3D Transformations & Cameras

Drawing in 3D – The story so far

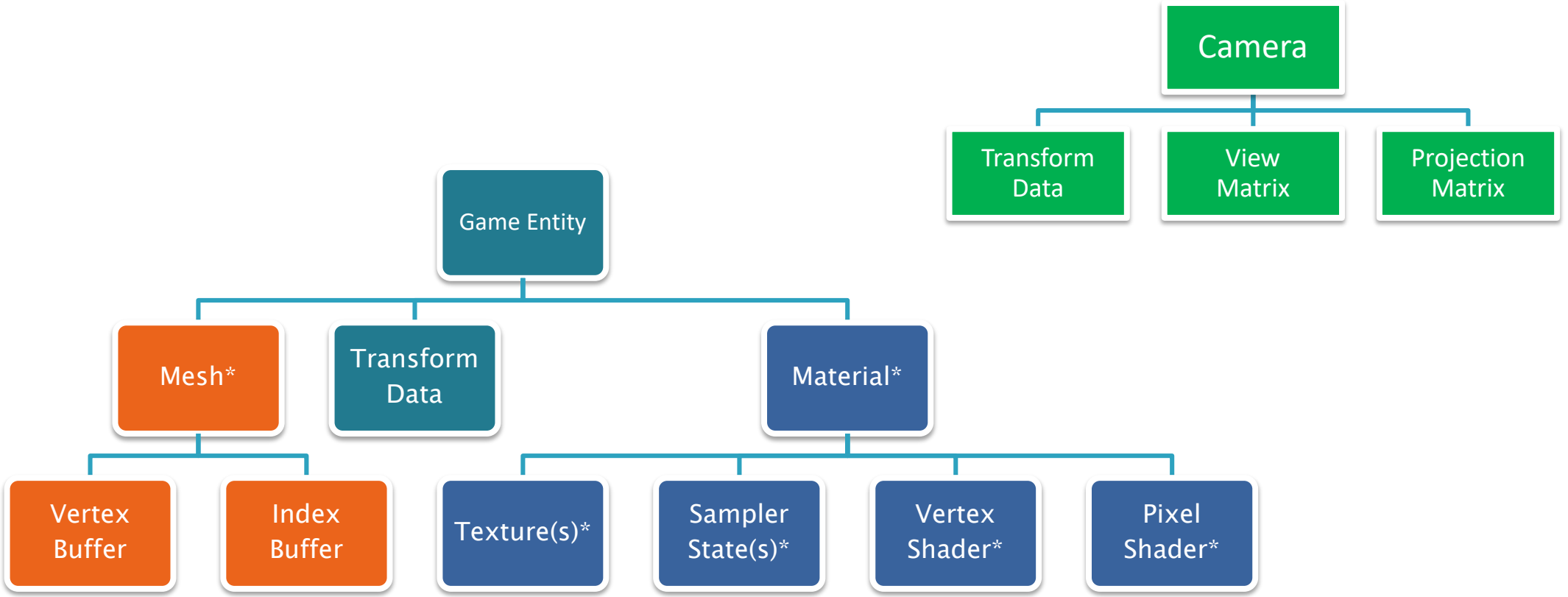
- Triangles & Vertices
  - The actual “stuff” we draw
  - Stored in vertex & index buffers
- The Rendering Pipeline
  - The steps the triangles go through to become pixels
- Representation of game entities
  - Visual component: mesh
  - Transformation info: matrices

What else do we need?

- A way to view the 3D space: The “camera”
  - Defines our viewpoint into the scene
  - Needs camera transformation & projection info – more matrices

- Material definitions
  - What the surface of a mesh looks like
  - Colors, textures, shaders, etc.

Where does everything go?



Drawing in 3D

A step back: Drawing in 2D

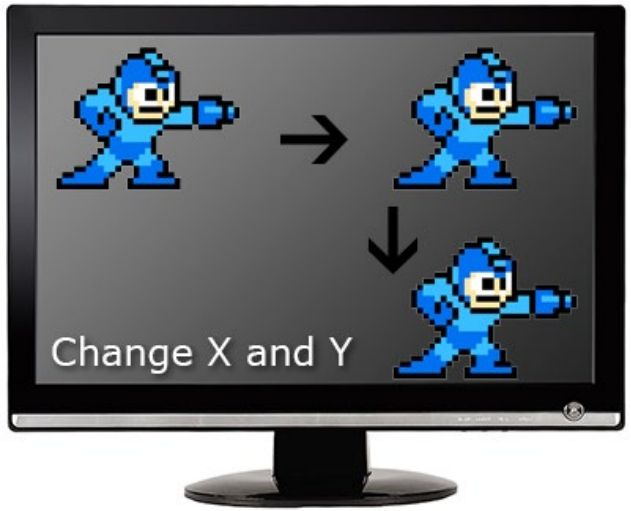
- You've probably done this before with…
  - MonoGame
  - Photoshop
  - A piece of paper
- It's easy to wrap your head around

Drawing in 2D

- Coordinate systems (x, y) match
  - Measurements (pixels) usually match as well

- You have a 2D surface
  - Your monitor
  - It has an origin (top left)

- You have 2D images
  - Bitmaps



Drawing in 2D

- Transformations are fairly easy to visualize

- Scaling
  - Your window has some amount of pixels
  - Your image has some amount of pixels
  - Adjust the relative size by scaling the image

- Rotating
  - Basically one way to rotate: around the “Z axis”
  - Like spinning a piece of paper on a desk

From 2D to 3D: Just one more D

- Some considerations:
  - We need to map a 3D scene to a 2D surface
  - Our “game space” isn’t measured in pixels anymore
- Where is the origin?
  - It’s at (0,0,0)! Well that doesn’t help…
  - It all depends on your perspective

2D screens vs. 3D worlds

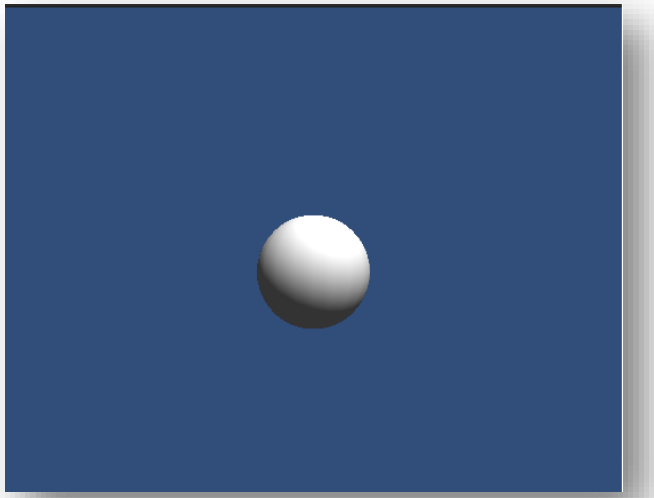
- Your screen cares about pixels
- But your 3D “world” does not
  - It cares about **units**
  - Can’t rely on pixel measurements anymore
- Ok, so how big is 1 unit?
  - It’s 1 unit big.
  - Great…

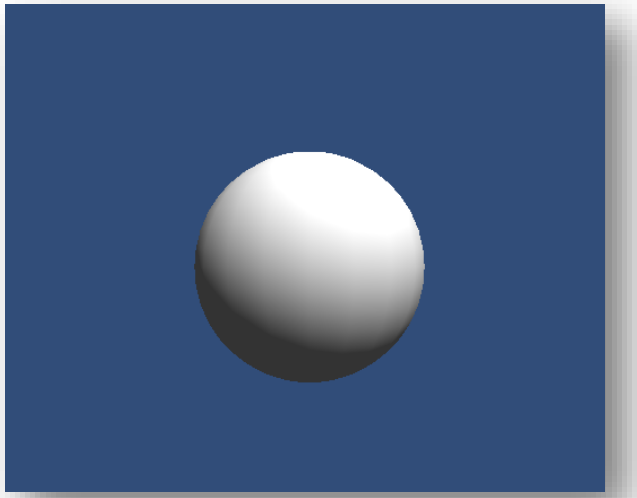
3D units

- What you might have meant was:
  - How big, in PIXELS, is 1 Unit?

- That depends…
  - How close are you to the object?
  - What's the object's scale?
  - What's the object's rotation?
  - How big is your viewport?
  - How wide is your field of view?
  - Is it a perspective or orthographic projection?
  - Where are you looking?

3D scene example





- Did the sphere move, or did the camera?
- Would the results be the same either way?

3D Transformations

3D transformations

- Each object usually has:
  - Translation
  - Rotation
  - Scale

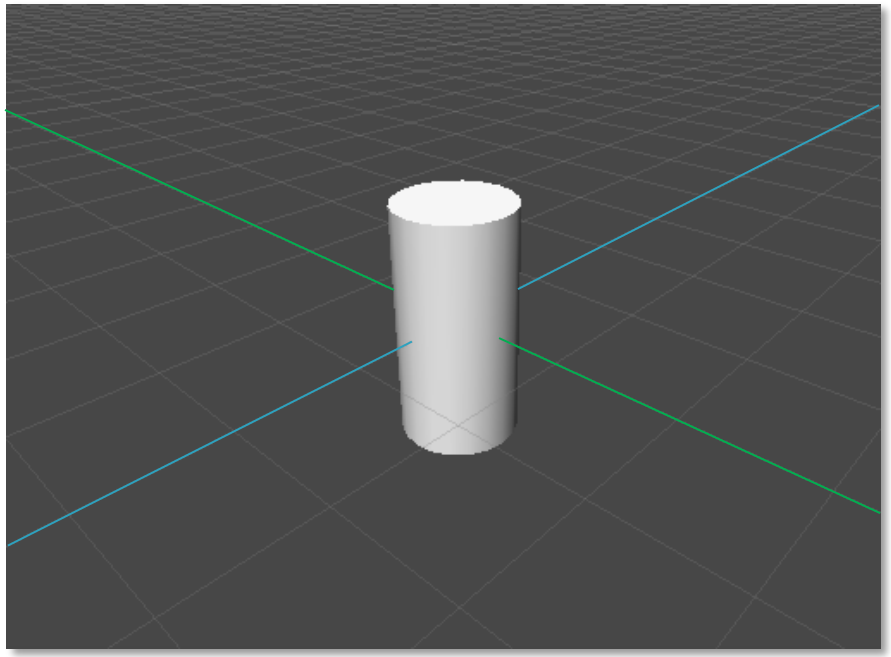
- You (the “Camera”) also needs:
  - Translation
  - Rotation
  - Mapping (projecting) from 3D back to 2D

Matrix math

- Matrices all the way down
  - Transformations can be represented with matrices
  - You can combine them to get more matrices!
  - Multiply them together

- The overall goal:
  - Get **a single matrix** that represents all of this math
    - Object's translation, rotation & scale
    - Camera's translation, rotation & projection
  - Apply it to each vertex of each triangle of a mesh
  - Each vertex will then be in 2D "screen" coordinates

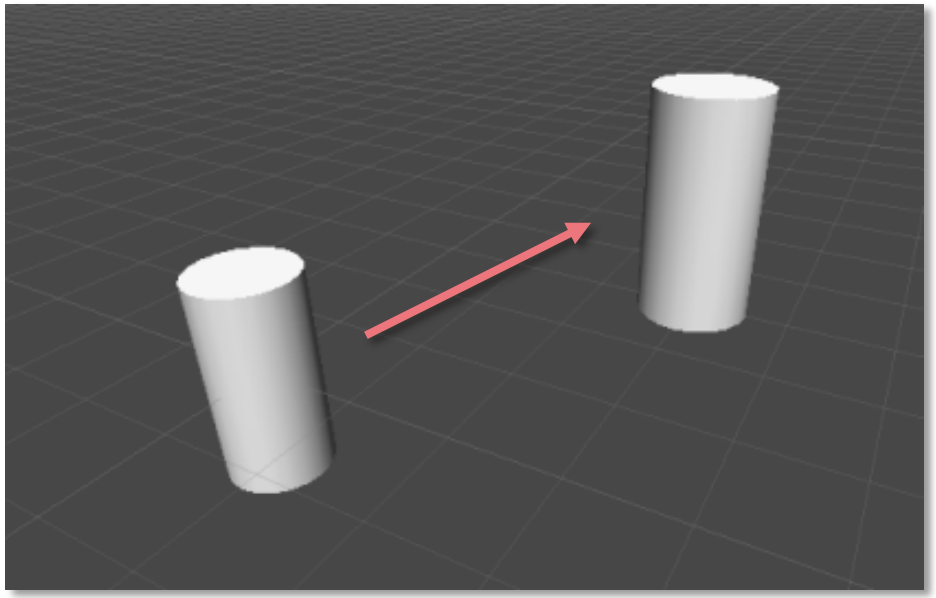
Local space



- Here's an object at the origin
- All of its vertices are relative to its center
  - Basically its own origin
- This is called "local" or "object" space

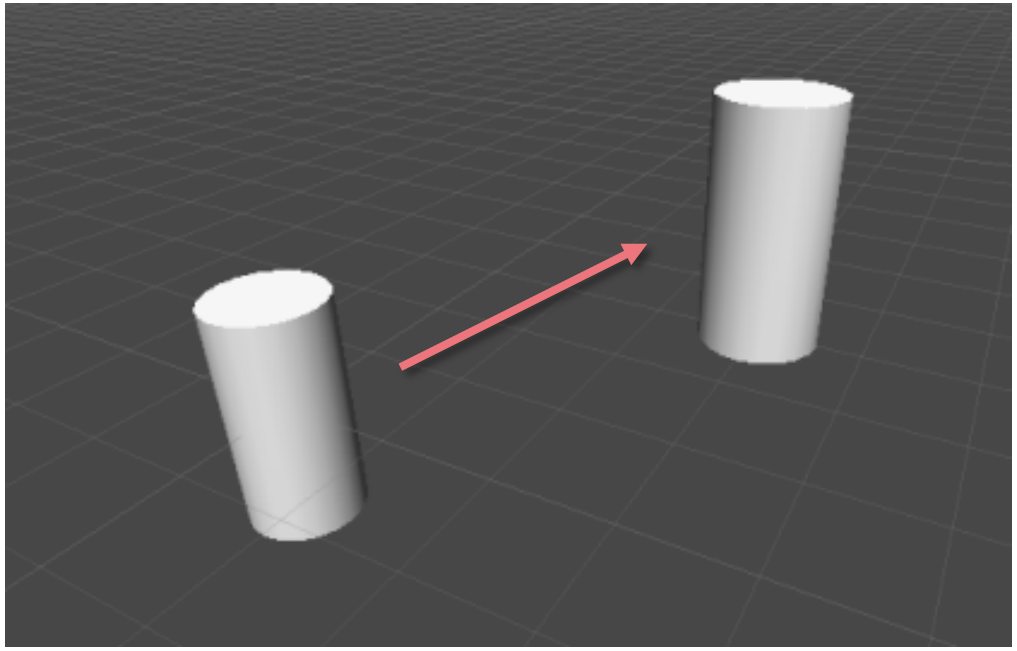
World space – Translation

- Moving the object to a different location
- Move all vertices by same amount
  - The same (x, y, z)
- If all vertices have the same translation, it appears as if the entire object has moved



World space – Translation matrix

- Use a Translation Matrix
  - Will translate (move) a vertex
  - We'll apply this to ALL vertices
- Vertices now in "world" space

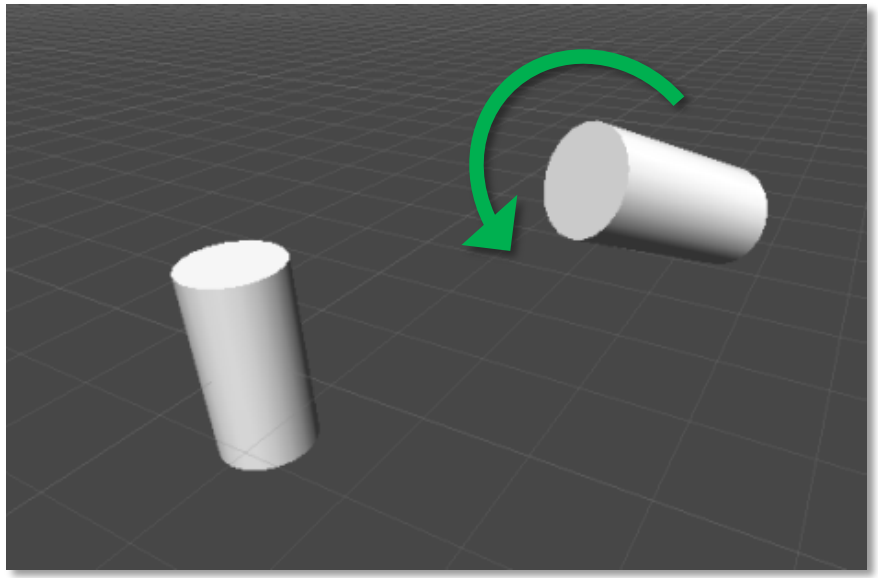


Translation Matrix
Move by (x,y,z)

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x & y & z & 1 \end{matrix}$$

World space – Rotation

- Might also want to rotate object
- We need to rotate **all** vertices around a one or more axes
- Generally done one axis at a time



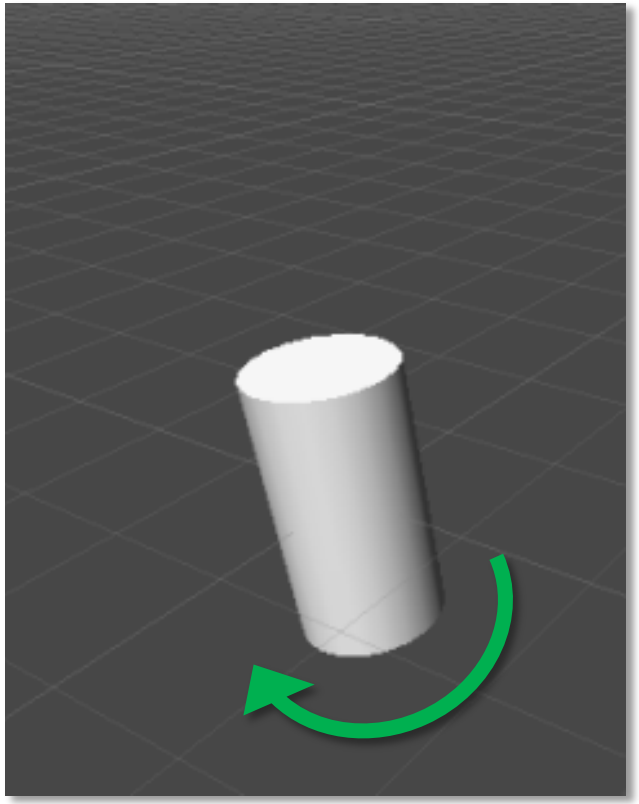
- Or with a more robust rotation representation (quaternions)

World space – Rotation matrix

- Will need a matrix to handle rotation
- Since we need rotation in our final world matrix for the shader

Rotation Matrix
Rotation 45° around Y

$$\begin{bmatrix} .707 & 0 & .707 & 0 \\ 0 & 1 & 0 & 0 \\ -.707 & 0 & .707 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



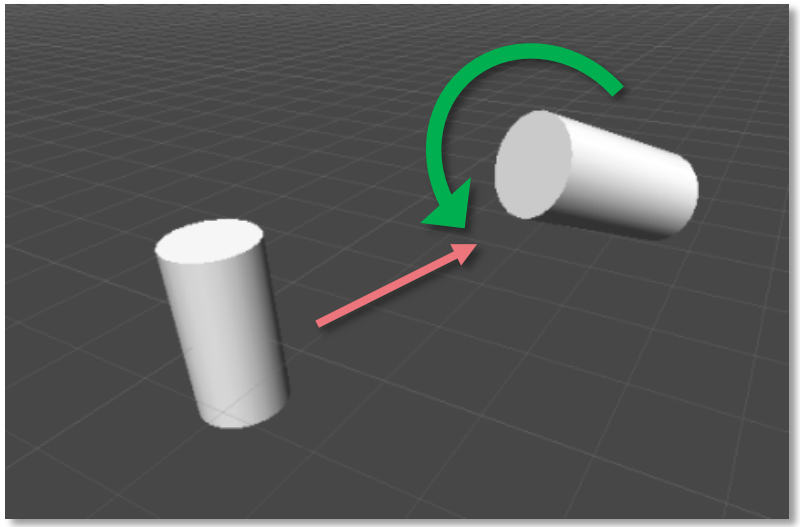
World space – Translation & rotation combined

- Combine (multiply) Translation and Rotation matrices
  - Resulting matrix represents both transformations

- This is still “world” space!

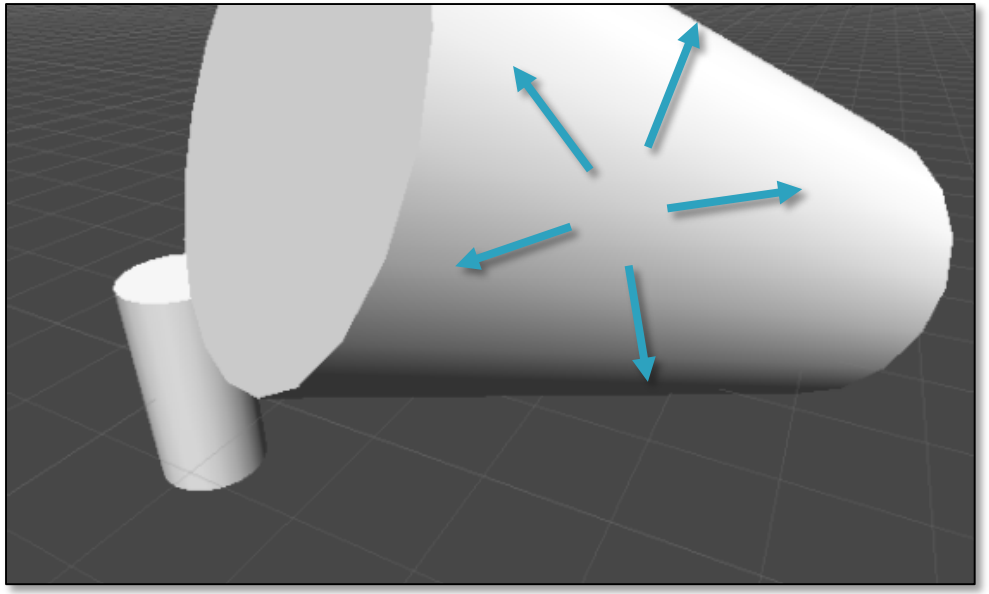
Combined Matrix
Rotation 45° around Y
and move by (x,y,z)

$$\begin{bmatrix} .707 & 0 & .707 & 0 \\ 0 & 1 & 0 & 0 \\ -.707 & 0 & .707 & 0 \\ x & y & z & 1 \end{bmatrix}$$



World space – Scale

- Let's also scale the object
- Move all of the vertices towards or away from the origin

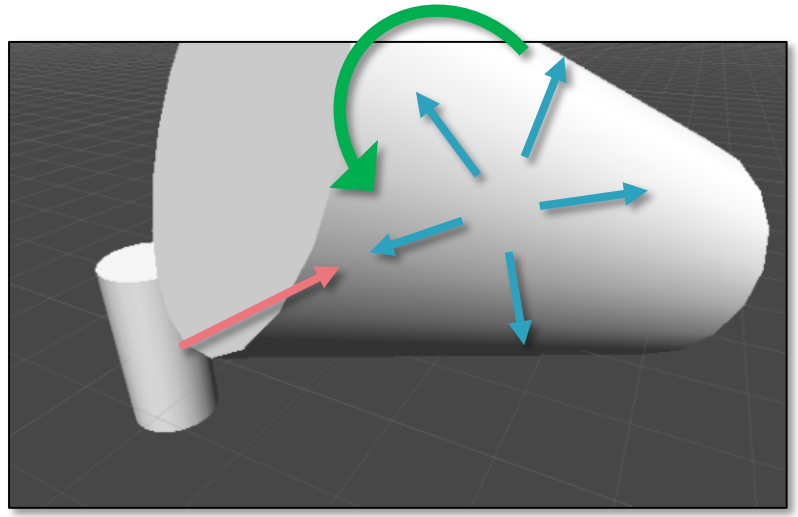


World space – Scale matrix

- Use a Scale Matrix
- Represents the amount to “stretch” on each axis

Scale Matrix
Scale by (2,3,4)

$$\begin{matrix} 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

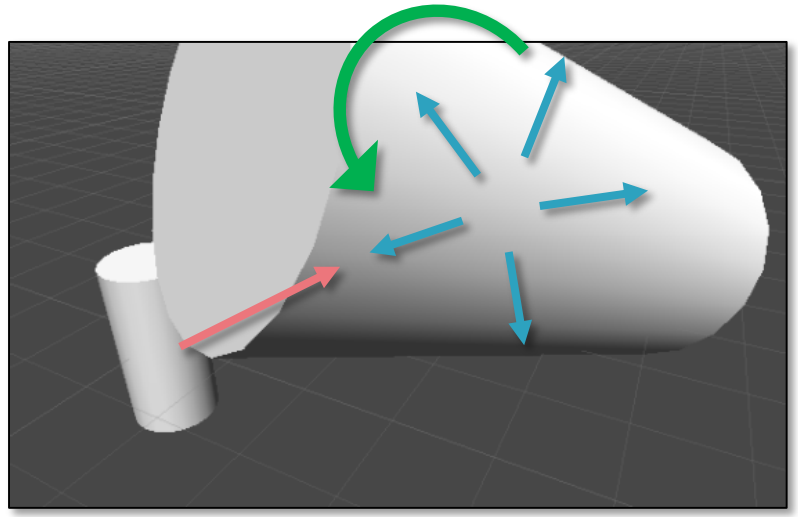


World space – All matrices combined

- Combine all 3 (Translation, Rotation & Scale)
  - One matrix representing full “world” transformation
- Still in “world” space

Final World Matrix
Contains translation (x,y,z),
rotation (45° around Y)
and scale (2,3,4)

$$\begin{matrix} 1.414 & 0 & 1.414 & 0 \\ 0 & 3 & 0 & 0 \\ -2.828 & 0 & 2.828 & 0 \\ x & y & z & 1 \end{matrix}$$



Matrix multiplication reminder

- A quick reminder about Matrix multiplication:
  - Not commutative
  - Order matters!

- (T * R * S) is not the same as (S * R * T)

- Each transformation will be affected by the other matrices
  - Scale by 2, Move by 1 = 2x size, center moved by 1
  - Move by 1, Scale by 2 = 2x size, center moved by 2

What's the best matrix mul order?

- We have three matrices to multiply
- Which order?
  - T * R * S ?
  - T * S * R ?
  - S * R * T ?
  - S * T * R ?
  - R * S * T ?
  - R * T * S ?
- Each has a different result!

Let's look at T and S

Translation Matrix
Move by (5,6,7)

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 5 & 6 & 7 & 1 \end{matrix}$$

Scale Matrix
Scale by (2,3,4)

$$\begin{matrix} 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

T * S
Translation is SCALED

$$\begin{matrix} 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 10 & 18 & 28 & 1 \end{matrix}$$

S * T
Translation is NOT scaled

$$\begin{matrix} 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 5 & 6 & 7 & 1 \end{matrix}$$

Which is better?

T * S
Translation is SCALED

$$\begin{matrix} 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 10 & 18 & 28 & 1 \end{matrix}$$

S * T
Translation is not scaled

$$\begin{matrix} 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 5 & 6 & 7 & 1 \end{matrix}$$

- Want our basic transforms to be predictable
- S * T is ideal here
  - Translation is not affected by scale
  - Object will end up exactly where we expect!

Rotation vs. translation?

- As we just saw:
  - T * S means translation is scaled
  - S * T means translation is not scaled
- Rotation vs. Translation?
  - T * R means translation is rotated
  - R * T means translation is not rotated
- Ideally, T comes last in our multiplication
  - __ * __ * T

Reminder: Rotation matrix = 3 axes

- In a 4x4 matrix
  - Top-left 3x3 portion is rotation
  - Each row corresponds to an axis

X axis	1	0	0	0
Y axis	0	1	0	0
Z axis	0	0	1	0
	5	6	7	1

How about rotation vs. scale?

Rotation Matrix
Rotation 45° around Y

$$\begin{matrix} .707 & 0 & .707 & 0 \\ 0 & 1 & 0 & 0 \\ -.707 & 0 & .707 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

Scale Matrix
Scale by (2,3,4)

$$\begin{matrix} 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

R * S
Axes do not match anymore
(Point in different directions)

$$\begin{matrix} 1.414 & 0 & 2.828 & 0 \\ 0 & 3 & 0 & 0 \\ -1.414 & 0 & 2.828 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

S * R
Axes still match!
(Point in same direction)

$$\begin{matrix} 1.414 & 0 & 1.414 & 0 \\ 0 & 3 & 0 & 0 \\ -2.828 & 0 & 2.828 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

How about rotation vs. scale?

Rotation Matrix
Rotation 45° around Y

$$\begin{bmatrix} .707 & 0 & .707 & 0 \\ 0 & 1 & 0 & 0 \\ -.707 & 0 & .707 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scale Matrix
Scale by (2,3,4)

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

R * S
Axes do not match anymore
(Point in different directions)

$$\begin{bmatrix} 1.414 & 0 & 2.828 & 0 \\ 0 & 3 & 0 & 0 \\ -1.414 & 0 & 2.828 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

S * R
Axes still match!
(Point in same direction)

$$\begin{bmatrix} 1.414 & 0 & 1.414 & 0 \\ 0 & 3 & 0 & 0 \\ -2.828 & 0 & 2.828 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Which is better?

R * S
Axes do not match anymore
(Point in different directions)

$$\begin{bmatrix} 1.414 & 0 & 2.828 & 0 \\ 0 & 3 & 0 & 0 \\ -1.414 & 0 & 2.828 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

S * R
Axes still match!
(Point in same direction)

$$\begin{bmatrix} 1.414 & 0 & 1.414 & 0 \\ 0 & 3 & 0 & 0 \\ -2.828 & 0 & 2.828 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- S * R is ideal
  - Scale won't affect our axes
  - More predictable results!

Altogether now

- S * T is ideal – don't scale the translation
- R * T is ideal – don't rotate the translation
- S * R is ideal – keep axes the same

- What combination satisfies all these orders?
  - S before T
  - R before T
  - S before R

- S * R * T

Matrix multiplication is associative

- *(A * B) * C* is the same as *A * (B * C)*
- Even though it is not commutative
- What does this mean for us?
  - (S * R) * T == S * (R * T)
  - Doesn't matter how you group them

Transformation Representation

Matrices & entities

- Each *game entity* needs its own transformations matrices
  - But NOT each mesh!
  - Since a single mesh can be shared among multiple entities
- Multiply transforms together in C++ to get a single matrix
  - The “world” matrix
  - Represents all transformations on a single entity
- Send that single world matrix to the shader when drawing

Reminder: Storing transformation data

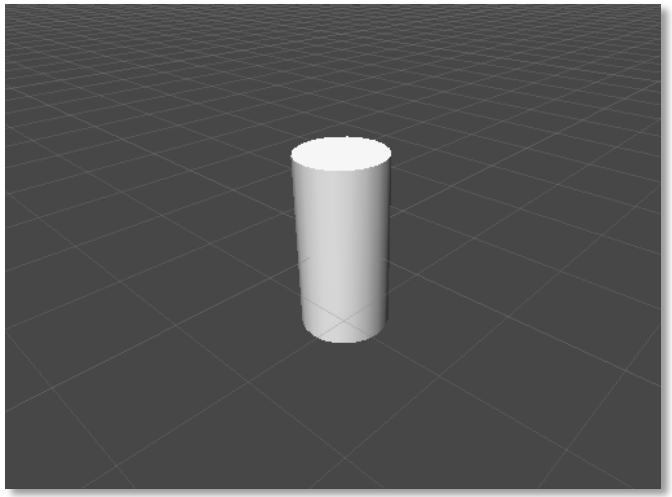
- Store your transformations separately from your world matrix
  - Position – X, Y, Z
  - Rotation – X, Y, Z (or yaw, pitch, roll)
  - Scale – X, Y, Z
- Recompute world matrix from this data
  - Do NOT “decompose” old matrix each frame
- Wrap this up into a Transform class!

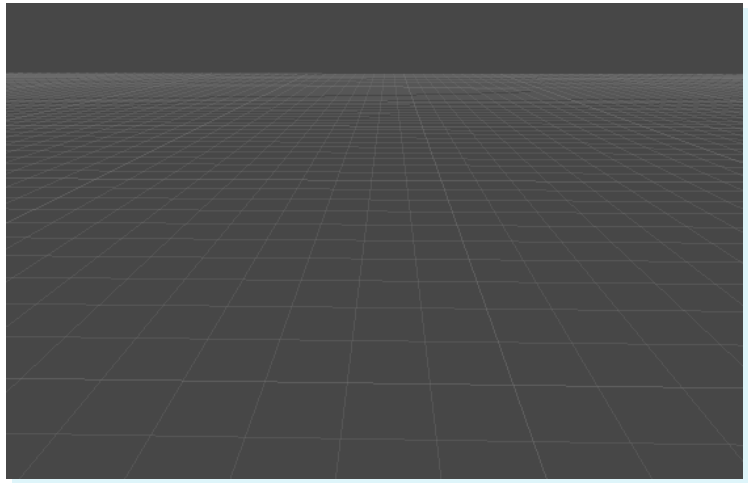
3D Cameras

What about *our* movement?

- What happens if our view point changes?

- The object should stay still in the WORLD
- But we might not see it on the SCREEN





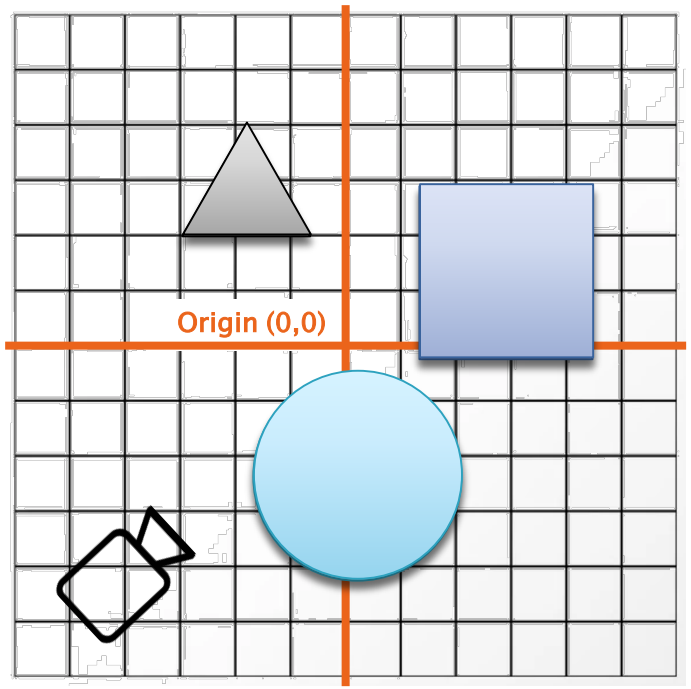
3D cameras

- Represents the viewer of the 3D scene
  - Defines where “we” are in the scene
  - And the direction we’re facing
- A 3D camera needs:
  - A position
  - An orientation (rotation)
- Represented by a camera’s *View Matrix*

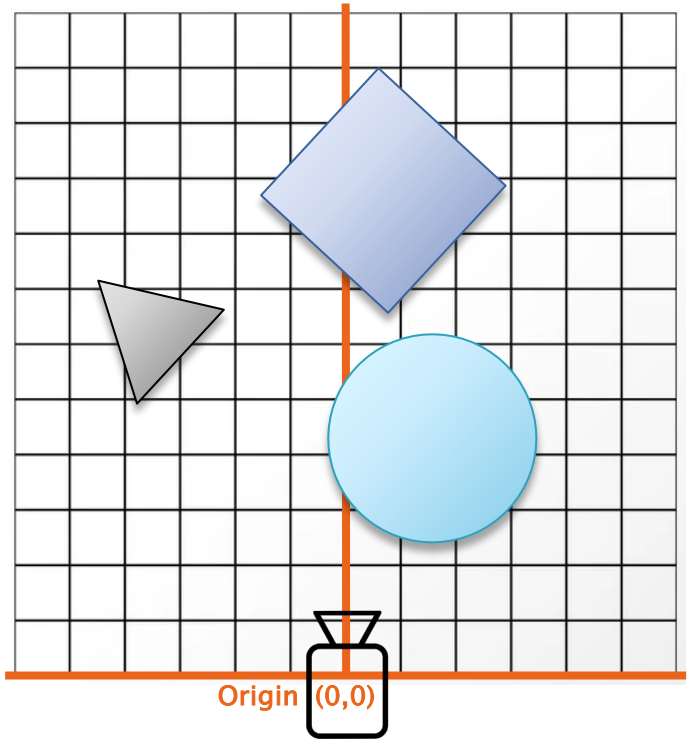
Camera space

- World matrix gets vertices into world space
  - Relative to the world's origin
- We now need them relative to our viewpoint
- We want the **camera** to now be the **origin**
  - Adjust all positions so they're measured *from the camera!*
  - Adjust rotations so the camera is looking along the Z axis
- We'll create a matrix that does this

Camera space – Example



World Space



Camera Space

View Matrix

View matrix

- Matrix that converts from world space to camera space

- Essentially applies a “reverse transformation”
  - Example: Camera is at (x, y, z), but we want it at (0,0,0)
  - If we move everything (-x, -y, -z), we’ll be back at (0,0,0)
  - Apply the same transformation to all vertices we draw!

View Matrix (without rotation)

Camera is at (x,y,z)

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x & -y & -z & 1 \end{matrix}$$

Creating a view matrix

- We could do it manually (ugh) or use DirectX Math
- Two main ways of defining a view matrix:
  - **Look At** (a point in space)
  - **Look To** (along a particular vector)
- Can use either method
  - Two different ways of defining a “direction” for the camera
  - Each produces a valid view matrix

“LookAt” view matrix – DirectX Math

- `XMMatrixLookAtLH()`
- 3 Parameters:
  - EyePosition – Camera’s location in the world
  - FocusPosition – A point in space to *look at*
  - UpDirection – Which way is up for the camera?
- Parameters describe looking at a particular point in space (focus) from another point in space (eye)
- Returns a left-handed View Matrix (+Z is forward)

“LookTo” view matrix

- `XMMatrixLookToLH()`
- 3 Parameters:
  - EyePosition – Camera’s location in the world
  - EyeDirection – Direction to look *(not a position in space!)*
  - UpDirection – Which way is up for the camera?
- Parameters describe looking *in a particular direction* from a specific point in space (eye position)
- Returns a left-handed View Matrix (+Z is forward)

Combining World & View

- Multiply the World and View matrices
- Result is cleverly named “World View” matrix
- Single matrix that transforms *local* → *world* → *camera* space

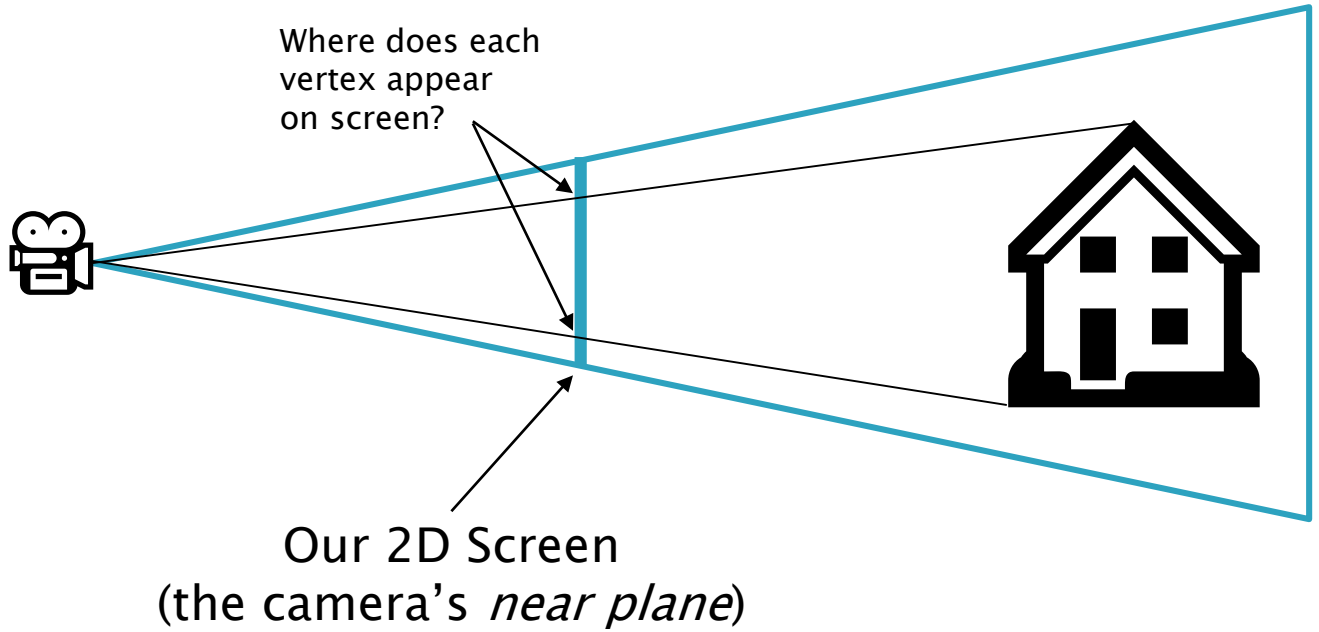
Projection Matrix

Screen space

- We have vertices somewhere in the world
  - World Matrix
- We know where “we” are
  - View Matrix
- How do we map back into 2D?
  - Need to get the vertices into “screen space”
  - We need a Projection Matrix

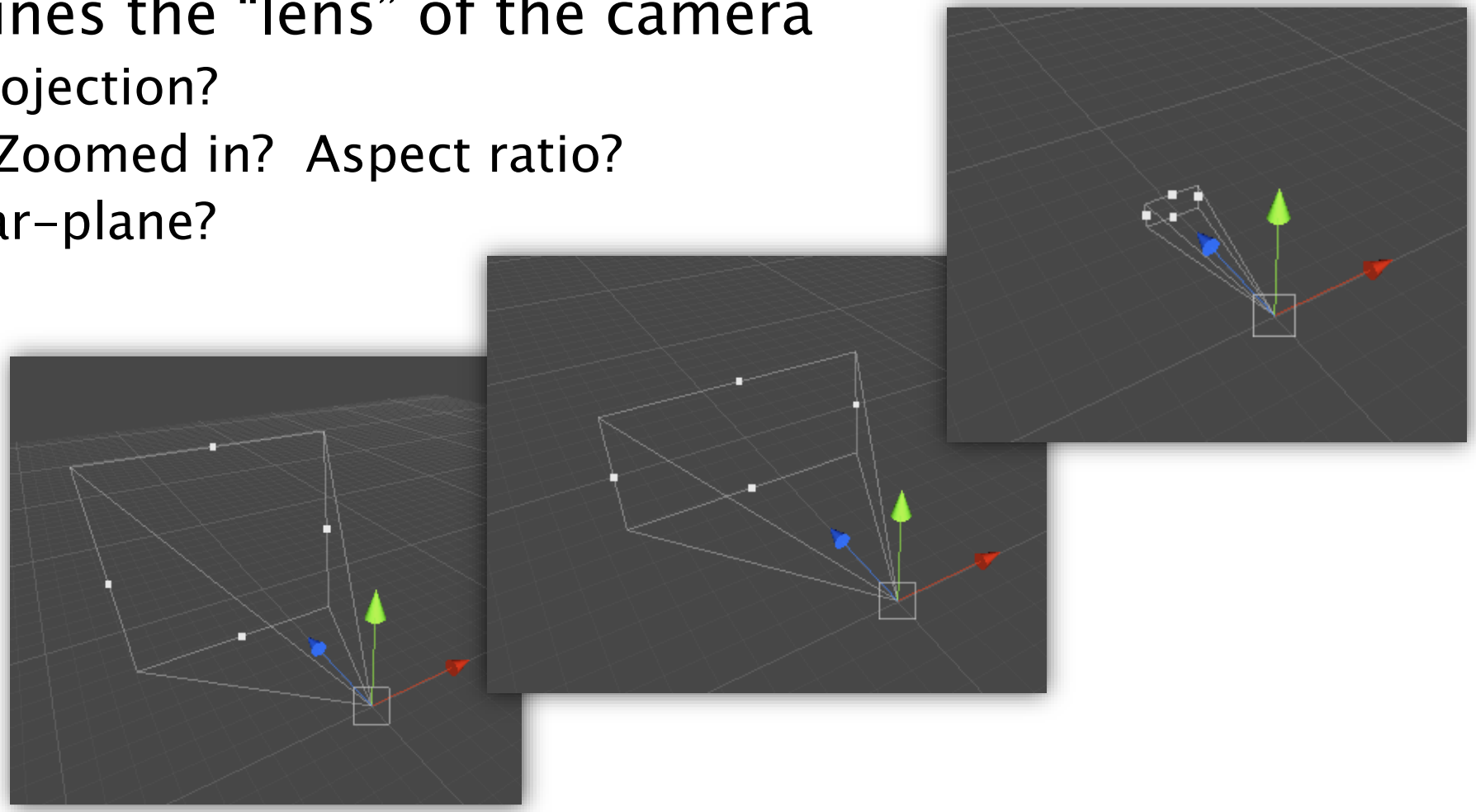
Projecting 3D back to 2D

- We have a 3D representation of geometry
- We need to know what it looks like on our screen (a 2D plane)



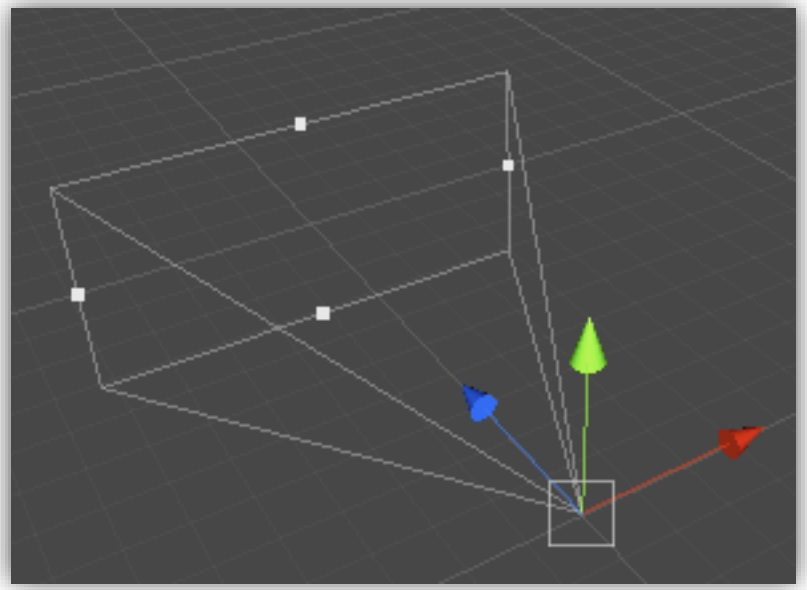
Projection matrix

- Essentially defines the “lens” of the camera
  - Shape of the projection?
  - Wide-angled? Zoomed in? Aspect ratio?
  - Near-plane? Far-plane?

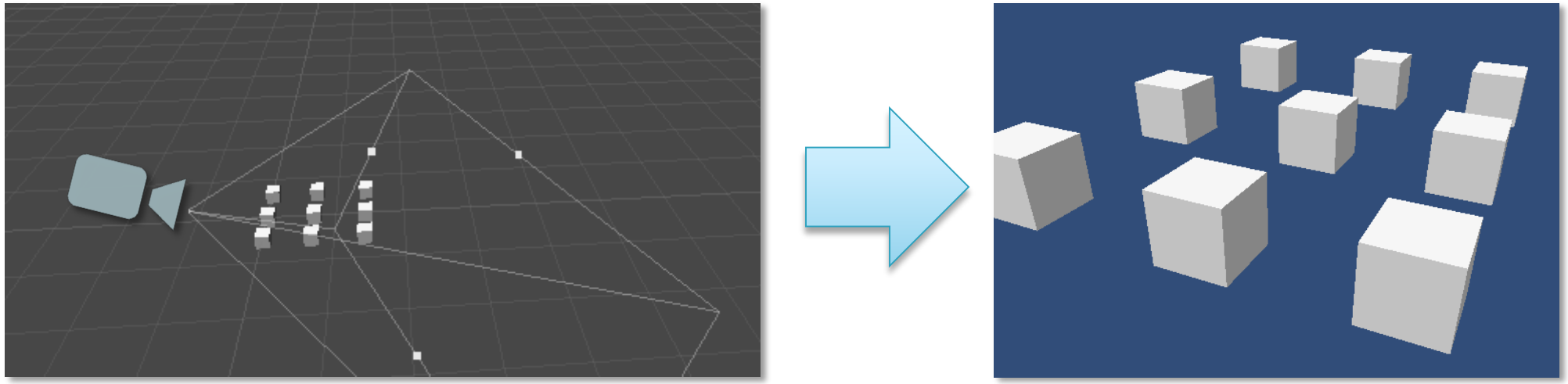


Perspective projections

- Similar to how our eyes work
- Closer objects appear to be larger
- Area the camera can "see" is trapezoidal



Perspective projection – Example

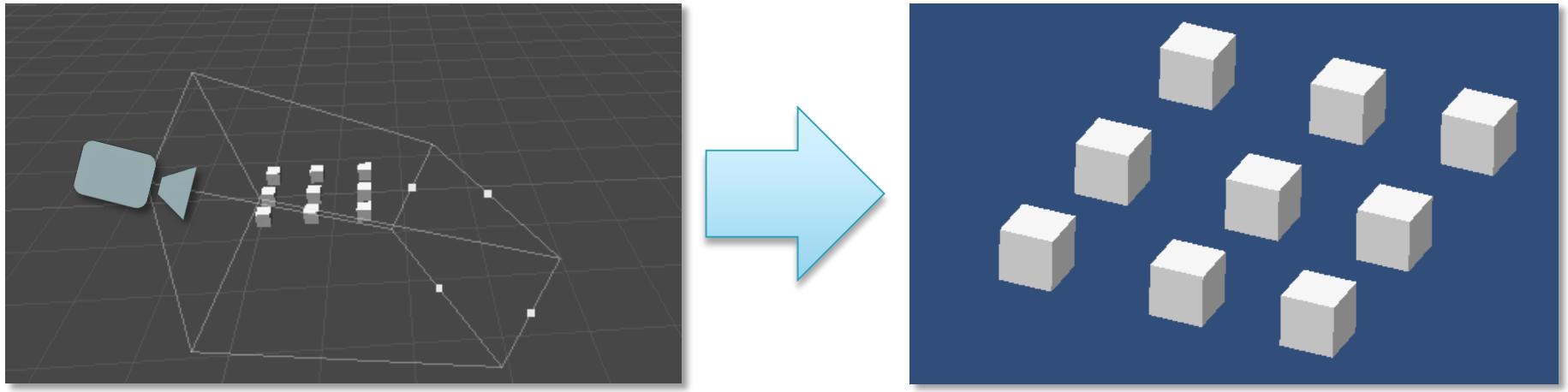


- Objects closer to the camera appear larger

Orthographic projections

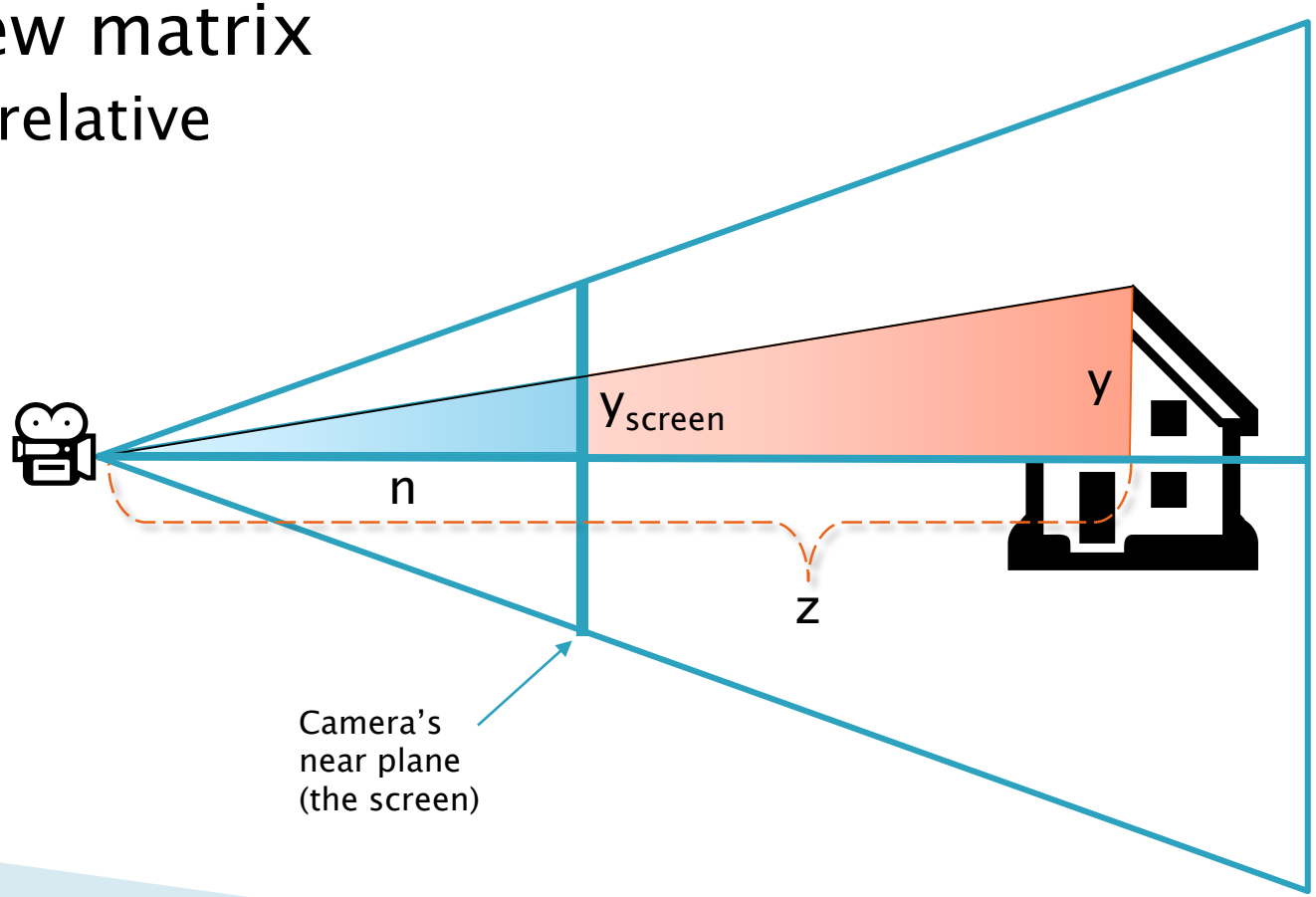
- Distance doesn't affect relative size – unlike our eyes!

- Area the camera can "see" is rectangular
  - Essentially, the "rays" that hit the camera are parallel



How does projection work?

- Find each vertex's (x, y) position on screen
- Done after applying view matrix
  - So everything is camera-relative
- Example: Find y_{screen}



Solving for y_{screen}

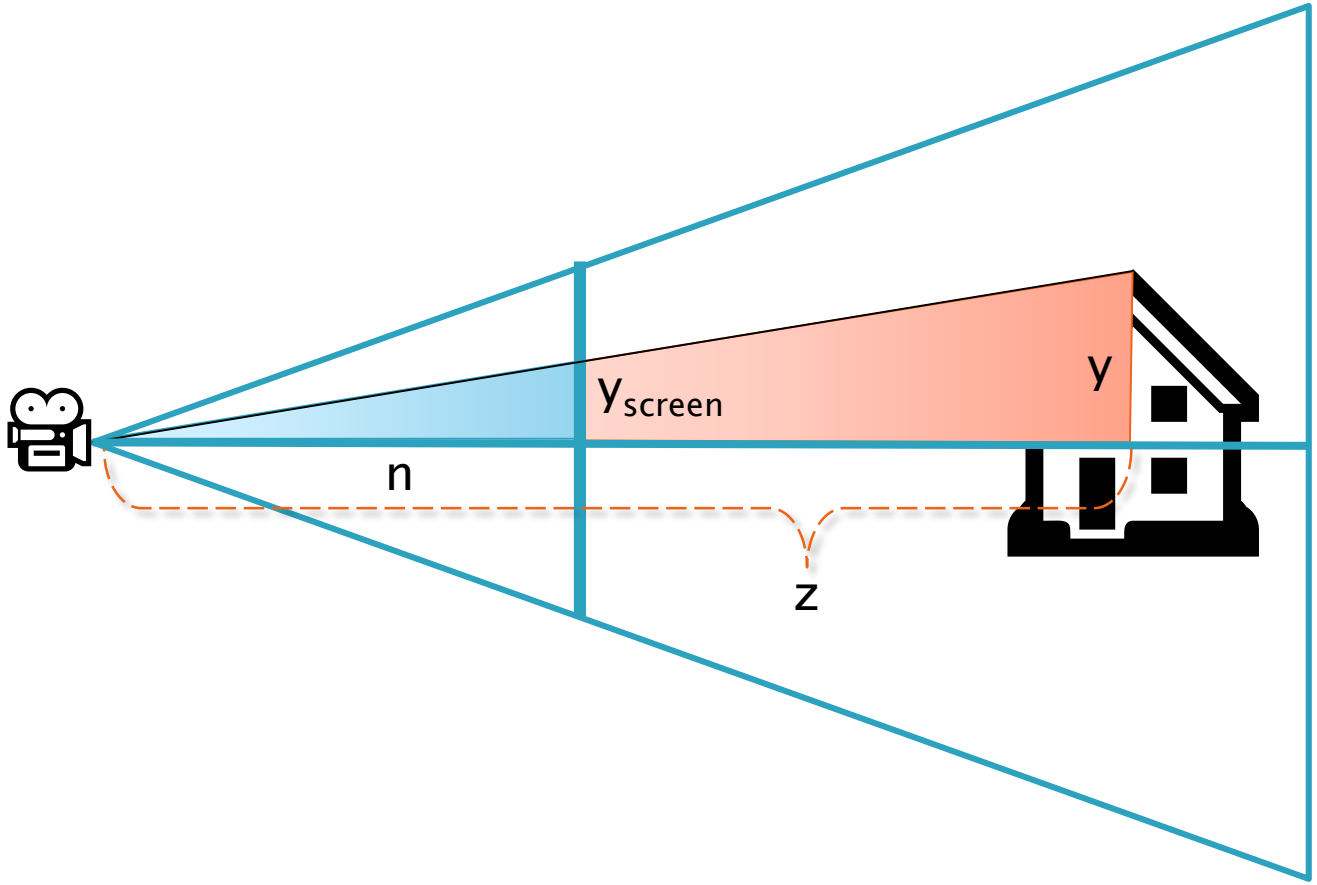
- Seems easy enough!

$$\frac{y_{screen}}{n} = \frac{y}{z}$$

$$y_{screen} = \frac{ny}{z}$$

- X is similar

$$x_{screen} = \frac{nx}{z}$$



Now do it with a matrix

- What matrix will give us the following result?

$$\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} nx/z \\ ny/z \\ z \\ 1 \end{bmatrix}$$

- Uh-oh…there's a problem

There *isn't* a matrix that does this

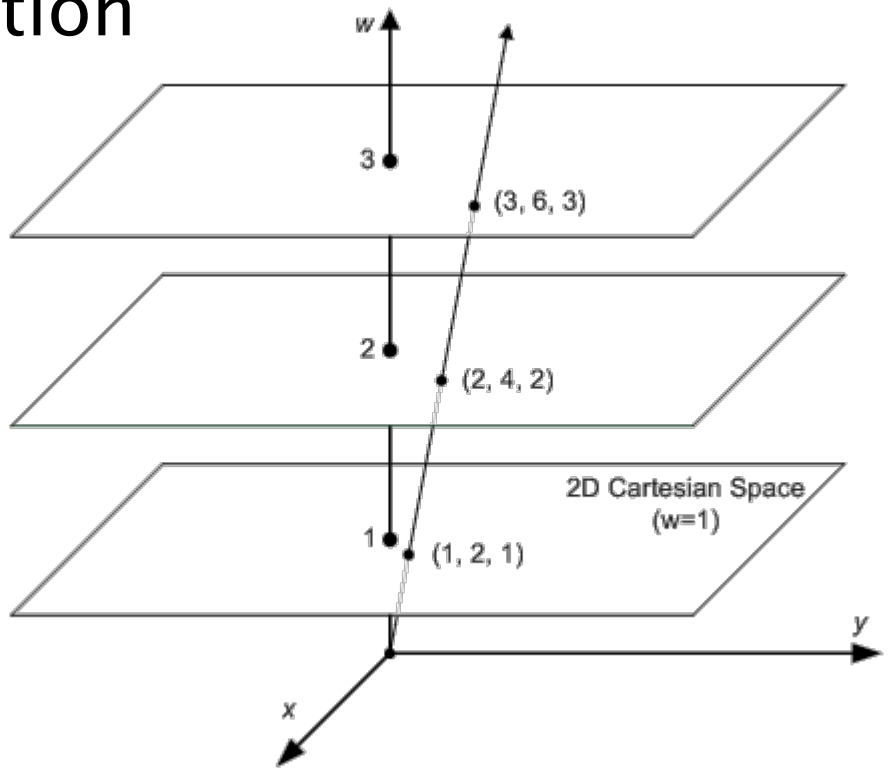
$$\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} nx/z \\ ny/z \\ z \\ 1 \end{bmatrix}$$

Doesn't work!

- So…how do we handle it?
- Homogenous coordinates

Homogenous coordinates

- Also known as projective coordinates
- A way of defining points within a projection
  - All points along a projected “ray” are considered to be the same
  - They all end up representing the same point after the projection



What do homogenous coordinates look like?

- Contain an extra component: *w*

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

- To convert to Cartesian coordinates, divide components by w

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \Leftrightarrow \begin{bmatrix} x/w \\ y/w \\ z/w \end{bmatrix}$$

Example homogenous coordinates

- Divide x,y,z by w:

$$\begin{bmatrix}2\\3\\4\\1\end{bmatrix} \Rightarrow \begin{bmatrix}2/1\\3/1\\4/1\end{bmatrix} \Rightarrow \begin{bmatrix}2\\3\\4\end{bmatrix}$$

Homogenous point

$$\begin{bmatrix}4\\6\\8\\2\end{bmatrix} \Rightarrow \begin{bmatrix}4/2\\6/2\\8/2\end{bmatrix} \Rightarrow \begin{bmatrix}2\\3\\4\end{bmatrix}$$

Homogenous point

- Homogenous points (2,3,4,1) and (4,6,8,2)…
  - Represent the same Cartesian point
  - They're *homogenous!*

$$\begin{bmatrix}2\\3\\4\\1\end{bmatrix} = \begin{bmatrix}4\\6\\8\\2\end{bmatrix} = \begin{bmatrix}20\\30\\40\\10\end{bmatrix} = \begin{bmatrix}2\\3\\4\end{bmatrix}$$

Ok, cool…how does that help us?

- What we're trying to accomplish →

$$\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} nx/z \\ ny/z \\ z \\ 1 \end{bmatrix}$$

- Let's assume the result is a homogenous coordinate
  - We'll need to perform a division by w afterwards
  - We really want to divide by z
  - Put z in the w component!
  - Is there a matrix to do this?

$$\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ ? \\ z \end{bmatrix} = \begin{bmatrix} nx/z \\ ny/z \\ ?/z \end{bmatrix}$$

Filling in the projection matrix: x,y scale

$$\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ ? \\ z \end{bmatrix} = \begin{bmatrix} nx/z \\ ny/z \\ ?/z \end{bmatrix}$$

- Where do we put *n* so that we get nx and ny?

$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ ? & ? & ? & ? \\ ? & ? & ? & ? \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ ? \\ z \end{bmatrix} = \begin{bmatrix} nx/z \\ ny/z \\ ?/z \end{bmatrix}$$

Filling in the projection matrix: moving z

$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ ? & ? & ? & ? \\ ? & ? & ? & ? \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ ? \\ z \end{bmatrix} = \begin{bmatrix} nx/z \\ ny/z \\ ?/z \end{bmatrix}$$

- How do we get the z in the last component?

$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ ? & ? & ? & ? \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ ? \\ z \end{bmatrix} = \begin{bmatrix} nx/z \\ ny/z \\ ?/z \end{bmatrix}$$

Projection matrix: What about Z itself?

$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ ? & ? & ? & ? \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ ? \\ z \end{bmatrix} = \begin{bmatrix} nx/z \\ ny/z \\ ?/z \end{bmatrix}$$

- After the divide, want 3rd component to still be z
  - What can I divide by z to get z?

    $?/z = z$
    $? = z^2$

  - How do we square z?

$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & ? & ? \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ z^2 \\ z \end{bmatrix} = \begin{bmatrix} nx/z \\ ny/z \\ z^2/z \end{bmatrix}$$

Projection matrix: Squaring z

$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & m_1 & m_2 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ z^2 \\ z \end{bmatrix} = \begin{bmatrix} nx/z \\ ny/z \\ z^2/z \end{bmatrix}$$

- What values do we use for m_1 and m_2?
- Result is a quadratic equation: $m_1 * z + m_2 = z^2$
  - Solve for two finite points
  - Near plane and far plane!

Solving for m_1 and m_2

$$m_1 z + m_2 = z^2$$

for $z = n$, $z = f$

① $m_1 n + m_2 = n^2$

② $m_1 f + m_2 = f^2$

$$m_1 = f + n$$

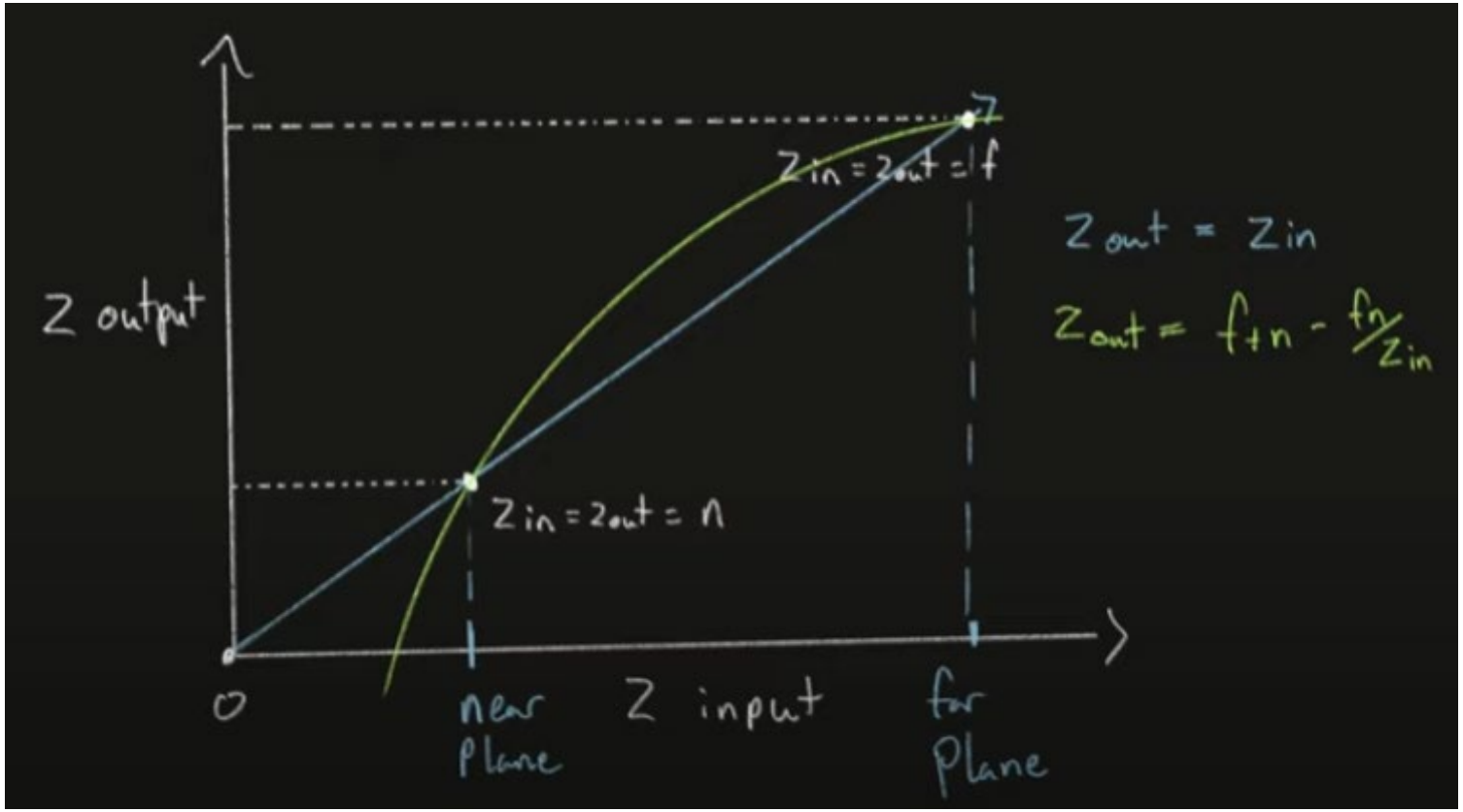
$$m_2 = -fn$$

Resulting perspective projection matrix

- This will give us the exact results we expect *at the clip planes*

$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & (f+n) & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ (f+n)z - fn \\ z \end{bmatrix}$$

Note: Z is not linear!



Projection matrices – DirectX Math

- Example functions for defining a Projection Matrix

- `XMMatrixPerspectiveLH()`
- `XMMatrixPerspectiveFovLH()`
- `XMMatrixOrthographicLH()`

- Several others exist as well

Creating a Perspective Projection

- `XMMatrixPerspectiveFovLH()` – 4 Params:
  - FovAngleY – Top-down field of view angle *(in radians)*
  - AspectRatio – Aspect ratio of view space X:Y
  - NearZ – Distance to the near clipping plane
  - FarZ – Distance to the far clipping plane

- Details
  - AspectRatio should always match window's ratio
  - Changing field of view angle can simulate "zooming"
  - Please don't make FoV crazy high in your assignments

Storing data – Cameras

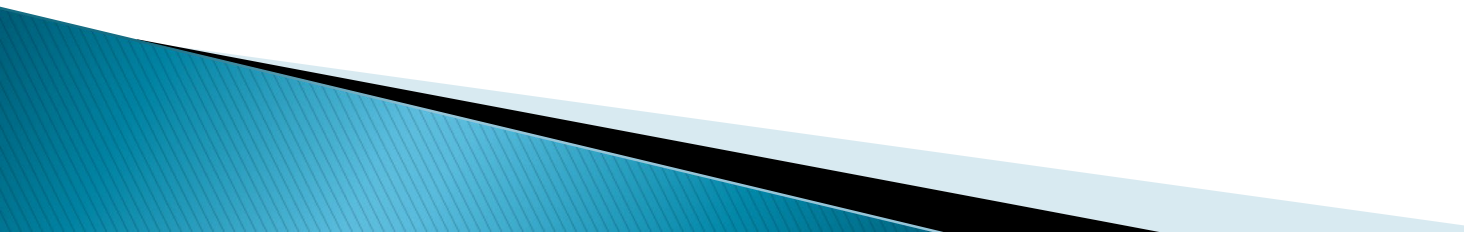
- View Matrix: Same as entities
  - Store raw position & direction
  - Update view matrix any time the camera moves/rotates
- EZ Mode:
  - Use a Transform object for camera, too
  - Use Transform's data to make view matrix
- Projection doesn't change often
  - Update aspect ratio when resizing the window
  - Update field of view if zooming

Screen space

- Ok, I have a projection matrix. Now what?
  - Combine all three: World * View * Projection
- Apply WVP matrix to each vertex's *position*
  - This should ***always*** happen in the Vertex Shader
- The vertices will now be in "screen space"
  - X & Y will be a screen position
  - Z will represent the depth

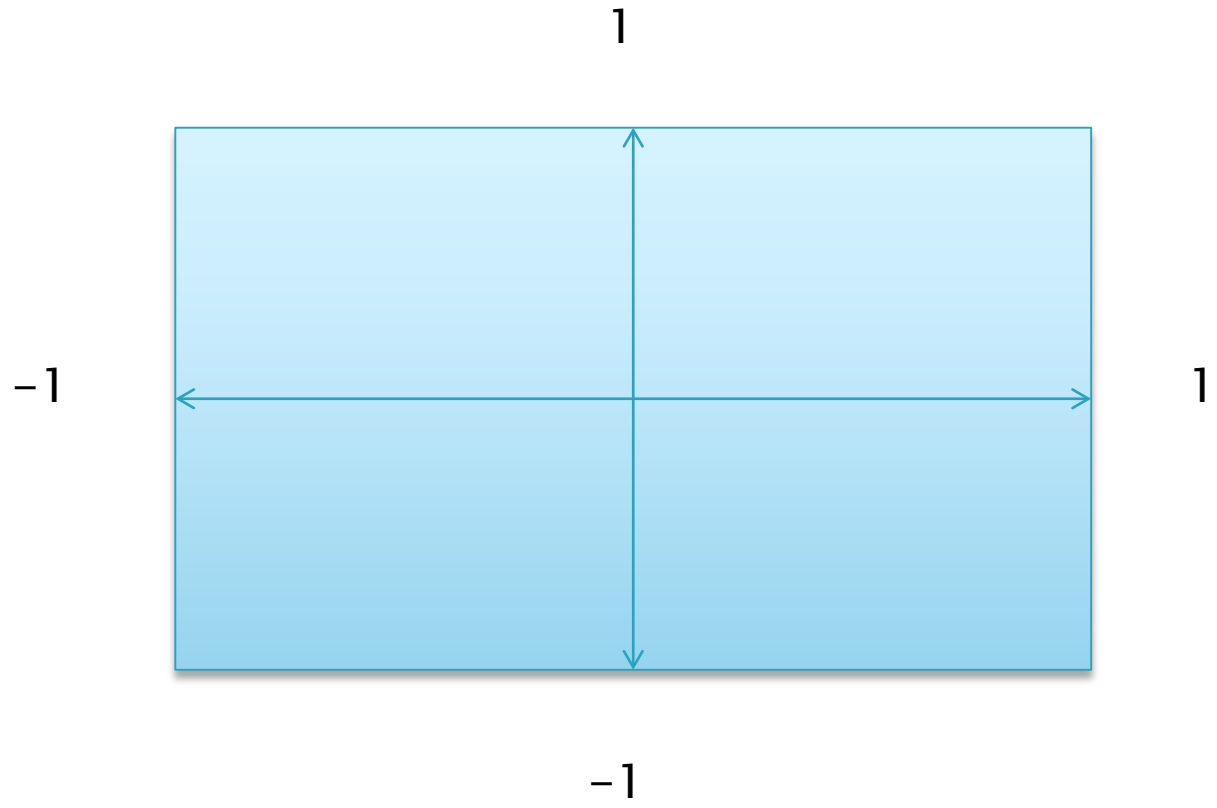
Screen space coordinates

- X & Y are in normalized device coordinates
  - Not specific pixels!
- Normalized Device Coordinates
  - [-1 to 1] on X and Y axis
- A vertex *inside* that range will be “on screen”
- A vertex *outside* that range cannot be seen



Normalized Device Coordinates

- Resolution-agnostic coordinates



What about the Z?

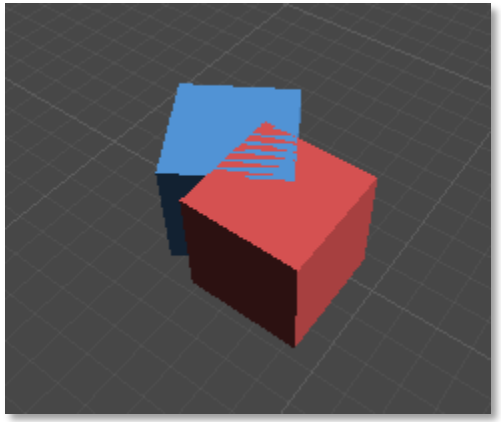
- Z represents depth
- Distance of vertex between camera's near and far clip planes
- A number between 0.0 and 1.0
  - If Z value < 0, it's "behind" the camera lens
  - If Z value > 1, it's "too far away"

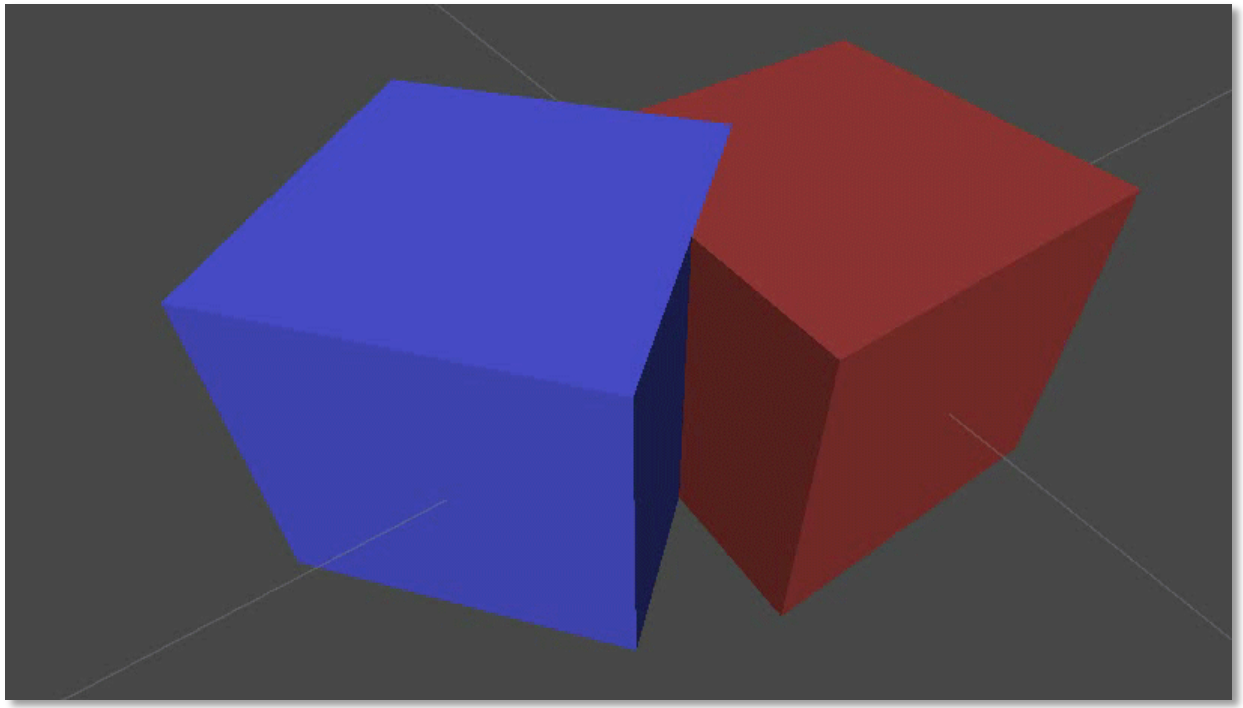
Depth & clip planes

- Your camera can't see infinitely into the distance
- Can only see between the "near" and "far" clip planes
- Depth is represented with a floating point number 0 – 1
  - There is only so much precision in a floating point number
  - Small distance between planes = more precise depths
  - Large distance between planes = less precise depths

Depth issues – Z fighting

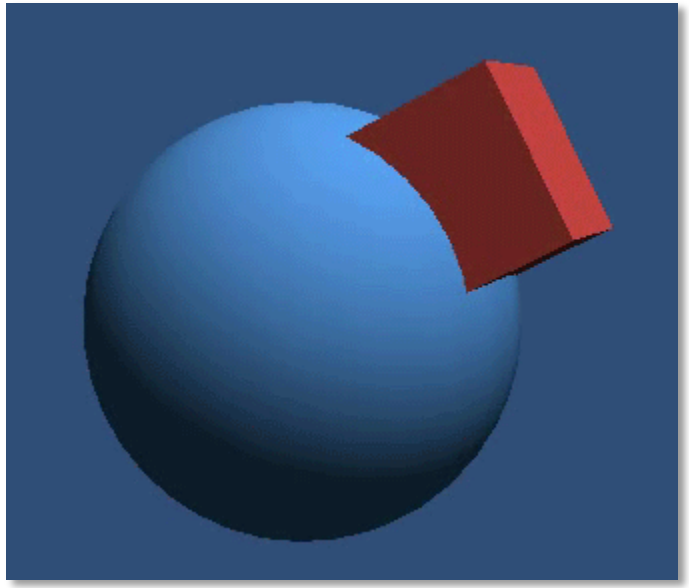
- What if two triangles have the same Depth?
- Z fighting!



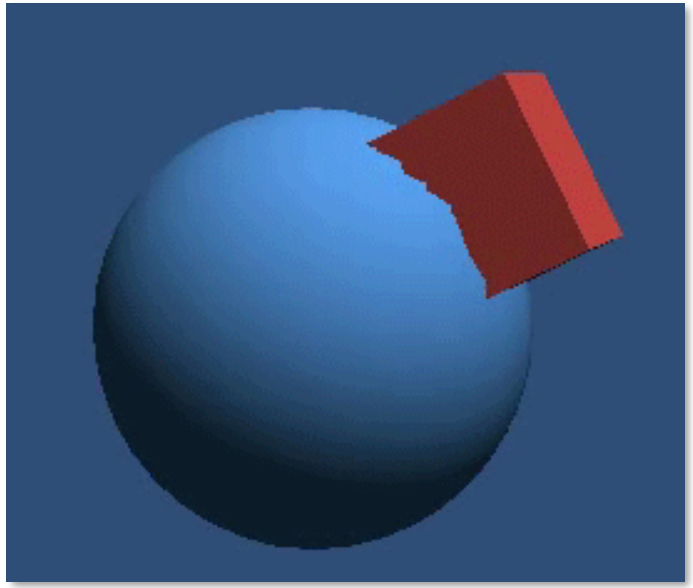


Depth values – Precision is not linear

- More precision closer to the camera



- Objects are **small**
- ~500 Units away
- Far plane Distance: 1000
- More precision



- Objects are **large**
- ~20,000 Units away
- Far plane Distance: >20,000
- Much less precision