PA199
Advanced Game Design

Lecture 3

Mathematics for Game Design

Dr. Fotis Liarokapis

9th March 2017

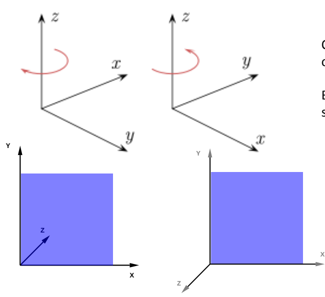
Basis and Coordinate Systems

- In any scene, we need a way to be able to position and orientate points, vectors, objects, etc:
  - We do this be defining a basis
- The basis is defined by an origin and a number of basis vectors
  - Can think of the basis as a ‘starting point’
- We employ a Cartesian basis

Basis and Coordinate Systems .

- The basis vectors are mutually orthogonal and unit length
  - Unit length:
    - Have a length of 1
  - Mutually Orthogonal:
    - Each vector is at a right angle to the others
- Basis vectors for 3 dimensions
  - Use ‘x’ and ‘y’ for 2 dimensions
  - Position in Cartesian coordinates specified by (x, y, z)

Left-handed vs. Right-Handed



OpenGL is right handed in object space and world space

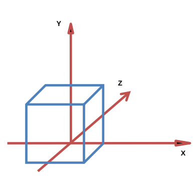
But in window space (screen space) left handed

left handed
Z axis goes "into" the page

right handed
Z axis goes "out" of the page

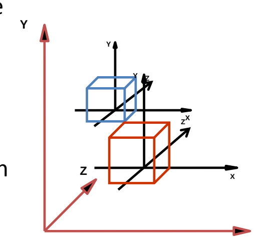
Local Coordinate System

- Preferred system for construction of object parts
- 3D Cartesian system
- Object vertices centered about the local origin



3D World Coordinate System

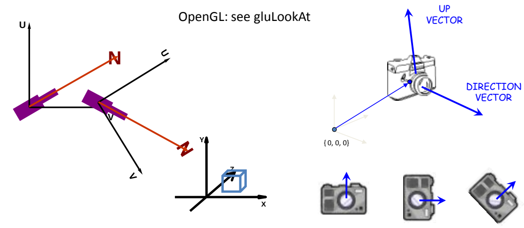
- 3D Cartesian coordinate system
- Arbitrary centre, handedness and orientation
- Used in the construction stage



Camera Coordinate System

- Used to define the view onto the 3D world that the user will see on the screen
- Centre (0,0,0) is located at the Imaginary User's eye
- Axes oriented such that:
  - One indicates the direction in which the user looks
  - The second indicates roughly the 'up' direction
  - The third indicates the handedness

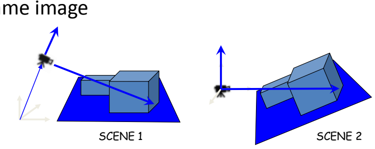
Example of Camera Coordinate System



https://www.opengl.org/sdk/docs/man2/xhtml/gluLookAt.xml

3D Camera Viewing

- Note that the following scenes should produce the same image



- In 'Scene 2', relative position of objects and camera remain constant but the actually scene has been changed
  - It has been transformed

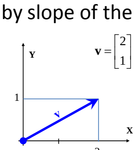
Notation: Scalars, Vectors, Matrices

- Scalar — a
  - Lower case, italic
- Vector — $\mathbf{a} = [a_1 \quad a_2 \quad \dots \quad a_n]$
  - Lower case, bold
- Matrix
  - Upper case, bold

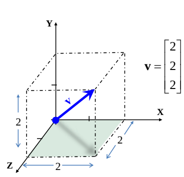
$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Vectors

- A quantity characterized by a **magnitude** and **direction**
  - Can be represented by an arrow, where **magnitude** is the length of the arrow and the **direction** is given by slope of the line

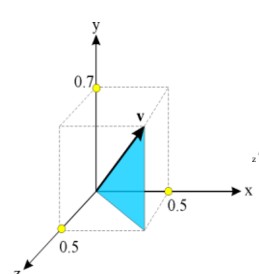


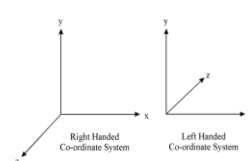
A vector in 2D



A vector in 3D

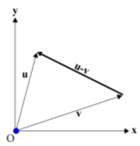
Vectors in 3D Co-ordinates



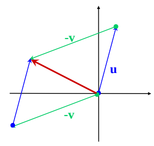


$$\mathbf{v} = \underbrace{\begin{bmatrix} 0.5 \\ 0.7 \\ 0.5 \end{bmatrix}}_{\text{RHS}} = \underbrace{\begin{bmatrix} 0.5 \\ 0.7 \\ -0.5 \end{bmatrix}}_{\text{LHS}}$$

Vector Subtraction

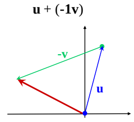


$$\underset{\mathbf{u}}{\begin{bmatrix}1\\4\end{bmatrix}}-\underset{\mathbf{v}}{\begin{bmatrix}5\\2\end{bmatrix}}=\underset{\mathbf{u+v}}{\begin{bmatrix}-4\\2\end{bmatrix}}$$



Can be seen as an addition of

u + (-1**v**)



Vector Magnitude

- The magnitude or "norm" of a vector of dimension n is given by the standard Euclidean distance metric:

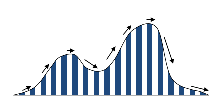
$$\|\mathbf{v}\| = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}$$

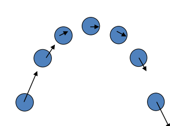
Vectors for Direction

- Vectors represent
  - Direction
  - Magnitude
- In games can be used for representing:
  - Position
  - Velocity
  - Forces / impulses

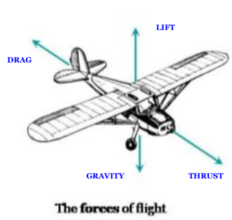
Example Vectors for Direction

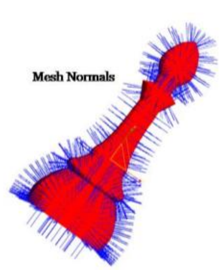
- Describing velocity (direction and speed) of roller coaster and ball at different points in time





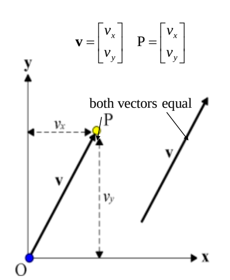
Other Vectors





Vertices and Points

- Vectors can however communicate a position
- Referred to as a point or vertex
- A vertex is actually represented by its displacement from the origin { 0, 0, 0 }



With the origin O, we can use this to represent a unique position in space

Unit Vectors

- Vectors of length 1 are often termed unit vectors
  - Normalised vectors
- When we only wish to describe direction we use normalised vectors
  - Often to avoid redundancy
- For this (and other reasons), we often need to normalise a vector: $\hat{\mathbf{v}} = \frac{\mathbf{v}}{\|\mathbf{v}\|} = \frac{1}{\sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}}\mathbf{v}$

Dot Product Definition

- Dot product (inner product) is defined as:

$$\mathbf{u} \cdot \mathbf{v} = \sum_i u_i v_i$$

$$\mathbf{u} \cdot \mathbf{v} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = u_1 v_1 + u_2 v_2$$

$$\mathbf{u} \cdot \mathbf{v} = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = u_1 v_1 + u_2 v_2 + u_3 v_3$$

Dot Product Magnitude

- Therefore we can redefine magnitude in terms of the dot-product operator:

$$\mathbf{u} \cdot \mathbf{u} = u_1^2 + u_2^2 + u_3^2 = \|\mathbf{u}\|^2 \Rightarrow \|\mathbf{u}\| = \sqrt{\mathbf{u} \cdot \mathbf{u}}$$

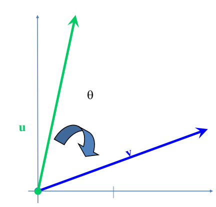
- Note that the dot product operator is **commutative** and **associative**
  - Changing the order of the operands does not change the result
  - (x * y) * z = x * (y * z)

Dot Product Using Angle

- The Dot Product can also be obtained from the following equation:

$$\mathbf{u} \cdot \mathbf{v} = \|\mathbf{u}\|\|\mathbf{v}\| \cos\theta$$

- where θ is the **angle** between the two vectors



Angle Between Two Vectors

- So, if we know the vectors u and v, then the dot product is useful for finding the angle between two vectors:

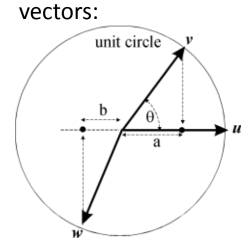
$$\mathbf{u} \cdot \mathbf{v} = \|\mathbf{u}\|\|\mathbf{v}\| \cos\theta \quad \Rightarrow \quad \theta = \cos^{-1}\left[\frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\|\|\mathbf{v}\|}\right]$$

- Note that if we had already normalised the vectors u and v then it would simply be:

$$\theta = \cos^{-1}[\hat{\mathbf{u}} \bullet \hat{\mathbf{v}}]$$

Dot Product Special Case

- If both vectors are normal, the dot product defines the cosine of the angle between the vectors:



$$\mathbf{u} \cdot \mathbf{v} = \cos\theta$$

But in general:

$$\mathbf{u} \cdot \mathbf{v} = \|\mathbf{u}\|\|\mathbf{v}\| \cos\theta$$

$$\Rightarrow \theta = \cos^{-1}\left[\frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\|\|\mathbf{v}\|}\right]$$

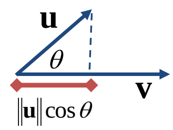
if θ > 90 then the dot product is negative

Projection Using Dot Product

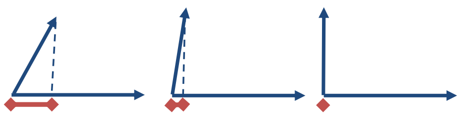
- Can find length of projection of u onto v

$$\mathbf{u}\bullet\mathbf{v}=\|\mathbf{u}\|\|\mathbf{v}\|\cos\theta$$

$$\|\mathbf{u}\|\cos\theta=\frac{\mathbf{u}\bullet\mathbf{v}}{\|\mathbf{v}\|}$$



- As lines become perpendicular:



Cross Product

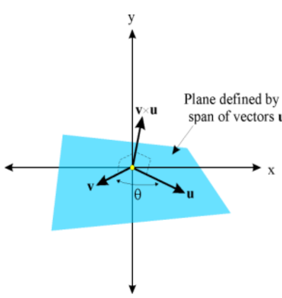
- Used for defining **orientation** and constructing **co-ordinate axes**
- Cross product defined as:

$$\mathbf{u}\times\mathbf{v}=\begin{bmatrix}u_1\\u_2\\u_3\end{bmatrix}\times\begin{bmatrix}v_1\\v_2\\v_3\end{bmatrix}=\begin{bmatrix}u_2v_3-u_3v_2\\u_3v_1-u_1v_3\\u_1v_2-u_2v_1\end{bmatrix}$$

- The result is a vector, perpendicular to the plane defined by **u** and **v**:

$$\mathbf{u}\times\mathbf{v}=\mathbf{w}\,\|\mathbf{u}\|\|\mathbf{v}\|\sin\theta$$

Cross Product Examples



Right Handed Coordinate System

Cross Product Properties

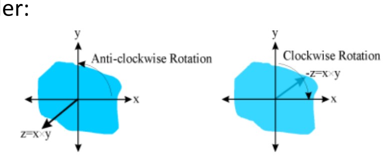
- Cross product is **anti-commutative**:

$$\mathbf{u}\times\mathbf{v}=-(\mathbf{v}\times\mathbf{u})$$

- It is **not associative**:

$$\mathbf{u}\times(\mathbf{v}\times\mathbf{w})\neq(\mathbf{u}\times\mathbf{v})\times\mathbf{w}$$

- Direction of resulting vector defined by operand order:



Vector Class

```
class TVector {
    public:
          double _x, _y, _z;

          // Constructors
          TVector(double x, double y, double z, TStatus s) : _x(x),    _y(y),
             _z(z), _Status(s) {}
          TVector(double x, double y, double z) : _x(x), _y(y),          _z(z),
             _Status(DEFAULT) {}

          // Functions here

};
#endif
```

Vector Subtraction Example

```
TVector &TVector::subtract(const TVector &v1, const TVector
    &v2, TVector &result)
{
    if (v1.isValid() && v2.isValid())
    {
          result._x = v1._x - v2._x;
          result._y = v1._y - v2._y;
          result._z = v1._z - v2._z;
          result._Status = DEFAULT;
    } else
          result = TVector();
    return result;
}
```

What is a Matrix?

- A matrix is a set of elements, organized into rows (m) and columns (n)

$$\begin{array}{cc} \text{Rows, } i = 1, \ldots, m & \left\{ \underbrace{\begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & & a_{2n} \\ \vdots & & & \ddots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{pmatrix}}_{\text{Columns, } j = 1, \ldots, n} \right. \end{array}$$

Why Use Matrices?

- Variety of engineering problems lead to the need to solve systems of linear equations

$$\boldsymbol{Ax} = \boldsymbol{b}$$

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & & a_{2n} \\ \vdots & & & \ddots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{pmatrix} \quad \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \quad \boldsymbol{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

matrix **column vectors**

Row and Column Matrices (vectors)

- Row matrix (or row vector) is a matrix with one row

$$\boldsymbol{r} = \begin{pmatrix} r_1 & r_2 & r_3 & \cdots & r_n \end{pmatrix}$$

- Column vector is a matrix with only one column

$$\boldsymbol{c} = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{pmatrix}$$

Square Matrix

- When the row and column dimensions of a matrix are equal (m = n) then the matrix is called **square**

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & & a_{2n} \\ \vdots & & \ddots & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

Matrix Transpose

- The transpose of the (m x n) matrix A is the (n x m) matrix formed by interchanging the rows and columns such that row i becomes column i of the **transposed** matrix

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & & a_{2n} \\ \vdots & & \ddots & \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad \boldsymbol{A}^T = \begin{pmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & & a_{m2} \\ \vdots & & \ddots & \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{pmatrix}$$

Matrix Equality

- Two (m x n) matrices A and B are **equal** if and only if each of their elements are equal
- That is when:
  - A = B
- If and only if:
  - $a_{ij} = b_{ij}$
  - For i = 1,…,m & j = 1,…,n

Matrix Addition General Format

$$\mathbf{A} + \mathbf{B} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & & b_{2n} \\ \vdots & & \ddots & \vdots \\ b_{m1} & b_{m2} & \dots & b_{mn} \end{pmatrix}$$
$$= \begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} & \cdots & a_{1n} + b_{1n} \\ a_{21} + b_{21} & a_{22} + b_{22} & & a_{2n} + b_{2n} \\ \vdots & & \ddots & \vdots \\ a_{m1} + b_{m1} & a_{m2} + b_{m2} & \dots & a_{mn} + b_{mn} \end{pmatrix}$$

Scalar Matrix Multiplication

- **Multiplication** of a matrix A by a **scalar** is defined as:

$$\alpha \boldsymbol{A} = \begin{pmatrix} \alpha a_{11} & \alpha a_{12} & \cdots & \alpha a_{1n} \\ \alpha a_{21} & \alpha a_{22} & & \alpha a_{2n} \\ \vdots & & \ddots & \\ \alpha a_{m1} & \alpha a_{m2} & \dots & \alpha a_{mn} \end{pmatrix}$$

Matrix Multiplication with Matrix General Format

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

$$\boldsymbol{C} = \boldsymbol{AB} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1p} \\ a_{21} & a_{22} & & a_{1p} \\ \vdots & & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mp} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & & b_{1n} \\ \vdots & & \ddots & \vdots \\ b_{p1} & b_{p2} & \dots & b_{pn} \end{pmatrix}$$
$$= \begin{pmatrix} a_{11}b_{11} + \cdots + a_{1p}b_{p1} & a_{11}b_{12} + \cdots + a_{1p}b_{p2} & \cdots & a_{11}b_{1n} + \cdots + a_{1p}b_{pn} \\ a_{21}b_{11} + \cdots + a_{2p}b_{p1} & a_{21}b_{12} + \cdots + a_{2p}b_{p2} & & a_{21}b_{1n} + \cdots + a_{2p}b_{pn} \\ \vdots & & \ddots & \vdots \\ a_{m1}b_{11} + \cdots + a_{mp}b_{p1} & a_{m1}b_{12} + \cdots + a_{mp}b_{p2} & \dots & a_{m1}b_{1n} + \cdots + a_{mp}b_{pn} \end{pmatrix}$$

Diagonal Matrices

- Simple diagonal Matrix

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & 0 & 0 & 0 \\ 0 & a_{22} & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & a_{nn} \end{pmatrix}$$

Identity Matrix

- The identity matrix has the property that if A is a square matrix, then:

$$\boldsymbol{IA} = \boldsymbol{AI} = \boldsymbol{A}$$

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

Matrix Inverse

- If A is an (n x n) square matrix and there is a matrix X with the property that:

$$\boldsymbol{AX} = \boldsymbol{I}$$

- X is defined to be the inverse of A and is denoted A^{-1}

$$\boldsymbol{AA}^{-1} = \boldsymbol{I} \qquad \boldsymbol{A}^{-1}\boldsymbol{A} = \boldsymbol{I}$$

Matrix Class

```
class TMatrix33
{
    public:
        double _Mx[3][3];

        // Constructors
        TMatrix33();
        TMatrix33(double Phi, double Theta, double Psi);
        TMatrix33(TVector& Axis,double Psi);

        // Functions here
};
#endif
```

Matrix Addition

```
TMatrix33 &TMatrix33::add(const TMatrix33 &m1, const
    TMatrix33 &m2, TMatrix33 &result) {
        result._Mx[0][0] = m1._Mx[0][0] + m2._Mx[0][0];
        result._Mx[0][1] = m1._Mx[0][1] + m2._Mx[0][1];
        result._Mx[0][2] = m1._Mx[0][2] + m2._Mx[0][2];
        result._Mx[1][0] = m1._Mx[1][0] + m2._Mx[1][0];
        result._Mx[1][1] = m1._Mx[1][1] + m2._Mx[1][1];
        result._Mx[1][2] = m1._Mx[1][2] + m2._Mx[1][2];
        result._Mx[2][0] = m1._Mx[2][0] + m2._Mx[2][0];
        result._Mx[2][1] = m1._Mx[2][1] + m2._Mx[2][1];
        result._Mx[2][2] = m1._Mx[2][2] + m2._Mx[2][2];
        return result;
}
```

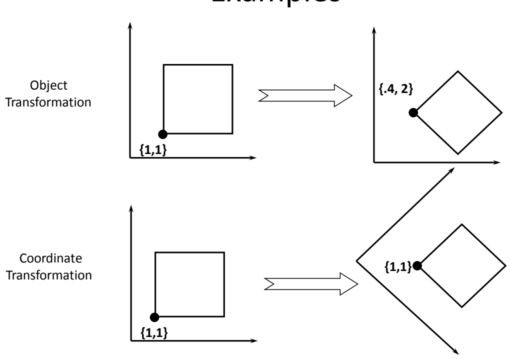
Transformations

- Allow us to move, orientate and change the primitives in our scene
  - Move, Rotate, Stretch, Squash, Shear
- Represented as matrices, such as:
  - We can store a translation and a rotation in a matrix
  - When we apply this matrix to an object, it will be translated and rotated as specified by the matrix
- Two ways of understanding a transformation:
  - Object Transformation
  - Coordinate Transformation

Object vs Coordinate Transformations

- Object Transformation
  - Alters the coordinates of each point according to some rule
  - The underlying coordinate system remains unchanged
- Coordinate Transformation
  - Produces a different coordinate system
  - Then represents all original points in this new system

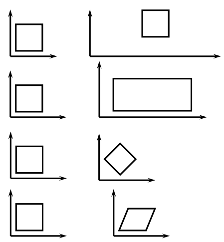
Examples



Affine Transformations Definition

- An affine transformation is any transformation that preserves:
  - Collinearity
    - i.e. All points lying on a line initially still lie on a line after transformation
  - Ratios of distances
    - i.e. The midpoint of a line segment remains the midpoint after transformation

Elementary Transformations



Homogeneous Coordinates

- Introduced in mathematics:
  - For projections and drawings
  - Used in artillery, architecture
  - Used to be classified material (in the 1850s)
- Add a third coordinate, *w*
- A 2D point is a 3 coordinates vector: $\begin{bmatrix} x \\ y \\ w \end{bmatrix}$

Homogeneous Coordinates .

- Two points are equal if and only if:
  - $x'/w' = x/w$ and $y'/w' = y/w$
- *w=0:* points at infinity
  - Useful for projections and curve drawing
- Homogenize = divide by *w*
- Homogenized points: $\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$

Translations with Homogeneous

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

$$\begin{cases} x' = x + wt_x \\ y' = y + wt_y \\ w' = \quad w \end{cases}$$

$$\begin{cases} \frac{x'}{w'} = \frac{x}{w} + t_x \\ \frac{y'}{w'} = \frac{y}{w} + t_y \end{cases}$$

Scaling with Homogeneous

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

$$\begin{cases} x' = s_x x \\ y' = s_y y \\ w' = w \end{cases}$$

$$\begin{cases} \frac{x'}{w'} = s_x \frac{x}{w} \\ \frac{y'}{w'} = s_y \frac{y}{w} \end{cases}$$

Rotation with Homogeneous

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

$$\begin{cases} x' = \cos\theta x - \sin\theta y \\ y' = \sin\theta x + \cos\theta y \\ w' = \qquad w \end{cases}$$

$$\begin{cases} \frac{x'}{w'} = \cos\theta \frac{x}{w} - \sin\theta \frac{y}{w} \\ \frac{y'}{w'} = \sin\theta \frac{x}{w} + \cos\theta \frac{y}{w} \end{cases}$$

Composition of Transformations

- To compose transformations, multiply the matrices:
  - Composition of a rotation and a translation:
    M = **RT**
- All transformations can be expressed as matrices
  - Even transformations that are not translations, rotations and scaling

Rotation Around a Point Q

- Rotation about a point Q:
  - translate Q to origin ($\mathbf{T}_Q$),
  - rotate about origin ($\mathbf{R}_\Theta$)
  - translate back to Q ($-\mathbf{T}_Q$).

$$\longrightarrow \quad P' = (-\mathbf{T}_Q)\mathbf{R}_\Theta \mathbf{T}_Q\, P$$

Beware!

- Matrix multiplication is *not* commutative
- The order of the transformations is vital
  - Rotation followed by translation is *very* different from translation followed by rotation
  - Careful with the order of the matrices!
- Small commutativity:
  - Rotation commute with rotation, translation with translation…

Matrices in OpenGL

- To initialise a matrix in OpenGL:
  - glLoadIdentity()
  - This clears the currently selected OpenGL matrix to the identity matrix
- To select a matrix as the current matrix:
  - glMatrixMode(mode)
    - GL_MODELVIEW, GL_PROJECTION, GL_TEXTURE

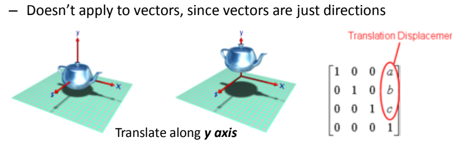
$$\begin{bmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{bmatrix}$$

How do we do this ?

- OpenGL does most of it for us !
  - OpenGL keeps a current matrix that allows us to orientate our primitives
    - This is known as the *model-view* matrix
  - All primitives placed are altered by the transformation stored in the model-view matrix
  - Model-view matrix acts as a state parameter; once set it remains until altered
    - Use calls such as *glTranslate()* to modify the current model-view matrix

Translation

- Think of translations as 'moving' without rotating
- Translation only applies to points
  - Doesn't apply to vectors, since vectors are just directions



- The translation displacement is written in the 12th, 13th, and 14th positions of our OpenGL matrix
- These correspond to the displacements in the x, y and z directions
  - So 12th position is the translation in the x direction

The Translation Matrix

- Example: Translate the point (x,y,z) by a displacement (a,b,c):
  - Gives us our translated point (x+a, y+b, z+c)
- glTranslate(dx, dy, dz)
  - Translates by a displacement (dx, dy, dz)
  - Calling glTranslate() concatenates the specified translation to the current model-view matrix
  - Any primitives drawn after this will be modified by the specified translation

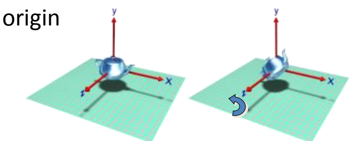
Translation Displacement

$$\begin{bmatrix}1&0&0&a\\0&1&0&b\\0&0&1&c\\0&0&0&1\end{bmatrix}\begin{bmatrix}x\\y\\z\\1\end{bmatrix}=\begin{bmatrix}x+a\\y+b\\z+c\\1\end{bmatrix}$$

Point to be translated

Rotation

- Change the orientation of a primitive, without affecting its position
- Rotation applies to both points and vectors
  - Rotating a vector will change its direction
- Rotations are conducted anti-clockwise about the origin

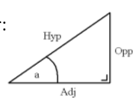


rotation of 45° about the Z axis

Rotation

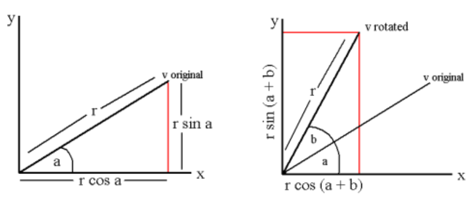
- Remember:

Sin a = Opp / Hyp
Cos a = Adj / Hyp
Tan a = Opp / Adj



$$\text{original } v = \begin{bmatrix} r\cos a \\ r\sin a\end{bmatrix}$$

$$\text{rotated } v = \begin{bmatrix} r\cos(a+b) \\ r\sin(a+b)\end{bmatrix}$$



Rotation

- Derivation:
- Expanding (a + b) from log tables:
  - Rotated x = r cos a cos b – r sin a sin b
  - Rotated y = r cos a sin b + r sin a cos b
- But:
  - Original x = r cos a
  - Original y = r sin a
- So:
  - Rotated x = original x cos b – original y sin b
  - Rotated y = original x sin b + original y cos b
- Elements 0, 1, 2, 4, 5, 6, 8, 9, 10 define any rotations in our transformation matrix

Rotation

$$\begin{bmatrix}0&4&8&12\\1&5&9&13\\2&6&10&14\\3&7&11&15\end{bmatrix}$$

The Rotation Matrix

- Rotations around the: x-axis (Rx), y-axis (Ry) and z-axis (Rz)

$$R_x=\begin{bmatrix}1&0&0&0\\0&\cos\theta&-\sin\theta&0\\0&\sin\theta&\cos\theta&0\\0&0&0&1\end{bmatrix}\quad R_y=\begin{bmatrix}\cos\theta&0&\sin\theta&0\\0&1&0&0\\-\sin\theta&0&\cos\theta&0\\0&0&0&1\end{bmatrix}\quad R_z=\begin{bmatrix}\cos\theta&-\sin\theta&0&0\\\sin\theta&\cos\theta&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}$$

- glRotatef(angle, vx, vy, vz)
  - Rotates around the axis (vx, vy, vz) by *angle* degrees
  - Calling glRotate() concatenates the specified rotate to the current model-view matrix
  - Any primitives drawn after this will be modified by the specified rotation

Scaling

- Allows us to make primitives larger and smaller, without changing the vertex positions of the original

Original   scale all axes   scale Y axis

$$\begin{bmatrix}s_1&0&0\\0&s_2&0\\0&0&s_3\end{bmatrix}\begin{bmatrix}x\\y\\z\end{bmatrix}=\begin{bmatrix}s_1x\\s_2y\\s_3z\end{bmatrix}$$

- Elements 1, 6, 11 define scales in our transformation matrix:
- glScalef(sx, sy, sz)
  - Scale a scene by sx in the y axis, sy in the y axis and sz in the z axis
  - The default value for sx,sy,sz is (1.0,1.0,1.0), which doesn't scale a scene at all
  - Any primitives drawn after this will be modified by the specified scaling

OpenGL Perspective Projection

- The call glFrustum(l, r, b, t, n, f) generates R, where:

$$R = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \text{ and } R^{-1} = \begin{bmatrix} \frac{r-l}{2n} & 0 & 0 & \frac{r+l}{2n} \\ 0 & \frac{t-b}{2n} & 0 & \frac{t+b}{2n} \\ 0 & 0 & 0 & -1 \\ 0 & 0 & \frac{-(f-n)}{2fn} & \frac{f+n}{2fn} \end{bmatrix}$$

- **R** is defined as long as *l* x *r*, *t* x *b*, and *n* x *f*

OpenGL Orthographic Projection

- The call glOrtho(l, r, b, t, n, f) generates R, where:

$$R = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & \frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & \frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & \frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ and } R^{-1} = \begin{bmatrix} \frac{r-l}{2} & 0 & 0 & \frac{r+l}{2} \\ 0 & \frac{t-b}{2} & 0 & \frac{t+b}{2} \\ 0 & 0 & \frac{f-n}{-2} & \frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- R is defined as long as *l* x *r*, *t* x *b*, and *n* x *f*

2D and 3D Lines

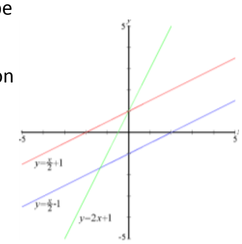
- In 2D, two different lines can either be
  - Parallel, meaning they never meet
  - May intersect at one and only one point
- In 3D (or more dimensions), lines may also be skew (meaning they don't meet) but also don't define a plane
- Two distinct planes intersect in at most one line
- Three or more points that lie on the same line are called collinear

2D Line Equation

- Lines in a Cartesian plane can be described algebraically by
  - Linear equations and functions
- In 2D the characteristic equation is often given by the slope-intercept form:

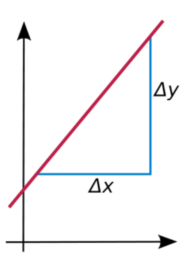
  y = mx + b

- where:
  - m is the slope of the line
  - b is the y-intercept of the line
  - x is the independent variable of the function y



Slope Definition

- Slope is often used to describe the measurement of the steepness, incline, gradient, or grade of a straight line
  - A higher slope value indicates a steeper incline
- The slope is defined as the <u>ratio</u> of the <u>altitude change</u> to the <u>horizontal distance</u> between any <u>two points</u> on the line
  - Using calculus, one can calculate the slope of the tangent to a curve at a point



Slope Calculation

- Slope is defined as the change in the y coordinate divided by the corresponding change in the x coordinate, between two distinct points on the line

$$m = \frac{\Delta y}{\Delta x}$$

- Given two points (x_1, y_1) and (x_2, y_2), the change in x from one to the other is $x_2 - x_1$, while the change in y is $y_2 - y_1$

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

Slope Special Cases

- The larger the absolute value of a slope, the steeper the line
  - A horizontal line has slope 0
    - Note that a vertical line's slope is undefined
  - A 45° rising line has a slope of +1
  - A 45° falling line has a slope of -1
- The angle **θ** a line makes with the positive **x** axis is closely related to the slope **m** via the tangent function:

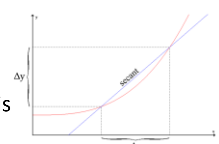
$$m = \tan\theta \qquad \theta = \arctan m$$

Slope Special Cases .

- Two lines are parallel if and only if
  - Their slopes are equal and they are not coincident or if
  - They both are vertical and therefore have undefined slopes
- Two lines are perpendicular if and only if
  - The product of their slopes is -1 or
  - One has a slope of 0 (a horizontal line) and the other has an undefined slope (a vertical line)

Derivative

- By moving the two points closer together Δy and Δx decreases
  - The line more closely approximates a tangent line to the curve
  - The slope of the secant approaches that of the tangent
- If y is dependent on x, then it is sufficient to take the limit where only Δx approaches zero
- Therefore, the slope of the tangent is the limit of Δy/Δx as Δx approaches zero



$$m = \frac{\Delta y}{\Delta x}$$

Differentiation and the Derivative

- **Differentiation** is a method to compute the rate at which a quantity, **y**, changes with respect to the change in another quantity, **x**, upon which it is **dependent**
- This **rate of change** is called the **derivative** of **y** with respect to **x**
- In more precise language, the dependency of y on x means that y is a function of x
- If x and y are real numbers, and if the graph of y is plotted against x, the **derivative** measures the **slope** of this graph at each point

Differentiation and the Derivative .

- This functional relationship is often denoted y = f(x), where **f** denotes the function
- The simplest case is when y is a linear function of x, meaning that the graph of y against x is a straight line
- In this case, y = f(x) = m x + c, for real numbers m and c, and the slope m is given by

$$m = \frac{\text{change in } y}{\text{change in } x} = \frac{\Delta y}{\Delta x}$$

- where the symbol Δ is an abbreviation for 'change in'

Differentiation and the Derivative ..

- It follows that Δy = m Δx
  - This gives an exact value for the slope of a straight line
- If the function f is not linear, then the change in y divided by the change in x varies
  - Differentiation is a method to find an exact value for this rate of change at any given value of x
- In Leibniz's notation, such an infinitesimal change in x is denoted by dx, and the derivative of y with respect to x is written:

$$\frac{dy}{dx}$$

3D Line Parametric Equations

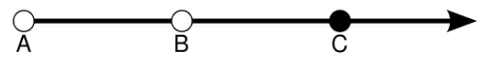
- In 3D a line is often described by parametric equations:

$$x = x_0 + at$$
$$y = y_0 + bt$$
$$z = z_0 + ct$$

- where:
  - x, y, and z are all functions of the independent variable t
  - x_0, y_0, and z_0 are the initial values of each respective variable
  - a, b, and c are related to the slope of the line, such that the vector (a, b, c) is a parallel to the line

Ray

- In Euclidean geometry, a ray, or half-line, given two distinct points A (the origin) and B on the ray, is the set of points C on the line containing points A and B such that A is not strictly between C and B
- In geometry, a ray starts at one point, then goes on forever in one direction

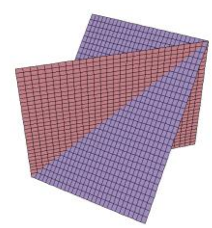


Example Class Tray.h

```
class TRay
{
    private:
        TVector _P; // Any point on the line
        TVector _V; // Direction of the line

    public:
        // Constructor
        TRay() {}

        // Line betwen two points OR point and a direction
        TRay(const TVector &point1, const TVector &point2);

        // Adjacent points on both lines
        bool adjacentPoints(const TRay &ray, TVector &point1, TVector &point2) const;

        // Distances
        double dist(const TRay &ray) const;
        double dist(const TVector &point) const;

        // More functions here
};
#endif
```

Plane Definition

- A plane can be uniquely determined by any of the following (sets of) objects:
  - Three non-collinear points
    - i.e. not lying on the same line
  - A line and a point not on the line
  - Two lines with one point of intersection
  - Two parallel lines



Plane Properties

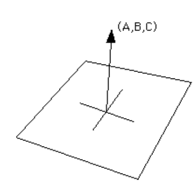
- Two planes are either parallel or they intersect in a line
- A line is either parallel to a plane or intersects it at a single point or is contained in the plane
- Two lines normal (perpendicular) to the same plane must be parallel to each other
- Two planes normal to the same line must be parallel to each other

Standard Plane Equation

- The standard equation of a plane in 3 space is:

$$Ax + By + Cz + D = 0$$

- The normal to the plane is the vector (A,B,C)



Plane Definition with a Point and a Normal Vector

- In a 3D space, another important way of defining a plane is by specifying a point and a normal vector to the plane
- Let **p** be the point we wish to lie in the plane, and let **n** be a nonzero normal vector to the plane
- The desired plane is the set of all points **r** such that:

$$\vec{n} \cdot (\vec{r} - \vec{p}) = 0$$

Plane Definition with a Point and a Normal Vector .

- If we write

$$\vec{n} = \begin{bmatrix} a \\ b \\ c \end{bmatrix} \qquad \mathbf{r} = (x, y, z)$$

- and d as the dot product

$$\vec{n} \cdot \mathbf{p} = -d$$

- then the plane Π is determined by the condition

$$ax + by + cz + d = 0$$

- where a, b, c and d are real numbers and a, b, and c are not all zero

Define a Plane using three Points

- The plane passing through three points $p_1=(x_1, y_1, z_1)$, $p_2=(x_2, y_2, z_2)$ and $p_3=(x_3, y_3, z_3)$ can be defined as the set of all points (x, y, z) that satisfy the following determinant equations:

$$\begin{vmatrix} x - x_1 & y - y_1 & z - z_1 \\ x_2 - x_1 & y_2 - y_1 & z_2 - z_1 \\ x_3 - x_1 & y_3 - y_1 & z_3 - z_1 \end{vmatrix} = \begin{vmatrix} x - x_1 & y - y_1 & z - z_1 \\ x - x_2 & y - y_2 & z - z_2 \\ x - x_3 & y - y_3 & z - z_3 \end{vmatrix} = 0$$

Determinant

- A determinant is a function depending on n that associates a scalar, det(A), to every n×n square matrix A
- The determinant of a matrix A is also sometimes denoted by |A|

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \qquad |A| = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix}$$

Dihedral Angle

- Given two intersecting planes described by

$$\Pi_1 : a_1 x + b_1 y + c_1 z + d_1 = 0$$

and

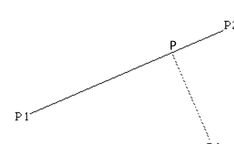
$$\Pi_2 : a_2 x + b_2 y + c_2 z + d_2 = 0$$

- the dihedral angle between them is defined to be the angle α between their normal directions

$$\cos\alpha = \hat{n}_1 \cdot \hat{n}_2 = \frac{a_1 a_2 + b_1 b_2 + c_1 c_2}{\sqrt{a_1^2 + b_1^2 + c_1^2}\sqrt{a_2^2 + b_2^2 + c_2^2}}.$$

Minimum Distance between a Point and a Line

- Find the shortest distance from a point to a line or line segment
- The equation of a line defined through two points P_1 (x_1, y_1) and P_2 (x_2, y_2) is:



$$P = P_1 + u\,(P_2 - P_1)$$

Minimum Distance between a Point and a Line .

- The point P_3 (x_3, y_3) is closest to the line at the tangent to the line which passes through P3, that is, the dot product of the tangent and line is equal to zero, thus:

$$(P_3 - P) \bullet (P_2 - P_1) = 0$$

- Substituting the equation of the line gives:

$$[P_3 - P_1 - u(P_2 - P_1)] \bullet (P_2 - P_1) = 0$$

Minimum Distance between a Point and a Line ..

- Solving this gives the value of u

$$u = \frac{(x3 - x1)(x2 - x1) + (y3 - y1)(y2 - y1)}{\|p2 - p1\|^2}$$

- Substituting this into the equation of the line gives the point of intersection (x, y) of the tangent as

$$x = x_1 + u\,(x_2 - x_1)$$
$$y = y_1 + u\,(y_2 - y_1)$$

- The distance therefore between the point P3 and the line is the distance between (x, y) above and P3

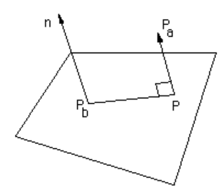
Minimum Distance between a Point and a Line ...

- Notes
  - The only special testing for a software implementation is to ensure that P_1 and P_2 are not coincident (denominator in the equation for u is 0)
  - If the distance of the point to a line segment is required then it is only necessary to test that u lies between 0 and 1
  - The solution is similar in higher dimensions

Minimum Distance between a Point and a Plane

- Let $P_a = (x_a, y_a, z_a)$ be the point in question
- A plane can be defined by its normal n = (A, B, C) and any point on the plane $P_b = (x_b, y_b, z_b)$
- Any point P = (x, y, z) lies on the plane if it satisfies the following

$$A\,x + B\,y + C\,z + D = 0$$



Minimum Distance between a Point and a Plane .

- Consider the projection of the line $(P_a - P_b)$ onto the normal of the plane n, that is just $||P_a - P_b||\cos\theta$
  - Where θ is the angle between $(P_a - P_b)$ and the normal n
- This projection is the minimum distance (D) of P_a to the plane and can be written in terms of the dot product:

$$D = (P_a - P_b) \bullet n \,/\, ||n||$$

- That is: $D = (A\,(x_a - x_b) + B\,(y_a - y_b) + C\,(z_a - z_b)) \,/\, \mathrm{sqrt}(A^2 + B^2 + C^2)$

Minimum Distance between a Point and a Plane ..

- Since point (x_b, y_b, z_b) is a point on the plane

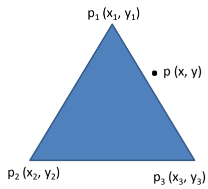
$$Ax_b + By_b + Cz_b + D = 0$$

- Substituting gives:

$$(Ax_a + By_a + Cz_a + D) \,/\, \mathrm{sqrt}(A^2 + B^2 + C^2)$$

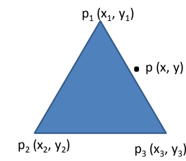
Point inside a Triangle

- Consider a triangle T defined by 3 points $p_1(x_1, y_1)$, $p_2(x_2, y_2)$, $p_3(x_3, y_3)$ and a single point p(x, y)
- Four solutions:
  - Barycentric solution
  - Parametric solution
  - Dot product solution
  - Cross product solution



Barycentric Solution

- Barycentric coordinate allows to express new p coordinates as a linear combination of p_1, p_2, p_3
- More precisely, it defines 3 scalars a, b, c such that :
  - $x = a * x_1 + b * x_2 + c * x_3$
  - $y = a * y_1 + b * y_2 + c * y_3$
  - $a + b + c = 1$



http://totologic.blogspot.fr/2014/01/accurate-point-in-triangle-test.html

Barycentric Solution .

- The way to compute a, b, c is not difficult :
  - $a = ((y_2 - y_3)*(x - x_3) + (x_3 - x_2)*(y - y_3)) / ((y_2 - y_3)*(x_1 - x_3) + (x_3 - x_2)*(y_1 - y_3))$
  - $b = ((y_3 - y_1)*(x - x_3) + (x_1 - x_3)*(y - y_3)) / ((y_2 - y_3)*(x_1 - x_3) + (x_3 - x_2)*(y_1 - y_3))$
  - $c = 1 - a - b$
- Then p lies in T if and only if:
  - $0 <= a <= 1$ and $0 <= b <= 1$ and $0 <= c <= 1$

http://totologic.blogspot.fr/2014/01/accurate-point-in-triangle-test.html

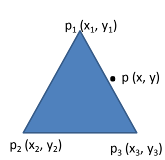
Barycentric Code Sample

```
function pointInTriangle(x1, y1, x2, y2, x3, y3, x, y: Number):
    Boolean
{
    var denominator:Number = ((y2 - y3)*(x1 - x3) + (x3 - x2)*(y1 - y3));
    var a:Number = ((y2 - y3)*(x - x3) + (x3 - x2)*(y - y3)) / denominator;
    var b:Number = ((y3 - y1)*(x - x3) + (x1 - x3)*(y - y3)) / denominator;
    var c:Number = 1 - a - b;

    return 0 <= a && a <= 1 && 0 <= b && b <= 1 && 0 <= c && c
    <= 1;
}
```

http://totologic.blogspot.fr/2014/01/accurate-point-in-triangle-test.html

Parametric Solution

- Consider the parametric expressions of the 2 edges $[p_1, p_2]$ and $[p_1, p_3]$ in T :
  - $x(t1) = t1*(x_2 - x_1)$
  - $y(t1) = t1*(y_2 - y_1)$
  - $x(t2) = t2*(x_3 - x_1)$
  - $y(t2) = t2*(y_3 - y_1)$
- Then express p(x, y) as a linear combination of them:
  - $x = x_1 + x(t1) + x(t2)$
  - $y = y_1 + y(t1) + y(t2)$



http://totologic.blogspot.fr/2014/01/accurate-point-in-triangle-test.html

Parametric Solution .

- Solving the system:
  - $t1 = (x*(y_3 - y_1) + y*(x_1 - x_3) - x_1*y_3 + y_1*x_3) / (x_1*(y_2 - y_3) + y_1*(x_3 - x_2) + x_2*y_3 - y_2*x_3)$
  - $t2 = (x*(y_2 - y_1) + y*(x_1 - x_2) - x_1*y_2 + y_1*x_2) / -(x_1*(y_2 - y_3) + y_1*(x_3 - x_2) + x_2*y_3 - y_2*x_3)$
- Then p lies in T if and only if:
  - $0 <= t1 <= 1$ and $0 <= t2 <= 1$ and $t1 + t2 <= 1$

http://totologic.blogspot.fr/2014/01/accurate-point-in-triangle-test.html

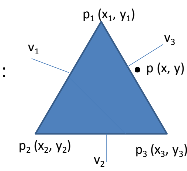
Parametric Code Sample

```
function pointInTriangle(x1, y1, x2, y2, x3, y3, x, y: Number):
    Boolean
{
    var denominator:Number = (x1*(y2 - y3) + y1*(x3 - x2) + x2*y3
      - y2*x3);
    var t1:Number = (x*(y3 - y1) + y*(x1 - x3) - x1*y3 + y1*x3) /
      denominator;
    var t2:Number = (x*(y2 - y1) + y*(x1 - x2) - x1*y2 + y1*x2) / -
      denominator;
    var s:Number = t1 + t2;

    return 0 <= t1 && t1 <= 1 && 0 <= t2 && t2 <= 1 && s <= 1;
}
```

http://totologic.blogspot.fr/2014/01/accurate-point-in-triangle-test.html

Dot Product Solution

- Assume that p_1, p_2, p_3 are ordered in counterclockwise and check if p lies at left of the 3 oriented edges
  - $[p_1, p_2]$, $[p_2, p_3]$, $[p_3, p_1]$
- First consider the 3 vectors v_1, v_2 and v_3 that are left-orthogonal to $[p_1, p_2]$, $[p_2, p_3]$ and $[p_3, p_1]$:
  - $v_1 = \langle y_2 - y_1, -x_2 + x_1 \rangle$
  - $v_2 = \langle y_3 - y_2, -x_3 + x_2 \rangle$
  - $v_3 = \langle y_1 - y_3, -x_1 + x_3 \rangle$
- Then we get the 3 following vectors :
  - $v_1' = \langle x - x_1, y - y_1 \rangle$
  - $v_2' = \langle x - x_2, y - y_2 \rangle$
  - $v_3' = \langle x - x_3, y - y_3 \rangle$



http://totologic.blogspot.fr/2014/01/accurate-point-in-triangle-test.html

Dot Product Solution .

- Compute the 3 dot products:
  - dot1 = v1 . v1' = $(y_2 - y_1)*(x - x_1) + (-x_2 + x_1)*(y - y_1)$
  - dot2 = v1 . v2' = $(y_3 - y_2)*(x - x_2) + (-x_3 + x_2)*(y - y_2)$
  - dot3 = v3 . v3' = $(y_1 - y_3)*(x - x_3) + (-x_1 + x_3)*(y - y_3)$
- Check if p lies in T if and only if
  - 0 <= dot1 and 0 <= dot2 and 0 <= dot3

http://totologic.blogspot.fr/2014/01/accurate-point-in-triangle-test.html

Dot Product Sample Code

```
function side(x1, y1, x2, y2, x, y:Number):Number
{
    return (y2 - y1)*(x - x1) + (-x2 + x1)*(y - y1);
}

function pointInTriangle(x1, y1, x2, y2, x3, y3, x, y:
    Number): Boolean
{
    var checkSide1:Boolean = side(x1, y1, x2, y2, x, y) >= 0;
    var checkSide2:Boolean = side(x2, y2, x3, y3, x, y) >= 0;
    var checkSide3:Boolean = side(x3, y3, x1, y1, x, y) >= 0;
    return checkSide1 && checkSide2 && checkSide3;
}
```

http://totologic.blogspot.fr/2014/01/accurate-point-in-triangle-test.html

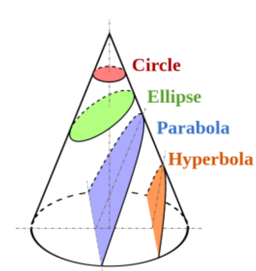
Cross Product Solution

- Calculate:
  - c1 = p_1 x p
  - c2 = p_2 x p
  - c3 = p_3 x p
- P is inside triangle if:
  - Clockwise order if
    - c1> 0 && c2> 0 && c3 > 0
  - Counterclockwise if
    - c1< 0 && c2< 0 && c3 < 0
  - No information if
    - (c1> 0 && c2> 0 && c3 > 0) || (c1< 0 && c2< 0 && c3 < 0)

http://www.sunshine2k.de/coding/java/PointInTriangle/PointInTriangle.html

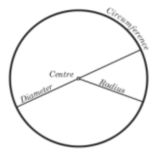
Conic Sections

- A conic section is a curve obtained as the intersection of a cone (more precisely, a right circular conical surface) with a plane



Circle

- The *circumference* of a circle means the length of the circle
- The interior of the circle is called a *disk*
- An *arc* is any continuous portion of a circle
- A *diameter* is a straight line through the center and terminating in both directions on the circumference



Equation of a Circle

- In an x-y coordinate system, the circle with centre (a, b) and radius r is the set of all points (x, y) such that:

$$(x-a)^2+(y-b)^2=r^2$$

- The equation of the circle follows from the Pythagorean theorem applied to any point on the circle

Circle at Origin

- If the circle is centred at the origin (0, 0), then the above formula can be simplified:

$$x^2+y^2=r^2$$

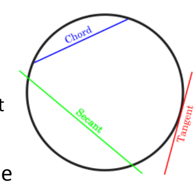
- and its tangent will be:

$$xx_1+yy_1=r^2$$

- where x_1, y_1 are the coordinates of the common point

Tangent and Secant Lines

- A line is tangent to a curve, at some point, if both line and curve pass through the point with the same direction
  - This is called the tangent line
  - Tangent line is the best straight-line approximation to the curve at that point
- The slope of a tangent line can be approximated by a secant line
  - It is a mistake to think of tangents as lines which intersect a curve at only one single point



Circle Parametric Equations

- When expressed in parametric equations (x, y) can be written using the trigonometric functions sine and cosine as:

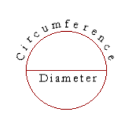
  x = a + r cost

  y = b + r sint

- where t is a parametric variable
  - Understood as the angle the ray to (x, y) makes with the x-axis

Definition of π

- π symbolizes the ratio
  - The relationship with respect to relative size of the circumference of circle to its diameter, whatever that relationship might be
  - So when we say that π ≈ 3.14, we mean that the circumference of circle is a little more than three times longer than the diameter:



C / D = π ≈ 3.14

  - π indicates the ratio of a curved line to a straight

Circumference of a Circle

- Since: C / D = π
- Can use that as a formula for calculating the circumference of a circle:

$$C = \pi D$$

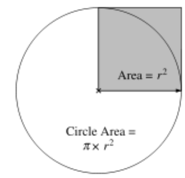
- Or, since D = 2r

$$C = \pi \cdot 2r = 2\pi r$$

Calculation of Area Enclosed

- The area enclosed by a circle is the radius squared, multiplied by π

$$A = r^2 \cdot \pi$$



Calculation of Area Enclosed .

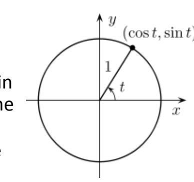
- Using a square with side lengths equal to the diameter of the circle, then dividing the square into four squares with side lengths equal to the radius of the circle, take the area of the smaller square and multiply by π

$$d = 2r = 2 \cdot \sqrt{\frac{A}{\pi}} \approx 1.1284 \cdot \sqrt{A}$$

  – approximately 79% of the circumscribing square

Unit Circle

- A unit circle is a circle with a unit radius
  – This is a circle whose radius is 1
- Often, the unit circle is the circle of radius 1 centered at the origin (0, 0) in the Cartesian coordinate system in the Euclidean plane
  – If (x, y) is a point on the unit circle in the first quadrant, then x and y are the lengths of the legs of a right triangle whose hypotenuse has length 1, then:

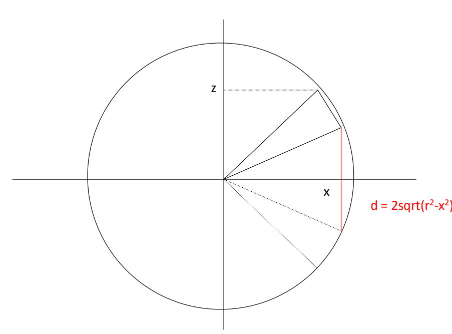


$$x^2 + y^2 = 1$$

Circle Properties

- The circle is the shape with the highest area for a given length of perimeter
- The circle is a highly symmetric shape
  – Every line through the centre forms a line of reflection symmetry and it has rotational symmetry around the centre for every angle
- All circles are similar:
  – A circle's circumference and radius are proportional
  – The area enclosed and the square of its radius are proportional
  – The constants of proportionality are 2π and π, respectively

Calculate x, y of Segment



Calculate x, y of Segment .

d = 2sqrt(r^2-x^2) → z = sqrt(r^2 - x^2) →
z = sqrt(r^2 - r^2/(1 + tanθ^2) →
z = r sqrt(1 - 1 / (1 + tanθ^2)

z = sqrt(r^2-x^2) → z^2 = r^2-x^2

But tanθ = z/x, so:

x^2tanθ^2 = r^2-x^2 → x^2(1+ tanθ^2) = r^2 →
x = r/sqrt(1 + tanθ^2)

Ground Implementation in C++

- TGround Class
  - Variables
    - Define 37 points in the surface of the ground
    - Define the normal of the ground
  - Functions
    - Ground Constructor
    - Draw Ground

TGround Class

```
class TGround
{
   public:
        TVector _points[37];  // points in the surface of the ground
        TVector _normal;                  // normal of the ground

   public:
        // Default constructor
        TGround();

        // Function that draws the ground
        void DrawGround();
};
```

Constructor

```
for(i=0; i<=36; i++)
{
        // Transform degrees in rads
        rad_angle=(pi*angle)/180.0;

        // Calculate the x and z co-ordinates of a circle ground
        x = radius/(sqrt(1.0 + tan(rad_angle)*tan(rad_angle)));
        z = radius*(sqrt(1.0 - (1.0/(1.0 +  tan(rad_angle)*tan(rad_angle)))));
        angle+= 10;

        // Check the co-ordinates in all the quadrants of the circle ground
        if ((i>=0) && (i<10))                          // First quadrant
        {
                if (i==9) {  _points[i] = TVector(0.0, 0.0, 1.0);  }
                                _points[i] = TVector(x, y, z);
        }
        if ((i>=10) && (i<19))                         // Second quadrant
        {
                _points[i] = TVector(-x, y, z);
        }
        if ((i>=19) && (i<28))                         // Third quadrant
        {
                if (i==27) { _points[i] = TVector(0.0, 0.0, -1.0); }
                _points[i] = TVector(-x, y, -z);
        }
        if ((i>=27) && (i<=36))                        // Fourth quadrant
        {
                _points[i] = TVector(x, y, -z);
        }
}

// Construct the vector for the normal of the ground
_normal = TVector(0.0, 1.0, 0.0);
```

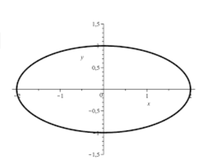
Draw Ground Function

```
void TGround::DrawGround()
{
    int i=0.0;

    glPushMatrix();
    glPushAttrib(GL_ENABLE_BIT);
    glCallList(50);

    glBegin(GL_POLYGON);
    for(i=0; i<=36; i++)
    {
            glNormal3f(0.0, 1.0, 0.0);
            glVertex3f(_points[i].X(), _points[i].Y(), _points[i].Z());
    }
    glEnd();

    glPopAttrib();
    glPopMatrix();
}
```

Ellipse

- An ellipse is a curve on a plane surrounding two focal points such that the sum of the distances to the two focal points is constant for every point on the curve

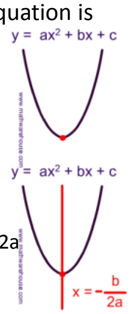
$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$



- You can think of an ellipse as an oval

Parabola

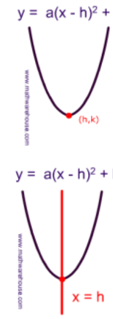
- The standard form of a parabola's equation is generally expressed:
  - $y = ax^2 + bx + c$
- The role of 'a'
  - If a> 0, the parabola opens upwards
  - If a< 0, it opens downwards
- The axis of symmetry
  - The axis of symmetry is the line x = -b/2a



http://www.mathwarehouse.com/geometry/parabola/standard-and-vertex-form.php

Vertex Form of Parabola

- The vertex form of a parabola's equation is generally expressed as:
  - $y = a(x-h)^2+k$ where (h,k) is the vertex
- If a is + then the parabola opens upwards like a regular "U"
- If a is - then the graph opens downwards like an upside down "U"
- If |a| < 1, the graph of the parabola widens
- If |a| > 1, the graph of the graph becomes narrower
  - The effect is the opposite of |a| < 1



http://www.mathwarehouse.com/geometry/parabola/standard-and-vertex-form.php

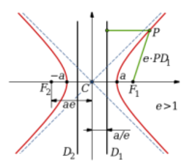
Hyperbola

- The equation of the hyperbola can be written as:

$$\frac{x^2}{a^2} - \frac{y^2}{b^2} = 1$$

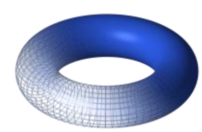
- If c is the distance from the center to either focus, then:

$$a^2+b^2 = c^2$$



Parametric surfaces

- A torus with major radius R and minor radius r may be defined parametrically as
  - x = cos(t)(R + r cos(u))
  - y = sin(t)(R + r cos(u))
  - z = r sin(u)
- where the two parameters t and u both vary between 0 and 2π



Equation of a Sphere

- Pythagoras theorem generalises to 3D giving:
  - $a^2 + b^2 + c^2 = d^2$
- Based on that we can easily prove that the general equation of a sphere is:
  - $(x- x_c)^2 + (y- y_c)^2 + (z- z_c)^2 = r^2$
- and at origin:
  - $x^2 + y^2 + z^2 = r^2$

Ray-Sphere Intersection

Ray: $P = P_0 + tV$
Sphere: $(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 = r^2$ or $|P - C|^2 - r^2 = 0$

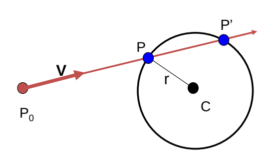
Substituting for P, we get: $|\mathbf{P_0 + tV} - C|^2 - r^2 = 0$

Solve quadratic equation: $at^2 + bt + c = 0$

where:

$a = |V|^2 = 1$
$b = 2 V \cdot (P_0 - C)$
$c = |P_0 - C|^2 - r^2$

$P = P_0 + tV$

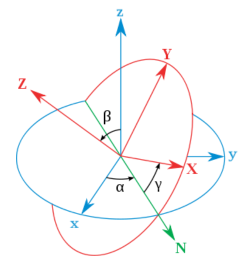


Advanced Methods of Rotation

- Different advanced methods exist including:
  - Euler angles
  - Quaternions

Euler Angles

- In many fields Euler angles are used to represent rotations
- Any rotation can be broken down into a series of three rotations about the major axes

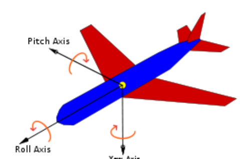


Euler Angles .

- We can simulate any arbitrary rotation with one rotation about the x-axis, one about the y-axis, and then one about the z-axis
  - i.e. consider an airplane pointing along the x-axis with the z-axis pointing up

Roll, Pitch, Yaw

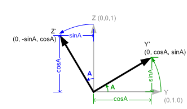
- Can represent any pose as a vector (roll, pitch, yaw)
  - The "roll" about the x-axis along the plane
  - The "pitch" about the y-axis which extends along the wings of the plane
  - The "yaw" or "heading" about the z-axis



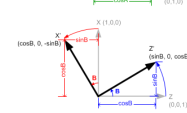
Disadvantages

- No universal standard for Euler rotations
  - Different fields use different sequences
    - i.e. some use z-y-z as opposed to the x-y-z system
- Although any rotation can be represented by either a set of Euler angles or a matrix
  - Computing the required angles is expensive and can introduce errors
- Interpolation does not work well!

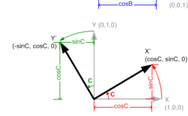
Standard Rotations



$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos A & -\sin A \\ 0 & \sin A & \cos A \end{pmatrix}$$



$$\begin{pmatrix} \cos B & 0 & \sin B \\ 0 & 1 & 0 \\ -\sin B & 0 & \cos B \end{pmatrix}$$



$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos A & -\sin A \\ 0 & \sin A & \cos A \end{pmatrix}$$

Angles to Axis

- Can combine the separate axis rotations into one matrix by multiplying standard rotational matrices together
  - Multiplication of matrices is not commutative
    - Different order of matrix multiplication results in a different outcome
- 6 different combinations are possible;
  - $R_xR_yR_z$, $R_xR_zR_y$, $R_yR_xR_z$, $R_yR_zR_x$, $R_zR_xR_y$ and $R_zR_yR_x$

$R_xR_yR_z$ & $R_xR_zR_y$ Rotations

$$
\begin{aligned}
&R_X R_Y R_Z \\
&= \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos A & -\sin A \\ 0 & \sin A & \cos A \end{pmatrix} \begin{pmatrix} \cos B & 0 & \sin B \\ 0 & 1 & 0 \\ -\sin B & 0 & \cos B \end{pmatrix} \begin{pmatrix} \cos C & -\sin C & 0 \\ \sin C & \cos C & 0 \\ 0 & 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} \cos B & 0 & \sin B \\ \sin A \sin B & \cos A & -\sin A \cos B \\ -\cos A \sin B & \sin A & \cos A \cos B \end{pmatrix} \begin{pmatrix} \cos C & -\sin C & 0 \\ \sin C & \cos C & 0 \\ 0 & 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} \cos B \cos C & -\cos B \sin C & \sin B \\ \sin A \sin B \cos C + \cos A \sin C & -\sin A \sin B \sin C + \cos A \cos C & -\sin A \cos B \\ -\cos A \sin B \cos C + \sin A \sin C & \cos A \sin B \sin C + \sin A \cos C & \cos A \cos B \end{pmatrix}
\end{aligned}
$$

$$
\begin{aligned}
&R_X R_Z R_Y \\
&= \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos A & -\sin A \\ 0 & \sin A & \cos A \end{pmatrix} \begin{pmatrix} \cos C & -\sin C & 0 \\ \sin C & \cos C & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos B & 0 & \sin B \\ 0 & 1 & 0 \\ -\sin B & 0 & \cos B \end{pmatrix} \\
&= \begin{pmatrix} \cos C & -\sin C & 0 \\ \cos A \sin C & \cos A \cos C & -\sin A \\ \sin A \sin C & \sin A \cos C & \cos A \end{pmatrix} \begin{pmatrix} \cos B & 0 & \sin B \\ 0 & 1 & 0 \\ -\sin B & 0 & \cos B \end{pmatrix} \\
&= \begin{pmatrix} \cos C \cos B & -\sin C & \cos C \sin B \\ \cos A \sin C \cos B + \sin A \sin B & \cos A \cos C & \cos A \sin C \sin B - \sin A \cos B \\ \sin A \sin C \cos B - \cos A \sin B & \sin A \cos C & \sin A \sin C \sin B + \cos A \cos B \end{pmatrix}
\end{aligned}
$$

$R_yR_xR_z$ & $R_yR_zR_x$ Rotations

$$
\begin{aligned}
&R_Y R_X R_Z \\
&= \begin{pmatrix} \cos B & 0 & \sin B \\ 0 & 1 & 0 \\ -\sin B & 0 & \cos B \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos A & -\sin A \\ 0 & \sin A & \cos A \end{pmatrix} \begin{pmatrix} \cos C & -\sin C & 0 \\ \sin C & \cos C & 0 \\ 0 & 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} \cos B & \sin B \sin A & \sin B \cos A \\ 0 & \cos A & -\sin A \\ -\sin B & \cos B \sin A & \cos B \cos A \end{pmatrix} \begin{pmatrix} \cos C & -\sin C & 0 \\ \sin C & \cos C & 0 \\ 0 & 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} \cos B \cos C + \sin B \sin A \sin C & -\cos B \sin C + \sin B \sin A \cos C & \sin B \cos A \\ \cos A \sin C & \cos A \cos C & -\sin A \\ -\sin B \cos C + \cos B \sin A \sin C & \sin B \sin C + \cos B \sin A \cos C & \cos B \cos A \end{pmatrix}
\end{aligned}
$$

$$
\begin{aligned}
&R_Y R_Z R_X \\
&= \begin{pmatrix} \cos B & 0 & \sin B \\ 0 & 1 & 0 \\ -\sin B & 0 & \cos B \end{pmatrix} \begin{pmatrix} \cos C & -\sin C & 0 \\ \sin C & \cos C & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos A & -\sin A \\ 0 & \sin A & \cos A \end{pmatrix} \\
&= \begin{pmatrix} \cos B \cos C & -\cos B \sin C & \sin B \\ \sin C & \cos C & 0 \\ -\sin B \cos C & \sin B \sin C & \cos B \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos A & -\sin A \\ 0 & \sin A & \cos A \end{pmatrix} \\
&= \begin{pmatrix} \cos B \cos C & -\cos B \sin C \sin A + \sin B \sin A & \cos B \sin C \sin A + \sin B \cos A \\ \sin C & \cos C \cos A & -\cos C \sin A \\ -\sin B \cos C & \sin B \sin C \cos A + \cos B \sin A & -\sin B \sin C \sin A + \cos B \cos A \end{pmatrix}
\end{aligned}
$$

$R_zR_xR_y$ & $R_zR_yR_x$ Rotations

$$
\begin{aligned}
&R_Z R_X R_Y \\
&= \begin{pmatrix} \cos C & -\sin C & 0 \\ \sin C & \cos C & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos A & -\sin A \\ 0 & \sin A & \cos A \end{pmatrix} \begin{pmatrix} \cos B & 0 & \sin B \\ 0 & 1 & 0 \\ -\sin B & 0 & \cos B \end{pmatrix} \\
&= \begin{pmatrix} \cos C & -\sin C \cos A & \sin C \sin A \\ \sin C & \cos C \cos A & -\cos C \sin A \\ 0 & \sin A & \cos A \end{pmatrix} \begin{pmatrix} \cos B & 0 & \sin B \\ 0 & 1 & 0 \\ -\sin B & 0 & \cos B \end{pmatrix} \\
&= \begin{pmatrix} \cos C \cos B - \sin C \sin A \sin B & -\sin C \cos A & \cos C \sin B + \sin C \sin A \cos B \\ \sin C \cos B + \cos C \sin A \sin B & \cos C \cos A & \sin C \sin B - \cos C \sin A \cos B \\ -\cos A \sin B & \sin A & \cos A \cos B \end{pmatrix}
\end{aligned}
$$

$$
\begin{aligned}
&R_Z R_Y R_X \\
&= \begin{pmatrix} \cos C & -\sin C & 0 \\ \sin C & \cos C & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos B & 0 & \sin B \\ 0 & 1 & 0 \\ -\sin B & 0 & \cos B \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos A & -\sin A \\ 0 & \sin A & \cos A \end{pmatrix} \\
&= \begin{pmatrix} \cos C \cos B & -\sin C & \cos C \sin B \\ \sin C \cos B & \cos C & \sin C \sin B \\ -\sin B & 0 & \cos B \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos A & -\sin A \\ 0 & \sin A & \cos A \end{pmatrix} \\
&= \begin{pmatrix} \cos C \cos B & -\sin C \cos A + \cos C \sin B \sin A & \sin C \sin A + \cos C \sin B \cos A \\ \sin C \cos B & \cos C \cos A + \sin C \sin B \sin A & -\cos C \sin A + \sin C \sin B \cos A \\ -\sin B & \cos B \sin A & \cos B \cos A \end{pmatrix}
\end{aligned}
$$

Code Example

```
void anglesToAxes(const Vector3 angles, Vector3& left, Vector3& up, Vector3& forward)
{
   const float DEG2RAD = 3.141593f / 180;
   float sx, sy, sz, cx, cy, cz, theta;

   // rotation angle about X-axis (pitch)
   theta = angles.x * DEG2RAD;
   sx = sinf(theta);
   cx = cosf(theta);

   // rotation angle about Y-axis (yaw)
   theta = angles.y * DEG2RAD;
   sy = sinf(theta);
   cy = cosf(theta);

   // rotation angle about Z-axis (roll)
   theta = angles.z * DEG2RAD;
   sz = sinf(theta);
   cz = cosf(theta);
}
```

Code Example .

$$
\left.\begin{array}{l} \text{left.x = cy*cz;} \\ \text{left.y = sx*sy*cz + cx*sz;} \\ \text{left.z = -cx*sy*cz + sx*sz;} \end{array}\right\} \quad \textit{determine left axis}
$$

$$
\left.\begin{array}{l} \text{up.x = -cy*sz;} \\ \text{up.y = -sx*sy*sz + cx*cz;} \\ \text{up.z = cx*sy*sz + sx*cz;} \end{array}\right\} \quad \textit{determine up axis}
$$

$$
\left.\begin{array}{l} \text{forward.x = sy;} \\ \text{forward.y = -sx*cy;} \\ \text{forward.z = cx*cy;} \end{array}\right\} \quad \textit{determine forward axis}
$$

Rotation About Arbitrary Axis

- In addition to the set of three Euler angles and the rotation matrix, a rotation can also be represented by a vector specifying the rotation axis and the angle of rotation around this axis
- The problem of rotation about an arbitrary axis in three dimensions arises in many fields including computer graphics and computer games

3x3 Matrix Representing Axis

- We can express the 3×3 rotation matrix in terms of a 3×3 matrix representing the axis:
  - [R] = [I] + s*[~axis] + t*[~axis]^2
- or equivalently:
  - [R] = c*[I] + s*[~axis] + t*([~axis]^2 + [I])

Matrix Expansion

$$[R] = \begin{bmatrix} t*x*x + c & t*x*y - z*s & t*x*z + y*s \\ t*x*y + z*s & t*y*y + c & t*y*z - x*s \\ t*x*z - y*s & t*y*z + x*s & t*z*z + c \end{bmatrix}$$

- where:
  - c = cos(angle)
  - s = sin(angle)
  - t = 1 - c
  - x = normalised axis x coordinate
  - y = normalised axis y coordinate
  - z = normalised axis z coordinate

Code Example

```
public void matrixFromAxisAngle(AxisAngle4d a1)
{
   double c = Math.cos(a1.angle);
   double s = Math.sin(a1.angle);
   double t = 1.0 - c;

   m00 = c + a1.x*a1.x*t;
   m11 = c + a1.y*a1.y*t;
   m22 = c + a1.z*a1.z*t;

   double tmp1 = a1.x*a1.y*t;
   double tmp2 = a1.z*s;

   m10 = tmp1 + tmp2;
   m01 = tmp1 - tmp2;
   tmp1 = a1.x*a1.z*t;
   tmp2 = a1.y*s;
   m20 = tmp1 - tmp2;
   m02 = tmp1 + tmp2;    tmp1 = a1.y*a1.z*t;
   tmp2 = a1.x*s;
   m21 = tmp1 + tmp2;
   m12 = tmp1 - tmp2;
}
```

Quaternions

- Complex numbers were discovered in 1800's and had the characteristic property to be defined in terms of i, where i is the square root of –1
- In 1843, sir William Rowan Hamilton discovered a number called the quaternion, which has a very similar form to complex numbers

Quaternion Definition

- A quaternion is based on three different numbers that are all square roots of –1 and are labeled i, j and k, where:

$$\mathrm{q} = (z_1, z_2, z_3, s) = (\mathbf{z}, s)$$

$$\mathrm{q} = iz_1 + jz_2 + kz_3 + s$$

$$i^2 = j^2 = k^2 = -1 \qquad \begin{array}{ll} ij = k & ji = -k \\ jk = i & kj = -i \\ ki = j & ik = -j \end{array}$$

Quaternion Properties

Multiplication natural consequence of defn.

$$q \cdot p = (\mathbf{z}_q s_p + \mathbf{z}_p s_q + \mathbf{z}_p \times \mathbf{z}_q \ , \ s_p s_q - \mathbf{z}_p \cdot \mathbf{z}_q)$$

Conjugate

$$q^* = (-\mathbf{z}, s)$$

Magnitude

$$||q||^2 = \mathbf{z} \cdot \mathbf{z} + s^2 = q \cdot q^*$$

Quaternion Rotations

Vectors as quaternions

$$v = (\mathbf{v}, 0)$$

Rotations as quaternions

$$r = (\hat{\mathbf{r}} \sin\frac{\theta}{2}, \cos\frac{\theta}{2})$$

Rotating a vector

$$x' = r \cdot x \cdot r^*$$

Composing rotations

$$r = r_1 \cdot r_2$$ ← Compare to Exp. Map

Quaternion Representation

- Defined like complex numbers but with 4 coordinates
  - q[w, (x, y, z)] also written q[w, v] where v = (x, y, z)
  - q = w + xi + yj + zk
    - Here, *w* is real part, and *(x,y,z)* are imaginary parts
    - Think of *w* as angle in an angle-axis representation
    - Think of *(x, y, z)* as axis in an axis-angle representation
- Based on three different roots of -1:
  - $i^2 = j^2 = k^2 = -1$

Quaternion Representation .

- For a right-hand rotation of θ **radians** about unit vector *v*, quaternion is:

  q = (cos(θ/2); **v** sin(θ/2))

  - Note how the 3 imaginary coordinates are noted as a vector
  - Only **unit quaternions** represent rotations
    - Such a quaternion describes a point on the 4D unit hyper-sphere
  - Important note: q and –q represent the **exact same** orientation

Quaternion Toolbox

- Addition
  - $q^1 + q^2 = [w^1 + w^2, v^1 + v^2]$
- Multiplication
  - $q^1q^2 = [w^1w^2 – v^1.v^2, v^1 \times v^2 + w^1v^2 + w^2v^1]$ (note: $q^1q^2 \neq q^2q^1$)
- Magnitude
  - $| q | = \text{sqrt}(w^2 + x^2 + y^2 + z^2)$
- Normalisation
  - N(q) = q / | q |
- Conjugate
  - q* = [w , -v]
- Inverse
  - $q^{-1} = q^* / | q |^2$
- Unit quternion
  - q is unit if | q | = 1 and $q^{-1} = q^*$
- Identity
  - $q^{Identity}$ = [1,(0,0,0)] for multiplication, $q^{Identity}$ = [0,(0,0,0)] for addition

Transforming a Point or Vector

- To transform a vector P by the rotation specified by the quaternion q, there are two options:
  - Multiply conj(q) by (0, Px, Py, Pz)
    - See next slide
  - Convert q to matrix and use matrix transformation

First Method

- Rotate vector P angle θ around unit axis R:
  - Form the quaternion representing the vector P
    - q1 = (0,Px,Py,Pz)
  - Form the rotation quaternion from the axis R and angle θ
    - q2 = (cos(θ/2),Rx sin(θ/2),Ry sin(θ/2),Rz sin(θ/2))
  - The rotated vector is given by v entry of the quaternion:
    - q3 = q2 q1 q2*
      - q2 must be of unit magnitude for this to work properly

Quaternion and Axis-Angle

- From axis-angle to quaternion:
  - q = (cos(θ/2); v sin(θ/2))
  - where:
    - v is the axis
    - θ is the angle
- From quaternion to axis-angle:
  - Axis v = (x, y, z) / sqrt($x^2 + y^2 + z^2$)
  - Angle θ = acos(w) * 2

Quaternion to Matrix

- From quaternion to a 3x3 rotation matrix:

$$\begin{vmatrix} 1-2y^2-2z^2 & 2yz+2wx & 2xz-2wy \\ 2xy-2wz & 1-2x^2-2z^2 & 2yz-2wx \\ 2xz+2wy & 2yz-2wx & 1-2x^2-2y^2 \end{vmatrix}$$

Euler Angles to Quaternion

- From Euler angles (pitch, yaw, roll)
  - Create three quaternions
  - One for each of pitch, roll, yaw
  - Then multiply them together
- Here, *P* = pitch/2, *Y* = yaw/2, *R* = roll/2
  - w = cos(R)*cos(P)*cos(Y) + sin(R)*sin(P)*sin(Y)
  - x = sin(R)*cos(P)*cos(Y) – cos(R)*sin(P)*sin(Y)
  - y = cos(R)*sin(P)*cos(Y) + sin(R)*cos(P)*sin(Y)
  - z = cos(R)*cos(P)*sin(Y) – sin(R)*sin(P)*cos(Y)

Quaternion Code Example

- Three functions:
  - Convert an axis and angle rotation to a quaternion
  - Convert a quaternion to a rotation matrix
  - Rotate the quaternion

Convert an axis and angle rotation to a quaternion

```
void Tquaternion::CreateFromAxisAngle(float X,
   float Y, float Z, float degree)
{
   float angle = float((degree / 180.0f) * PI);
   float result = (float)sin( angle / 2.0f );
   w = (float)cos( angle / 2.0f );
   x = float(X * result);
   y = float(Y * result);
   z = float(Z * result);
}
```

Convert a quaternion to a rotation matrix

```
void Tquaternion::CreateMatrix(float *pMatrix)
{
    pMatrix[ 0] = 1.0f - 2.0f * ( y * y + z * z );
    pMatrix[ 1] = 2.0f * (x * y + z * w);
    pMatrix[ 2] = 2.0f * (x * z - y * w);
    pMatrix[ 3] = 0.0f;
    pMatrix[ 4] = 2.0f * ( x * y - z * w );
    pMatrix[ 5] = 1.0f - 2.0f * ( x * x + z * z );
    pMatrix[ 6] = 2.0f * (z * y + x * w );
    pMatrix[ 7] = 0.0f;
    pMatrix[ 8] = 2.0f * ( x * z + y * w );
    pMatrix[ 9] = 2.0f * ( y * z - x * w );
    pMatrix[10] = 1.0f - 2.0f * ( x * x + y * y );
    pMatrix[11] = 0.0f;
    pMatrix[12] = 0;
    pMatrix[13] = 0;
    pMatrix[14] = 0;
    pMatrix[15] = 1.0f;
}
```

Rotate the Quaternion

```
void Tquaternion::quaternionRotation(double x, double y, double z)
{
    float matrixX[16], matrixY[16], matrixZ[16];
    static float rotation = 0;

    Tquaternion qRotationX;
    Tquaternion qRotationY;
    Tquaternion qRotationZ;
    qRotationX.CreateFromAxisAngle(1, 0, 0, x);
    qRotationY.CreateFromAxisAngle(0, 1, 0, y);
    qRotationZ.CreateFromAxisAngle(0, 0, 1, z);

    qRotationX.CreateMatrix(matrixX);
    qRotationY.CreateMatrix(matrixY);
    qRotationZ.CreateMatrix(matrixZ);

    glMultMatrixf(matrixX);
    glMultMatrixf(matrixY);
    glMultMatrixf(matrixZ);
}
```

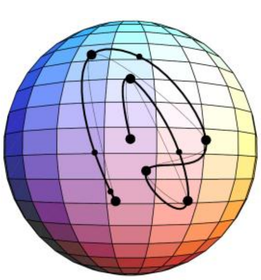
Quaternion Interpolation

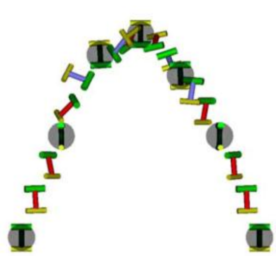
- One of the most important reasons for using quaternions is that they are very good at representing rotations in space
- Quaternions overcome the issues that plague other methods of rotating points in 3D space such as Gimbal lock
  - An issue when you represent your rotation with Euler angles

Quaternion Interpolation Methods

- Using quaternions can define several methods that represents a rotational interpolation in 3D space:
  - SLERP
    - Used to smoothly interpolate a point between two orientations
  - SQAD (extension of SLERP)
    - Used to interpolate through a sequence of orientations that define a path

Examples





SLERP

- SLERP provides a method to smoothly interpolate a point about two orientations
  - SLERP stands for Spherical Linear Interpolation
- Represent the first orientation as q_1 and the second orientation as q_2
- The point that is interpolated will be represented by P and the interpolated point will be represented by P'
- The interpolation parameter t will interpolate P from q_1 when t=0 to q_2 when t=1

SLERP Interpolation

- The standard linear interpolation formula is:

  $p' = p_1 + t(p_2 - p_1)$
- The general steps to apply this equation are:
  - Compute the difference between p_1 and p_2
  - Take the fractional part of that difference
  - Adjust the original value by the fractional difference between the two points
- Can use the same principle to interpolate between two quaternion orientations

SQAD

- SQUAD (Spherical and Quadrangle) can be used to smoothly interpolate over a path of rotations
  - Just as a SLERP can be used to compute an interpolation between two quaternions
- If we have the sequence of quaternions:

$$q_1, q_2, q_3, \cdots, q_{n-2}, q_{n-1}, q_n$$

SQAD Representation

- And we also define the "helper" quaternion (s_i) which we can consider an intermediate control point:

$$s_i = \exp\left(-\frac{\log\left(q_{i+1}q_i^{-1}\right) + \log\left(q_{i-1}q_i^{-1}\right)}{4}\right) q_i$$

SQAD Orientation

- Then the orientation along the sub-curve defined by:

$$q_{i-1}, q_i, q_{i+1}, q_{i+2}$$

- at time t is given by:

$$\text{squad}(q_i, q_{i+1}, s_i, s_{i+1}, t) = \text{slerp}(\text{slerp}(q_i, q_{i+1}, t), \text{slerp}(s_i, s_{i+1}, t), 2t(1-t))$$

Quaternion Advantages

- Quaternion interpolation using SLERP and SQUAD provide a way to interpolate smoothly between orientations in space
- Rotation concatenation using quaternions is faster than combining rotations expressed in matrix form
- Converting quaternions to matrices is slightly faster than for Euler angles
- Quaternions only require 4 numbers
  - 3 if they are normalized
  - The Real part can be computed at run-time
    - To represent a rotation where a matrix requires at least 9 values

Quaternion Disadvantages

- Very hard to understand
- Can become invalid because of floating-point round-off error
  - This can be resolved by re-normalizing the quaternion

References

- http://www.songho.ca/opengl/gl_anglestoaxes.html
- http://inside.mines.edu/~gmurray/ArbitraryAxisRotation/
- http://www.euclideanspace.com/maths/geometry/rotations/conversions/angleToMatrix/
- http://en.wikipedia.org/wiki/Euler_angles
- http://en.wikipedia.org/wiki/Line_(mathematics)
- http://en.wikipedia.org/wiki/Plane_(mathematics)
- http://inst.eecs.berkeley.edu/~cs283/sp13/lectures/283-lecture18.pdf
- http://www.gamedev.net/page/resources/_/technical/math-and-physics/quaternion-powers-r1095
- http://www.gamasutra.com/view/feature/3278/
- http://3dgep.com/?p=1815

Programming Links

- http://pages.cs.wisc.edu/~cs368-2/CppTutorial/NOTES/CLASSES-INTRO.html
- http://www.quantstart.com/articles/Matrix-Classes-in-C-The-Header-File
- http://www.programiz.com/article/c%2B%2B-programming-pattern
- http://stackoverflow.com/questions/564877/point-and-line-class-in-c
- http://www.seasite.niu.edu/CS240/Old_CPP_Notes/lines_class_cpp_program.htm
- http://www.linuxfocus.org/English/March1998/article28.html
- http://www.cs.stanford.edu/~acoates/quaternion.h

Questions