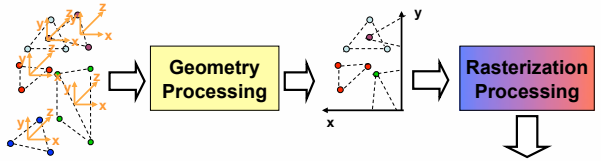
GPU Programming for Video Games

3D Coordinates & Transformations

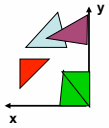
Prof. Aaron Lanterman
(Based on slides by Prof. Hsien-Hsin Sean Lee)
School of Electrical and Computer Engineering
Georgia Institute of Technology

Georgia Institute of Technology

3D graphics rendering pipeline (1)

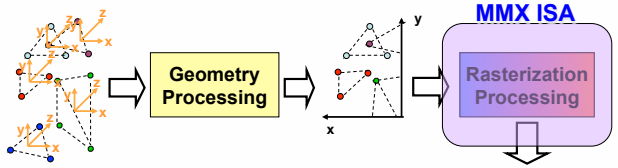


- Geometry pipeline
  - Processing vertices
  - Mainly **floating-point** operations
- Rasterization pipeline
  - Processing pixels
  - Mainly dealing with **integer** operations

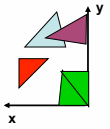


Georgia Institute of Technology

3D graphics rendering pipeline (2)

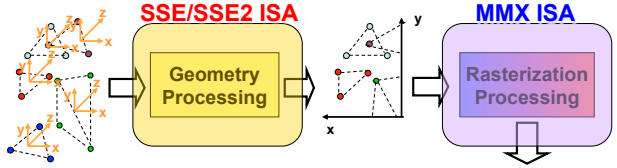


- Geometry pipeline
  - Processing vertices
  - Mainly floating-point operations
- Rasterization pipeline
  - Processing pixels
  - Mainly dealing with Integer operations
  - **MMX was originally designed to accelerate this type of functionality**

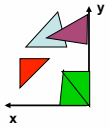


Georgia Institute of Technology

3D graphics rendering pipeline (3)

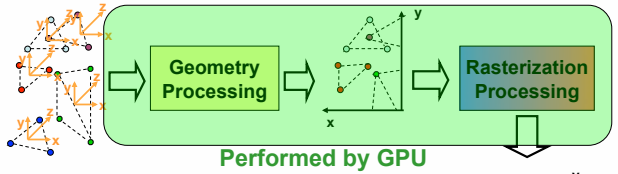


- Geometry pipeline
  - Processing vertices
  - Mainly floating-point operations
  - **SSE/SSE2 were designed for this part**
- Rasterization pipeline
  - Processing pixels
  - Mainly dealing with Integer operations
  - **MMX was originally designed to accelerate this type of functionality**

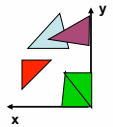


Georgia Institute of Technology

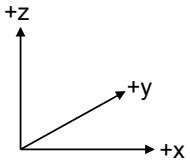
Fixed-function 3D graphics pipeline



- Geometry pipeline
  - Processing vertices
  - Mainly floating-point operations
  - ~~SSE/SSE2 were designed for this part~~
- Rasterization pipeline
  - Processing pixels
  - Mainly dealing with integer operations
  - ~~MMX was originally designed to accelerate this type of functionality~~

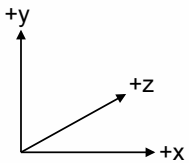


3D Coord: Math textbooks use z-up



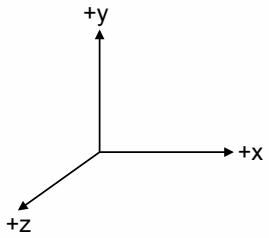
Z-up, Right-Handed System

3D Coord: Real games tend to use y-up



Left-Handed System

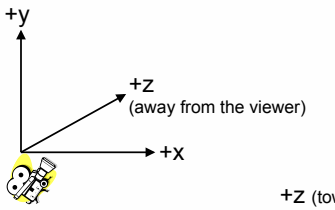
- Direct3D
- Unity3D



Right-Handed System

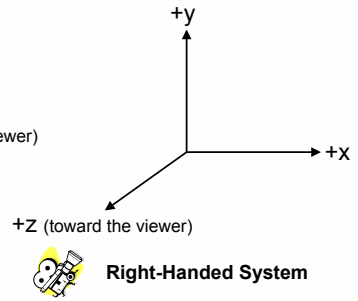
- OpenGL
- XNA

X-Y natural for screen coordinates



Left-Handed System

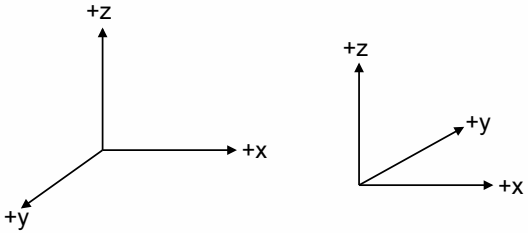
- Direct3D
- Unity3D



Right-Handed System

- OpenGL
- XNA

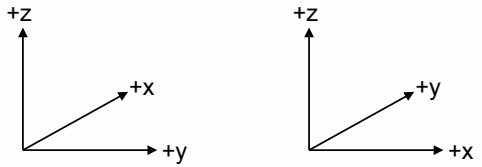
Some use Z-up for world coordinates



Left-Handed System **Right-Handed System**

- Z-up, LHS: Unreal
- Z-up, RHS: Quake/Radiant, Source/Hammer
- Nearly everything still uses Y-up for screen coordinates!

Another view



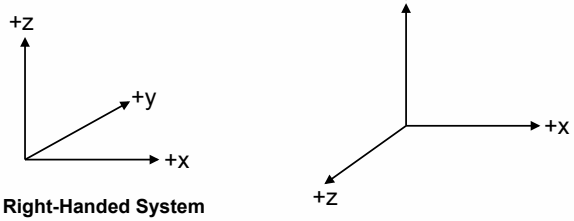
Left-Handed System

- Unreal

Right-Handed System

- Quake/Radiant
- Source/Hammer

3D "object" modeling software



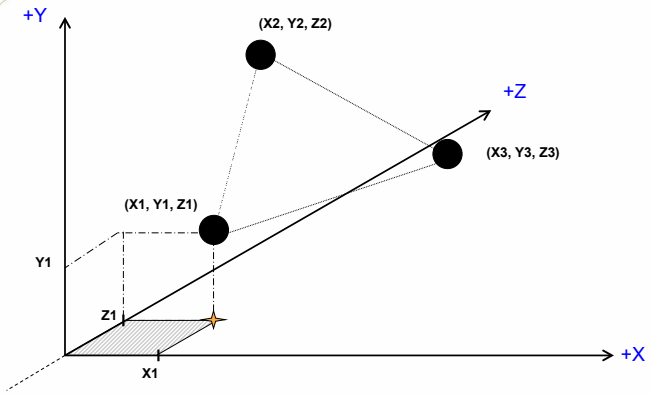
Right-Handed System

3D Studio Max, Blender

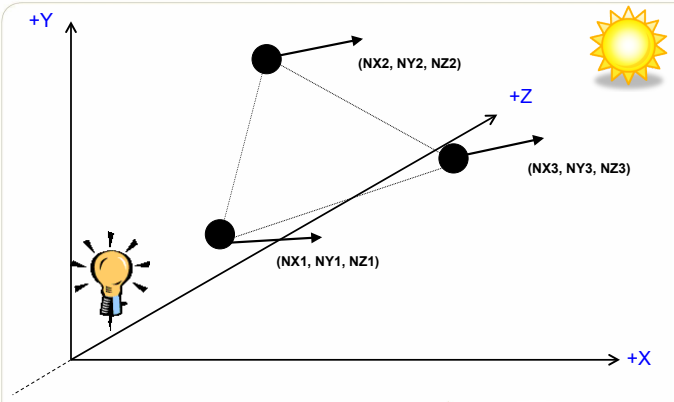
Right-Handed System

Maya, Milkshape

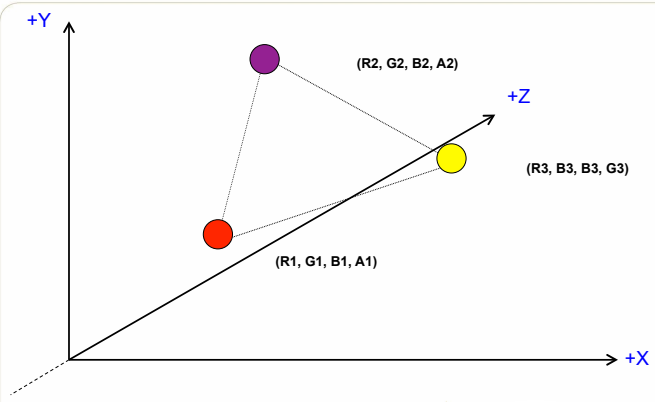
Geometry format – vertex coordinates



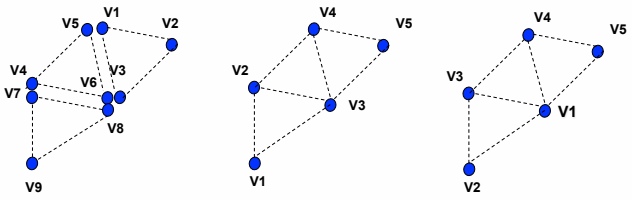
Geometry format – vertex normals



Geometry format – vertex colors

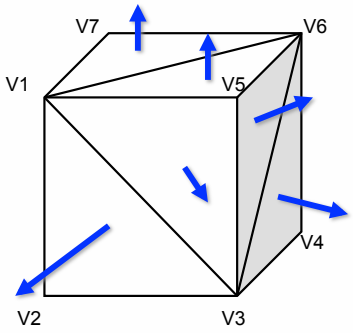


Triangle-based geometry representation



Triangle List Triangle Strip Triangle Fan

Specifying a 3D object (1)

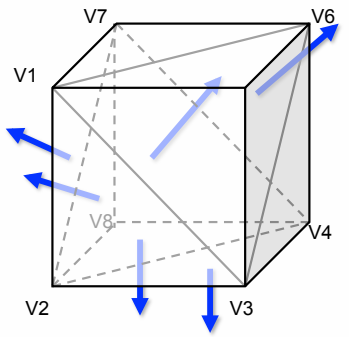


Triangle list
{v1, v3, v2},
{v1, v5, v3},
{v5, v6, v3},
{v4, v3, v6},
{v1, v7, v6},
{v1, v6, v5}

Triangle strip
{v5, v3, v1, v2},
{v5, v6, v3, v4},
{v7, v6, v1, v5}

- Vertex ordering is critical when culling mode enabled
- We will discuss normal computation later

Specifying a 3D object (2)

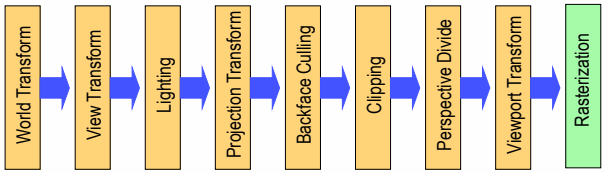


Triangle list
{v1, v2, v7},
{v2, v8, v7},
{v2, v3, v4},
{v2, v4, v8},
{v4, v7, v8},
{v4, v6, v7}

Triangle strip
{v1, v2, v7, v8},
{v3, v4, v2, v8},
{v6, v7, v4, v8}

- Vertex ordering is critical when culling mode enabled
- We will discuss normal computation later

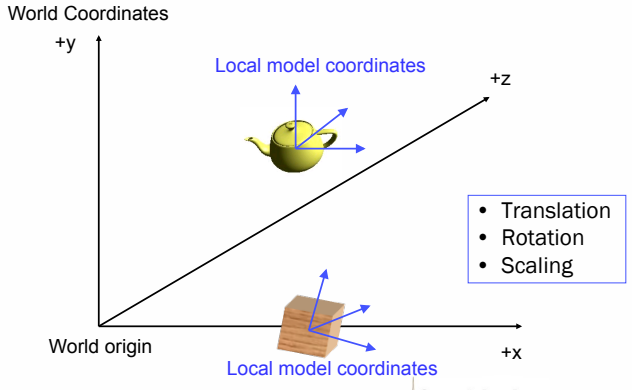
3D rendering pipeline



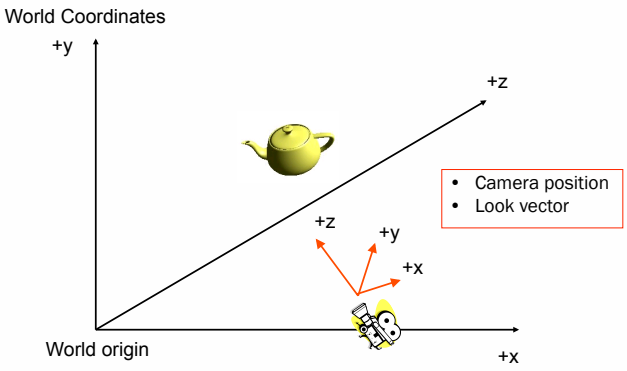
Transformation pipeline

- World Transformation
  - Model (object) coordinates → World coordinates
- View Transformation
  - World coordinates → View (camera) space
- Projection Transformation
  - View (camera) space → Screen plane
- These are a series of matrix multiplications

World transformation



View transformation



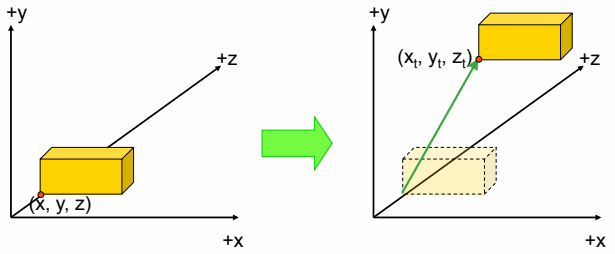
Projection transformation

- Set up camera internals
- Set up
  - Field of View (FOV)
  - View frustum
  - View planes
- Will discuss in the next lecture

Homogeneous coordinates

- Enable all transformations to be done by "*multiplication*"
  - Primarily for translation (see next few slides)
- Add one coordinate (w) to a 3D vector
- Each vertex has [x, y, z, w]
  - w will be useful for perspective projection
  - w should be 1 in a Cartesian coordinate system

Transformation 1: translation (Offset)



Translation matrix (1)

$$[x_t, y_t, z_t, 1] = [x, y, z, 1] \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

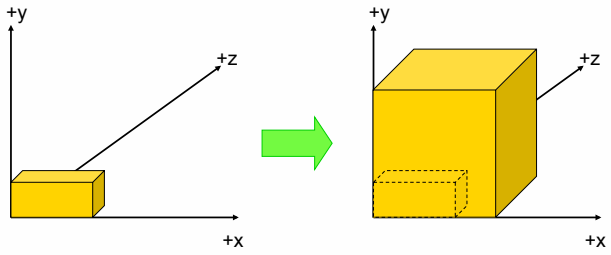
- Direct3D and XNA use a row-coordinate convention (as shown above)
- Transformation matrices operate "left to right"

Translation matrix (2)

$$\begin{bmatrix} x_t \\ y_t \\ z_t \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- OpenGL, Unity & non-graphics world uses column coordinates (as shown above)
- Transformation matrices operate "right to left"
- Subtle points concerning HLSL/Cg will come up

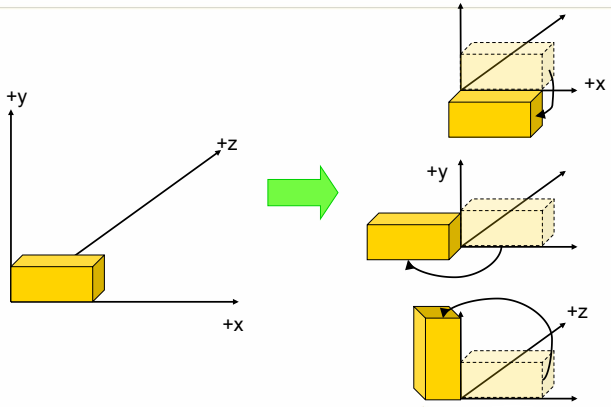
Transformation 2: scaling



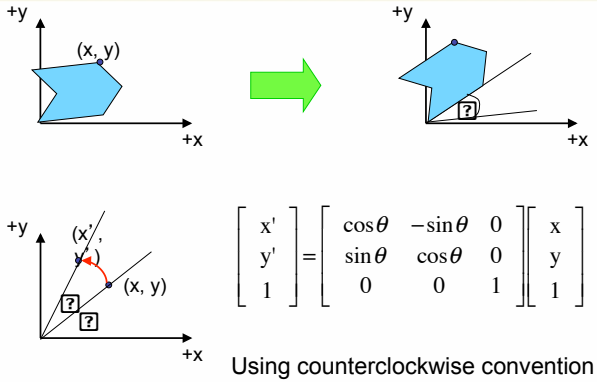
Scaling matrix

$$\begin{bmatrix} x_s \\ y_s \\ z_s \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Transformation 3: rotation



2D rotation



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

Using counterclockwise convention

3D rotation matrices (LHS)

Rotation along **Z** axis

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

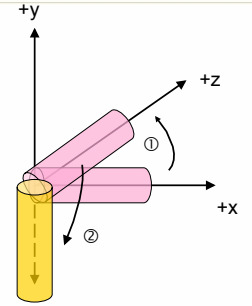
Rotation along **Y** axis

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

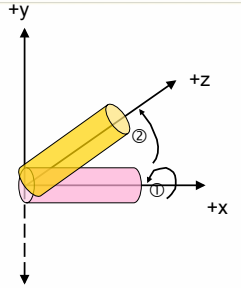
Rotation along **X** axis

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

Non-commutative property (1)

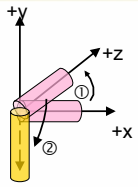


1. Clockwise 90° looking down y
2. Counter-CW 90° looking down x



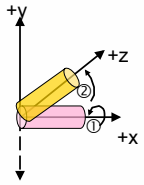
1. Counter-CW 90° looking down x
2. Clockwise 90° looking down y

Non-commutative property (2)



$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\frac{\pi}{2}) & -\sin(\frac{\pi}{2}) & 0 \\ 0 & \sin(\frac{\pi}{2}) & \cos(\frac{\pi}{2}) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(-\frac{\pi}{2}) & 0 & \sin(-\frac{\pi}{2}) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(-\frac{\pi}{2}) & 0 & \cos(-\frac{\pi}{2}) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & -1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

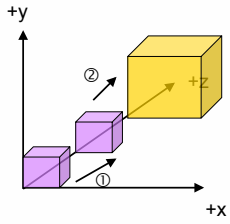
(x'', y'', z'') = (-z, -x, y)



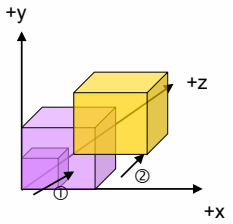
$$\begin{bmatrix} \cos(-\frac{\pi}{2}) & 0 & \sin(-\frac{\pi}{2}) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(-\frac{\pi}{2}) & 0 & \cos(-\frac{\pi}{2}) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\frac{\pi}{2}) & -\sin(\frac{\pi}{2}) & 0 \\ 0 & \sin(\frac{\pi}{2}) & \cos(\frac{\pi}{2}) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(x'', y'', z'') = (-y, -z, x)

Non-commutative property (3)

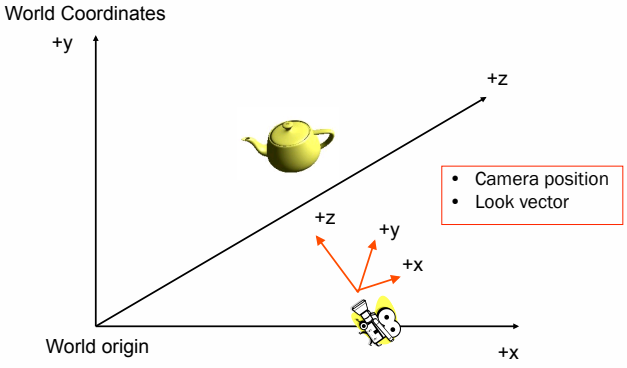


1. Translation by (x, y, z)
2. Scale by 2 times



1. Scale by 2 times
2. Translation by (x, y, z)

View transformation revisited



Specifying the view transformation

- Most commonly parameterized by:
  - Position of camera
  - Position of point to look at
  - Vector indicating "up" direction of camera
- In Direct3D: `D3DXMatrixLookAtLH`
  - D3D uses a LHS, but also have `D3DXMatrixLookAtRH`
- In XNA: `Matrix.CreateLookAt` (RHS)
- In OpenGL: `gluLookAt` (RHS)
- Can also build a rotation+translation matrix as if the camera was an object in scene, then take the inverse of that matrix

msdn.microsoft.com/en-us/library/bb205342(VS.85).aspx
msdn.microsoft.com/en-us/library/bb205343(VS.85).aspx

Using "LookAt" in Unity

```
// This complete script can be attached to a camera to make it
// continuously point at another object.

// The target variable shows up as a property in the inspector.
// Drag another object onto it to make the camera look at it.
using UnityEngine;
using System.Collections;

public class ExampleClass : MonoBehaviour {
    public Transform target;

    void Update() {
    // Rotate the camera every frame so it keeps looking at the target
        transform.LookAt(target);     // optional parameter for "up"
    }
}
```

docs.unity3d.com/ScriptReference/Transform.LookAt.html

Trouble with transforming normals

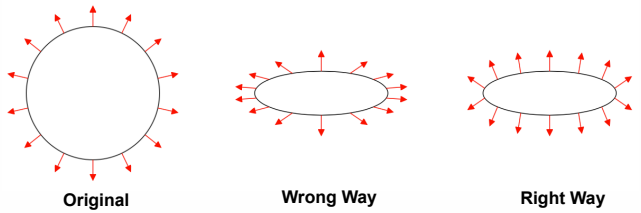


Image from Jason L. McKesson, "Learning Modern 3D Graphics Programming," 2012

http://alfonse.bitbucket.org/oldtut/Illumination/Tut09%20Normal%20Transformation.html#d0e9295

Transforming normals

Only using upper left 3x3 part of the 4x4 matrix:

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

$$(\mathbf{Wn}) \bullet (\mathbf{Mt}) = 0$$

$$(\mathbf{Wn})^T (\mathbf{Mt}) = 0$$

$$(\mathbf{n}^T \mathbf{W}^T)(\mathbf{Mt}) = 0$$

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

Since $\mathbf{n}^T \mathbf{t} = 0$ we can set $\mathbf{W}^T \mathbf{M} = \mathbf{I}$

$$\mathbf{W} = (\mathbf{M}^{-1})^T = \mathbf{M}^{-T}$$

From http://en.wikipedia.org/wiki/Normal_(geometry)#Transforming_normals