Transformation and Camera Matrices

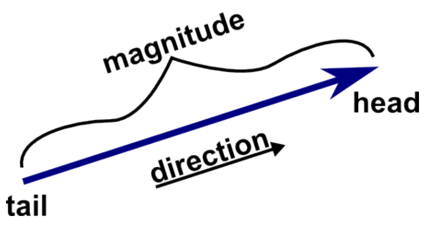
Assumptions

This reading assumes you have experience with vectors, matrices and quaternions, and the combination of these mathematical constructs. Not that you can necessarily do all of the math by hand, but that you have used these before, either through an existing math library or by implementing the structures & operations yourself.

Reminders

A few quick reminders about some of these basic constructs and their operations.

Vectors



Vectors are small arrays of numbers, often representing a position or direction with magnitude. We'll be using 2D, 3D and 4D vectors for various purposes. Vector **addition** and vector **subtraction** are common operations between two vectors. Scalar operations like **multiplication** and **division** require a vector and a single number (the scalar). More advanced operations include the **dot product**, which geometrically represents the cosine of the angle between the vectors multiplied by their magnitudes, and the **cross product**, which yields a new vector that is perpendicular to the vectors being crossed.

Matrices

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

When dealing with positions in 3D space, we can use 4x4 matrices to compactly represent one or more **transformations**, such as **translation**, **rotation** and **scale**, that we'd like to apply to those positions. **Multiplying** two transformation matrices together results in a single matrix that represents *both* transformations in a particular order. However, matrix multiplication is not commutative – AB ≠ BA – so the order of the transformations dictates the exact outcome.

We can also get the **inverse** of a transformation matrix, which is effectively the opposite transformation: multiplying a matrix by its inverse gets us back to the identity matrix. The **transpose** of a matrix mirrors values across the diagonal, swapping the rows and columns.

Multiplying a vector and a matrix is possible as long as the dimensions match. This will apply the matrix's transformations to the position represented by the vector. Since our matrices will be 4x4, we'll need to add an extra dimension to our 3D vectors for this to work.

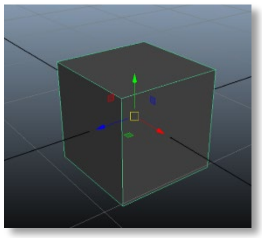
Quaternions

Quaternions are an alternative way of describing **3D rotations**. While they are not required for this course, they do have many useful properties and are often used behind the scenes in most modern game engines for rotations. Even if using quaternions to represent rotations, we'll eventually need to convert them to a matrix to be combined with other transformations.

Coordinate Spaces

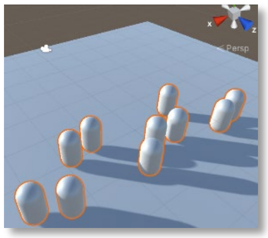
The overall goal of much of this matrix math is to take a set of positions in 3D space – usually from the vertices of our 3D meshes – and turn them into 2D screen coordinates. This requires several steps, each with its own matrix. These steps each represent transforming from one coordinate system to the next, eventually expressing our mesh as positions on the screen. Since we can multiply matrices together to combine their transformations, what we really want is a single matrix that represents all of these transformations that we can apply to vertex positions when rendering.

Local Space



The initial positions of a mesh's vertices are defined in what is called **local space**: the coordinate system in which they were modeled, often centered on, or just above, the origin. This is the space a 3D artist is generally working in. At this stage, we have no idea where the mesh will end up in our game world.

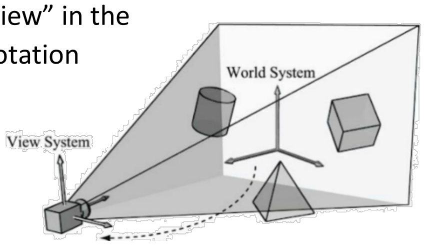
World Space



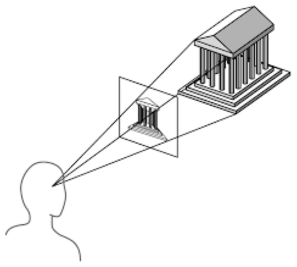
When a level designer is arranging 3D models within an editor like Unity, they're generally working in **world space**. World space represents where our mesh should be located in our 3D scene (or "world"). This is where transformations like translation, rotation and scale are applied to the vertices of the 3D model with a matrix called the **world matrix**.

View Space

If we want to determine if mesh is on our screen, it's not enough to know where that mesh is in our 3D world; we need to know where it is in relation to our point of view in that 3D world. We often refer to that point of view as our "camera", "eye" or "view" in the scene. It consists of a position (where the camera is) and a rotation (which way is the camera looking). Thus, **view space** represents where each vertex is located *relative to the camera*, as if the camera was the origin and the Z axis matched the camera's viewing direction. This is achieved with another matrix called the **view matrix**.



Screen Space



Now that our vertices are described as 3D offsets from the camera (that's *view space*), we need to project them onto a 2D surface to find their position on the screen. This is called **screen space** (or post-projection space) and is achieved using a **projection matrix**. In this space, the X and Y coordinates describe on our screen position within the range -1 to 1 (normalized device coordinates), as we're not dealing with specific pixels until rasterization.

Matrix Details

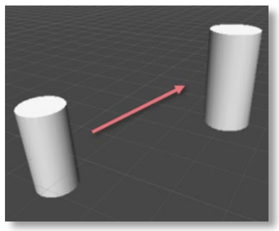
We'll dig into the individual matrices we need to transform from local to screen space.

Since DirectX Math stores its matrices in row-major order, this document will display its matrices in the same way. If this seems "backwards" to you, you've probably worked with column-major matrices instead. Column-major is simply the transpose of row-major layout and vice versa.

World Matrix

The world matrix contains the transformations to go from *local space* (as the model was originally created) to *world space* (where it should be in our virtual 3D world). These transformations generally include translation, rotation and/or scale. Let's look at each of these individually first.

Translation

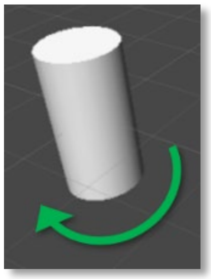


Translation is basically an offset. To translate by an amount, you add that amount to the existing position. In this case, the offset is an entire vector representing the offsets in X, Y and Z. If we move all of the vertices of a mesh by the same offset, the resulting vertices maintain the same overall shape, they're just in a different location.

In a row-major transformation matrix that represents only translation, the individual x, y and z offsets are placed in the last row, as seen below. This essentially means "move by (x,y,z)". Note that this always moves on the world's major axes and does not inherently take the object's orientation into account, though there are ways of doing that.

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

Rotation



This causes a position to rotate around the origin. A rotation matrix really defines three axes – positive X, positive Y and positive Z – after that rotation is applied. You can see this by looking at the top-left 3x3 portion of a 4x4 rotation matrix: each row is effectively a vector pointing along one of the positive axes.

$$\begin{array}{l|cccc}
\text{X axis} & 1 & 0 & 0 & 0 \\
\text{Y axis} & 0 & 1 & 0 & 0 \\
\text{Z axis} & 0 & 0 & 1 & 0 \\
 & 5 & 6 & 7 & 1
\end{array}$$

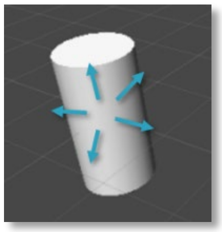
If we introduce rotation around the Y axis, the "vectors" representing the X and Z axes change and the Y axis remains the same. Below you'll see a matrix representing a 45-degree rotation around the Y axis. The Y axis is still pointing "up", while X and Z have "rotated". You can mimic this by holding your arms out, one in front and one to the right, while turning around.

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

A matrix that rotates around a single, major axis is fine, though we often want to rotate around multiple axes for a single overall transformation. For instance, we might want a character to both "turn left" and "look up". Without the use of quaternions, this requires two separate rotation matrices that must be combined together. While quaternions can be used in place of a rotation matrix to define overall orientation, we'll need to convert that quaternion into a matrix to combine it with our other transformations.

Our math library provides a function to create a single rotation matrix from three separate Euler angles: XMMatrixRotationRollPitchYaw(). Behind the scenes, it is effectively just creating three separate matrices and combining them for us.

Scale



Scaling an object has the effect of making it larger or smaller. What we're really doing is moving each vertex towards or away from the origin. If the origin is at the very center of the object, the object's relative size changes but effectively remains "in the same place". However, if the vertices are not centered around the origin, the object will appear to move when scaled.

X, Y, Z scale values are stored along the diagonal of a 4x4 transformation matrix. A *uniform scale* is when all three axes have the same scale value. To the right is an example of a *non-uniform scale* matrix:

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

Combining These Transformations Predictably

To combine multiple transformation matrices, we multiply them together. Remember that matrix multiplication is not commutative. That is, AB ≠ BA. This means *Translation * Scale* and *Scale * Translation* yield different results. With three total matrices to combine, that gives us 6 potential permutations: *T*R*S*, *T*S*R*, *R*T*S*, etc. So, what's the correct order?

While any order will give us *an* answer, what we really want is a *predictable answer*. For instance, we probably want the center of an object to be at its exact translation position, regardless of its scale. Depending on the multiplication order, however, we might end up *scaling the translation*, which is undesirable. Let's look at an example to see this in action:

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)

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

Next, let's see the results of multiplying these matrices in different orders: T*S and S*T.

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

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

In each case the resulting matrix contains both transformations. But, in the left example, the final translation has been scaled, resulting in the object moving quite far from the position we've specified. This means **S * T** is the preferable multiplication order.

In the case of rotation and translation, we'd run into the same issue. With T * R, the translation will be rotated around the origin. With R * T, the object's center will still be at our exact translation. This means that **R * T** is the preferable order with rotation and translation.

If S * T is best and R * T is best, then either way, *translation should come last*. But what about scale and rotation? Which order is preferable there? Let's look at another example:

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)

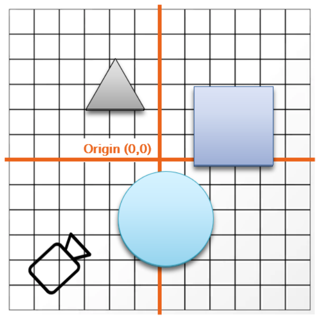
$$\begin{matrix} 1.414 & 0 & 1.414 & 0 \\ 0 & 3 & 0 & 0 \\ -2.828 & 0 & 2.828 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

As we can see, with S * R, each axis is scaled but points in the same direction as before. Thus, the most predictable order for row-major transformation matrix multiplication is **S * R * T**.

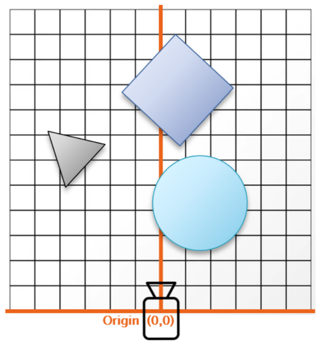
View Matrix

Now that we have a world matrix that describes a mesh in our 3D scene, the next task is to figure out what it looks like from our point of view in that scene. We call this point of view our "camera" or "eye" and it is generally defined by a position and a rotation. In other words: where is it located and which way is it facing?

Once we know those two pieces of information, we can build a matrix that transforms into **view space** (also called camera space). We no longer want to describe points at their absolute positions in the 3D scene and instead want to describe them as *offsets from our camera*. In effect, we want our camera to be the new origin and our camera's local axes (its *left*, *up* and *forward* directions) to be our major axes (*x*, *y* and *z* directions).



World Space



Camera Space

In both illustrations above, the camera and objects have the same *relative* positions and orientations. However, in the example on the right, the camera's position is at the origin and its forward (Z) axis matches the world's forward (Z) axis. This is exactly what we want, as it means that each object's new position is also its offset from the camera itself. With this setup, an object that is directly in front of the camera has an X and Y of (0, 0), though its Z value will be non-zero.

> *This sounds similar to the <u>normalized device coordinates</u> we need for screen space, where the very center of the screen is (0,0). Creating a proper view matrix is a necessary step to eventually getting our vertices into screen space.*

If we want each object to remain relative to each other and the camera, then we need to apply the same transformation to each one. This is achieved by created a matrix that effectively "undoes" the camera's transformations. If you recall from earlier, multiplying a matrix by its inverse results in the identity matrix; this means that our view matrix can simply be the inverse of a matrix that describes the camera's position and rotation.

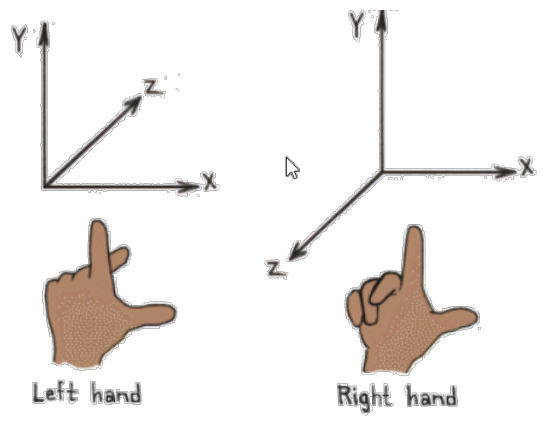
So, if we were to multiply the camera's transform by its inverse, the camera essentially has no transformations (it's been moved back to the origin and un-rotated). If we apply that transformation to each object instead, those objects still have the same relative position & orientation to the camera but are now described as if the camera was the origin. Perfect.

Side Note: Handedness

We often assume that the positive X axis points to the right and the positive Y axis points up. However, should the *positive* Z axis or the *negative* Z axis point forward (in front of the camera)? Either way is possible, and neither is wrong, but we need to make that choice and stick with it for our entire engine.

This is referred to as the handedness of our coordinate system, with Z-positive forward being *left-handed* and Z-negative forward being *right-handed*. Why this terminology? To answer that, first extend the thumbs, pointer fingers and middle fingers of your hands to match the axes of a coordinate system, as seen in the images below. Now align your hands so that the X and Y axes (thumbs and pointer fingers) match each other and also point along the positive X and Y directions. Notice that:

- The middle finger on your *left hand* now points away from you, signifying that the positive Z axis is forward. This is a *left-handed coordinate system*.
- The middle finger on your *right hand* now points towards you, signifying that the positive Z axis is backwards (and negative Z is forwards). This is a *right-handed coordinate system*.



What does this mean for us? We need to choose one and stick with it. Some engines and math libraries are hardcoded with one of these two coordinate systems, while others allow you to customize your handedness. Back when rendering pipelines were fixed-function, it was generally assumed that Direct3D was left-handed and OpenGL was right-handed. In modern graphics APIs, where we write the vertex shader code ourselves, there is no default; it all depends on the code we write. In other words, modern graphics APIs are not implicitly tied to either option.

Creating a View Matrix in Code

Luckily for us, the DirectX Math library has several functions to create a view matrix. Rather than having to do the math ourselves, these functions take very simple parameters and do the heavy lifting for us, returning a valid view matrix (assuming our parameters are valid). To define a camera, we need a position and an orientation. Position is pretty simple: just a point in space. But what about orientation? There are two common ways of handling view matrix orientation, known as *Look To* and *Look At*.

If we're creating a *Look To* view matrix, that simply means we provide a position and a direction, as if we're saying "go here and then rotate to look along this vector". To prevent unintended *roll* (looking in the correct direction but with our head tilted), we also need to provide the direction we consider to be "up". This is almost always just (0,1,0), or the world's positive Y axis. This aligns the resulting orientation so we don't accidentally roll the camera.

Alternatively, we could create a *Look At* view matrix. Instead of a position and a direction, this requires two positions – where the *camera is* and a point in space to *look at* – in addition to the "up" vector. This simply calculates the direction between those two positions and uses that as our camera's direction.

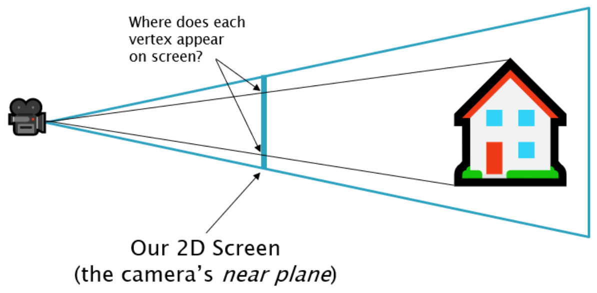
In either case, we end up with a matrix that represents the inverse of the camera's translation and rotation. Also note that there are two versions of each of these functions – ending with either LH (left-hand) or RH (right-hand) – so we can decide for ourselves the handedness to use. I'll be assuming left-handed in all of my code and demos for the course, but you could use either for your own assignments.

Using a View Matrix

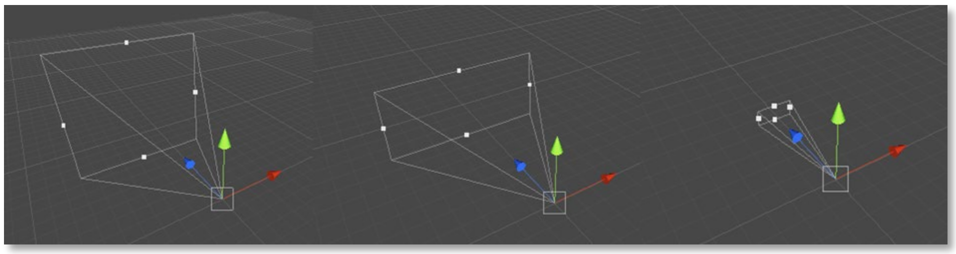
Once we have a view matrix, we can combine it with a world (transformation) matrix. The resulting matrix, often called the *world-view matrix*, represents the total of all of this work. If we multiply a position in local space by just a world matrix, it ends up in world space. If we instead multiply that local-space position by a world-view matrix, it essentially has both sets of transformations applied and ends up in view space. Only one more matrix left!

Projection Matrix

The last piece of the puzzle. The goal of this matrix is to map the 3D coordinates of our meshes to the 2D coordinates (normalized device coordinates, not exact pixels) on our screen.


Where does each vertex appear on screen?
Our 2D Screen
(the camera's *near plane*)

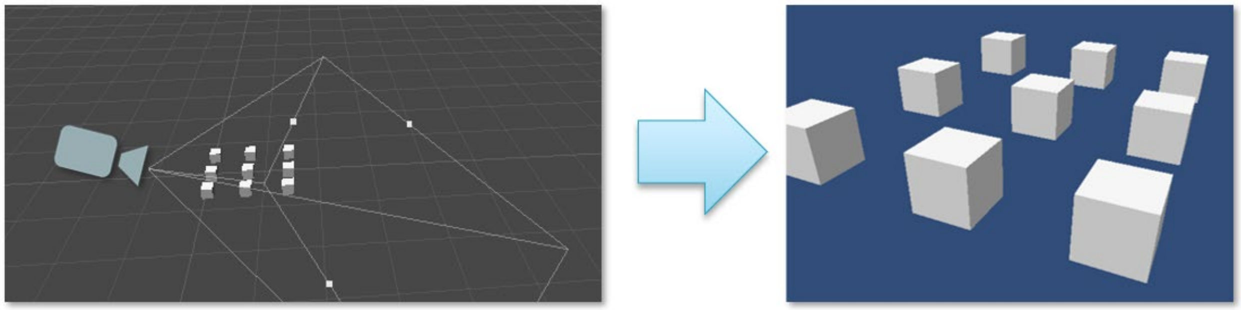
While the view matrix helps us describe where objects are relative to the camera, the projection matrix describes how much of the 3D world is actually visible from the camera. In other words: what is the overall shape of the camera's "vision"? This is analogous to the lens of a real-world camera and can be configured in many different ways, as shown below. This shape is called the camera's **frustum**.



Projection Types

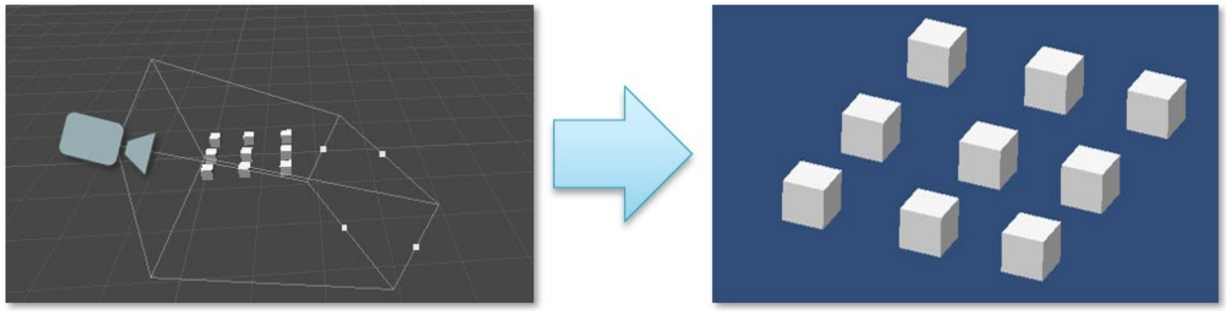
There are two major types of projections: *perspective* and *orthographic*. In a **perspective projection**, objects that are closer to camera appear larger. This is similar to how human vision works. The overall shape of the frustum is a trapezoid (it may look like a pyramid, but the top is actually flat). As an object gets closer to the camera, it takes up a larger portion of the available viewing space.

Likening this to a real-world camera, the object "blocks" more light rays as it gets closer to the camera (though that's not explicitly what's happening here as we're not simulating light rays).



On the other hand, in an **orthographic projection**, objects do not appear larger as they move towards the camera. This is because the shape of the frustum is a box. This is often used for isometric, top-down or 2D games, as the results appear very flat.

Using the "light ray" example, with this kind of projection, all of the light rays are effectively parallel, meaning each object "blocks" the same number of rays regardless of its position.



Most math libraries provide functions for creating either type of projection. In addition, the handedness of our coordinate system matters when creation a projection matrix, so DirectX Math has both LH and RH versions of each projection matrix function. I'll be using left-handed perspective matrices for my examples and demos, but feel free to play around with others.

World Units to Normalized Device Coordinates

Regardless of the type of projection, one of the goals of this matrix is to map the world units of our vertices to the [-1 to 1] range of our normalized device coordinates. If you've ever mapped one number range to another, you've probably used this kind of math. For instance, mapping [0-5] to [20-50] requires scaling and offsetting numbers in the [0-5] range so they end up stretching across the [20-50] range.

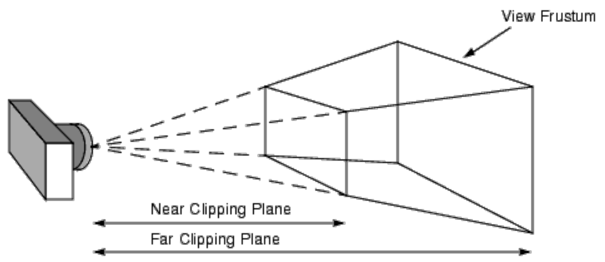
For a projection matrix, we need to map numbers in the [-*projectionSize + offset* to *projectionSize + offset*] range down to the [-1 to 1] range. This simple offset and scale and can be represented by a matrix with translation and scale transformations. However, the "size" of our projection (how much of the world we can see) is defined differently for perspective and orthographic projections.

For *orthographic*, we just pick a size for the box in world units: how wide and how tall? We use this information to create the aforementioned matrix to translate and scale to the [-1 to 1] range and we're done; we have an orthographic projection matrix.

For *perspective*, there's a bit more work to do. We usually define a horizontal field of view angle (in radians) and an aspect ratio (which should generally match our window's aspect ratio). Because the shape of our field of vision isn't a box, we need to get a bit fancier with our matrix to ensure our screen positions are correct. More on this soon.

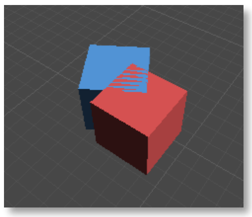
Clip Planes & Depth

The definition of a camera frustum generally includes what are called near and far *clip planes*. These planes are perpendicular to the camera's forward vector and define the "front" and "back" (or near and far sides) of the view frustum. Anything farther away than the far clip plane is excluded from the frustum (it's not visible, hence *clipped*). Similarly, anything closer than the near clip plane is also not within the frustum. Since these planes are always perpendicular to the camera's direction, they can be defined by a single value each, representing the distance to the plane from the camera.



Why do we need these? Can't we just include *everything* in front of the camera? In addition to the X and Y positions of our vertices being mapped to normalized values, we also need the **depth** (or distance from the camera) of each vertex to be mapped to a predictable range: [0 to 1]. For this mapping to work, we need finite distances for the front and back of the frustum, hence the need to explicitly choose near and far clip plane distances.

What might *not* be obvious from this, however, is why we can't just make the far clip plane a *really, really big number* (like 999,999,999). Ultimately, the depth of a vertex after projection will be stored as a 32-bit floating-point value. There is only so much precision available in those bits to represent unique numbers. When mapping the range [0 to 1000] to [0 to 1], we'll be able to accurately describe very minor changes in depth. If we instead mapped the range [0 to 999,999,999] to [0 to 1], very small changes in depth would result in the same 32-bit floating-point value and we'd have visual artifacts when objects are very close or overlap. This is known as *depth fighting* or *Z-fighting*.

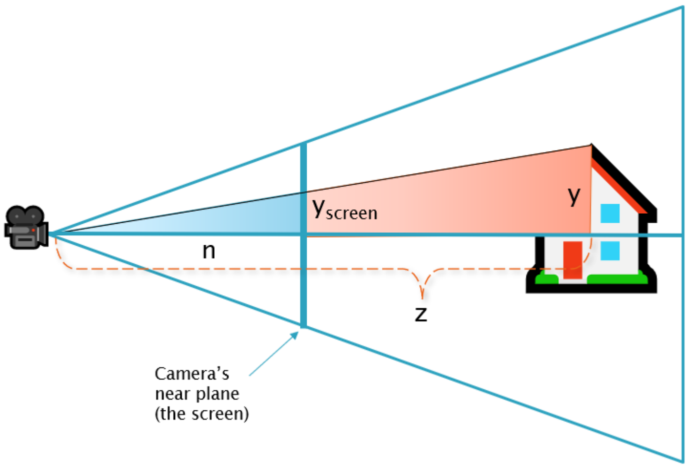


Ok, so what about the near clip plane? Why can't we just choose zero for its distance? Our camera's position is defined by a point in 3D space. The purpose of a projection matrix is to project the 3D scene onto *something*. In reality, we're projecting the entire frustum *onto the near clip plane*. Take a look at the diagram of the view frustum above again; in essence, the near clip plane *is* our window! If the near clip plane's distance is 0, then we'd be trying to project the scene onto a single point in space, which is infinitely small. That doesn't give us a useable image!

Perspective Projection Details

We're going to get a little math heavy for a moment. You don't strictly need to know this, as our math library will handle it for us, but it can help uncover some of the mystery of projection matrices.

How does a perspective projection actually work? The goal is to figure out where each vertex is on the screen. This is can be done with a pretty simple equation based on the diagram to the right.



We have coordinates of the vertex (x, y, z) in view space and the distance to the near clip plane (n) of the camera. We want to find the vertex's position on the near clip plane itself. Thus, we have the following ratios: $\frac{y_{screen}}{n} = \frac{y}{z}$

Solving for y_{screen} gives us: $y_{screen} = \frac{ny}{z}$. Hey, that's pretty easy! And solving for x is just as easy. So, what's the big deal?

Remember this needs to be in a matrix! What matrix can we use below to generate the proper result?

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

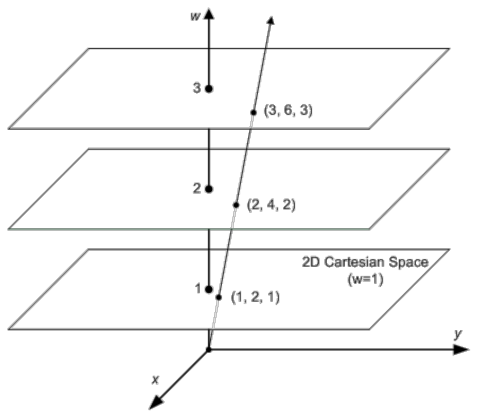
As it turns out, there *isn't* a matrix that does this! Regardless of the projection matrix we use, based on the camera's data alone, we cannot divide the resulting x and y values by z.

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

Doesn't work!

We're going to need an extra step in here and a slightly different way to think about our numbers.

Homogenous Coordinates



Homogenous coordinates, or projective coordinates, are an alternative way of defining points within a projection. In the example diagram to the right, all points along the projected "ray" are considered to be the same, as they all represent the same point after any one of those projections.

To define homogenous coordinates, we use an extra component. For 3D points, that means we add a fourth component, *w*, resulting in a 4D vector.

> *Hey, we've already been using 4D vectors since our matrices are 4x4. That's pretty handy!*

To convert from homogenous coordinates (the 4D vector) back to Cartesian coordinates (the 3D vector), we divide the (x,y,z) components by the *w* component:

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

For another example, take the homogenous points below. They all represent the same Cartesian point (2,3,4) after dividing. In other words, they're *homogenous*.

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

Back to the Projection Matrix

Ok, homogenous coordinates are a thing, but how do they help us with our projection issue? If we assume that the result of our projection matrix multiplication is not just a 3D vector with a useless w component, but a *4D homogenous vector*, then we know we can convert it back to 3D Cartesian coordinates with one extra division step. This means we don't have to divide by z just yet!

Remember, this is what we ultimately want: the x and y divided by z.

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

If we assume we'll be dividing by the w component during a later step, we can do this instead:

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

Let’s start filling in this mystery matrix. First up, where do we put *n* so that we get *nx* and *ny*?

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

Boom, half the matrix is done! Next: how do we ensure that the z winds up in the last component?

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

Ok, one more row to go. It gets slightly tricker here, but nothing we can’t handle. We know that when we convert from homogenous coordinates to Cartesian coordinates, we’ll end up dividing each component by *w.* In our case, that *w* value will simply be our original vector’s *z* value, as seen above. However, for the z component of the final 3D vector (the one all the way on the right above), we actually want that final value to just be our original z value. If we’re definitely dividing by z, but we want the result to *actually be **z***, then the numerator needs to be z squared! Let’s fill in a few more blanks:

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

Ok, we have two unknowns left. This row will be multiplied by the vector and the result must be z^2. Let’s plug some variables in and see what we get.

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

The result is a quadratic equation: $m_1 * z + m_2 = z^2$ We don’t need to solve this for every single possible depth value, just two: the near clip plane (*n*) and the far clip plane (*f*). After doing so, we’ll the following results: $m_1 = f + n$ and $m_2 = -fn$ Let’s plug them in!

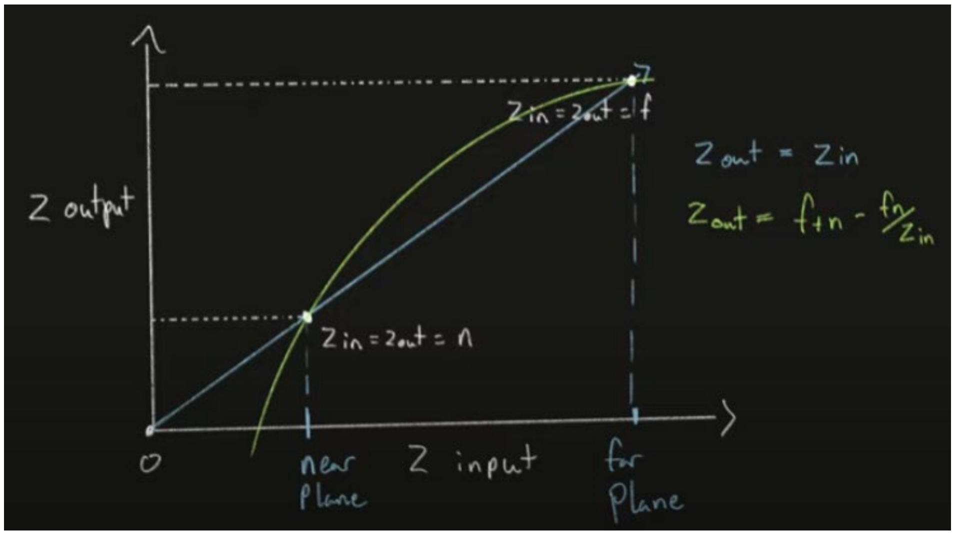
$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & (f+n) & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ (f+n)z - fn \\ z \end{bmatrix}$$

We now have a projection matrix that gives us homogenous coordinates. The last step is to perform the conversion back to Cartesian coordinates.

Perspective Divide and Non-Linear Depth

The last step, where we divide the resulting (*x,y,z*) by *w*, is known as the perspective divide. It's actually so common that most rendering pipelines *do it for us*. That's right: we won't actually perform this step anywhere in our shader code. Since the rasterizer stage will do it for us, we can just output the result of our matrix math. This is why the output of a vertex shader is required to be a 4D vector instead of a 3D vector.

There is an interesting side effect to this math: the depth values we end up with are *not linear*. We solved the quadratic equation above for two finite distances: the near and far clip planes. If we plug those values in, we get our expected depth values as output. But if we plug in a value between the near and far clip planes? We get a number *higher* than we expect. If we graph it out, we'll see that the depth values resulting from a perspective projection matrix are, in fact, logarithmic instead of linear:



Will this affect us at all? Not necessarily, as this is expected and normal in modern rendering pipelines. The only time it might bite us in the butt is with advanced techniques that actually need to retrieve these values and compare them to linear depth values. But that's not something we need to worry about right now.

Using a Projection Matrix

We have a world matrix. We have a view matrix. We have a projection matrix. We combine them all into the aptly named *world-view-projection* (or WVP) matrix. This is a single matrix that will perform all of these transformations, in order, to our local-space vertices. The output will be homogenous screen coordinates that the rasterizer stage will use when converting our geometry to pixels.

And that's pretty much all the matrix math we'll really need for basic 3D rendering!