# Cameras and Projection Transforms

The mathematics behind the Render Pipeline

Programming – Computer Graphics

aie

SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

## Contents

## Virtual Cameras

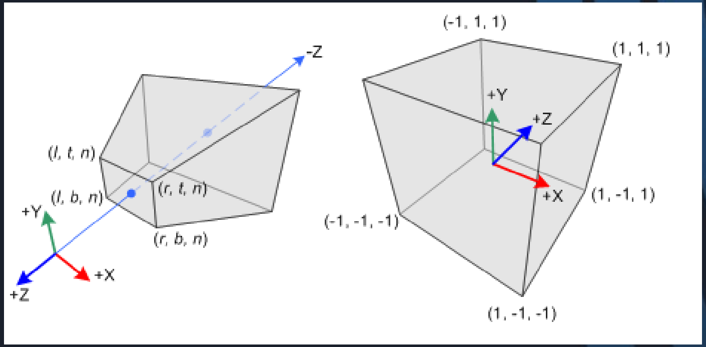
- The GPU has no concept of a camera
  - There is no object moving around in virtual space that displays your scene for you
- The GPU will display any geometry that is positioned within Clip Space
  - Typically an area in the range [-1,1] in all axis, centred at (0,0,0)
  - Any geometry not in this space wont appear
- In Computer Graphics we create Virtual Cameras
  - The job of the camera is to mathematically transform a 3-D or 2-D scene to fit within Clip Space, even if the geometry is positioned way off in the distance!
  - In essence we move the world around us when we render computer graphics!

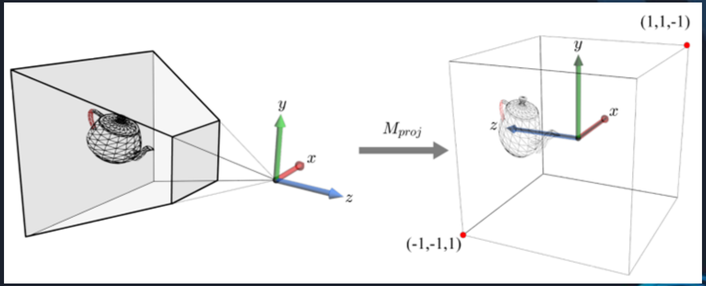
aie
SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

# Virtual Cameras

- We typically create two matrix transforms to represent a virtual camera and one transform for each 3D object in the scene
  - A World, Global or Model transform to represent where a 3D object is in the scene
  - A View or Camera transform that represents the inverse of the camera's world transform
    - Used to transform the scene as if the camera is at the centre of the world
  - A Projection transform that represents the lens of the camera
    - Used to distort a 3D scene into a Clip Space cube

- Combined Projection and View transforms define a Camera Frustum





## The Journey of a Vertex

- So we’ve now set the stage to discuss the transforms and pipeline further

- To render 3D points onto a screen of pixels, we need to:
  - Convert Local Space vertices to World Space vertices
  - Convert World Space vertices to View Space vertices
  - Convert View Space vertices to Clip Space vertices

- Clip Space vertices are then converted to Screen Space pixels during Rasterisation

- All of these conversions are achieved with Matrix multiplication

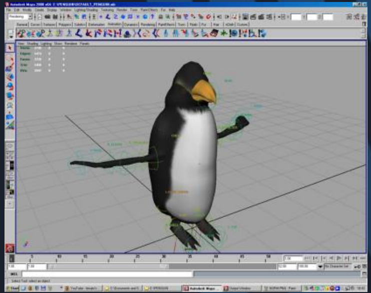
# The Journey of a Vertex

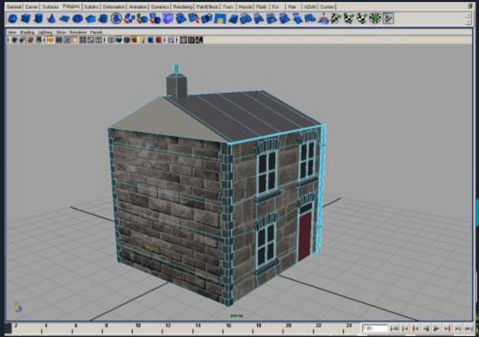
$$\begin{tikzcd}
\text{Local Space vertices} \arrow[r] & \begin{bmatrix} Model \\ Matrix \end{bmatrix} \arrow[dr] & \\
& & \text{World Space vertices} \arrow[dl] \\
& \begin{bmatrix} View \\ Matrix \end{bmatrix} \arrow[dl] & \\
\text{View Space vertices} \arrow[dr] & & \\
& \begin{bmatrix} Projection \\ Matrix \end{bmatrix} \arrow[r] & \text{Clip Space vertices}
\end{tikzcd}$$

- The Rasterisation step is handled by the GPU

# Local Space meshes

- Vertices start their journey towards pixeldom in Local Space
  - Sometimes called Object Space or Mesh Space
- All of their positions are stored relative to their own unique central origin
  - Often located at the base of the mesh, but sometimes at its center





# World Space coordinates

- In order to project our points onto the viewing plane we need to figure out where all the vertices are in relation to the camera

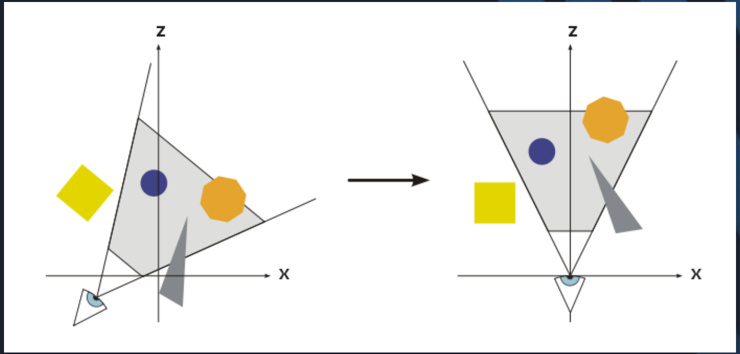
- All model vertices are currently in local space, but we need them in the same coordinate space as the camera, so we must:
  - Define a matrix with the rotation, scale and translation of the model as a whole in World Space or Global Space, called the World Transform or Model Transform
  - We transform each of the local space vertex coordinates into world space by multiplying by this transform
    - Since the camera's position is also stored in world space, this means we now have our points in the same coordinate space as the camera!

$$\begin{bmatrix} Model \\ Matrix \end{bmatrix} \times Local\ Space = World\ Space$$

# View Space Transform

- The triangles of a mesh are not yet relative to the camera, which is how we need them
  - We need to transform them to View Space, relative to the camera, as if the camera was at the centre of the world so that they can end up in Clip Space

- Multiplying each World Space vertex by the inverse of a camera's World Transform means the world space vertices are now oriented around the camera, such that the camera is the origin of the coordinate space
  - We can use a matrix to represent the position and rotation of the camera and calculate its inverse to get a View Matrix



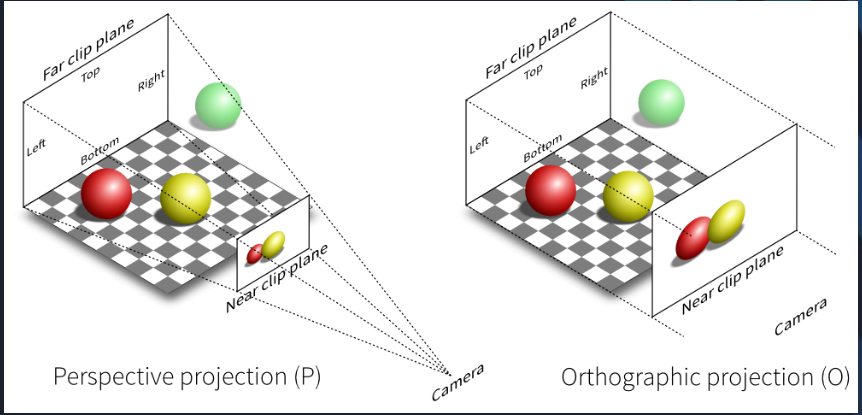
$$\begin{bmatrix} View \\ Matrix \end{bmatrix} \times World\ Space = View\ Space$$

## Clip Space

- So we now have all vertices in View Space
  - Transformed as if the camera is at the origin
- We now need to transform all visible geometry to fit within Clip Space
  - Fit within a cube with dimensions [-1,1] in all axis
- To do this we need a Projection Transform

# Projection Transforms

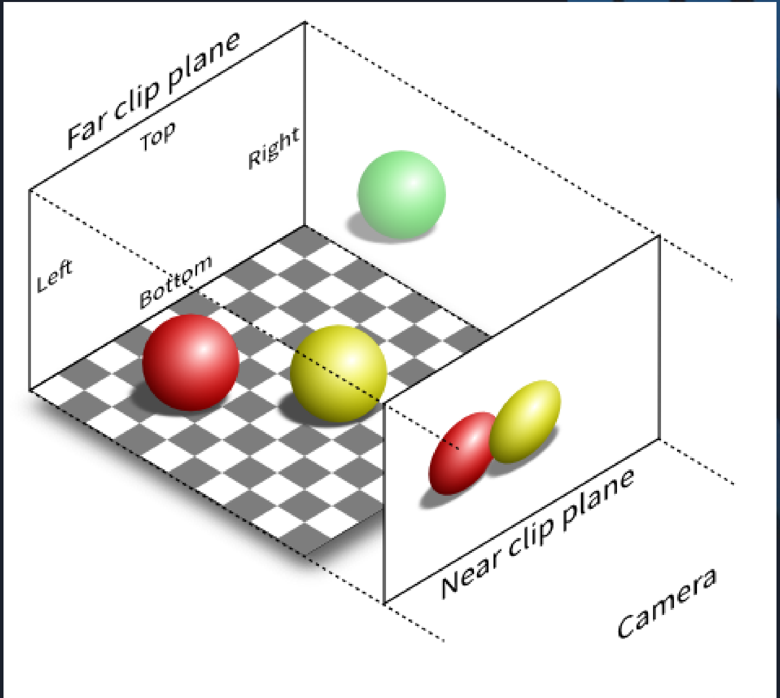
- There are two types of Projection Transform related to a camera
  - Perspective Transforms that use a perspective with a field-of-view and aspect ratio to convert a frustum shape to the Clip Space cube
  - Orthographic Transforms that have no field-of-view or aspect ratio and simply resize a rectangular space to fit the Clip Space cube
  - Both have a Near and Far clip plane
    - Can't see closer or further than them



# Orthographic Projections

- An Orthographic Projection matrix is the simplest to create and behaves like a camera with tunnel vision
  - No perspective for depth possible

- We can define 6 values to specify the limits of our rectangular space that we will convert to a Clip Space cube
  - Left, Right, Bottom, Top, Near and Far
  - Using these values we can create a matrix that scales the space, i.e. [Left,Right] would scale to be [-1,1]
  - The matrix would also offset the scene so that the origin is the middle of the 6 values

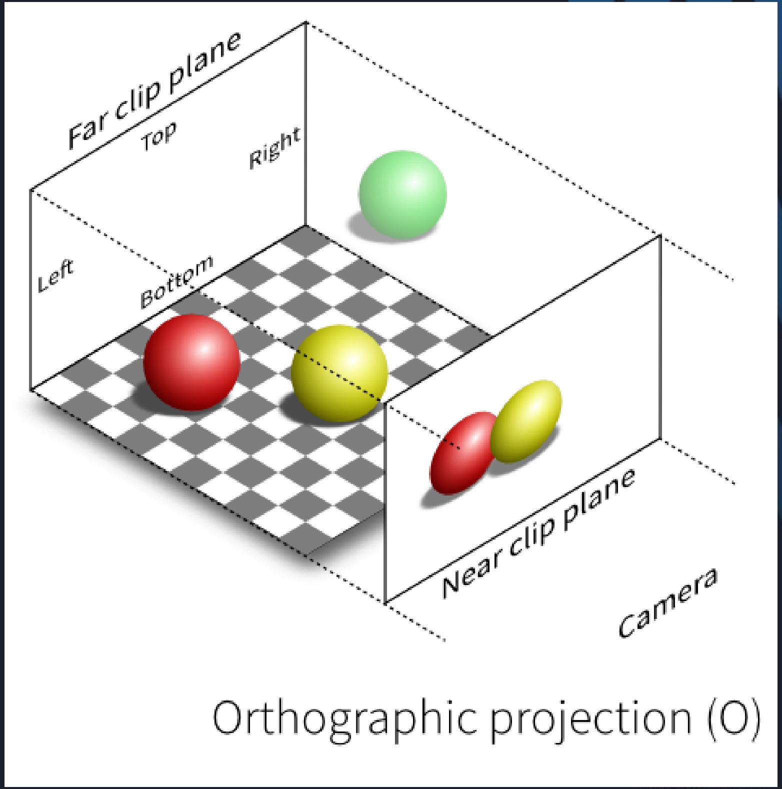


Orthographic projection (O)

# Orthographic Projections

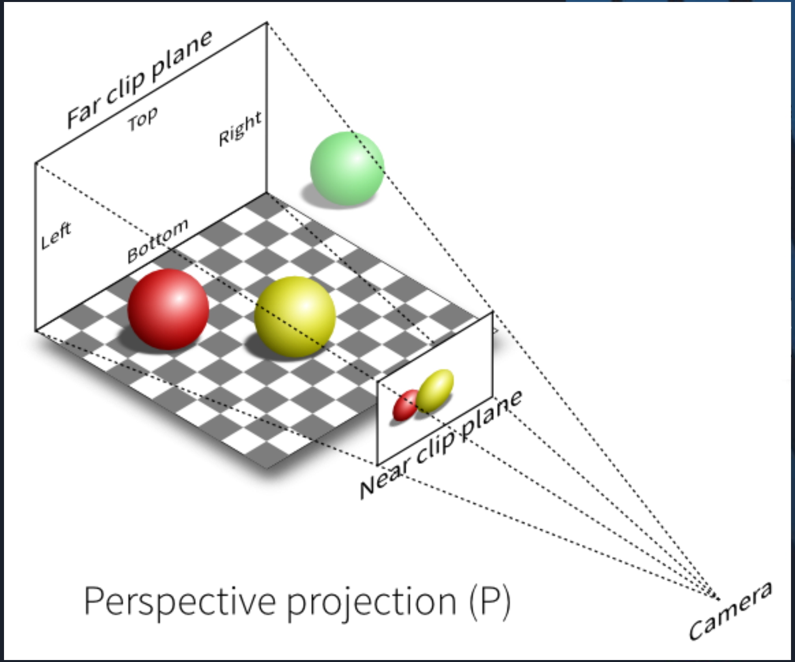
- The transform matrix can be built the following way
  - $l$ = left, $r$ = right, $t$ = top, $b$ = bottom, $f$ = far, $n$ = near

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



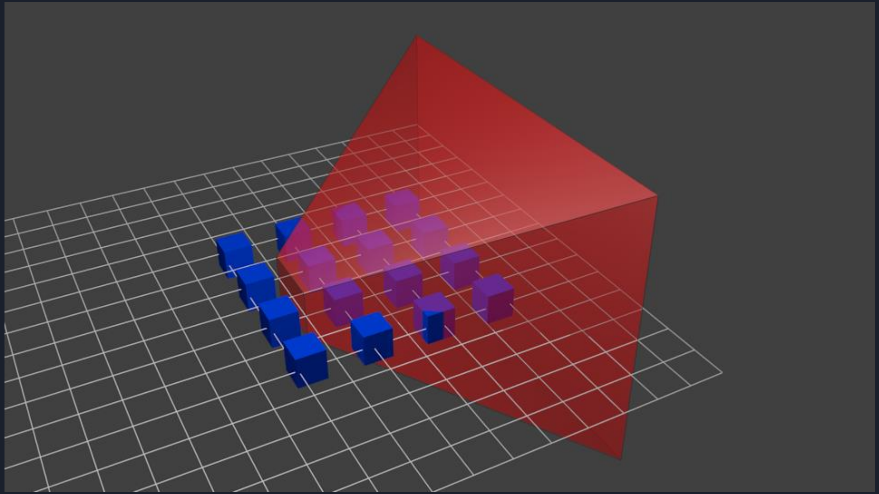
# Perspective Projection

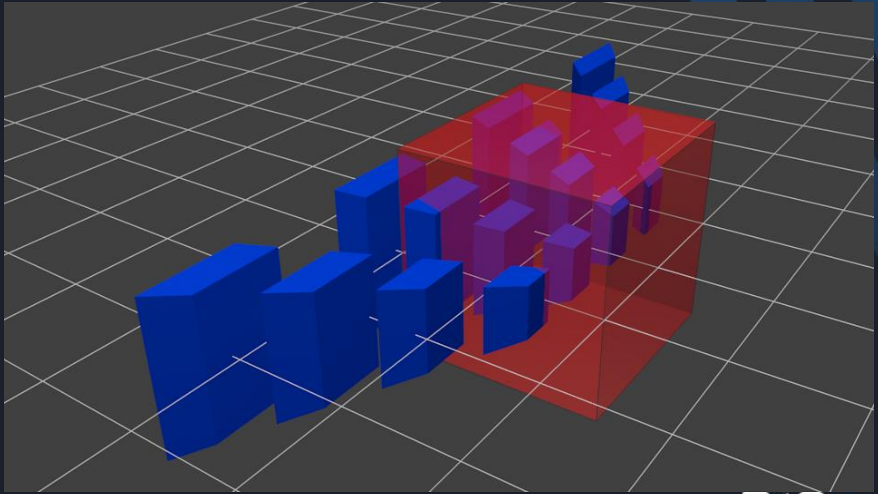
- A Perspective Projection uses a field-of-view and aspect ratio to distort a Frustum shape to fit the Clip Space cube
  - A Frustum is like a pyramid on its side with the top cut off
- The field-of-view (fov) is an angle usually specified as a vertical fov in radians
  - Typically $\frac{\pi}{4}$ or 45°
- The aspect ratio can simply be the width of the screen divided by the height
  - For example, $1920/1080$, or $16/9$ for widescreen



## Perspective Projection Distortion

- As mentioned, the Perspective Transform distorts the space to fit the Clip Space


View Space


Clip Space

aie SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

# Building a Perspective Projection

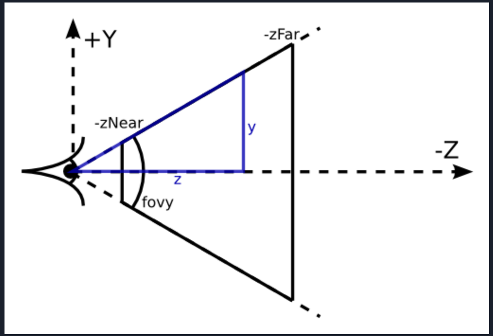
- To build a Perspective Projection we need to calculate a few things:
  - How the Field-of-View scales the X and Y axis
  - How the Aspect Ratio scales the X axis
  - How the Far and Near planes scale the Z axis
  - Offset the transform along the Z axis so that it fits between the [-1,1]

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

- We'll start with a simple Orthographic transform and build it up

# Field of View

- We use the vertical field-of-view as radians to calculate the scale for the X and Y axis in our transform
  - We use half of the angle rather than the full angle



- To calculate the scale amount we use the tangent function

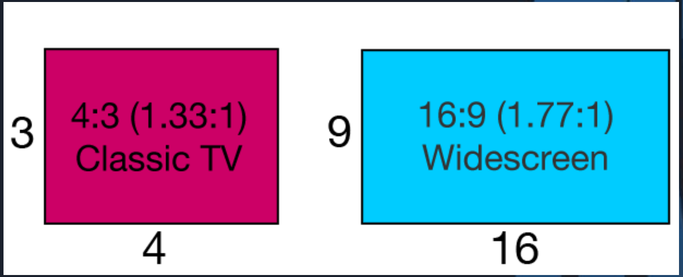
$$\tan(\frac{fov}{2})$$

- To use the value as a scale we simple divide 1 by the result and insert it in to the matrix

$$\begin{bmatrix} \frac{1}{\tan(\frac{fov}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{fov}{2})} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

## Aspect Ratio

- The aspect ratio represents the dimension of the screen as a ratio between the width and height
  - i.e. Width / Height, commonly 16:9, 16:10 and 4:3
- Incorporating the aspect ratio in to the equation is an easy step
  - Since the field-of-view was the vertical fov we simply need to scale the X axis of our matrix
  - We do this by first multiplying the tan() result by the aspect ratio before the divide



$$a = aspect\ ratio = \frac{width}{height}$$

$$\begin{bmatrix} \frac{1}{a \times \tan(\frac{fov}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{fov}{2})} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

# Distributing Z within Far and Near planes

- The last part of our perspective matrix is to incorporate the scale between the near and the far clip planes so that they map to [-1,1]
  - Also so that we offset the translation of the matrix to sit in the middle of these planes

$$\begin{bmatrix} \frac{1}{a \times \tan(\frac{fov}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{fov}{2})} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

- This is almost identical to the Orthographic projection

# Our Final Projection Matrix

$$a = aspect\ ratio = \frac{width}{height}$$

$$fov = vertical\ field\ of\ view$$

$$f = far\ clip\ distance$$

$$n = near\ clip\ distance$$

$$\begin{bmatrix} \frac{1}{a \times \tan(\frac{fov}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{fov}{2})} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

# Combining Matrices

- So we could multiple each matrix individually against vertices, or we could concatenate them together
  - We multiply the Projection, View and Model (aka Global or World) matrices together
  - This matrix becomes our Projection View Model transform
    - In Direct3D and related engines it is commonly called a World View Projection matrix due to it being a Left-Handed matrix system

$$PVM = Projection \times View \times Model$$

$$Clip\ Space\ Vertex = PVM \times Local\ Space\ Vertex$$

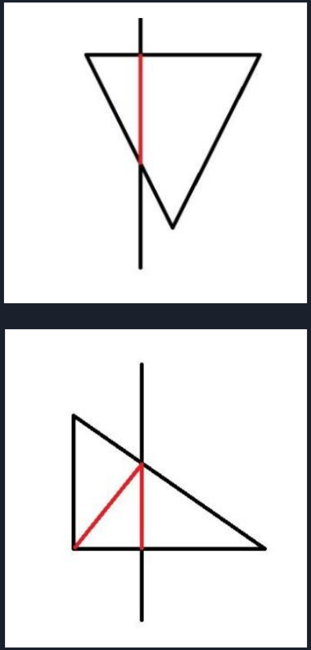
# Clip Space Clipping

- From here on out the GPU handles the conversion from Clip Space to Screen Space for rasterisation

- Before the final render stages the geometry is checked to see if its actually inside Clip Space
  - Single primitives are checked, which are the pieces that make up a complex model
  - Single points are easy
    - If X, Y and Z are all within [-1,1] then it is rasterised

- For a line, if both points are within [-1,1] it is rasterised, if neither are it is ignored
  - If one point is in and the other is out then the line is clipped to only rasterise the visible portion

- Triangles are clipped in a similar manner, with any visible portion being rasterised


*Example triangle clipping*

## Screen Space

- Up to this point the entire pipeline has been screen independent
  - Based on our transforms and Clip Space [-1,1], ignoring the dimensions of the screen
  - The Orthographic Projection didn't need to use the screen dimensions, it's just useful to use it for pixel accurate placement
- The GPU takes care of this last step based on a specified Viewport dimension
  - Usually the screen Width and Height
- The GPU simply performs the following calculation to get an XY Screen Space coordinate and writes the Z to the depth buffer

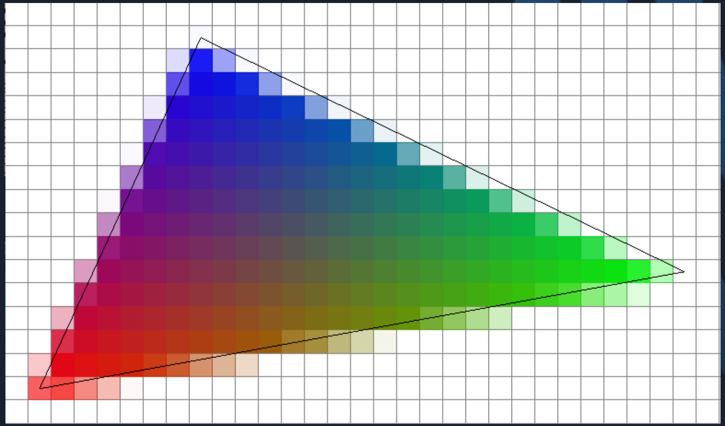
$$C = Clip\ Space\ point$$

$$V = Viewport\ (Width, Height, 1)$$

$$ScreenSpace = (C \times 0.5 + 0.5) \times V$$

## Rasterization

- Rasterisation is the final step that the GPU performs before the Fragment Shader takes over to colour pixels
- Screen Space vertex data is interpolated across the surface that the primitive fills in and the interpolated data is sent to the Fragment Shader to calculate whatever colour you want



## Summary

- On their road from 3D position to coloured pixels our vertices:
  - Start in Local Space, aka Model or Object Space
  - Convert to World Space, aka Global Space, via a matrix transform
  - Convert to View Space via another matrix transform
  - Convert to Clip Space with another matrix
  - GPU then converts to Screen Space based on viewport data
- Screen Space data is then rasterised and the Fragment Shader takes over

## Further Reading

- Dunn, F, Parberry, I, 2011, *3D Math Primer for Graphics and Game Development*, 2nd Edition, CRC Press

- Lengyel, E, 2011, *Mathematics for 3D Game Programming and Computer Graphics*, 3rd Edition, Cengage Learning