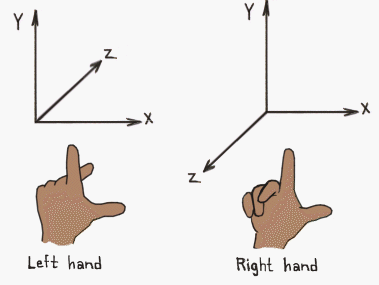
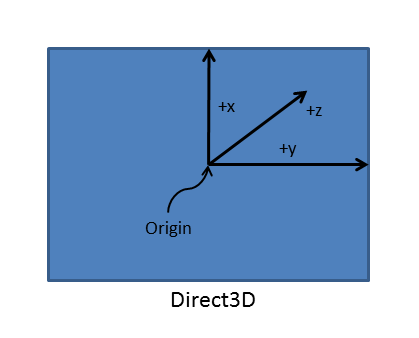
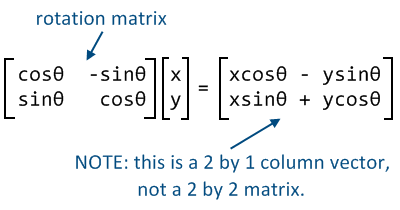
**Basics**: **Coordinate system**

**lefthanded** (Direct3D) vs **righthanded** (OpenGL) coordinates

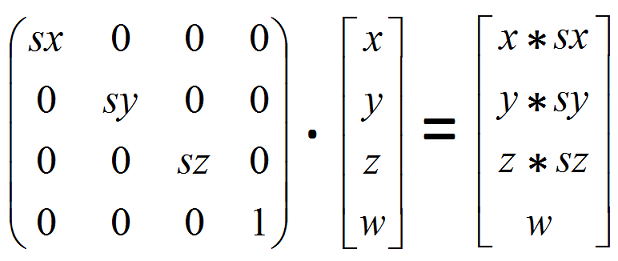
**Normalized device coordinates (NDC)**: -1.0 is the far left and 1.0 is the far right regardless of the size of the screen, the same applies to the y-axis, z-axis goes from 0 to 1 into the screen; used in Direct3D and OpenGL

Point (0,0,0) is at the **center** at the screen

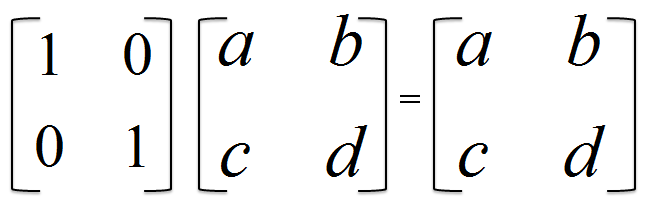
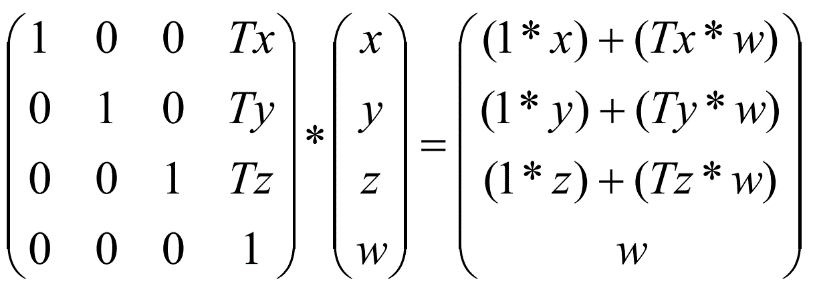
**Math**: **Matrices**

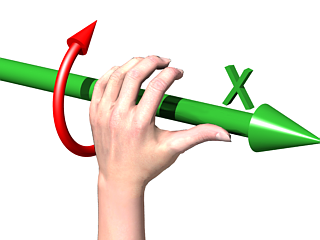
Add & subtract: same dimension, just add/ subtract numbers in same position

Multiplication: row \* column

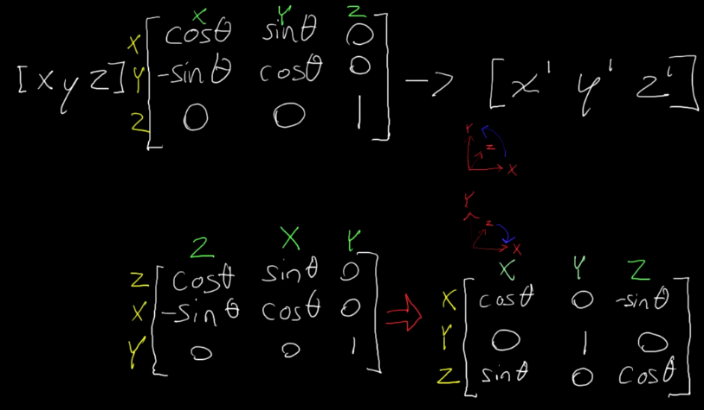
Rotation (applies to vector): angle sum formula; sin(A+B) = sin(A)cos(B) + sin(B)cos(A), cos(A+B) = cos(A)cos(B) - sin(A)sin(B)

Scaling: scaler on diagonal line, the scaler applies to every row

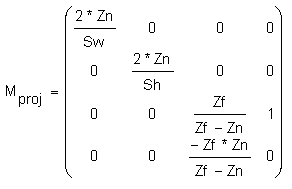
Identity matrix: doesn’t change the other matrix in multiplication

Translation: needs an extra dimension w

Rotation in 3D: left-hand rule (left-handed coordinate)

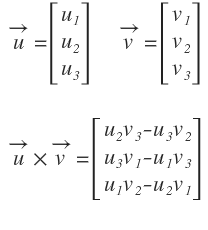
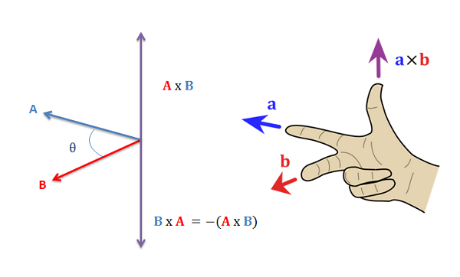
* Rotation around z-axis: from x to y
* Rotation around y-axis: from z to x
* Rotation around x-axis: from y to z

3d Projection:

Perspective division: divide x and y by z

**Math**: **Vectors**

**Dot product**: returns a **scalar**; projection of a on b times b; order doesn’t matter

* |a| |b| cos(θ)
* a1b1 + a2b2 + a3b3+ …
* value increases as θ gets smaller, negative if over 90, zero if 0

**cross product**: returns a **vector** **perpendicular** to both vectors; order matters

* magnitude: |a| |b| sin(θ)
* use left hand rule (left-handed coordinate)

**Rasterization**: **Drawing Triangles**

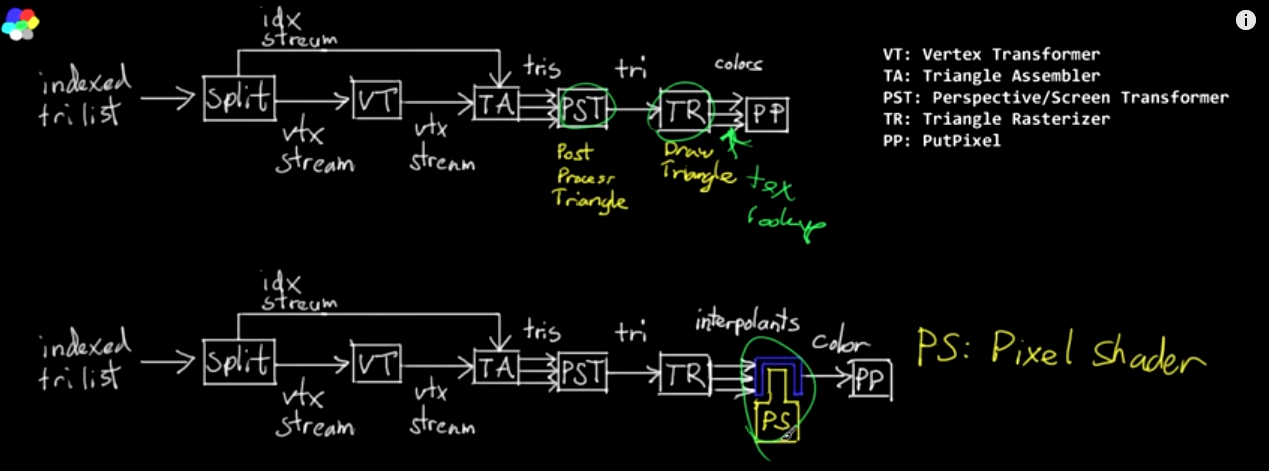
* using triangles to represent all shapes
* split all triangles into flat-top and flat-bottom triangles
* draw the triangles using scanlines from top down
* a pixel belongs to a triangle if (DirectX rules)
  + its center is in the triangle
  + its center is on the top edge of the triangle
  + its center is on the left edge of the triangle

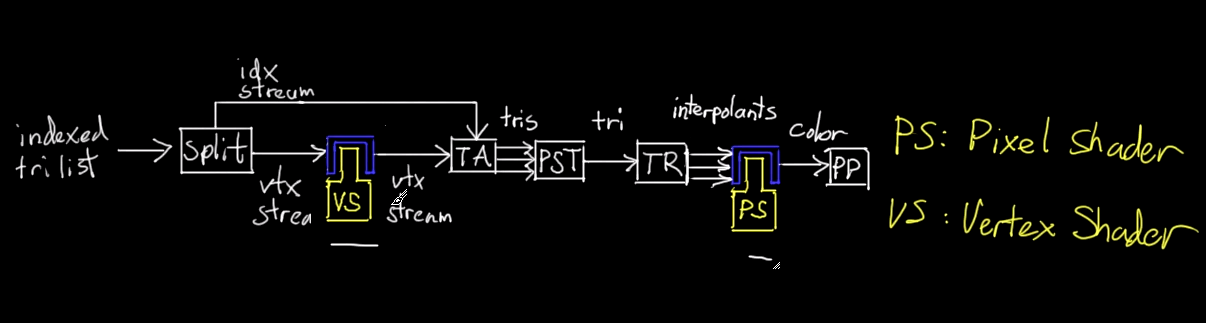
**Camera:**

* apply negative transformation to the world to get camera view
* store rotation as a rotation matrix
* movement: apply rotation to a unit vector of initial direction/heading to move where you are facing

**Pipeline**

* representation of objects in 3d
  + store vector of vertices + vector for faces or
  + store vertices as triangles
  + set origin at mean of vertices for better rotation
* representation of the camera
  + position (vector) and orientation (matrix)
* for each object:
  + apply rotation around objects’ origins (matrix multiplication)
  + transformation (position object in 3d space)
  + apply rotation for camera view (inverse of camera orientation)
  + apply transformation (opposite of camera’s position)
* project the object onto the screen
  + transform with projection matrix
* make things in front block things in the back
  + Back face culling (only for rendering convex shapes with no overlaps)
    - Create the triangles’ vertices in a particular order so that
    - Cross product gives a normal that points outwards
    - Find the vector from the camera to the triangle
    - Dot product to determine if the face is facing you
    - Only render the triangle if the face is facing you
  + Painter’s algorithm (used to render concave shapes on top of culling)
    - Sort triangles by average z value
    - Draw further ones first
  + Z-buffer (deals with overlapping triangles, replaces painter’s algorithm)
    - Store color for each pixel by z value
    - Pixel map of the screen
* draw the triangles in 2d
  + transform to screen space
  + apply rasterization rules
* Shaders – allows custom effects
  + Vertex shader: rotation & transformation of vertices
  + Geometry shader: deals with shapes
    - Split triangles into smaller ones
    - Determine the shading for each triangle
  + Pixel shader: 2d, applying texture and color to pixels on screen
    - Texture
      * Add texture coordinates to vertices
      * interpolate to find colors at that pixel
      * Use 1/z since z isn’t linear

Pipeline



Shading

