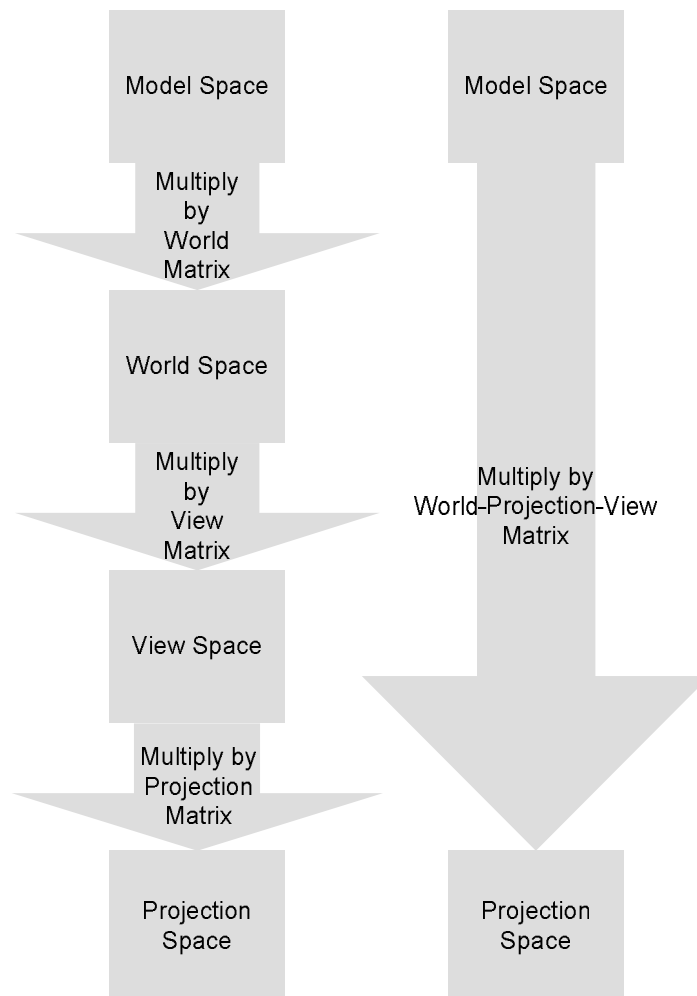


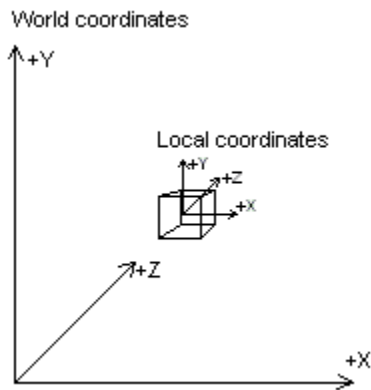
## Shader Series Supplemental: Coordinate Spaces

For every shader you create, it is important to understand the different coordinate spaces you'll be dealing with. Any position or direction belongs to a *coordinate space*. You can change a position or direction to a different coordinate space by using a transformation matrix. The most commonly used transformation matrices in graphics programming are the world, view, and projection matrices. The Shader Series will often refer to the context of a given position or direction by its coordinate space. The following image shows the standard set of transformations applied to geometry in a 3D scene.

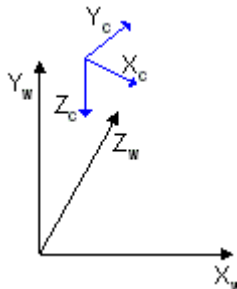


**Model space** (or Local Space) is made up of the raw vertex positions from a particular mesh. This is also known as *untransformed* geometry and is typically the raw data that is exported from a modeling application or geometry generation function.

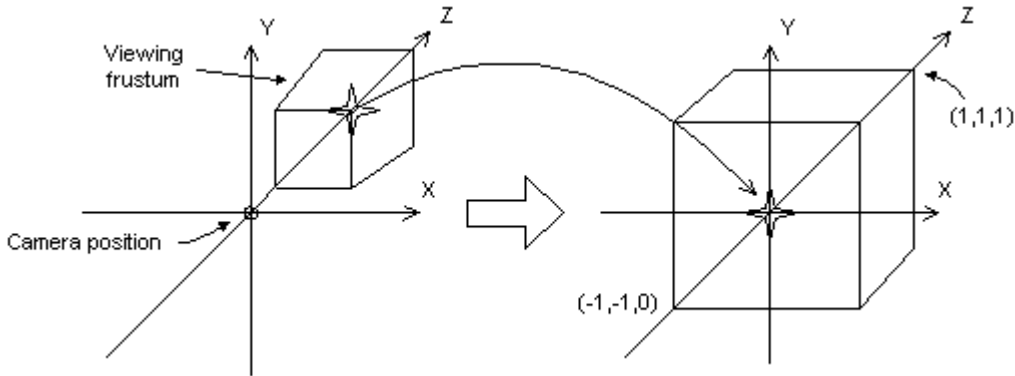
**World space** coordinates indicate a vertex's position in the world. When you multiply the model-space positions of a mesh by the *world matrix*, you get your model's position within the world. This position is typically important for calculating things such as lighting, when a particular light may also be represented as a position in the world.



*View space* coordinates are the coordinate positions within the view frustum. Positions and directions in view space are relative to the camera. When positions are in view space, the vertices are relative to their final positions on the screen. A *view matrix* is used to transform a world-space position to view-space. While this sounds complicated, the XNA Framework provides helper methods to create appropriate view matrices using simple camera coordinates.

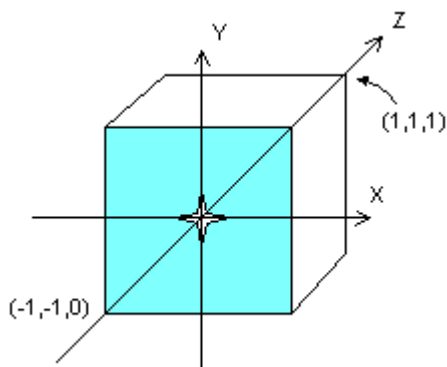


*Projection space* (also called homogenous coordinate space) can be a tricky concept to visualize, since the *projection matrix* takes the view space frustum and squashes it into a box. The reason this is important is that we can now take the aspect ratio and perspective into consideration. A *perspective* projection transform is an essential visual cue for depth in a 3D scene. A perspective projection matrix gives depth to a scene by shrinking objects that are farther away from the view point. Additionally, the projection matrix can collapse positions between a near and far clipping plane . the distance range in which geometry will be rendered. There's a great article on projection transformations in the DirectX SDK: <http://msdn2.microsoft.com/en-us/library/bb147302.aspx>.



One of the primary functions of most vertex shaders is to transform vertices from model space to projection space. As we have seen, this can be done by transforming the vertex by the world, view, and projection matrices in turn. However, there is a more efficient way to do this, by taking advantage of the transitive properties of matrix concatenation. You can concatenate the world, view, and projection matrices outside of the shader code, creating the *world-view-projection matrix* (often abbreviated as *wvp*.) Transforming a vertex by this matrix transforms the vertex from model space directly to projection space in one step, and minimizes the amount of work a vertex shader must do. Additionally, less data can be sent to shader via shader constants. If there's no reason to manipulate your vertices in world or view space, this is a way of optimizing your shader usage.

The part of projection space that you can actually see on the screen is called *homogenous clip space* (or sometimes simply *clip space*). It can be imagined as the area of your screen extruded into your monitor. The actual coordinates drawn are a square from  $(-1, -1)$  to  $(1, 1)$  on the x and y axes, and with z-coordinate values from 0 to 1. It's easier to show homogenous clip space than to describe it:



The shaded area is analogous to the front of the monitor; the volume behind it is the actual snapshot of the world that you'd be able to see. Primitives outside of this area are *clipped*, meaning that Direct3D will not actually draw them (since you couldn't see them anyway).

For more detailed information about 3D transforms, see the DirectX SDK documentation here: <http://msdn2.microsoft.com/en-us/library/bb206269.aspx>.