} - \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} a-e & b-f \\ c-g & d-h \end{bmatrix}$$

subtract elements

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix}$$

Multiply each row by each column

Matrix Operations

+ *Scalar-Matrix Multiplication*

$$\alpha\mathbf{A} = \left\lfloor \alpha a_{ij} \right\rfloor$$

+ *Matrix-Matrix Addition*

$$\mathbf{C} = \mathbf{A} + \mathbf{B} = \left\lfloor a_{ij} + b_{ij} \right\rfloor$$

+ *Matrix-Matrix Multiplication*
  + A: *n* x *l* matrix, B: *l* x *m* ➔ C: *n* x *m* matrix

$$\mathbf{C} = \mathbf{AB} = \left[c_{ij} \right]$$

$$c_{ij} = \sum_{k=1}^{l} a_{ik} b_{kj}$$

Matrix Operations

+ Properties of Scalar-Matrix Multiplication

$$\alpha(\beta \mathbf{A}) = (\alpha\beta)\mathbf{A}$$

$$\alpha\beta\mathbf{A} = \beta\alpha\mathbf{A}$$

+ Properties of Matrix-Matrix Addition
  + Commutative: $\mathbf{A} + \mathbf{B} = \mathbf{B} + \mathbf{A}$
  + Associative: $\mathbf{A} + (\mathbf{B} + \mathbf{C}) = (\mathbf{A} + \mathbf{B}) + \mathbf{C}$
+ Properties of Matrix-Matrix Multiplication

$$\mathbf{A}(\mathbf{BC}) = (\mathbf{AB})\mathbf{C}$$

$$\mathbf{AB} \neq \mathbf{BA}$$

+ ***Identity Matrix*** **I** (Square Matrix)

$$\mathbf{I} = [a_{ij}], \quad a_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

$$\mathbf{AI} = \mathbf{A}$$

$$\mathbf{IB} = \mathbf{B}$$

Identity Matrix

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Multiplication

- Is AB = BA? Maybe, but maybe not!

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae+bg & \dots \\ \dots & \dots \end{bmatrix} \qquad \begin{bmatrix} e & f \\ g & h \end{bmatrix}\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} ea+fc & \dots \\ \dots & \dots \end{bmatrix}$$

- Heads up: multiplication is NOT commutative!

Row and Column Matrices

- Column Matrix
  - p^T: row matrix
- *Concatenations*
  - Associative
- By Row Matrix

$$\mathbf{p} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$\mathbf{p}' = \mathbf{A}\mathbf{p}$$

$$\mathbf{p}' = \mathbf{A}\mathbf{B}\mathbf{C}\mathbf{p}$$

$$(\mathbf{A}\mathbf{B})^T = \mathbf{B}^T\mathbf{A}^T$$

$$\mathbf{p}'^T = \mathbf{p}^T\mathbf{C}^T\mathbf{B}^T\mathbf{A}^T$$

Inverse of a Matrix

- Identity matrix:
  $AI = A$
- Some matrices have an inverse, such that:
  $AA^{-1} = I$
- Inversion is tricky:
  $(ABC)^{-1} = C^{-1}B^{-1}A^{-1}$
- Derived from non-commutativity property

Determinant of a Matrix

- Used for inversion
- If det(A) = 0, then A has no inverse
- Can be found using factorials, pivots, and cofactors!
- And for Areas of Triangles

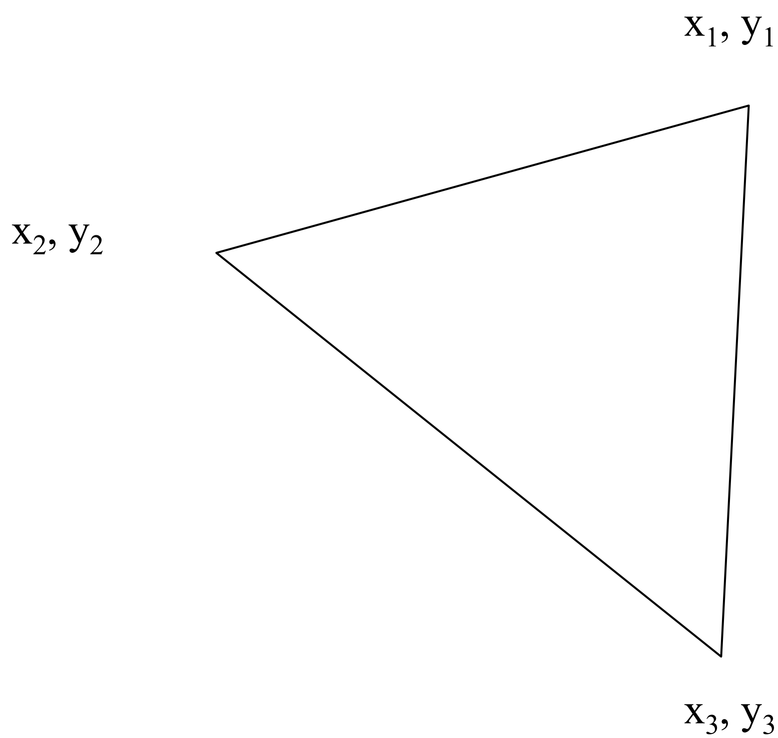
$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

$$\det(A) = ad - bc$$

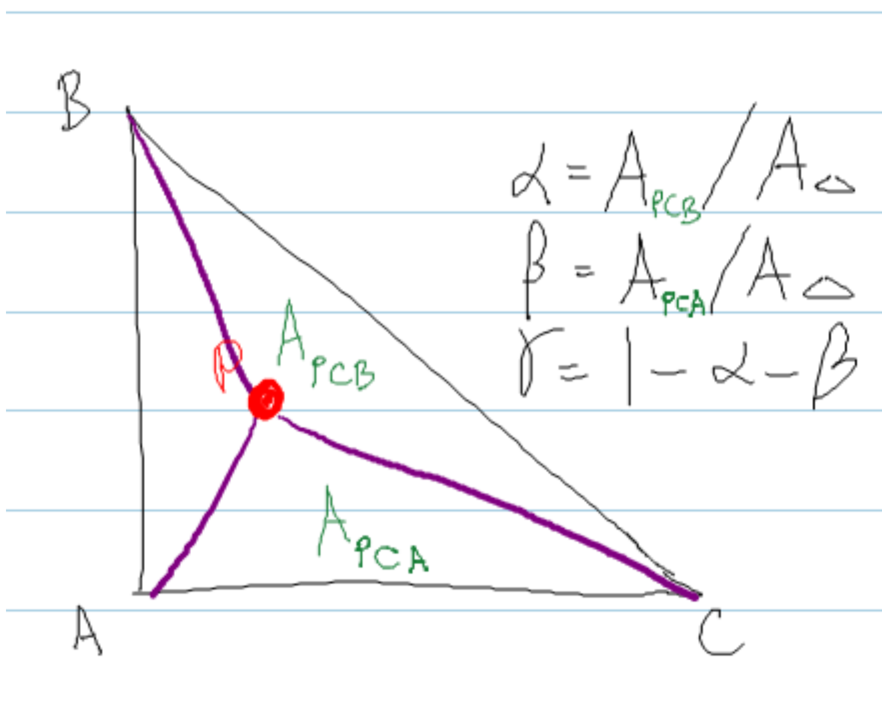
$$A^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

Area of Triangle – Cramer's Rule

$$Ans = \frac{1}{2} \bullet \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}$$



Use This Here



Transformations