

# Game Engine Architecture

## Chapter 5 3D Math for Games

Hooman Salamat

# Topics

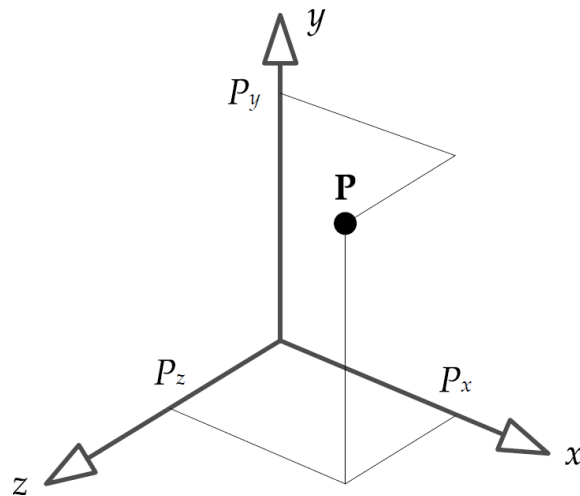
- Math for games
  - Points and Vectors
  - Matrices
  - Quaternions
  - Comparison of rotational representations
  - Other useful math objects
  - Hardware-accelerated SIMD Math
  - Random Number Generation

# 2D as a Start

- Most operations in 3D also make sense in 2D
- Always use 2D as a basis, it'll help

# Points and vectors

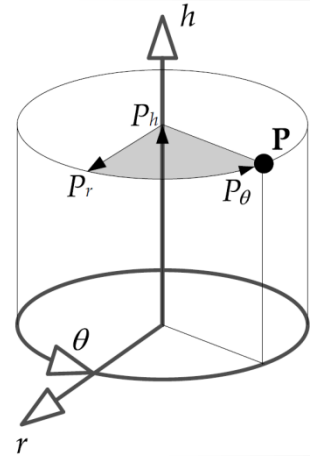
- A point is a location in n-dimensional space
- Usually represented in Cartesian space
  - Two or three mutually perpendicular axes
  - A point is a triple of numbers  $(P_x, P_y, P_z)$



# Other systems

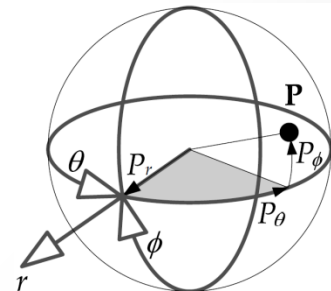
- Cylindrical

- Employs a height axis ( $h$ ), a radial axis ( $r$ ), and a yaw angle ( $\theta$ )
- Points represented as  $(P_h, P_r, P_\theta)$



- Spherical

- Pitch( $\phi$ ), yaw( $\theta$ ), and radial ( $r$ )
- Points represented as  $(P_r, P_\phi, P_\theta)$

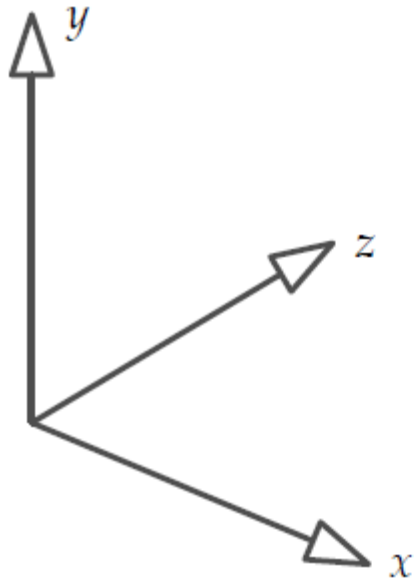


# Picking a system

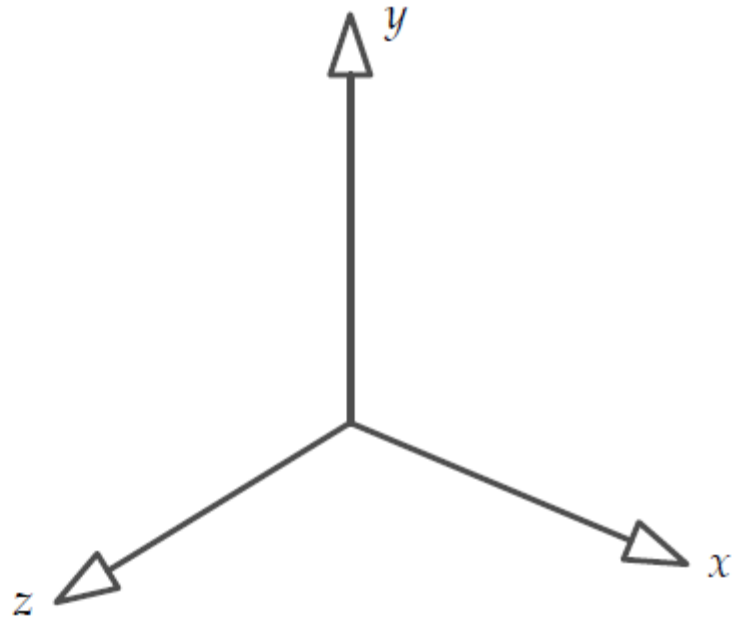
- Most people use Cartesian
- Sometimes it is easier to use something else
  - Swirling objects around a character are easier in cylindrical
  - Explosions may be easier in spherical

# Left vs Right

- When using Cartesian coordinates, you get to choose either left or right handed coordinate systems



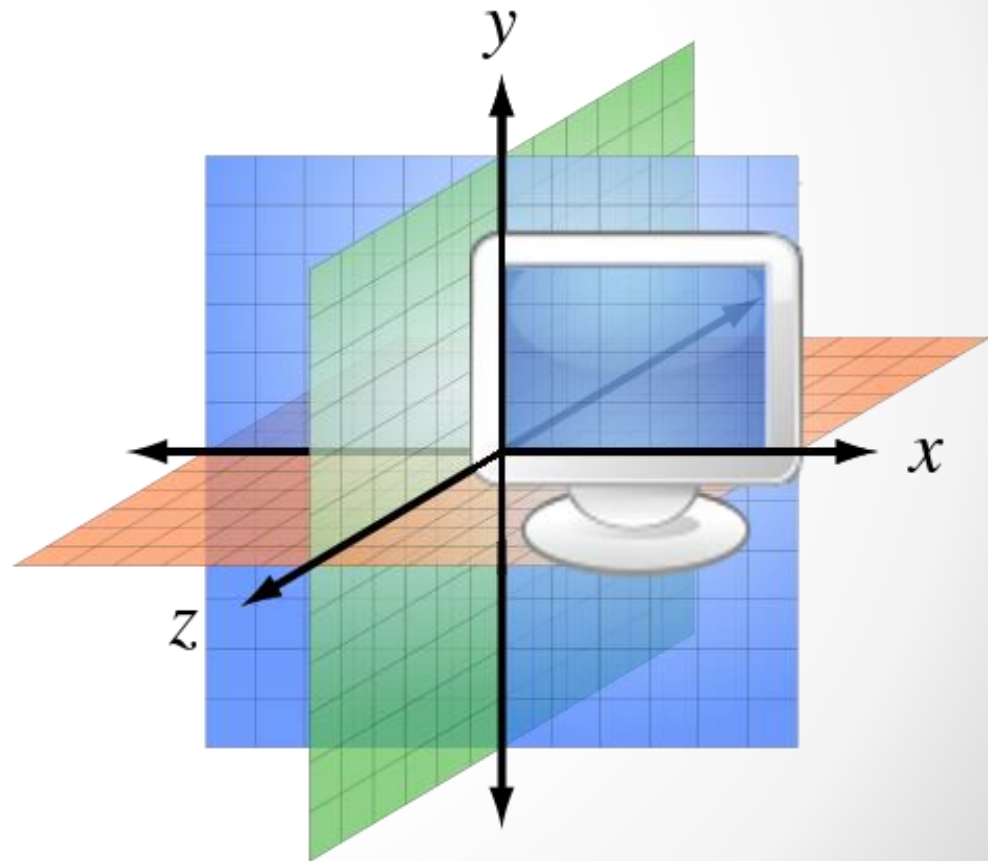
**Left-Handed**



**Right-Handed**

# Ogre

- **Ogre** is using a **right-handed** coordinate system like OpenGL (DirectX by default is left-handed)





# Left vs Right

- Converting is easy, flip the direction of any one axis.
- Graphics programmers typically pick left-handed with y pointing up, x to the right, and z pointing into the screen
- Helps with z-buffering

# Vectors

- Vector have
  - Magnitude
  - Direction
- Extend from the tail toward the head
- Differs from a scalar because of the direction
- Technically represents an offset relative to a known point
- Can be used to represent a point if it is relative to the origin
  - Called a position vector versus direction vector

# Basis Vectors

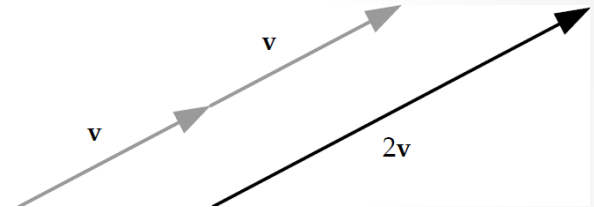
- Sometimes useful to define orthogonal unit vectors
  - **i** is along the x-axis
  - **j** is along the y-axis
  - **k** is along the z-axis
- Can now represent points as the sum of scalars multiplied by the unit vectors

$$(5, 3, -2) = 5\mathbf{i} + 3\mathbf{j} - 2\mathbf{k}$$

# Vector Operations

- Multiply by a scalar - scales the magnitude without affecting direction

$$s\mathbf{a} = (sa_x, sa_y, sa_z)$$



- Non-uniform scaling

$$\mathbf{s} \otimes \mathbf{a} = (s_x a_x, s_y a_y, s_z a_z)$$

$$a\mathbf{S} = \begin{bmatrix} a_x & a_y & a_z \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} = \begin{bmatrix} a_x s_x & a_y s_y & a_z s_z \end{bmatrix}$$

# Vector operations

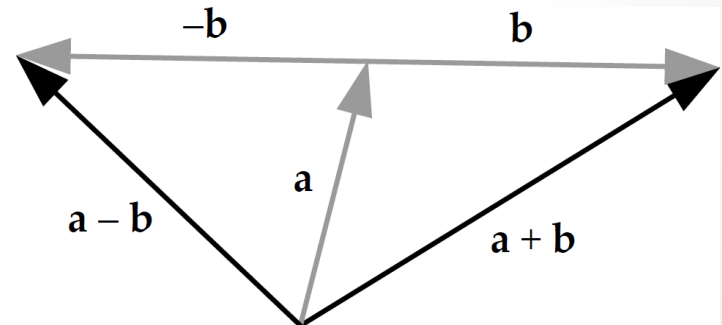
- Addition and Subtraction

$$a + b = [(a_x + b_x), (a_y + b_y), (a_z + b_z)]$$

$$a - b = [(a_x - b_x), (a_y - b_y), (a_z - b_z)]$$

- Points and directions

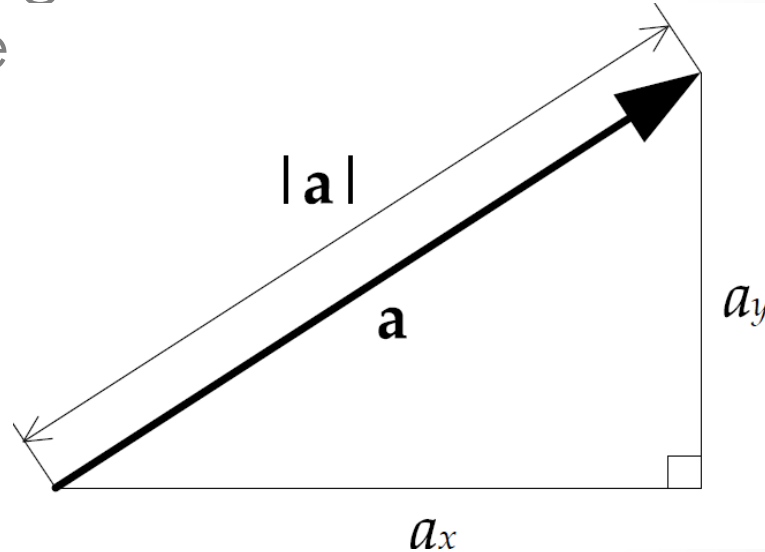
- direction + direction = direction
- direction - direction = direction
- point + direction = point
- point - point = direction
- point + point = crap



# Vector Operations

- Calculating magnitude – think of it as distance
- We can use the Pythagorean theorem to calculate a vector's magnitude

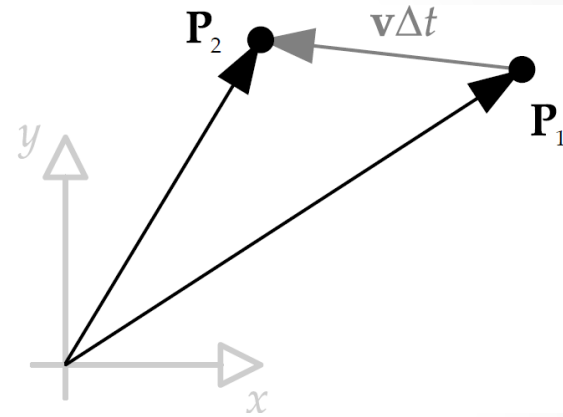
$$|\mathbf{a}| = \sqrt{a_x^2 + a_y^2 + a_z^2}$$



# Use of Vector Operations

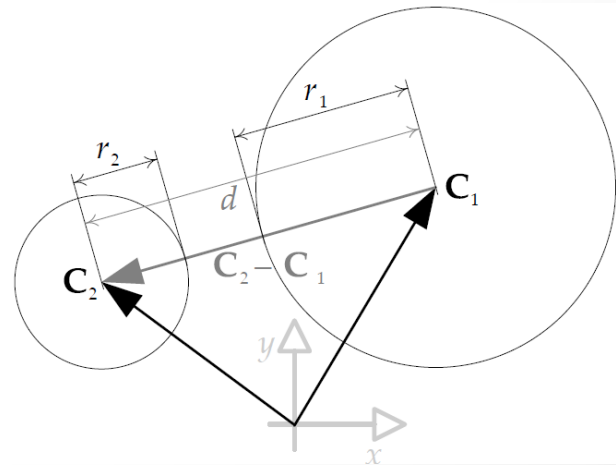
- Moving objects

- $\mathbf{P}_2 = \mathbf{P}_1 + \mathbf{v}\Delta t$



- Object collision

- if  $d < r_1 + r_2$  then they collide
  - Faster to compare  $d^2 < (r_1 + r_2)^2$



# Unit Vectors

- Sometimes you need a unit length vector in the same direction as the original – called normalization, not a normal vector  $\mathbf{u} = \frac{\mathbf{v}}{|\mathbf{v}|} = \frac{1}{v} \mathbf{v}$
- A vector is said to be *normal* to a surface if it is *perpendicular* to that surface.
- Lighting calculations make heavy use of normal vectors to define the direction of surfaces relative to the direction of the light rays.
- Normal vectors are usually of unit length



# DOT product

- Add the components of the vector

$$\mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y + a_z b_z$$

- Also written as

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| |\mathbf{b}| \cos(\theta)$$

- It is commutative, distributive, and works with scalar multiplication

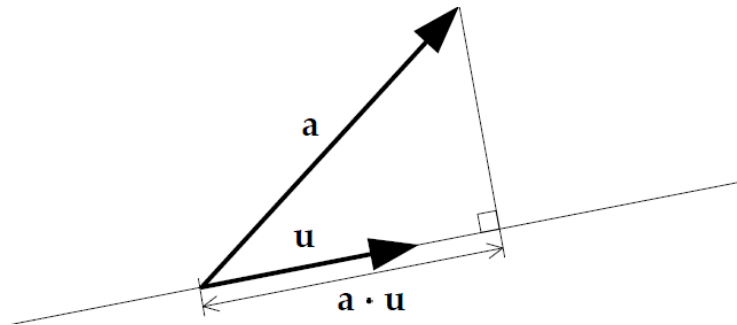
$$\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a}$$

$$\mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) = \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \cdot \mathbf{c}$$

$$s\mathbf{a} \cdot \mathbf{b} = \mathbf{a} \cdot s\mathbf{b} = s(\mathbf{a} \cdot \mathbf{b})$$

# Vector Projection

- If  $\mathbf{u}$  is a unit vector:  $|\mathbf{u}| = 1$
- then the dot product of  $\mathbf{a}$  and  $\mathbf{u}$  represents the length of the projection of  $\mathbf{a}$  onto  $\mathbf{u}$



# Magnitude

- You can use the dot product to find the magnitude as well

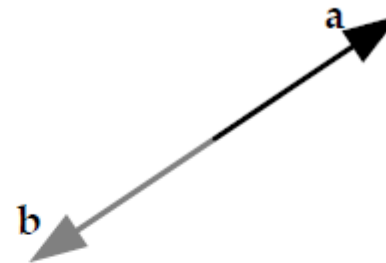
$$|\mathbf{a}|^2 = \mathbf{a} \cdot \mathbf{a}$$

$$|\mathbf{a}| = \sqrt{\mathbf{a} \cdot \mathbf{a}}$$

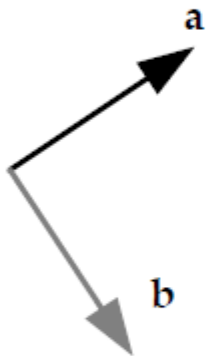
# Useful Dot Product Tests



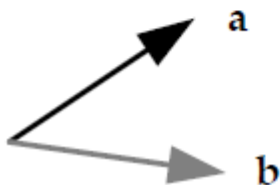
$$(a \cdot b) = ab$$



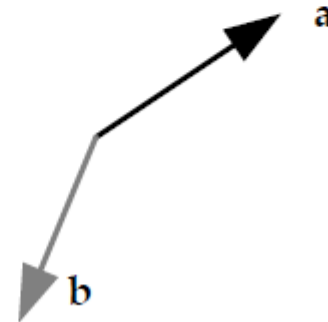
$$(a \cdot b) = -ab$$



$$(a \cdot b) = 0$$



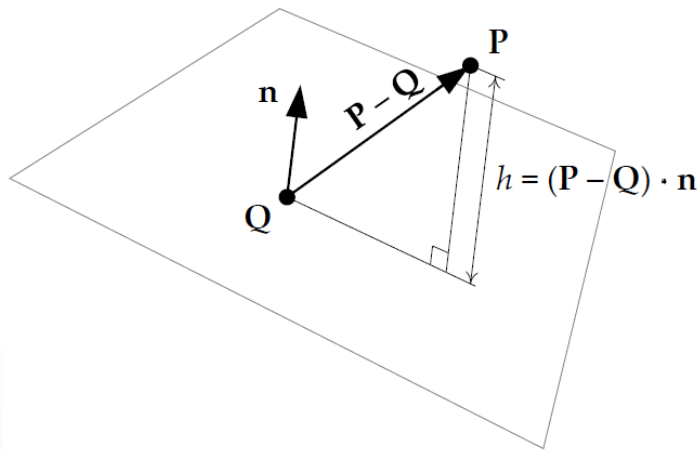
$$(a \cdot b) > 0$$



$$(a \cdot b) < 0$$

# Other applications

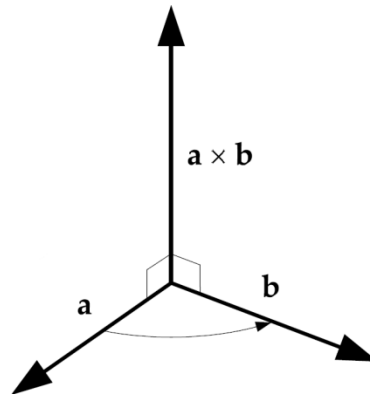
- Visibility – can player P see enemy E
  - $\mathbf{v} = \mathbf{E} - \mathbf{P}$  gives the relative position of E in relation to P
  - if  $\mathbf{f}$  is the facing vector of P then when  $\mathbf{v} \cdot \mathbf{f}$  is negative E is behind P
- If we define a plane as a point  $\mathbf{Q}$  and a normal  $\mathbf{n}$  then we can find the height  $h$  of a point P above the plane using projection



# Cross Product

- Yields another vector that is perpendicular to the vectors being multiplied

$$\mathbf{a} \times \mathbf{b} = [(a_y b_z - a_z b_y), (a_z b_x - a_x b_z), (a_x b_y - a_y b_x)]$$
$$= (a_y b_z - a_z b_y)\mathbf{i} + (a_z b_x - a_x b_z)\mathbf{j} + (a_x b_y - a_y b_x)\mathbf{k}$$

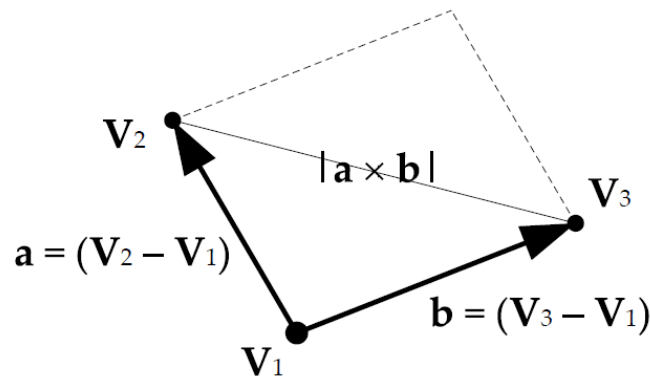


# Magnitude of X-product

- Magnitude of the cross product is the area of the parallelogram formed by the two vectors

$$|\mathbf{a} \times \mathbf{b}| = |\mathbf{a}||\mathbf{b}|\sin(\theta)$$

$$A_{triangle} = \frac{1}{2} |(\mathbf{V}_2 - \mathbf{V}_1) \times (\mathbf{V}_3 - \mathbf{V}_1)|$$



# Properties of the cross product

- It is not commutative

$$\mathbf{a} \times \mathbf{b} \neq \mathbf{b} \times \mathbf{a}$$

- It is anti-commutative

$$\mathbf{a} \times \mathbf{b} = -\mathbf{b} \times \mathbf{a}$$

- Distributive over addition

$$\mathbf{a} \times (\mathbf{b} + \mathbf{c}) = \mathbf{a} \times \mathbf{b} + (\mathbf{a} \times \mathbf{c})$$

- Combine with scalar multiplication

$$(s\mathbf{a}) \times \mathbf{b} = \mathbf{a} \times (s\mathbf{b}) = s(\mathbf{a} \times \mathbf{b})$$

- Cartesian basis vector are related by cross product

$$(\mathbf{i} \times \mathbf{j}) = \mathbf{k}$$

$$(\mathbf{j} \times \mathbf{k}) = \mathbf{i}$$

$$(\mathbf{k} \times \mathbf{i}) = \mathbf{j}$$



# Uses of the cross product

- Finding a vector that is perpendicular to two other vectors
- Finding the normal vector to a plane

$$\mathbf{n} = \textit{normalize}((\mathbf{P}_2 - \mathbf{P}_1) \times (\mathbf{P}_3 - \mathbf{P}_1))$$

- Calculate torque
  - Given a force  $\mathbf{F}$  and a vector  $\mathbf{r}$  from the center of mass the torque is

$$\mathbf{N} = \mathbf{r} \times \mathbf{F}$$

# Pseudovectors

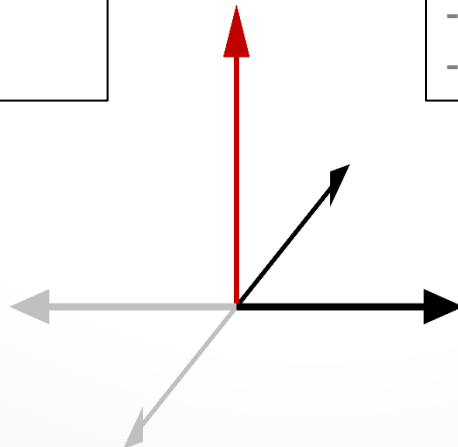
- Cross product doesn't actually produce a vector
  - Creates a pseudovector
- Difference is quite subtle – only apparent on reflection
  - Vectors reflect to mirror image
  - Pseudovectors reflect to mirror image, but also change direction

## Vectors

- Position
- Velocity
- Acceleration

## Pseudovectors

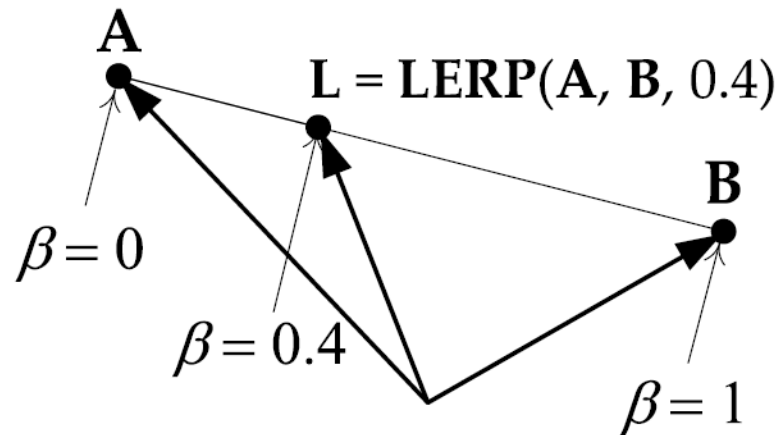
- Angular velocity
- Magnetic fields
- Normals to a surface



# LERP

- A simple linear interpolation between 2 points  
( $\beta$  ranges from 0 to 1)

$$\begin{aligned} \mathbf{L} &= \text{LERP}(\mathbf{A}, \mathbf{B}, \beta) = (1 - \beta)\mathbf{A} + \beta\mathbf{B} \\ &= [(1 - \beta)\mathbf{A}_x + \beta\mathbf{B}_x, (1 - \beta)\mathbf{A}_y + \beta\mathbf{B}_y, (1 - \beta)\mathbf{A}_z + \beta\mathbf{B}_z] \end{aligned}$$



# Matrices

- A matrix is a rectangular array of  $m \times n$  scalars

$$\mathbf{M} = \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix}$$

- They are convenient for representing translation, rotation, and scale operations
- When all rows and columns are of unit magnitude it is called isotropic or orthonormal – represent pure rotations

# 4x4 Matrix

- Sometimes 4x4 matrices are used to represent 3D transformations
  - They call them transformation matrices
- Affine matrix is a 4 x 4 transformation matrix that
  - Preserves parallelism of line
  - Relative distance ratio
  - Not necessarily absolute lengths and angles
- Affine matrix is any combination of translation, rotations, scale or shear

# Matrix Multiplication

- The product of two matrices is written as  $\mathbf{P}=\mathbf{AB}$ , if  $\mathbf{A}$  and  $\mathbf{B}$  are transformation matrices then so is  $\mathbf{P}$
- Multiplication is done by taking the dot products of the rows of  $\mathbf{A}$  and the columns of  $\mathbf{B}$ . The inner dimensions must be the same  $n_A=m_B$ .

# Multiplication

$$\mathbf{P} = \mathbf{AB}$$

$$\mathbf{P} = \begin{bmatrix} P_{11} = A_{row1} \cdot B_{col1} & P_{12} = A_{row1} \cdot B_{col2} & P_{13} = A_{row1} \cdot B_{col3} \\ P_{21} = A_{row2} \cdot B_{col1} & P_{22} = A_{row2} \cdot B_{col2} & P_{23} = A_{row2} \cdot B_{col3} \\ P_{31} = A_{row3} \cdot B_{col1} & P_{32} = A_{row3} \cdot B_{col2} & P_{33} = A_{row3} \cdot B_{col3} \end{bmatrix}$$

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

# Points and Vectors as Matrices

- Can represent points or vectors as row matrices (1 x n) or column matrices (n x 1)
- For example  $\mathbf{v} = (3, 4, -1)$  can be

$$\mathbf{v}_1 = [3 \quad 4 \quad -1]$$

$$\mathbf{v}_2 = \begin{bmatrix} 3 \\ 4 \\ -1 \end{bmatrix} = \mathbf{v}_1^T$$



# Vector X Matrix

- If multiplying a row vector ( $1 \times n$ ) by an  $n \times n$  matrix, the vector appears on the left

$$\mathbf{v}'_{1 \times n} = \mathbf{v}_{1 \times n} \mathbf{M}_{n \times n}$$

- If multiplying an  $n \times n$  matrix by a column vector ( $n \times 1$ ) the vector appears on the right

$$\mathbf{v}'_{n \times 1} = \mathbf{M}_{n \times n} \mathbf{v}_{n \times 1}$$

- $\mathbf{v}' = \left( ((v\mathbf{A})\mathbf{B})\mathbf{C} \right)$  row vectors: read left-to-right
- $\mathbf{v}'^T = \left( \mathbf{C}^T (\mathbf{B}^T (\mathbf{A}^T v^T)) \right)$  column vectors: read right-to-left

# Identity matrix

- Usually denoted by the symbol  $\mathbf{I}$
- Always square ( $n=m$ ) with the diagonals = 1 and 0 everywhere else

$$\mathbf{I}_{3 \times 3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{AI} \equiv \mathbf{IA} \equiv \mathbf{A}$$

# Inverse Matrix

- An inverse matrix **A** (denoted **A**<sup>-1</sup>) undoes the effects of matrix **A**
- When you multiply a matrix by its inverse, the result is always the identity matrix **A**(**A**<sup>-1</sup>)  $\equiv$  (**A**<sup>-1</sup>)**A**  $\equiv$  **I**
- Not all matrices have inverses. The ones in game development do.
- Undoing concatenated matrices requires special care

$$(\mathbf{ABC})^{-1} = \mathbf{C}^{-1}\mathbf{B}^{-1}\mathbf{A}^{-1}$$

# Transposition

- The transpose of matrix  $\mathbf{M}$  is  $\mathbf{M}^T$
- Found by reflecting across the diagonal

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

- Inverse of an orthonormal (pure rotation) matrix is equal to its transpose
- Transpose of concatenated matrices is handled like inverse

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

# Homogeneous Coordinates

- A 2x2 matrix can be used to represent rotations in 2 dimensions

$$\begin{bmatrix} r'_x & r'_y \end{bmatrix} = \begin{bmatrix} r_x & r_y \end{bmatrix} \begin{bmatrix} \cos \phi & \sin \phi \\ -\sin \phi & \cos \phi \end{bmatrix}$$

- The same is true for 3 dimensions

$$\begin{bmatrix} r'_x & r'_y & r'_z \end{bmatrix} = \begin{bmatrix} r_x & r_y & r_z \end{bmatrix} \begin{bmatrix} \cos \phi & \sin \phi & 0 \\ -\sin \phi & \cos \phi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

# Translations?

- Can you use a 3 x 3 to represent a translation?
- Nope  $\mathbf{r} + \mathbf{t} = [r_x + t_x \quad r_y + t_y \quad r_z + t_z]$
- We can however use a 4 x 4

$$\mathbf{r} + \mathbf{t} = [r_x \quad r_y \quad r_z \quad 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}$$

$$= [r_x + t_x \quad r_y + t_y \quad r_z + t_z \quad 1]$$

# Points versus Vectors

- Remember points and vectors are different
  - Points can be translated, vectors cannot
- We can achieve this by setting the last value in the homogenized vector to 1 for point and 0 for vectors
  - Eliminates the effects of translations

$$[\mathbf{v} \ 0] \begin{bmatrix} U & 0 \\ t & 1 \end{bmatrix} = [(\mathbf{v}\mathbf{U} + 0\mathbf{t}) \ 0] = [\mathbf{v}\mathbf{U} \ 0]$$

- Technically we can convert from 4D homogeneous to 3D non-homogeneous by dividing

$$[x \ y \ z \ w] \equiv \begin{bmatrix} x & y & z \\ w & w & w \end{bmatrix}$$

- Now makes sense why we set  $w=0$  for vectors

# Atomic transformation matrices

- Notice that a 4 x 4 transformation matrix can be partitioned into 4 components

$$M_{affine} = \begin{bmatrix} \mathbf{U}_{3 \times 3} & \mathbf{0}_{3 \times 1} \\ \mathbf{t}_{1 \times 3} & 1 \end{bmatrix}$$

- $\mathbf{U}$  represents rotation and/or scaling
- $\mathbf{t}$  represents the translation
- $\mathbf{0} = [0 \ 0 \ 0]^T$
- a scalar 1



# Point times Atomic

- When you multiply a point by a matrix the result is

$$[\mathbf{r}'_{1 \times 3} \quad 1] = [\mathbf{r}_{1 \times 3} \quad 1] \begin{bmatrix} \mathbf{U}_{3 \times 3} & \mathbf{0}_{3 \times 1} \\ \mathbf{t}_{1 \times 3} & 1 \end{bmatrix} = [\mathbf{r}\mathbf{U} + \mathbf{t} \quad 1]$$

# Atomic translation

Previous translation matrix

Now be written as

$$\mathbf{r} + \mathbf{t} = [r_x \quad r_y \quad r_z \quad 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix} \longrightarrow [r \quad 1] \begin{bmatrix} I & 0 \\ t & 1 \end{bmatrix} = [r + t \quad 1]$$

- The inversion is just the negation of  $\mathbf{t}$

# Atomic Rotation

- Pure rotations have the form

$$\begin{bmatrix} r & 1 \end{bmatrix} \begin{bmatrix} R & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} rR & 1 \end{bmatrix}$$

# Rotations

- The 1 within the rotation matrix always appears on the axis of rotation, the sine and cosine are off axis
- Positive rotations go from
  - x to y - about z
  - y to z - about x
  - z to x - about y - this explains the transpose of the matrix
- The inverse of a pure rotation is its transpose

- $\text{Rotate}_x(\mathbf{r}, \alpha) = \begin{bmatrix} r_x & r_y & r_z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha & 0 \\ 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

# Scaling

Written as

$$\mathbf{rS} = \begin{bmatrix} r_x & r_y & r_z & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

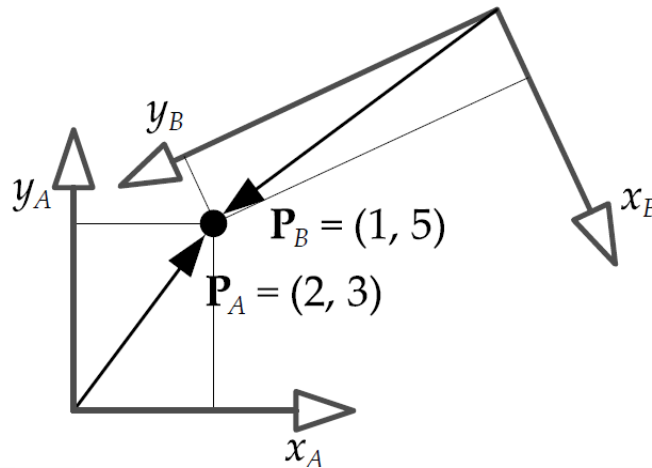
Shorthand of

$$\xrightarrow{\hspace{1cm}} \begin{bmatrix} r & 1 \end{bmatrix} \begin{bmatrix} \mathbf{S}_{3 \times 3} & \mathbf{0} \\ \mathbf{0} & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{rS}_{3 \times 3} & 1 \end{bmatrix}$$

- To invert scaling just use the reciprocals of the scaling factors
- If all the scaling factors are the same it is called uniform scaling

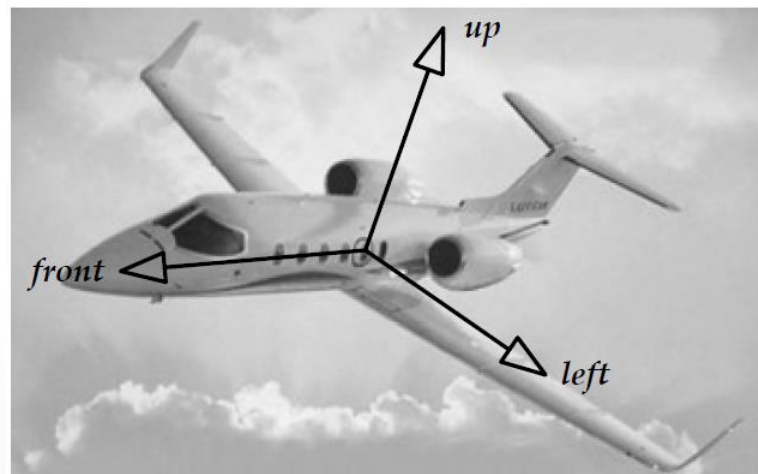
# Coordinate Spaces

- We can think of a point as being a vector relative to a given set of axes
- The axes are just for a frame of reference and are referred to as a coordinate space



# Model space

- When a model is created, the vertices are relative to a coordinate system called model space
- Model space origin is usually in the center of the object
- Model space axes are usually named something like front, left, and up



# Mapping model to world

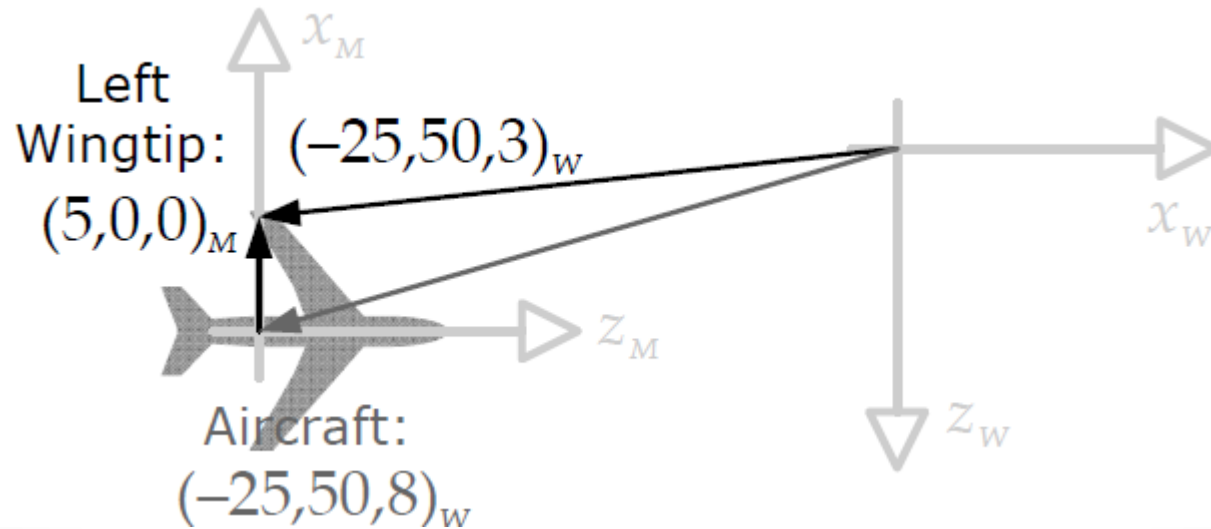
- The mapping can be arbitrary
  - Front =  $\mathbf{k}$ , Left =  $\mathbf{i}$ , Up =  $\mathbf{j}$  or Front =  $\mathbf{i}$ , Right =  $\mathbf{k}$ , Up =  $\mathbf{j}$
- Why use model space?
  - Consider pitch, yaw, and roll
  - They cannot be represented in terms of  $\mathbf{i}, \mathbf{j}, \mathbf{k}$



# World Space

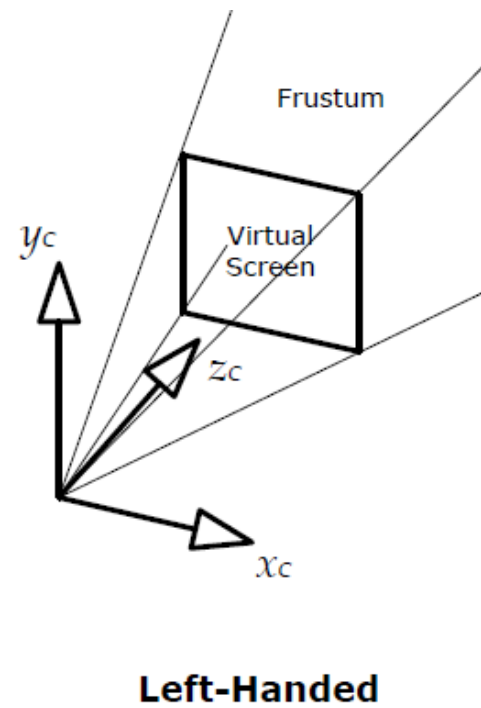
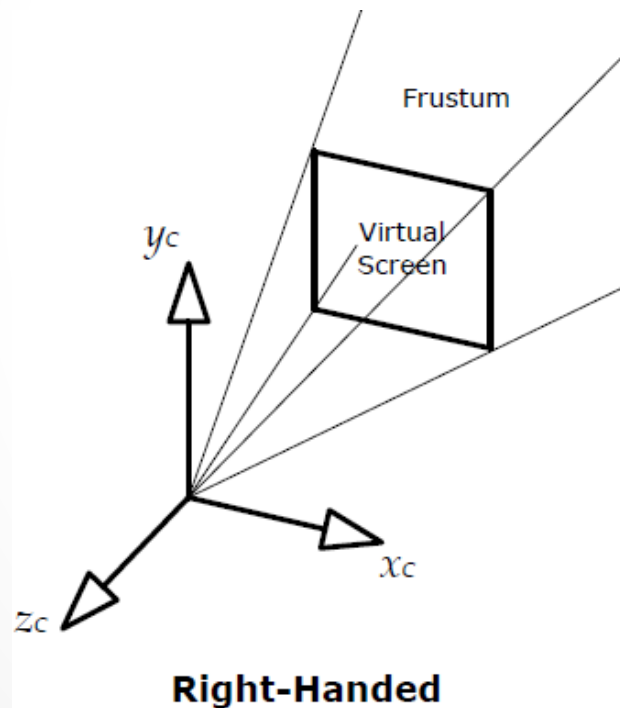
- A fixed coordinate system where the objects, orientations, and scales are defined
- Center usually placed at the center of the playable area
- The orientation is arbitrary but usually y or z is up

# Model to world



# View space

- A coordinate space fixed to the location of the camera



# Change of basis

- All coordinate spaces are relative
  - You must specify them in relation to another space
- This implies a hierarchy of spaces
  - The world space is at the root
- The camera has a location in world space, so its view space has the world space as a parent

# Change of basis matrix

- The matrix from transforming a child to parent space is called

$$\mathbf{M}_{C \rightarrow P}$$

- With the matrix any child-space position can be transformed into a parent-space position

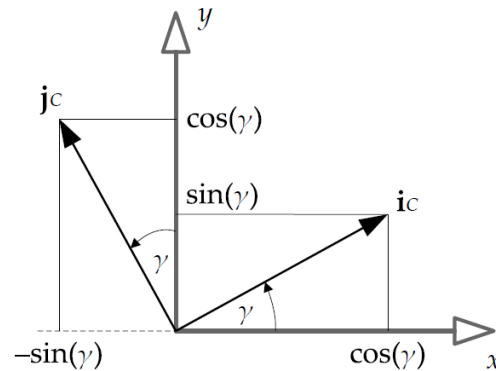
$$\mathbf{P}_P = \mathbf{P}_C \mathbf{M}_{C \rightarrow P}$$

$$\mathbf{M}_{C \rightarrow P} = \begin{bmatrix} \mathbf{i}_C & 0 \\ \mathbf{j}_C & 0 \\ \mathbf{k}_C & 0 \\ \mathbf{t}_C & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{i}_{Cx} & \mathbf{i}_{Cy} & \mathbf{i}_{Cz} & 0 \\ \mathbf{j}_{Cx} & \mathbf{j}_{Cy} & \mathbf{j}_{Cz} & 0 \\ \mathbf{k}_{Cx} & \mathbf{k}_{Cy} & \mathbf{k}_{Cz} & 0 \\ \mathbf{t}_{Cx} & \mathbf{t}_{Cy} & \mathbf{t}_{Cz} & 1 \end{bmatrix}$$

$\mathbf{i}_C$  unit x-axis basis vector of child in parent space  
 $\mathbf{j}_C$  unit y-axis basis vector of child in parent space  
 $\mathbf{k}_C$  unit z-axis basis vector of child in parent space  
 $\mathbf{t}_C$  translation of child relative to parent space

# An example

- Consider a child space rotating by  $\gamma$  around the z-axis



- We can see that  $i_c = [\cos \gamma \quad \sin \gamma \quad 0]$  and  $j_c = [-\sin \gamma \quad \cos \gamma \quad 0]$

# Child to Parent

- By putting these into our mapping equation with  $k_C = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$  we get

$$\mathbf{M}_{C \rightarrow P} = \begin{bmatrix} \cos \gamma & \sin \gamma & 0 & 0 \\ -\sin \gamma & \cos \gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = rotate_z(r, \gamma)$$

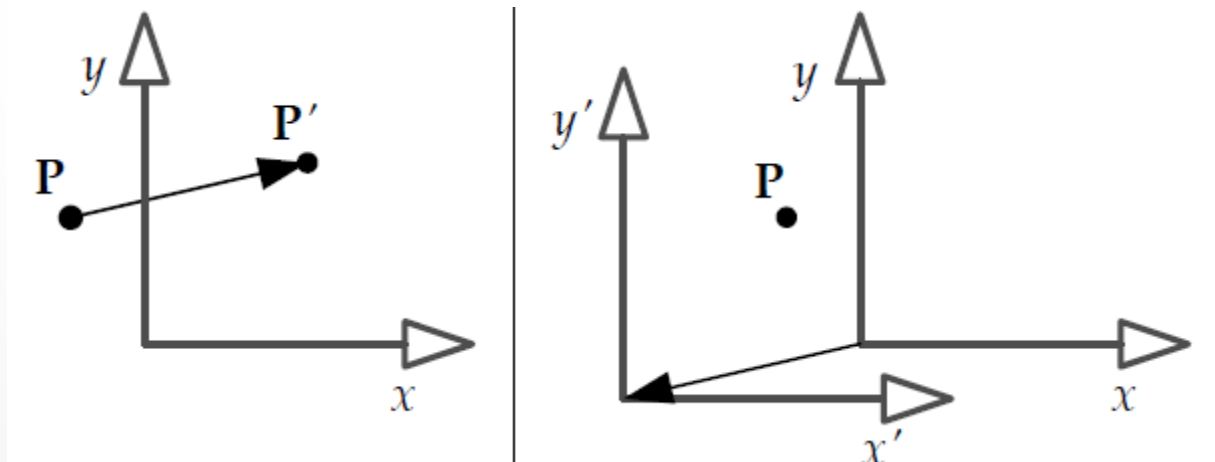
# Extracting the basis

- Given a 4 X 4 transformation matrix we can extract the unit basis vectors by grabbing the proper row
- Say you have a vehicle's model-to-world affine matrix and assume the vehicle always points in the positive z-axis
- We can extract  $\mathbf{k}_c$  from the transformation to get the vehicle facing direction (by grabbing its third row)



# Confusion


- You can transform a coordinate system as well as a point, but in the opposite direction
- So, if a matrix transforms a point from child to parent space, then it also transforms coordinate axes from parent to child space



# Transforming normal vectors

- Remember a normal vector has to remain perpendicular to the surface it is associated with
- In general if the point can be rotated from A to B with a 3X3 matrix  $\mathbf{M}_{A \rightarrow B}$  then the normal vector will be transformed by  $(\mathbf{M}_{A \rightarrow B}^{-1})^T$

## Transforming normal vectors



normal tangent

$\mathbf{n}^T \mathbf{t} = 0$

$\mathbf{t}' = \mathbf{M} \mathbf{t}$

$\mathbf{n}'^T \mathbf{t}' = 0$

If M is a rotation,  
 $(\mathbf{M}^{-1})^T = \mathbf{M}$

$(\mathbf{n}^T \mathbf{M}^{-1})(\mathbf{M} \mathbf{t}) = 0$

$\mathbf{n}' = (\mathbf{n}^T \mathbf{M}^{-1})^T = (\mathbf{M}^{-1})^T \mathbf{n}$

# Storing Matrices in Memory

- `Ogre::Matrix3 rotMat = Matrix3(0.28, 0.96, -0.04, 0.58, -0.20, -0.79, -0.77, 0.20, -0.61);`
- syntax, the first subscript is the row and the second is the column,

```
Matrix3(Real fEntry00, Real fEntry01, Real fEntry02,  
Real fEntry10, Real fEntry11, Real fEntry12,  
Real fEntry20, Real fEntry21, Real fEntry22)  
{  
    m[0][0] = fEntry00;  
    m[0][1] = fEntry01;  
    m[0][2] = fEntry02;  
    m[1][0] = fEntry10;  
    m[1][1] = fEntry11;  
    m[1][2] = fEntry12;  
    m[2][0] = fEntry20;  
    m[2][1] = fEntry21;  
    m[2][2] = fEntry22;  
}
```

# Quaternions

- We can use a 3X3 matrix to represent a rotations
- Not ideal
  - Nine floating point numbers for 3 DOF
  - Rotating a vector requires 9 multiplications and 6 additions
  - They are hard to interpolate
- We can use a quaternion

$$q = [q_x \quad q_y \quad q_z \quad q_w]$$

# Quaternion

- Developed by Sir William Rowan Hamilton in 1843
  - Used to solve problems in mechanics
- The unit length quaternions can represent 3D rotations
  - All of them obey the constraint

$$q_x^2 + q_y^2 + q_z^2 + q_w^2 = 1$$

# Quaternion

- Composed of two elements
  - A unit axis of rotation scaled by the sine of the half angle of rotation
  - The cosine of the half angle

$$\begin{aligned} \mathbf{q} &= [\mathbf{q}_V \quad q_S] \\ &= [\mathbf{a} \sin \frac{\theta}{2} \quad \cos \frac{\theta}{2}] , \end{aligned}$$

- Can also be written as a four element vector

$$\mathbf{q} = [q_x \quad q_y \quad q_z \quad q_w] , \text{where}$$

$$q_x = q_{V_x} = a_x \sin \frac{\theta}{2},$$

$$q_y = q_{V_y} = a_y \sin \frac{\theta}{2},$$

$$q_z = q_{V_z} = a_z \sin \frac{\theta}{2},$$

$$q_w = q_S = \cos \frac{\theta}{2}.$$

# Operations

- Magnitude and vector addition work the same
- Adding two quaternions together is not the sum of two angles though
- Multiplication is the most important operations – represents one rotation followed by another

$$pq = [(p_s \mathbf{q}_V + q_s \mathbf{p}_V + \mathbf{p}_V \times \mathbf{q}_V) \quad (p_s q_s - \mathbf{p}_V \cdot \mathbf{q}_V)]$$

# Conjugate and Inverse

- The inverse of a quaternion  $q$ , denoted  $q^{-1}$ , has the following property

$$qq^{-1} = 0\mathbf{i} + 0\mathbf{j} + 0\mathbf{k} + 1$$

- To determine the inverse we need the conjugate

$$q^* = [-\mathbf{q}_V \quad q_S]$$

- The inverse then is

$$q^{-1} = \frac{q^*}{|q|^2}$$



# It's simple

- Remember our quaternions are all unit length this means

$$q^{-1} = q^* = [-\mathbf{q}_V \quad q_S]$$

- Other useful facts

$$(pq)^* = q^* p^*$$

$$(pq)^{-1} = q^{-1} p^{-1}$$

# Rotating with quaternions

- To rotate a vector  $\mathbf{v}$  by a quaternion  $q$ 
  1. We first write the vector in quaternion form
$$\mathbf{v} = [\mathbf{v} \ 0] = [v_x \ v_y \ v_z \ 0]$$
  2. Then multiply  $q$  times  $\mathbf{v}$  and by the inverse of  $q$

$$\mathbf{v}' = \text{rotate}(q, \mathbf{v}) = q\mathbf{v}q^{-1}$$

or

$$\mathbf{v}' = \text{rotate}(q, \mathbf{v}) = q\mathbf{v}q^*$$

3. Finally extract the vector out of  $\mathbf{v}' = [\mathbf{v}' \ 0]$

# Concatenation

- For matrices we followed the rule

$$\mathbf{R}_{net} = \mathbf{R}_1 \mathbf{R}_2 \mathbf{R}_3$$
$$\mathbf{v}' = \mathbf{v} \mathbf{R}_1 \mathbf{R}_2 \mathbf{R}_3 = \mathbf{v} \mathbf{R}_{net}$$

- We do something similar with quaternion rotation

$$q_{net} = q_3 q_2 q_1$$
$$\mathbf{v}' = q_3 q_2 q_1 \mathbf{v} q_1^{-1} q_2^{-1} q_3^{-1}$$
$$= q_{net} \mathbf{v} q_{net}^{-1}$$

# Quaternion to matrix

- If

$$\mathbf{q} = [\mathbf{q}_V \quad q_s] = [q_{V_x} \quad q_{V_y} \quad q_{V_z} \quad q_s] = [x \quad y \quad z \quad w]$$

- Then

$$R = \begin{bmatrix} 1 - 2y^2 - 2z^2 & 2xy + 2zw & 2xz - 2yw \\ 2xy - 2zw & 1 - 2x^2 - 2z^2 & 2yz + 2xw \\ 2xz + 2yw & 2yz - 2xw & 1 - 2x^2 - 2y^2 \end{bmatrix}$$

- Much harder to go the other way

# Some useful (normalized) quaternions

- $[w, x, y, z] = [\cos(\frac{a}{2}), \sin(\frac{a}{2})n_x, \sin(\frac{a}{2})n_y, \sin(\frac{a}{2})n_z]$
- Where  $\mathbf{a}$  is the angle of rotation and  $(n_x, n_y, n_z)$  is the axis of rotation

w	x	y	z	Description
1	0	0	0	Identity quaternion, no rotation
0	1	0	0	180° turn around X axis
0	0	1	0	180° turn around Y axis
0	0	0	1	180° turn around Z axis
sqrt(0.5)	sqrt(0.5)	0	0	90° rotation around X axis
sqrt(0.5)	0	sqrt(0.5)	0	90° rotation around Y axis
sqrt(0.5)	0	0	sqrt(0.5)	90° rotation around Z axis
sqrt(0.5)	-sqrt(0.5)	0	0	-90° rotation around X axis
sqrt(0.5)	0	-sqrt(0.5)	0	-90° rotation around Y axis
sqrt(0.5)	0	0	-sqrt(0.5)	-90° rotation around Z axis

# How to use Quaternions in Ogre

```
cbool BasicTutorial1::keyPressed(const KeyboardEvent& evt)
{
    switch (evt.keysym.sym)

    {
        case 119: //ASCII code for "w"

        {
            Quaternion q(Degree(-45), Vector3::UNIT_X); //rotate around the X axis 45 degrees.
            ogreNode->rotate(q);
        }
        break;
        case 115: //ASCII code for "s"

        {
            Quaternion q(Degree(45), Vector3::UNIT_X); //rotate around the X axis -45 degrees.
            ogreNode->rotate(q);
        }
        break;
        case 97: //ASCII code for "a"

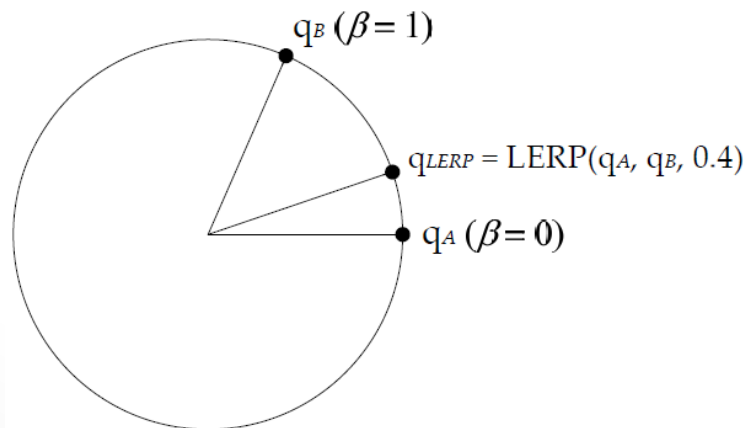
        {
            Quaternion q(Degree(-45), Vector3::UNIT_Y); //rotate around the Y axis -45 degrees.
            ogreNode->rotate(q);
        }
        break;
        case 100: //ASCII code for "d"

        {
            Quaternion q(Degree(45), Vector3::UNIT_Y); //rotate around the Y axis 45 degrees.
            ogreNode->rotate(q);
        }
        break;
    }
}
```

## Rotational Interpolation

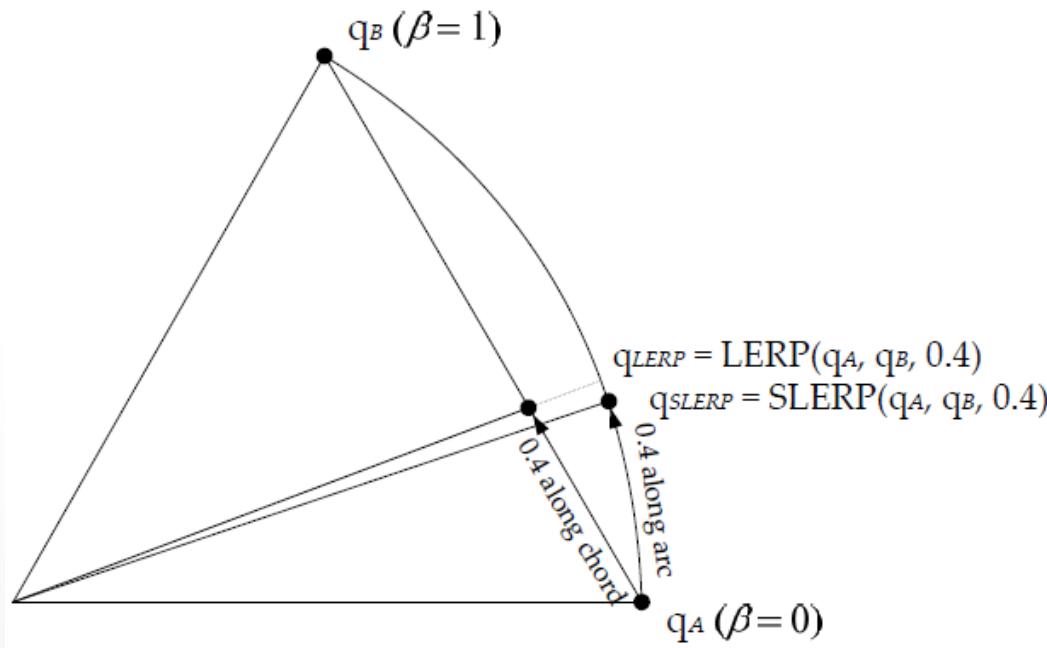
- Given two quaternions  $q_A$  and  $q_B$  we can find an intermediate rotation  $\beta$  percent between them as

$$q_{LERP} = LERP(q_A, q_B, \beta) = \frac{(1 - \beta)q_A + \beta q_B}{|(1 - \beta)q_A + \beta q_B|}$$
$$= \text{normalize} \left( \begin{bmatrix} (1 - \beta)q_{A_x} + \beta q_{B_x} \\ (1 - \beta)q_{A_y} + \beta q_{B_y} \\ (1 - \beta)q_{A_z} + \beta q_{B_z} \\ (1 - \beta)q_{A_w} + \beta q_{B_w} \end{bmatrix}^T \right)$$



# Problems with lerp

- Because LERP interpolate along a chord instead of the actual surface of the 4D hyper-sphere, the angular speed is not constant
  - The beginning and end are slower than the middle





# Slerp

- Spherical Linear Interpolation (SLERP) fixes this
- SLERP is like LERP, but the weighting of the two quaternions is modified based on the angle between them

$$\text{SLERP}(\mathbf{p}, \mathbf{q}, \beta) = w_p \mathbf{p} + w_q \mathbf{q},$$

where

$$w_p = \frac{\sin(1 - \beta)\theta}{\sin \theta},$$
$$w_q = \frac{\sin \beta\theta}{\sin \theta}.$$

# Find the angle

- To find the angle  $\theta$  we use the following

$$\cos(\theta) = \mathbf{p} \cdot \mathbf{q} = p_x q_x + p_y q_y + p_z q_z + p_w q_w$$

$$\theta = \cos^{-1}(\mathbf{p} \cdot \mathbf{q})$$

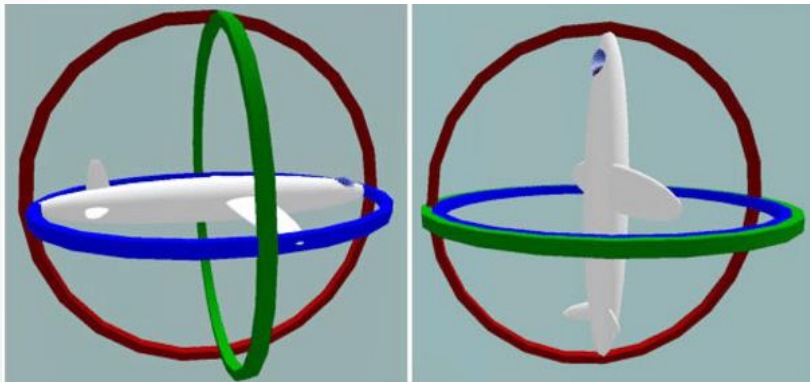
# Should you slerp?

- It depends
- If you can get it to be fast enough, then use it
- Otherwise LERP is fine and likely the user can't tell anyway

# Comparison of Rotational Representations

- Euler Angles
  - Simple: yaw, pitch and roll
  - represented by a 3D vector  $[q_Y \ q_P \ q_R]$ .
  - Easy to visualize
  - Easy to interpolate unless it is around an arbitrary axis
  - Prone to gimbal lock
  - Order of the rotations matters

**PYR  $\neq$  YPR  $\neq$  RYP**



# 3X3 Matrix

- Overall pretty good
- Not subject to gimbal lock
- Uses lots of computation and memory
- Not very intuitive to read

# Axis/Angle

- We can use a representation similar to quaternions called axis+angle  $[\mathbf{a} \ \theta]$
- Easy to understand
- Compact
- Cannot interpolate rotations
- Cannot be easily applied to points and vectors

# Quaternions

- Small
- Can be easily applied to vectors and points
- Easy to interpolate
- Hard to read

# SQT Transforms

- Quaternions can only represent rotations
- We can combine them with translation and scaling into a SQT transform
- Good for most things
  - Easy to understand
  - Easy to interpolate
  - Compact representation

$$SQT = [\mathbf{s} \quad \mathbf{q} \quad \mathbf{t}]$$



# Dual Quaternions

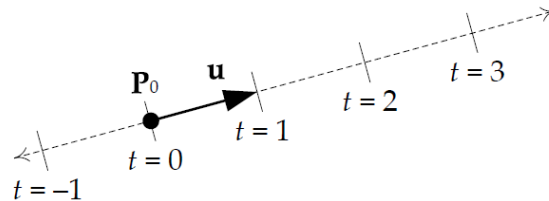
- A *rigid transformation* is a transformation involving a rotation and a translation—a “corkscrew” motion.
- A rigid transformation can be represented using a mathematical object known as a *dual quaternion*
- A dual quaternion is like an ordinary quaternion, except that its four components are *dual numbers* instead of regular real-valued numbers
- A dual number can be written as the sum of a non-dual part and a dual part as follows:  $a + \varepsilon b$ .
- Here  $\varepsilon$  is a magical number called the *dual unit*
- $\varepsilon^2 = 0$

# Rotations and Degrees of Freedom

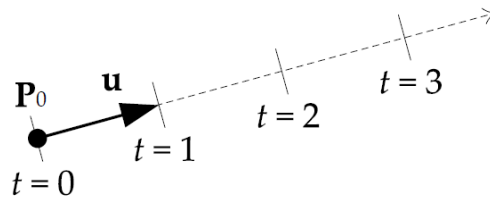
- DOF refers to the number of mutually independent ways in which an object's physical state (position and orientation) can change.
- a three-dimensional object has three degrees of freedom in its translation (along the x-, y- and z-axes) and three degrees of freedom in its rotation (about the x-, y- and z-axes) → 6 DOF
- For example, Euler angles require three floats, but axis+angle and quaternion representations use four floats, and a 3X3 matrix

# Other useful object

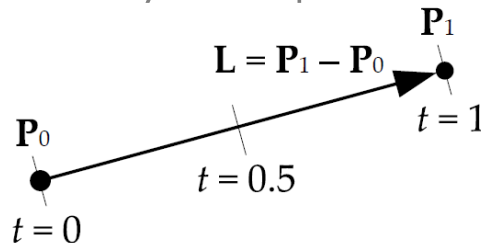
- Line – can be represented using a parametric equation  
$$P(t) = P_0 + t\mathbf{u} \quad \text{where} \quad -\infty < t < +\infty$$



- Ray – only extends in one direction so  $t \geq 0$

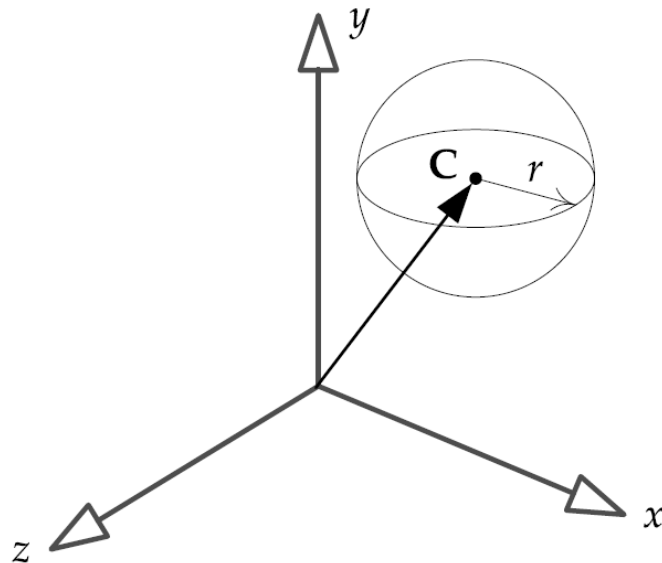


- Segment – bounded by two points  $0 \leq t \leq L$



# Spheres

- Defined using a center point  $C$  and a radius  $r$
- Fits nicely in a 4 element vector

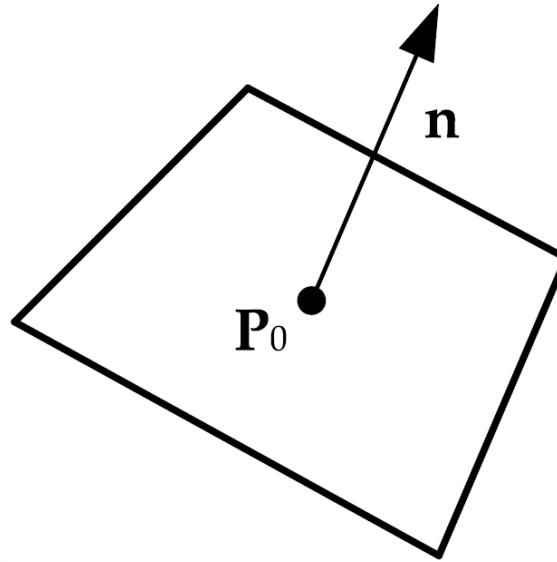


# Planes

- Plane is a 2D surface in 3D space
- Only satisfied when  $\mathbf{P}=[x \ y \ z]$  lies on the plane
$$Ax + By + Cz + D = 0$$

# Point-normal

- Planes can also be represented by
  - A point  $\mathbf{P}_0$
  - A unit vector  $\mathbf{n}$  that is normal to the plane



# Different, but the same

- If A,B,C from the traditional equation are interpreted as a vector
- The vector lies in the direction of the plane normal
- The normalized version  $[a \ b \ c] = \mathbf{n}$
- The normalized parameter  $d = D/\sqrt{A^2 + B^2 + C^2}$  is the distance to the origin

# AABB

- Axis aligned bounding box
- Useful for collision detection
- Represented using 6 values

$$[x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max}]$$

- Or two points  $\mathbf{P}_{min}$  and  $\mathbf{P}_{max}$
- Easy to test if a point is within the bounds

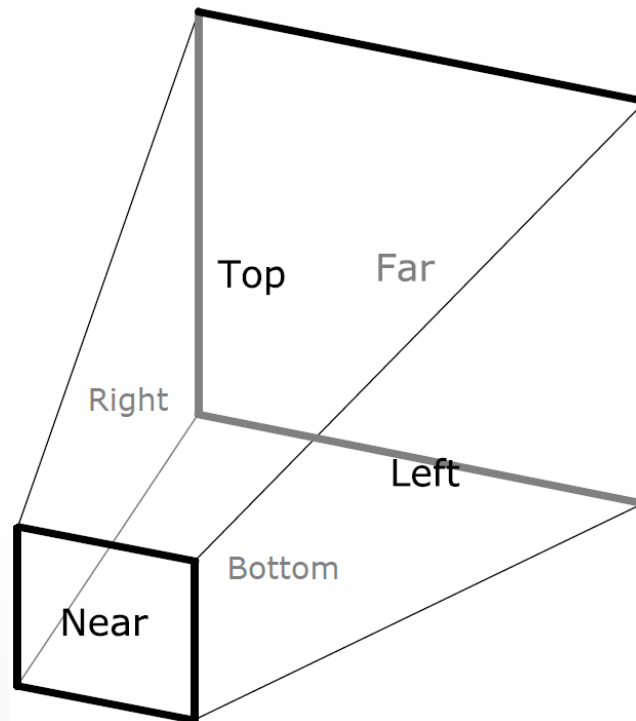


# oBB

- Oriented bounding box
- Differs from AABB because it aligns with the objects local coordinate space, not the world space
- Testing for intersection usually involves transforming the point into the OBB coordinate system

# Frusta

- A group of 6 planes that create a truncated pyramid shape
- Usually represented as an array of six planes in point-normal form

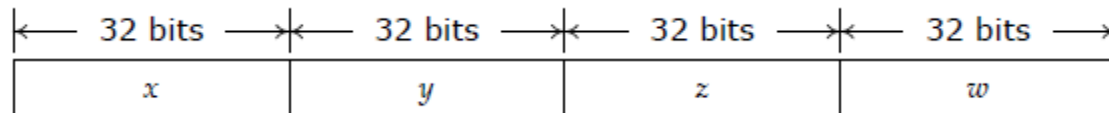


# Hardware-accelerated SIMD Math

- Stands for Single Instruction Multiple Data
- Allows a single operation to be performed on multiple data items in parallel
- For example you can multiply four pairs of floating point numbers with a single command

# History

- Intel introduced MMX instruction in the first Pentiums
- Could do SIMD on 8-eight bit, 4-16 bit or 2-32 bit integers
- Later enhancements resulted in the Streaming SIMD Extensions (SSE)
  - Uses special 128-bit registers
  - Can perform parallel operation on 32-bit floating point numbers



# SSE Registers

- In packed 32-bit floating point mode holds 4 -32 floating point numbers
- Elements are called  $[x \ y \ z \ w]$
- Consider the SIMD instruction

`addps xmm0, xmm1`

- This performs the following operations

`xmm0.x = xmm0.x + xmm1.x;`  
`xmm0.y = xmm0.y + xmm1.y;`  
`xmm0.z = xmm0.z + xmm1.z;`  
`xmm0.w = xmm0.w + xmm1.w;`

# Using SIMD

- Visual Studio has a special datatype `__m128`
  - Compiler understands how to use it

```
__m128 v= _mm_set_ps(-1.0f, 2.0f, 0.5f, 1.0f);
```

- GNU C/C++ uses the keyword `vector`

```
vector float = (vector float) (-1.0f, 2.0f, 0.5f, 1.0f);
```

# Coding with SSE

- Can use inline assembly

```
__m128 addWithAssembly(const __m128 a, const __m128 b)
{
    __asm addps xmm0, xmm1
}
```

- Can use intrinsics (# include <xmmintrin.h>)

```
__m128 addWithIntrinsics(const __m128 a, const __m128 b)
{
    return _mm_add_ps (a,b);
}
```

# Vector-matrix multiplication

- Say you want to multiply a 1X4 vector  $\mathbf{v}$  with a 4X4 matrix  $\mathbf{M}$  to get the solution  $\mathbf{r}$

$$\mathbf{r} = \mathbf{vM}$$

$$\begin{bmatrix} r_x & r_y & r_z & r_w \end{bmatrix} = \begin{bmatrix} v_x & v_y & v_z & v_w \end{bmatrix} \begin{bmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{bmatrix}$$

$$= \begin{bmatrix} v_x M_{11} + v_y M_{21} + v_z M_{31} + v_w M_{41} \\ v_x M_{12} + v_y M_{22} + v_z M_{32} + v_w M_{42} \\ v_x M_{13} + v_y M_{23} + v_z M_{33} + v_w M_{43} \\ v_x M_{14} + v_y M_{24} + v_z M_{34} + v_w M_{44} \end{bmatrix}$$



# Attempt 1

- We could store  $\mathbf{v}$  in an SSE and each column of  $\mathbf{M}$  in an SSE
- The multiple each  $\mathbf{M}$  SSE by the  $\mathbf{v}$  SSE
- We get

$$vMcol1 = [v_x M_{11} \quad v_y M_{21} \quad v_z M_{31} \quad v_w M_{41}];$$

$$vMcol2 = [v_x M_{12} \quad v_y M_{22} \quad v_z M_{32} \quad v_w M_{42}];$$

$$vMcol3 = [v_x M_{13} \quad v_y M_{23} \quad v_z M_{33} \quad v_w M_{43}];$$

$$vMcol4 = [v_x M_{14} \quad v_y M_{24} \quad v_z M_{34} \quad v_w M_{44}];$$

- Now we have to add across the vectors
  - Not very fast

# Attempt 2

- We could create SSEs with replicated values from  $\mathbf{v}$  and multiply each row of  $\mathbf{M}$  by the appropriate  $\mathbf{v}$  SSE
- We get
- Now we can add the vectors
  - Much better

# Random Number Generator

- Linear Congruent Generators
  - Found in the `c rand()` function
  - Not particular good
- Mersenne Twister
  - Very large period (before repeating a sequence)
  - Very high order of equidistribution
  - Passes numerous test for randomness
  - Fast
- Mother-of-All
  - Published by George Marsaglia
  - Reasonable period
  - Faster than Mersenne Twister

# What is Unity.Mathematics

- Shader/SIMD math library
- Implements vector and matrix types:
  - floatN, quaternion
  - Float3x3, float4x4
- And some elementary functions:
  - Min, max, fabs, etc
  - Sin, cos, sqrt, normalize, dot, cross, etc

# When to use Unity.Mathematics

- Data Oriented Technology Stack (DOTS)
  - Entity package
  - Unity.Physics package
  - Burst is aware of Unity.Mathematics
- Determinism (future)
  - Compute a floating number in different platforms
  - Arm or Intel CPU creates different floating point!
  - You can deterministically generate the same floating point across different platforms (Very important in multiplayer in networking).
- Performance (more on this later)

# When not to use Unity Mathematics

- Reliance on extremely specific behavior
- Using `UnityEngine.Mathf` functions with no clear equivalent
- Dependence on some libraries that expect `Vector3/Matrix4x4`

# How do I use Unity.Mathematics?

- Make sure it's installed?
- Window → Package Manager → mathematics
- In your code:
  - Using Unity.Mathematics

# Componentwise Vector Multiply

```
void ComponentwiseVectorMultiplyClassic()
{
    var v0 = new Vector4(2, 4, 6, 8);
    var v1 = new Vector4(1, -1, 1, -1);
    var result = Vector4.Scale(v0, v1);
    //result = new Vector4(2,-4,6,-8)
}
```

```
void ComponentwiseVectorMultiplyUnityMathematics()
{
    var v0 = new float4(2, 4, 6, 8);
    var v1 = new float4(1, -1, 1, -1);
    var result = v0 * v1;
    //result = new float4(2,-4,6,-8)
}
```



# Build 4X4 Matrices

```
void Build4x4Classic()  
{  
    //each vector creates a "column" in the matrix  
    var c0 = new Vector4(1, 0, 0, 0);  
    var c1 = new Vector4(0, 1, 0, 0);  
    var c2 = new Vector4(0, 0, 1, 0);  
    var c3 = new Vector4(0, 0, 0, 1);  
    var m = new Matrix4x4(c0, c1, c2, c3);  
}
```

```
void Build4x4UnityMathematics()  
{  
    var c0 = new float4(1, 0, 0, 0);  
    var c1 = new float4(0, 1, 0, 0);  
    var c2 = new float4(0, 0, 1, 0);  
    var c3 = new float4(0, 0, 0, 1);  
    var m = new float4x4(c0, c1, c2, c3);  
}
```

# Operator\*(4x4,4x4)

```
void OperatorMultiply4X4Classic()
{
    var c0 = new Vector4(1, 1, 1, 1);
    var c1 = new Vector4(1, 1, 1, 1);
    var c2 = new Vector4(1, 1, 1, 1);
    var c3 = new Vector4(1, 1, 1, 1);
    var Matrix4x4_Ones = new Matrix4x4(c0, c1, c2, c3);
    var d0 = new Vector4(0.5f, 0, 0, 0);
    var d1 = new Vector4(0, 0.5f, 0, 0);
    var d2 = new Vector4(0, 0, 0.5f, 0);
    var d3 = new Vector4(0, 0, 0, 0.5f);
    var m4x4_HalfIdentify = new Matrix4x4(d0, d1, d2, d3);
    Matrix4x4 result = Matrix4x4_Ones * m4x4_HalfIdentify;
    Debug.Log(result); //all elements are 0.5, a true matrix multiplication, element ij is a dot product of row i of the first
matrix and column j of the second matrix!
}

void OperatorMultiply4X4UnityMathematics()
{
    var c0 = new float4(1, 1, 1, 1);
    var c1 = new float4(1, 1, 1, 1);
    var c2 = new float4(1, 1, 1, 1);
    var c3 = new float4(1, 1, 1, 1);
    var f4x4_Ones = new float4x4(c0, c1, c2, c3);
    var d0 = new float4(0.5f, 0, 0, 0);
    var d1 = new float4(0, 0.5f, 0, 0);
    var d2 = new float4(0, 0, 0.5f, 0);
    var d3 = new float4(0, 0, 0, 0.5f);
    var f4x4_HalfIdentify = new float4x4(d0, d1, d2, d3);
    float4x4 result = f4x4_Ones * f4x4_HalfIdentify;
    Debug.Log(result); //the result is f4x4_HalfIdentify because each element is calculated by multiplying corresponding
components!
}
```

# Multiply4X4 And Vector4

```
void Multiply4x4AndVector4Classic()
{
    var c0 = new Vector4(1, 1, 1, 1);
    var c1 = new Vector4(1, 1, 1, 1);
    var c2 = new Vector4(1, 1, 1, 1);
    var c3 = new Vector4(1, 1, 1, 1);
    var m4x4 = new Matrix4x4(c0, c1, c2, c3);
    var v4 = new Vector4(0.5f, 0, 0, 0); //column vector!!!
    Vector4 result1 = m4x4 * v4; //4x4 * 4x1 = 4x1
    //Vector4 result2 = v4 * m4x4; //No operator*(Vector4, Matrix4x4)
}
```

```
void Multiply4x4AndVector4UnityMathematics()
{
    var c0 = new float4(1, 1, 1, 1);
    var c1 = new float4(1, 1, 1, 1);
    var c2 = new float4(1, 1, 1, 1);
    var c3 = new float4(1, 1, 1, 1);
    var f4x4 = new float4x4(c0, c1, c2, c3);
    var v4 = new float4(0.5f, 0, 0, 0); //row or column vector!!!
    float4 result1 = math.mul(f4x4, v4); //4x4 * 4x1 = 4x1
    float4 result2 = math.mul(v4, f4x4); //1x4 * 4x4 = 1x4
}
```

# Quaternions

[//https://quaternions.online/](https://quaternions.online/)

```
void QuaternionMultiplicationClassic()
{
    var axis = Vector3.up;
    var q = Quaternion.AngleAxis(45.0f, axis);
    var orientation = Quaternion.Euler(45.0f, 90.0f, 180.0f);
    var result = q * orientation;
}

void QuaternionMultiplicationUnityMathematics()
{
    var axis = new float3(0.0f, 1.0f, 0.0f);
    var q = quaternion.AxisAngle(axis, math.radians(45.0f));
    var orientation = quaternion.Euler(math.radians(45.0f),
    math.radians(90.0f), math.radians(180.0f));
    var result = math.mul(q, orientation);
}
```

# Generating Random Numbers

```
void RandomNumberClassic()
{
    //[0,1] inclusive
    float randomFloat1 = UnityEngine.Random.value;

    //[-5,5] inclusive
    float randomFloat2 = UnityEngine.Random.Range(-5.0f, 5.0f);
}

void RandomNumberUnityMathematics()
{
    //Choose some non-zero seed and set up the random number generator state.

    uint seed = 1;
    Unity.Mathematics.Random rng = new Unity.Mathematics.Random(seed);

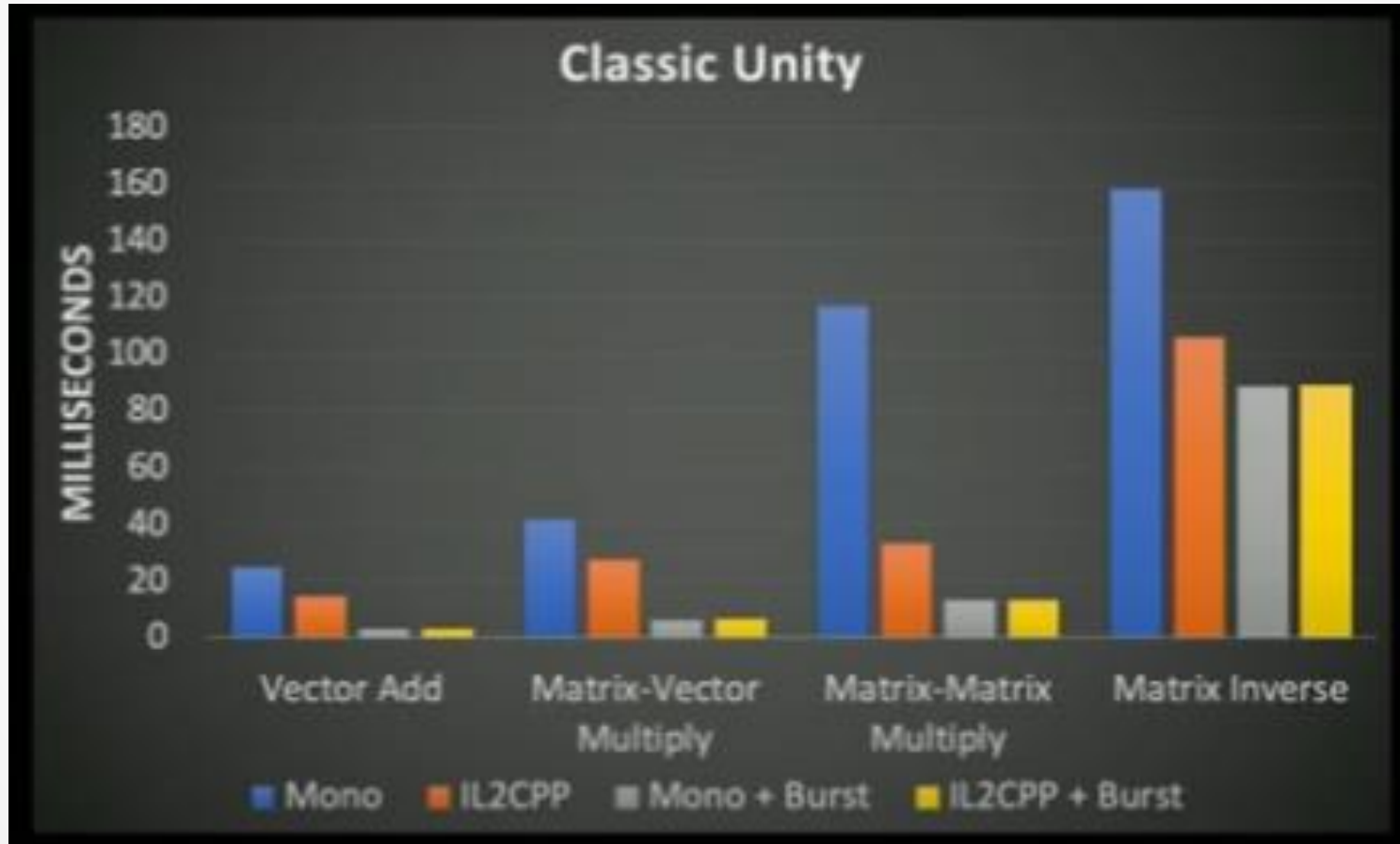
    //[0,1) exclusive
    float randomFloat1 = rng.NextFloat();

    //[-5,5) exclusive
    float randomFloat2 = rng.NextFloat(-5.0f, 5.0f);
}
```

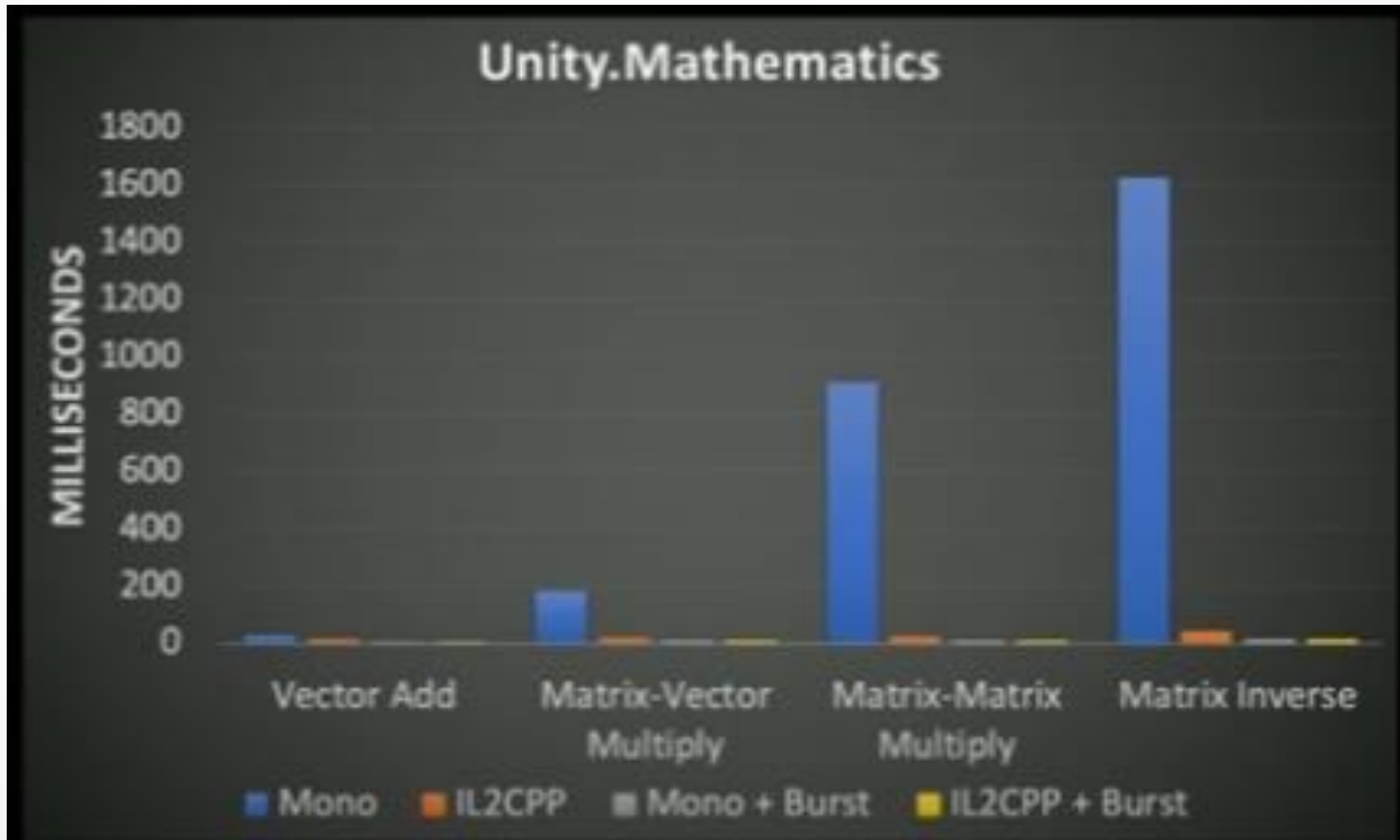
# Performance

- 1,000,000 of the following:
  - Vector 4 additions
  - Matrix4x4, Vector4 multiplies
  - Matrix4x4, Matrix4x4 multiples
  - Matrix4x4 inverse
- Mono, IL2CPP, Burst
- How does classic Unity fare?

# Performance: Classic Unity

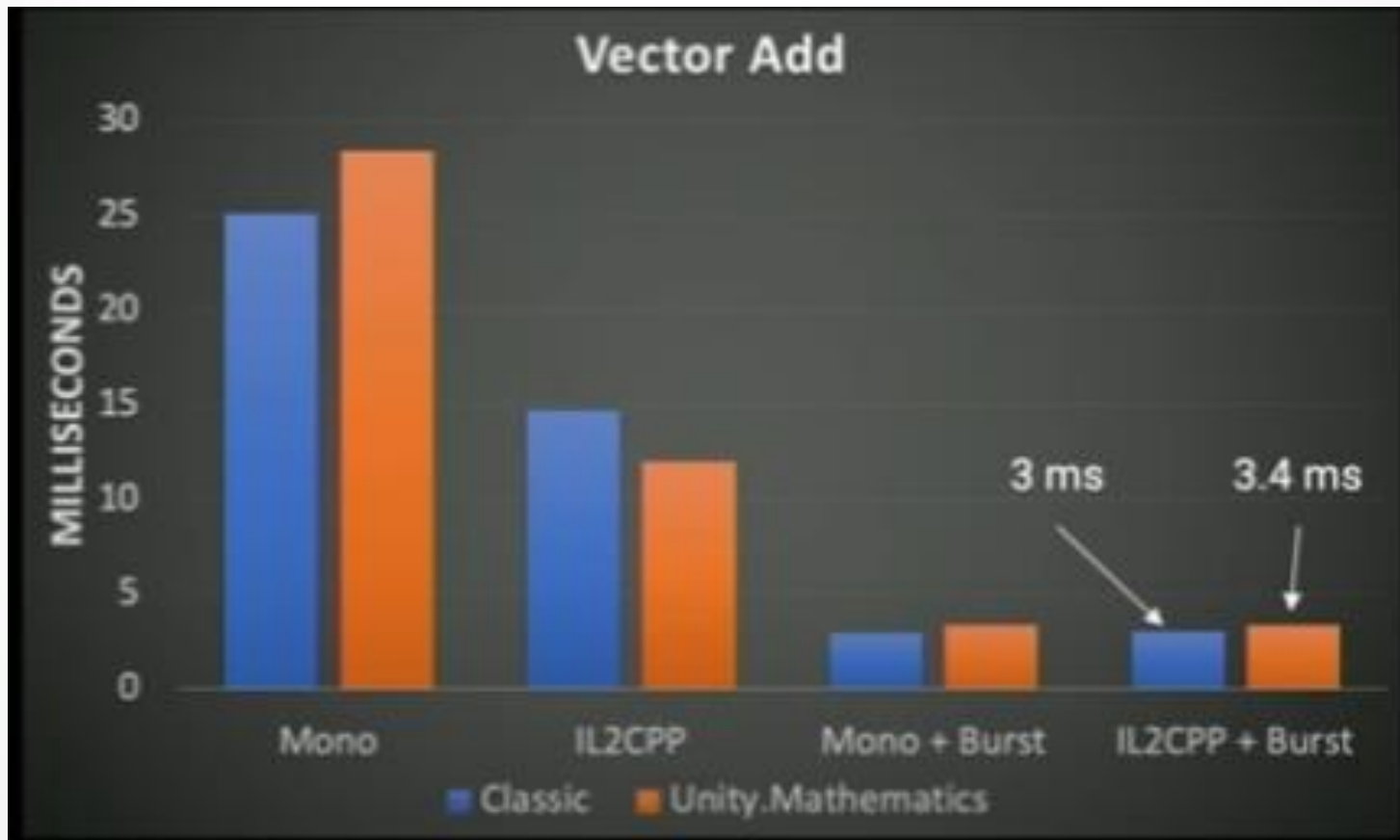


# Unity Mathematics

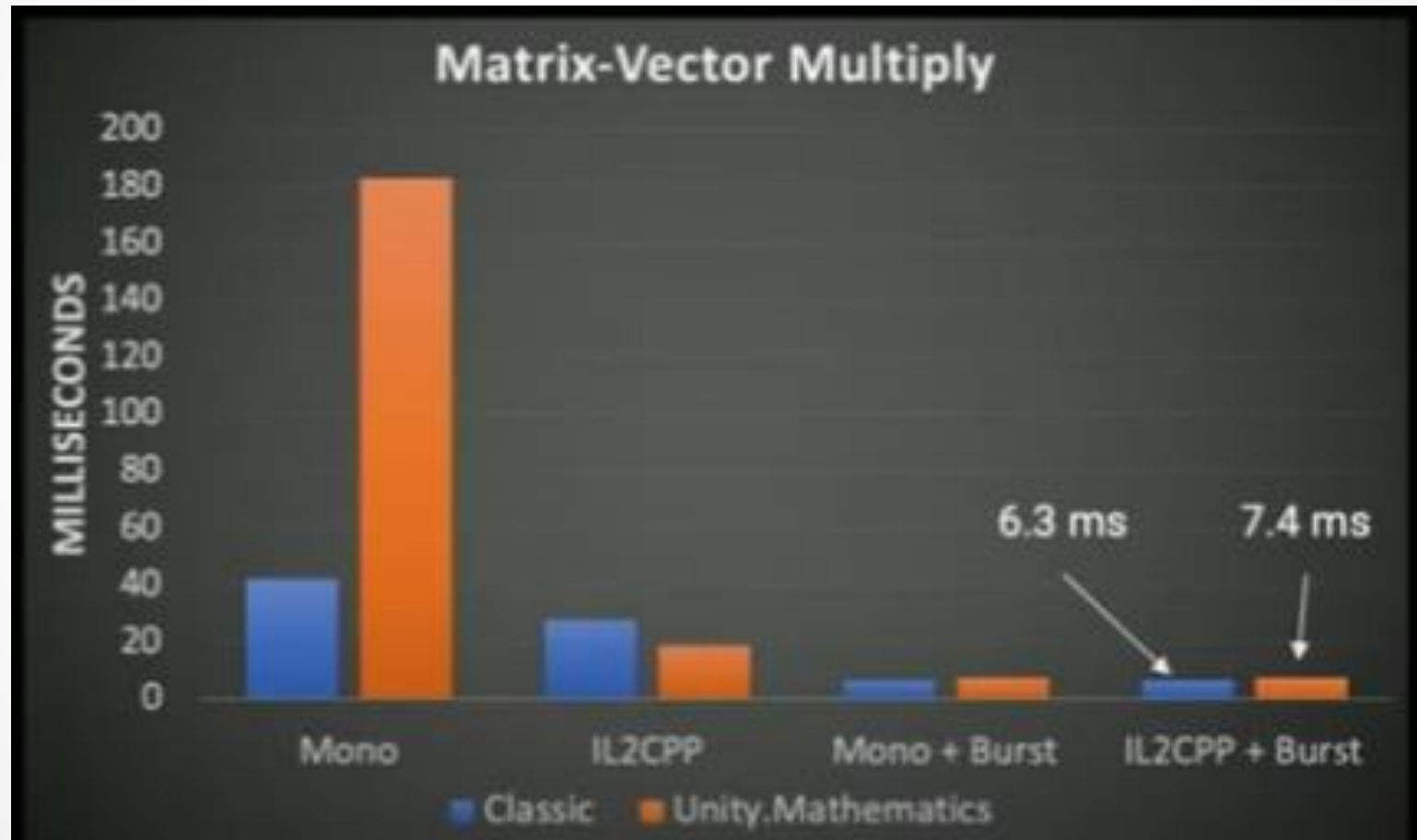




# Vector Add



# Matrix-Vector Multiply



# Matrix-Matrix Multiply



# Matrix Inverse

