# Computer Graphics Pipeline

This chapter is based on the lecture notes of Jonathan Thaler for the course System Architectures at FHV.

In this appendix we introduce and discuss the *Real-Time Computer Graphics Rendering Pipeline* (CG pipeline) as a concrete context in which the pipes and filters architecture does conceptually apply. In the accompanying seminar [Lab Exercise 1] you are writing a very simple software renderer which implements a CG pipeline where the pipes and filters architecture is used. It is important to understand that although the CG pipeline in modern GPUs is indeed an ultra high-performance massively parallel pipeline, the specific implementation is highly hardware dependent and due to performance implications has nothing to do with pipes and filteres. However, *conceptually* we can treat it as a variant of the pipes and filters architecture, which operates on a stream of data and transforms them in multiple sequential steps. We very specifically focus only on the *real-time* CG pipeline here, with a few additional topics related to rendering. A full treatment of real-time CG in general is far beyond the scope of this course as it is a tremenduously vast area, therefore we refer interested students to [2] for an in-depth introduction. See Figures 1.1, 1.2, 1.3 for examples of images of various computer games rendered with real-time CG.

Figure 1.1: Real-time rendering in the game Fortnite (Image courtesy of Epic).

Figure 1.2: Real-time rendering in the game Doom Eternal (Image courtesy of id Software).


Figure 1.3: Real-time rendering in the game Destiny 2 (Image courtesy of Bungie).

Note that there exists also non real-time CG, more specifically known as physically based / offline rendering which has the aim of rendering images that are physically correct. In this branch of computer graphics, the flow of the light is simulated in physically correct ways, with the aim of creating images that are indistinguishable from photographs. Due to its extreme computational requirements, rendering of such pictures takes hours or days even on parallel hardware and is therefore not realtime. This rendering technique follows a rather different approach and although there are also certain transformations involved which are similar to real-time CG, therefore the physically based rendering pipeline works very different. This topic is also beyond the scope of this course and we refer interested students to the excellent book [25] which is available also online for free.[14] See Figures 1.4, 1.5, 1.6 for examples of images rendered with phyiscally based rendering

Figure 1.4: Photorealistic rendering of various materials and surfaces (Image taken from the gallery of http://www.pbr-book.org/)

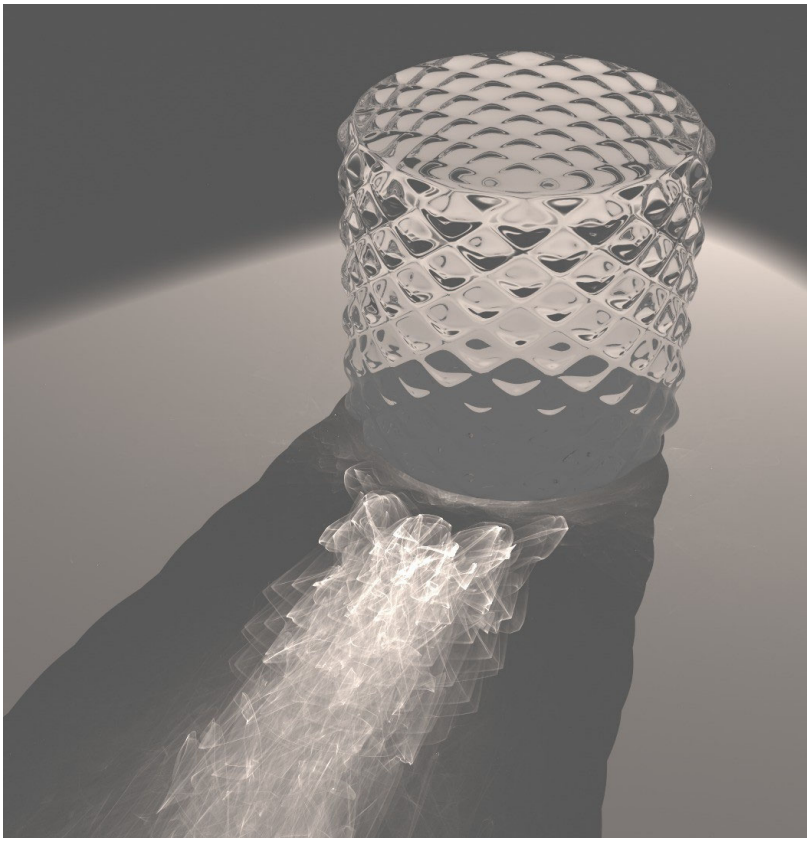

Figure 1.5: Photorealistic rendering of light scattering with photon tracing (Image taken from the gallery of http://www.pbr-book.org/)

Figure 1.6: Photorealistic rendering of translucent material with subsurface scattering (Image taken from the gallery of http://www.pbr-book.org/)

Both areas of real-time and physically-based rendering overlap in the use of a number of techniques but in general are quite distinct. However, in recent years there has been a trend in real-time CG towards more photorealistic rendering, utilising methods from physically based rendering. This became possible through the ever-increasing computing capabilities and generalisation of modern GPUs and incorporation of techniques from physically based rendering into hardware such as ray tracing. See Figures 1.7, 1.8, 1.9 for examples of images rendered with photorealistic real-time CG.

Figure 1.7: Towards photorealistic real-time rendering with the CryEngine (Image courtesy of Damians Stempniewski, taken from https://damians.artstation.com/).


Figure 1.8: Towards photorealistic real-time rendering with the Unreal Engine 4 (Image courtesy of NeoScape, taken from https://www.unrealengine.com/en-US/spotlights/datasmith-and-unreal-engine-drive-real-time-architectural-design-at-neoscape).

Figure 1.9: Towards photorealistic real-time rendering with the Unreal Engine 4 (Image courtesy of Epic, taken from https://www.unrealengine.com/en-US/blog/epic-games-demonstrates-real-time-ray-tracing-in-unreal-engine-4-with-ilmxlab-and-nvidia).

## 1.1 The Rendering Pipeline

The central problem of 3D computer graphics is how to arrive from 3D model coordinates at 2D screen coordinates, see Figure 1.10. This is achieved through the computer graphics rendering pipeline, which we will discuss more in depth in the remainder of this chapter as an example context where to apply the pipes and filters architecture.

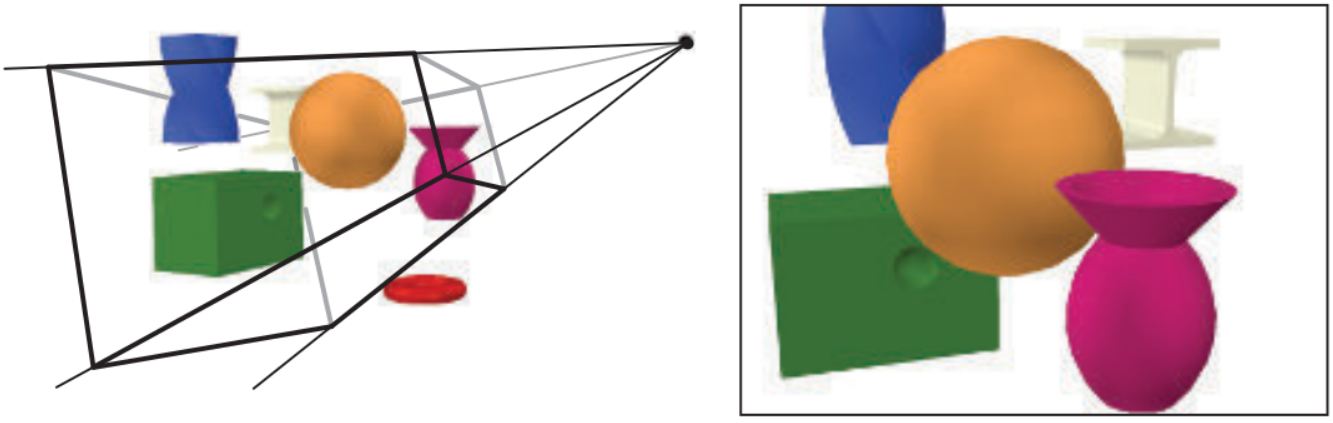

Figure 1.10: In the left image, a virtual camera is located at the tip of the pyramid (where four lines converge). Only the primitives inside the view volume are rendered. For an image that is rendered in perspective (as is the case here), the view volume is a *frustum* (plural: *frusta*), i.e., a truncated pyramid with a rectangular base. The right image shows what the

camera “sees”. Note that the red donut shape in the left image is not in the rendering to the right because it is located outside the view frustum. Also, the twisted blue prism in the left image is clipped against the top plane of the frustum (Figure taken from [2]).

The main function of the pipeline is to generate, or render, a two-dimensional image, given a virtual camera, three-dimensional objects, light sources, and more. The rendering pipeline is thus the underlying tool for real-time rendering. The locations and shapes of the objects in the image are determined by their geometry, the characteristics of the environment, and the placement of the camera in that environment. The appearance of the objects is affected by material properties, light sources, textures (images applied to surfaces), and shading equations. A coarse division of the real-time rendering pipeline into four main stages —application, geometry processing, rasterization, and pixel processing, is shown in Figure 1.11.

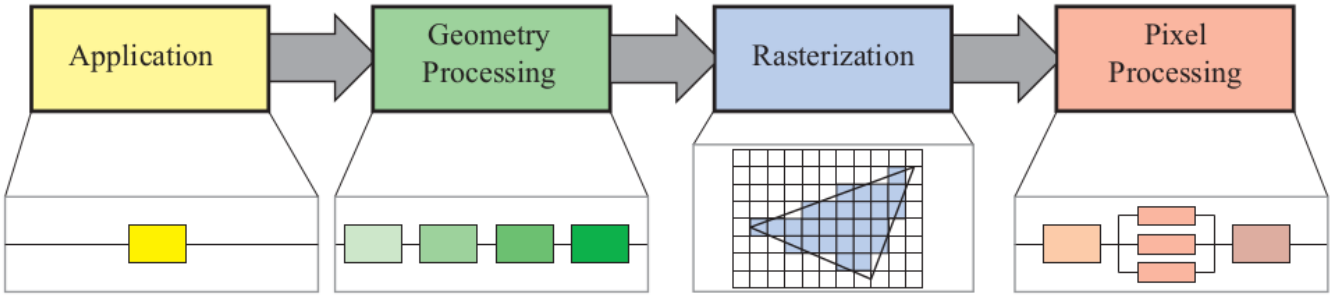


Figure 1.11: The basic construction of the rendering pipeline, consisting of four stages: application, geometry processing, rasterization, and pixel processing. Each of these stages may be a pipeline in itself, as illustrated below the geometry processing stage, or a stage may be (partly) parallelized, as shown below the pixel processing stage. (Figure taken from [2]).

The pipeline stages execute in parallel, with each stage dependent upon the result of the previous stage. Ideally, a nonpipelined system that is then divided into *n* pipelined stages could give a speedup of a factor of *n*. There are generally four pipeline stages:

1. The *application stage* is driven by the application and is therefore typically implemented in software running on general-purpose CPUs. These CPUs commonly include multiple cores that are capable of processing multiple threads of execution in parallel. This enables the CPUs to efficiently run the large variety of tasks that are the responsibility of the application stage. Some of the tasks traditionally performed on the CPU include collision detection, global acceleration algorithms, animation, physics simulation, and many others, depending on the type of application. Note that some computations like physics simulations tend to be executed more and more on the GPU using compute shaders, therefore the separation of tasks between CPU and GPU is blurred.

2. The *geometry processing stage* deals with transformations, projections, and all other types of geometry handling. This stage computes what is to be drawn, how it should be drawn, and where it should be drawn. The geometry stage is typically performed on a graphics processing unit (GPU) that contains many programmable cores as well as fixed-operation hardware. The geometry processing stage on the GPU is responsible for most of the per-triangle and per-vertex operations.

3. The *rasterization stage* typically takes as input three vertices, forming a triangle, and finds all pixels that are considered inside that triangle, then forwards these to the next stage. Therefore, all the primitives that survive clipping in the previous stage are then rasterized, which means that all pixels that are inside a primitive are found and sent further down the pipeline to pixel processing.

4. The *pixel processing stage* executes a program per pixel to determine its color and may perform depth testing to see whether it is visible or not. It may also perform per-pixel operations such as blending the newly computed color with a previous color. The rasterization and pixel processing stages are also processed entirely on the GPU. The goal here is to compute the color of each pixel of each visible primitive. Those triangles that have been associated with any textures (images) are rendered with these images applied to them as desired. Visibility is resolved via the z-buffer algorithm, along with optional discard and stencil tests. Each object is processed in turn, and the final image is then displayed on the screen.

In the following we discuss the geometry, rasterization and pixel processing stages more in depth as they are the relevant stages as the application stage is highly domain specific.

## 1.2 Geometry Processing Stage

In general, this stage deals with transformations, projections, and all other types of geometry handling. This stage is further divided into the following functional stages: vertex shading, projection, clipping, and screen mapping, see Figure 1.12.

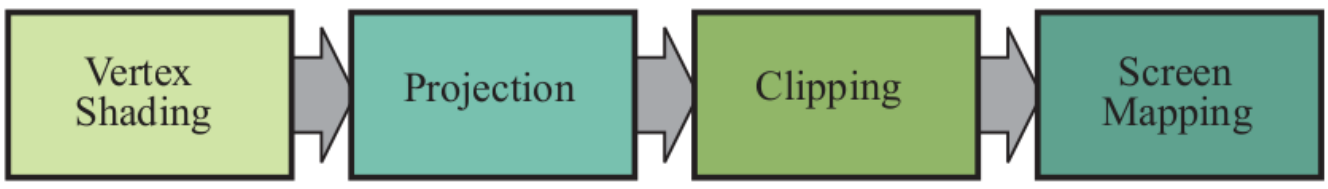


Figure 1.12: The geometry processing stage divided into a pipeline of functional stages (Figure taken from [2]).

### 1.2.1 Vertex Shading

There are two main tasks of vertex shading, namely, to compute the position for a vertex and to evaluate whatever the programmer may like to have as vertex output data, such as a normal and texture coordinates. Traditionally much of the shade of an object was computed by applying lights to each vertex's location and normal and storing only the resulting color at the vertex. These colors were then interpolated across the triangle. For this reason, this programmable vertex processing unit was named the vertex shader. With the advent of the modern GPU, along with some or all of the shading taking place per pixel, this vertex shading stage is more general and may not evaluate any shading equations at all, depending on the programmer's intent. The vertex shader is now a more general unit dedicated to setting up the data associated with each vertex. As an example, the vertex shader can animate an object using transformations.

We start by describing how the vertex position is computed, a set of coordinates that is always required. On its way to the screen, a model is transformed into several different spaces or coordinate systems. Originally, a model resides in its own model space, which simply means that it has not been transformed at all. Each model can be associated with a model transform so that it can be positioned and oriented. It is possible to have several model transforms associated with a single model. This allows several copies (called instances) of the same model to have different locations, orientations, and sizes in the same scene, without requiring replication of the basic geometry.

It is the vertices and the normals of the model that are transformed by the model transform. The coordinates of an object are called model coordinates, and after the model transform has been applied to these coordinates, the model is said to be located in world coordinates or in world space. The world space is unique, and after the models have been transformed with their respective model transforms, all models exist in this same space.

As mentioned previously, only the models that the camera (or observer) sees are rendered. The camera has a location in world space and a direction, which are used to place and aim the camera. To facilitate projection and clipping, the camera and all the models are transformed with the view transform. The purpose of the view transform is to place the camera at the origin and aim it, to make it look in the direction of the negative z-axis, with the y-axis pointing upward and the x-axis pointing to the right. We use the -z-axis convention; some texts prefer looking down the +z-axis. The difference is mostly semantic, as transform between one and the other is simple. The actual position and direction after the view transform has been applied are dependent on the underlying application

programming interface (API) (for example OpenGL, Vulcan or DirectX). The space thus delineated is called camera space, or more commonly, view space or eye space. Both the model transform and the view transform are implemented as 4x4 matrices (see Linear Transformations). An example of the way in which the view transform affects the camera and the models can be seen in Figure 1.13.

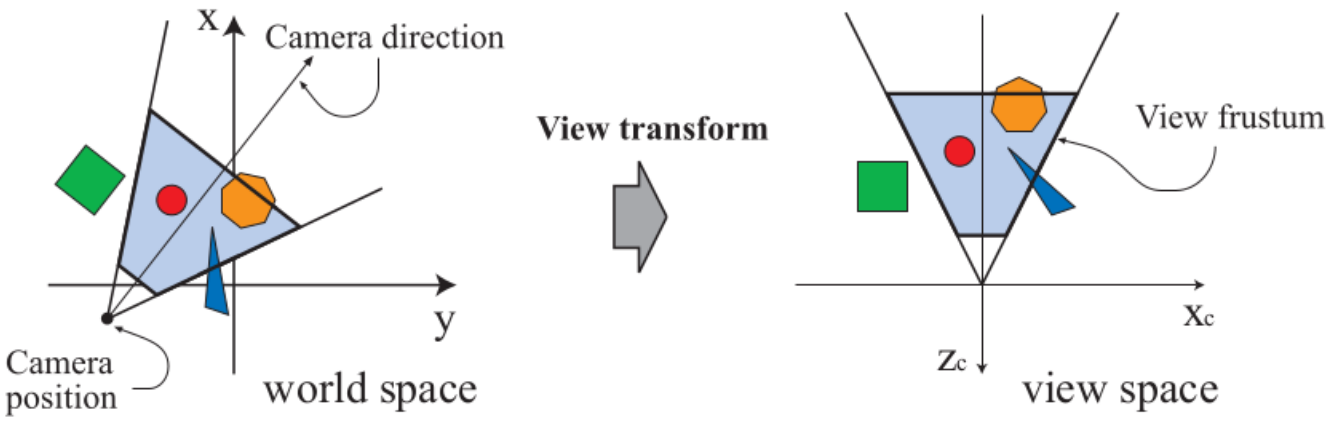


Figure 1.13: In the left illustration, a top-down view shows the camera located and oriented as the user wants it to be, in a world where the +z-axis is up. The view transform reorients the world so that the camera is at the origin, looking along its negative z-axis, with the camera's +y-axis up, as shown on the right. This is done to make the clipping and projection operations simpler and faster. The light blue area is the view volume. Here, perspective viewing is assumed, since the view volume is a frustum. Similar techniques apply to any kind of projection (Figure taken from [2]).

Next, we describe the second type of output from vertex shading. To produce a realistic scene, it is not sufficient to render the shape and position of objects, but their appearance must be modeled as well. This description includes each object's material, as well as the effect of any light sources shining on the object. Materials and lights can be modeled in any number of ways, from simple colors to elaborate representations of physical descriptions. This operation of determining the effect of a light on a material is known as shading. It involves computing a shading equation at various points on the object. Typically, some of these computations are performed during geometry processing on a model's vertices, and others may be performed during per-pixel processing. A variety of material data can be stored at each vertex, such as the point's location, a normal, a color, or any other numerical information that is needed to evaluate the shading equation. Vertex shading results (which can be colors, vectors, texture coordinates, along with any other kind of shading data) are then sent to the rasterization and pixel processing stages to be interpolated and used to compute the shading of the surface.

As part of vertex shading, rendering systems perform projection and then clipping, which transforms the view volume into a unit cube with its extreme points at (−1, −1, −1) and (1, 1,

1). Different ranges defining the same volume can and are used, for example, $0 \leq z \leq 1$. The unit cube is called the canonical view volume. Projection is done first, and on the GPU it is done by the vertex shader. There are two commonly used projection methods, namely orthographic (also called parallel) and perspective projection, see Figure 1.14.

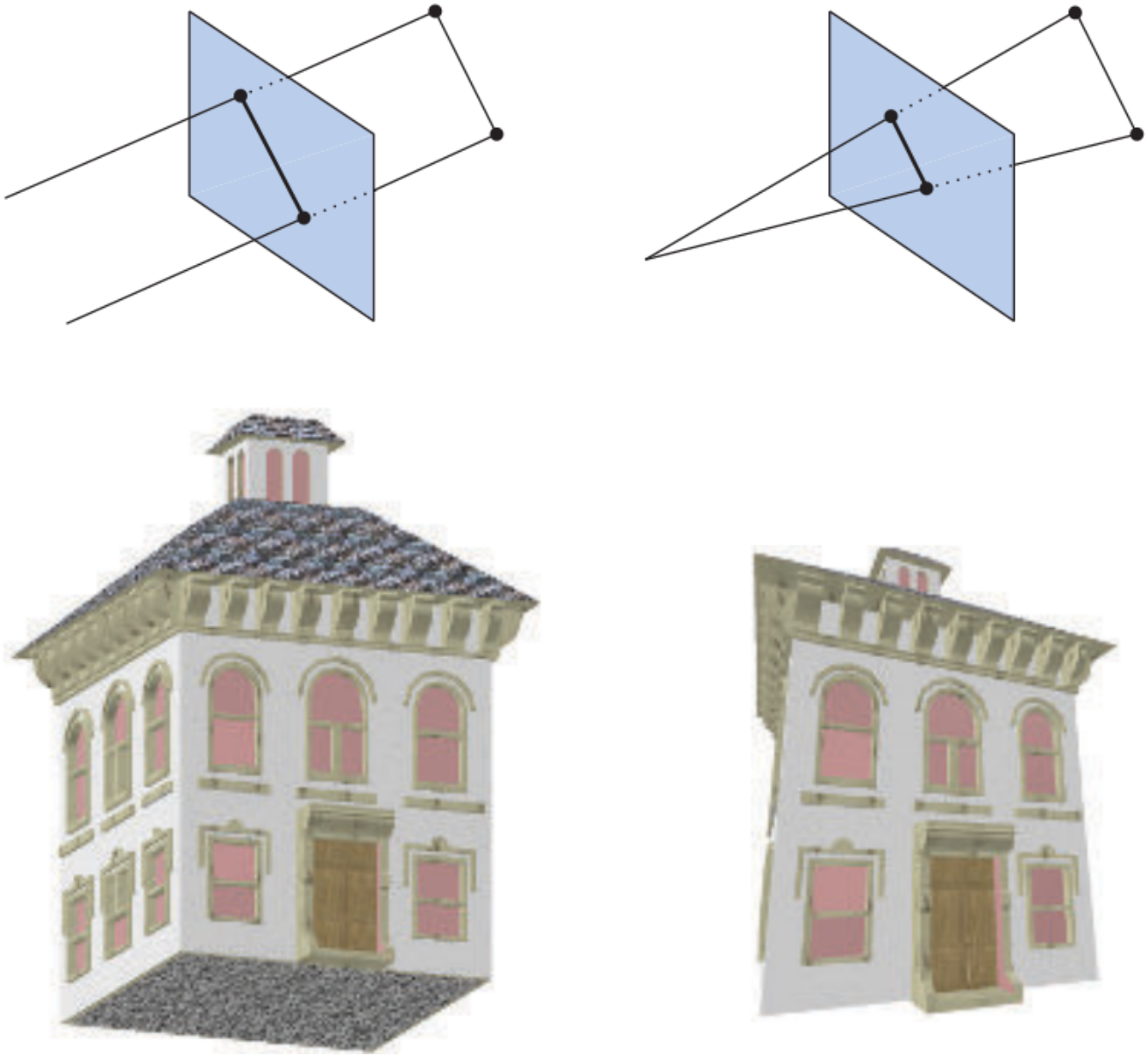

Figure 1.14: On the left is an orthographic, or parallel, projection; on the right is a perspective projection (Figure taken from [2]).

Note that projection is expressed as a matrix (see Linear Transformations) and so it may sometimes be concatenated with the rest of the geometry transform. The view volume of orthographic viewing is normally a rectangular box, and the orthographic projection transforms this view volume into the unit cube. The main characteristic of orthographic projection is that parallel lines remain parallel after the transform. This transformation is a combination of a translation and a scaling.

The perspective projection is a bit more complex. In this type of projection, the farther away

an object lies from the camera, the smaller it appears after projection. In addition, parallel lines may converge at the horizon. The perspective transform thus mimics the way we perceive objects' size. Geometrically, the view volume, called a frustum, is a truncated pyramid with rectangular base. The frustum is transformed into the unit cube as well. Both orthographic and perspective transforms can be constructed with 4x4 matrices (see Linear Transformations), and after either transform, the models are said to be in clip coordinates. These are in fact homogeneous coordinates, discussed below, and so this occurs before division by w. The GPU's vertex shader must always output coordinates of this type in order for the next functional stage, clipping, to work correctly. Although these matrices transform one volume into another, they are called projections because after display, the z-coordinate is not stored in the image generated but is stored in a z-buffer, described below. In this way, the models are projected from three to two dimensions.

Modern GPUs provide a number of optional vertex processing such as tesselation (amplifying geometry detail), geometry shading (generating new geometric primitives) and stream output (storing geometry in arrays for further processing). However these are quite advanced topics and beyond the scope of this short introduction. We refer interested students to [2] for an in-depth discussion of these topics.

### 1.2.2 Clipping

Only the primitives wholly or partially inside the view volume need to be passed on to the rasterization stage (and the subsequent pixel processing stage), which then draws them on the screen. A primitive that lies fully inside the view volume will be passed on to the next stage as is. Primitives entirely outside the view volume are not passed on further, since they are not rendered. It is the primitives that are partially inside the view volume that require clipping. For example, a line that has one vertex outside and one inside the view volume should be clipped against the view volume, so that the vertex that is outside is replaced by a new vertex that is located at the intersection between the line and the view volume. The use of a projection matrix means that the transformed primitives are clipped against the unit cube. The advantage of performing the view transformation and projection before clipping is that it makes the clipping problem consistent; primitives are always clipped against the unit cube. See Figure 1.15 for the clipping process.

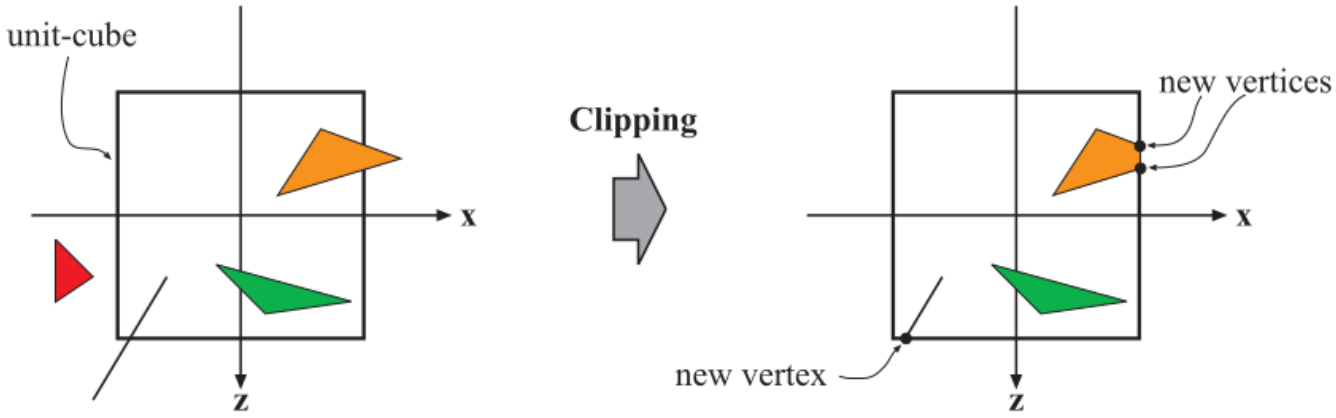


Figure 1.15: After the projection transform, only the primitives inside the unit cube (which correspond to primitives inside the view frustum) are needed for continued processing. Therefore, the primitives outside the unit cube are discarded, and primitives fully inside are kept. Primitives intersecting with the unit cube are clipped against the unit cube, and thus new vertices are generated and old ones are discarded (Figure taken from [2]).

The clipping step uses the 4-value homogeneous coordinates produced by projection to perform clipping. Values do not normally interpolate linearly across a triangle in perspective space. The fourth coordinate is needed so that data are properly interpolated and clipped when a perspective projection is used. Finally, perspective division is performed, which places the resulting triangles' positions into three-dimensional normalized device coordinates. As mentioned earlier, this view volume ranges from (-1, -1, -1) to (1, 1, 1). The last step in the geometry stage is to convert from this space to window coordinates.

### 1.2.3 Screen Mapping

Only the (clipped) primitives inside the view volume are passed on to the screen mapping stage, and the coordinates are still three-dimensional when entering this stage. The x- and y-coordinates of each primitive are transformed to form *screen coordinates*. Screen coordinates together with the z-coordinates are also called window coordinates. Assume that the scene should be rendered into a window with the minimum corner at ($x_1$ $y_1$) and the maximum corner at ($x_2$, $y_2$), where $x_1 < x_2$ and $y_1 < y_2$. Then the screen mapping is a translation followed by a scaling operation. The new x- and y-coordinates are said to be screen coordinates. The z-coordinate ([−1, +1] for OpenGL and [0, 1] for DirectX) is also mapped to [$z_1$ , $z_2$], with $z_1 = 0$ and $z_2 = 1$ as the default values. These can be changed with the API, however. The window coordinates along with this remapped z-value are passed on to the rasterizer stage. See Figure 1.16 for the process of screen mapping.

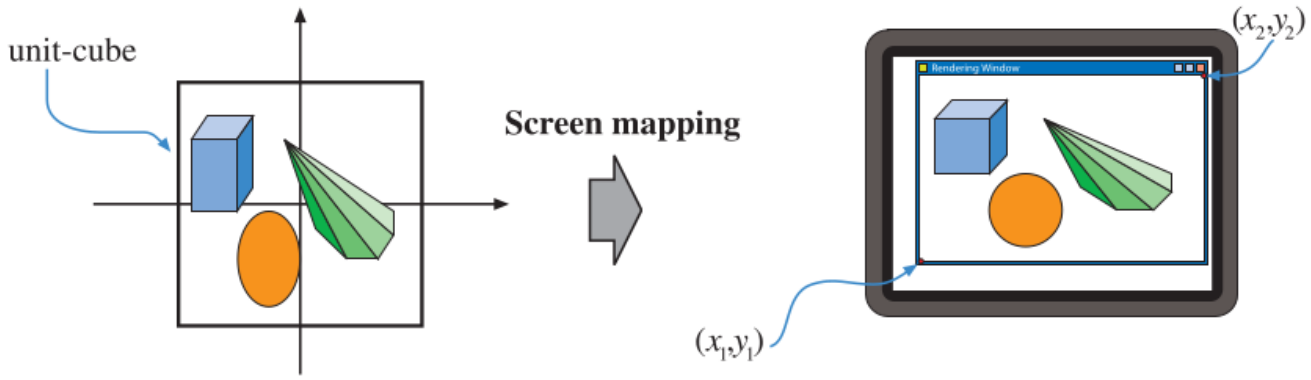


Figure 1.16: The primitives lie in the unit cube after the projection transform, and the screen mapping procedure takes care of finding the coordinates on the screen (Figure taken from [2]).

## 1.3 Rasterization Stage

Given the transformed and projected vertices with their associated shading data (all from geometry processing), the goal of the next stage is to find all pixels - short for picture elements - that are inside the primitive, e.g., a triangle, being rendered. We call this process *rasterization*, and it is split up into two functional substages: triangle setup (also called primitive assembly) and triangle traversal, see Figure 1.17.

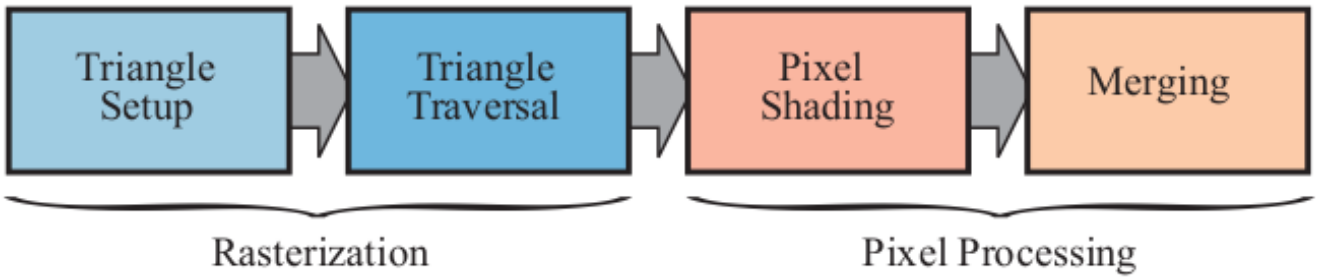


Figure 1.17: Left: rasterization split into two functional stages, called triangle setup and triangle traversal. Right: pixel processing split into two functional stages, namely, pixel processing and merging (Figure taken from [2]).

Rasterization, also called scan conversion, is thus the conversion from two-dimensional vertices in screen space - each with a z-value (depth value) and various shading information associated with each vertex - into pixels on the screen. Rasterization can also be thought of as a synchronization point between geometry processing and pixel processing, since it is here that triangles are formed from three vertices and eventually sent down to pixel processing.

### 1.3.1 Triangle Setup

In this stage the differentials, edge equations, and other data for the triangle are computed. These data may be used for triangle traversal, as well as for interpolation of the various shading data produced by the geometry stage. Fixed-function hardware is used for this task and is therefore not fully programmable through shaders.

### 1.3.2 Triangle Traversal

Here is where each pixel that has its center (or a sample) covered by the triangle is checked and a fragment generated for the part of the pixel that overlaps the triangle. Finding which samples or pixels are inside a triangle is often called triangle traversal. Each triangle fragment's properties are generated using data interpolated among the three triangle vertices. These properties include the fragment's depth, as well as any shading data from the geometry stage. It is also here that perspective-correct interpolation over the triangles is performed. All pixels or samples that are inside a primitive are then sent to the pixel processing stage, described next.

## 1.4 Pixel Processing Stage

At this point, all the pixels that are considered inside a triangle or other primitive have been found as a consequence of the combination of all the previous stages. The pixel processing stage is divided into pixel shading and merging, shown to the right of Figure 1.17. Pixel processing is the stage where per-pixel or per-sample computations and operations are performed on pixels or samples that are inside a primitive.

### 1.4.1 Pixel Shading

Any per-pixel shading computations are performed here, using the interpolated shading data as input. The end result is one or more colors to be passed on to the next stage. Unlike the triangle setup and traversal stages, which are usually performed by dedicated, hardwired silicon, the pixel shading stage is executed by programmable GPU cores. To that end, the programmer supplies a program for the pixel shader (or fragment shader, as it is known in OpenGL), which can contain any desired computations. A large variety of techniques can be employed here, one of the most important of which is texturing. Simply put, texturing an object means "gluing" one or more images onto that object, for a variety of purposes, see Figure 1.18.

Figure 1.18: A dragon model without textures is shown in the upper left. The pieces in the image texture are “glued” onto the dragon, and the result is shown in the lower left (Figure taken from [2]).

At its simplest, the end product is a color value for each fragment, and these are passed on to the next substage.

### 1.4.2 Merging

The information for each pixel is stored in the color buffer, which is a rectangular array of colors (a red, a green, and a blue component for each color). It is the responsibility of the merging stage to combine the fragment color produced by the pixel shading stage with the color currently stored in the buffer. This stage is also called ROP, standing for “raster operations (pipeline)”. Unlike the shading stage, the GPU subunit that performs this stage is typically not fully programmable. However, it is highly configurable, enabling various effects.

This stage is also responsible for resolving visibility. This means that when the whole scene has been rendered, the color buffer should contain the colors of the primitives in the scene that are visible from the point of view of the camera. For most or even all graphics hardware, this is done with the z-buffer (also called depth buffer) algorithm. A z-buffer is the same size and shape as the color buffer, and for each pixel it stores the z-value to the currently closest primitive. This means that when a primitive is being rendered to a certain pixel, the z-value on that primitive at that pixel is being computed and compared to the contents of the z-buffer at the same pixel. If the new z-value is smaller than the z-value in

the z-buffer, then the primitive that is being rendered is closer to the camera than the primitive that was previously closest to the camera at that pixel. Therefore, the z-value and the color of that pixel are updated with the z-value and color from the primitive that is being drawn. If the computed z-value is greater than the z-value in the z-buffer, then the color buffer and the z-buffer are left untouched. The z-buffer algorithm is simple, has O(n) convergence (where n is the number of primitives being rendered), and works for any drawing primitive for which a z-value can be computed for each (relevant) pixel. Also note that this algorithm allows most primitives to be rendered in any order, which is another reason for its popularity. However, the z-buffer stores only a single depth at each point on the screen, so it cannot be used for partially transparent primitives. These must be rendered after all opaque primitives, and in back-to-front order, or using a separate order-independent algorithm.

When the primitives have reached and passed the rasterizer stage, those that are visible from the point of view of the camera are displayed on screen. The screen displays the contents of the color buffer. To avoid allowing the human viewer to see the primitives as they are being rasterized and sent to the screen, double buffering is used. This means that the rendering of a scene takes place off screen, in a back buffer. Once the scene has been rendered in the back buffer, the contents of the back buffer are swapped with the contents of the front buffer that was previously displayed on the screen. The swapping often occurs during vertical retrace, a time when it is safe to do so.

## 1.5 From 3D Model to 2D Screen Coordinates

After having discussed the graphics rendering pipeline from a general, high-level perspective, in this section we provide a more detailed description in technical terms of how to arrive from a 3D model at 2D coordinates. The key tools for projecting three dimensions down to two are a *viewing model*, use of *homogeneous coordinates*, application of linear *transformations* by matrix multiplication, and setting up a viewport mapping. The common transformation process for producing the desired view is analogous to taking a photograph with a camera, see Figure 1.19.

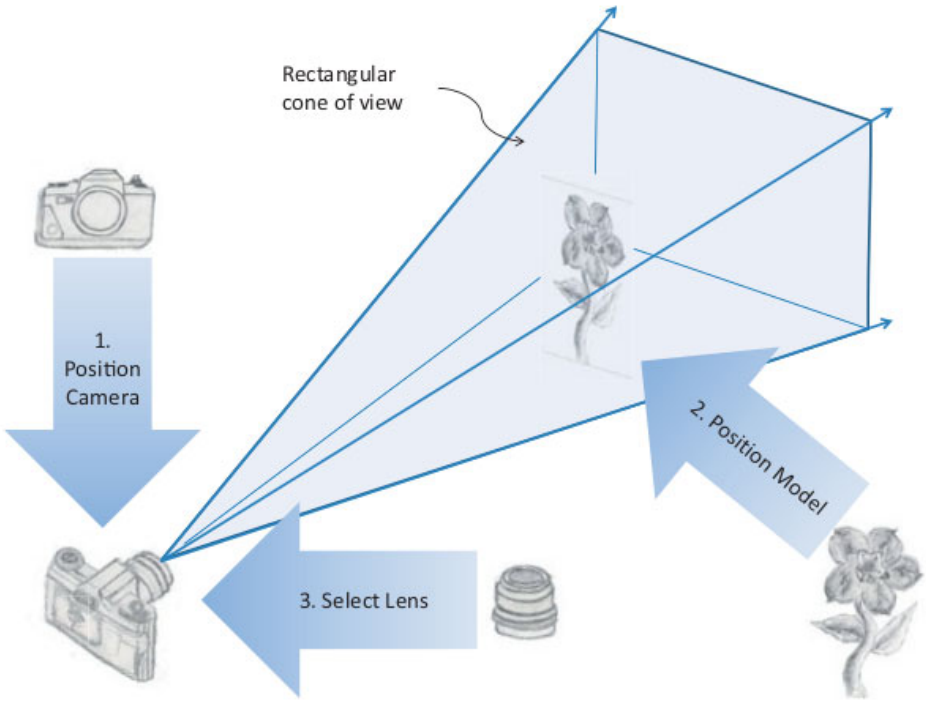


Figure 1.19: Steps in configuring and positioning the viewing frustum (Figure taken from [27]).

1. Move your camera to the location you want to shoot from and point the camera the desired direction (viewing transformation).
2. Move the subject to be photographed into the desired location in the scene (modeling transformation).
3. Choose a camera lens or adjust the zoom (projection transformation).
4. Take the picture (apply the transformations).
5. Stretch or shrink the resulting image to the desired picture size (viewport transformation).

Notice that Steps 1 and 2 can be considered doing the same thing, but in opposite directions. Because of this, these two steps are normally lumped together as the model-view transform. It will, though, always consist of some sequence of movements (translations), rotations, and scalings. The defining characteristic of this combination is in making a single, unified space for all the objects assembled into one scene to view, or eye space

Figure 1.20 shows a description of the full transformation process the model coordinates go through when rendering them onto screen.

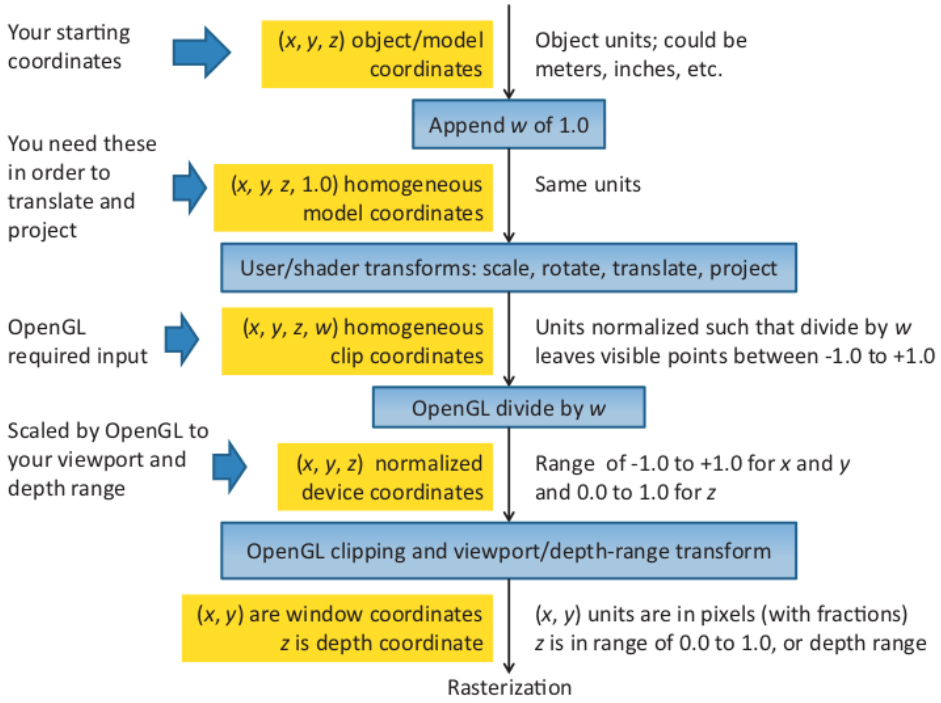


Figure 1.20: Coordinate systems for transforming from 3D model coordinates into 2D screen coordinates (Figure taken from [27]).

Lets have a look at the visualisation of the various transformation steps. We start out with the model matrix. A model is defined by a set of vertices. The x,y,z coordinates of these vertices are defined relative to the object's center: that is, if a vertex is at (0,0,0), it is at the center of the object, see Figure 1.21.

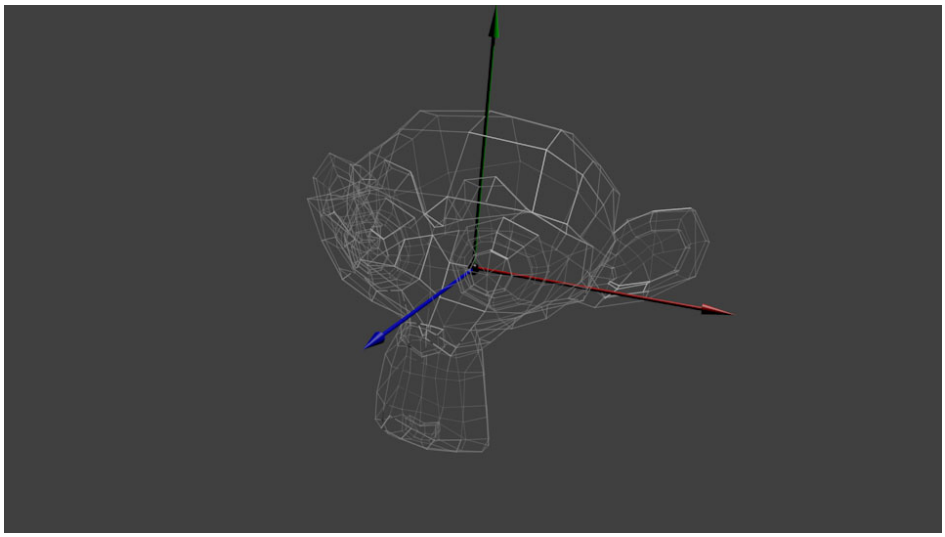

Figure 1.21: A model in model space (Figure taken from http://www.opengl-tutorial.org /beginners-tutorials/tutorial-3-matrices/).

To put the model into an absolute space which is the same for all models, putting models in relation to each other, the model transformation is used. As the point of reference the *center of the world* is used, see Figure 1.22.

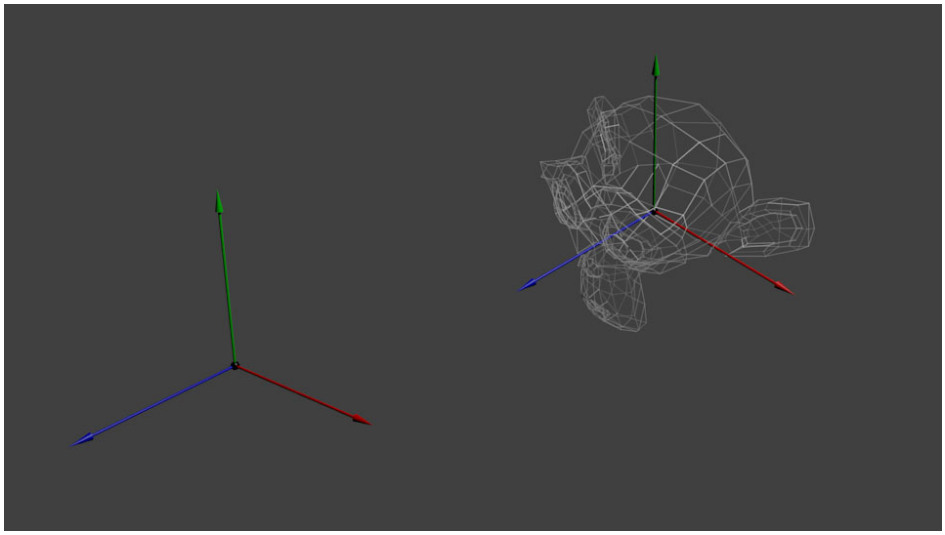

Figure 1.22: Transforming the model to world space (Figure taken from http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/).

The model is now in world space, which means that all vertices of the model are transformed relatively to the center of the world, see Figure 1.23.

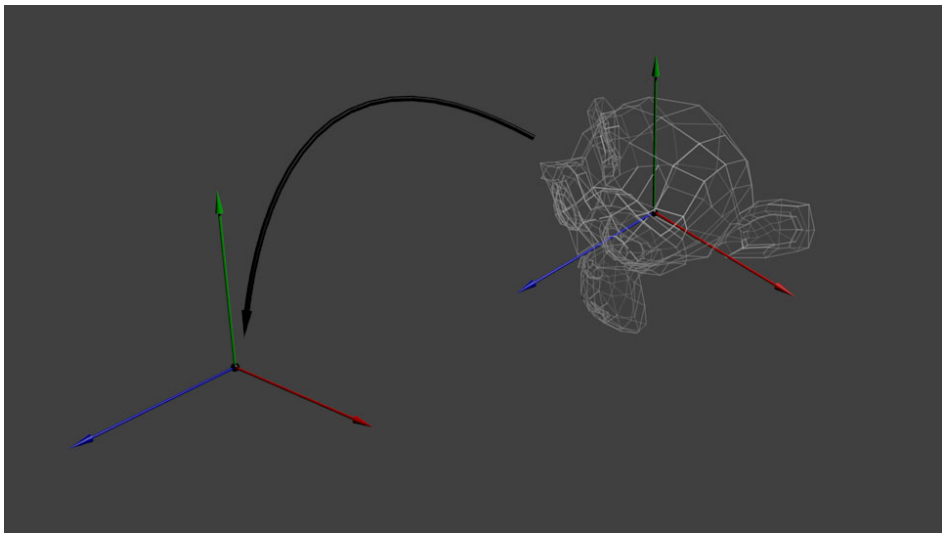

Figure 1.23: The center of the world as absolute reference point (Figure taken from http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/).

We went from *model coordinates* to *world coordinates* using the *model matrix*. Next follows the view matrix, which deals with the camera. It you want to view a moutain from another angle, you can either move the camera… or move the mountain. While not practical in real life, this is really simple and handy in computer graphics. Initially the camera is at the origin of the world space. In order to move the world, you simply introduce another matrix. Let's say you want to move your camera of 3 units to the right (+x). This is equivalent to moving your whole world (meshes included) 3 units to the left! (-x), see Figure 1.24.

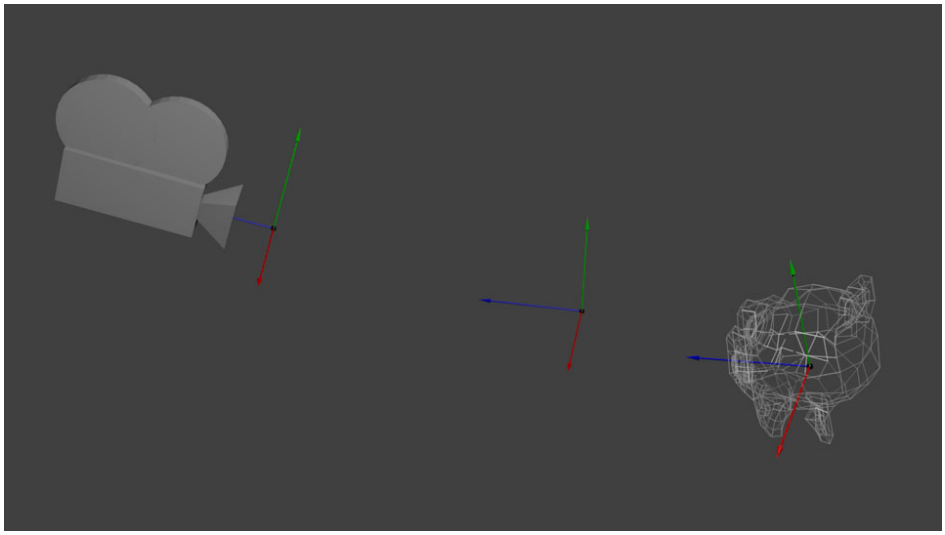

Figure 1.24: The camera in world space (Figure taken from http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/).

Now to move from model space to camera space we combine the modeling transformation and the viewing transforamtion, which transforms the model coordinates to world coordinates to viewing (camera) coordinates, relative to the camera, see Figure 1.25.

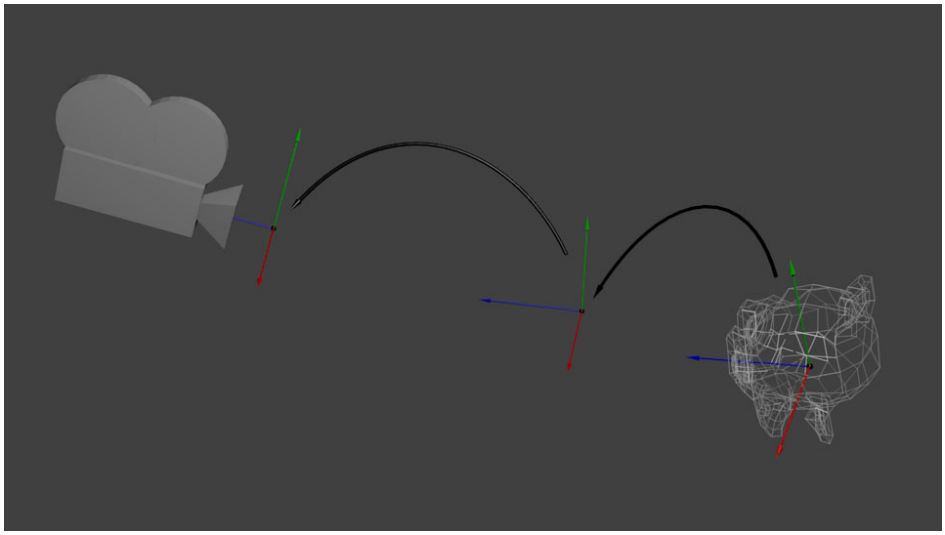

Figure 1.25: From model to camera space (Figure taken from http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/).

We went now from *world coordinates* to *camera coordinates* using the *viewing matrix*. We are now in camera space. Now we apply the perspective projection which will cause objects farther away from the camera appear smaller and objects closer to the camer appear larger, see Figure 1.26.

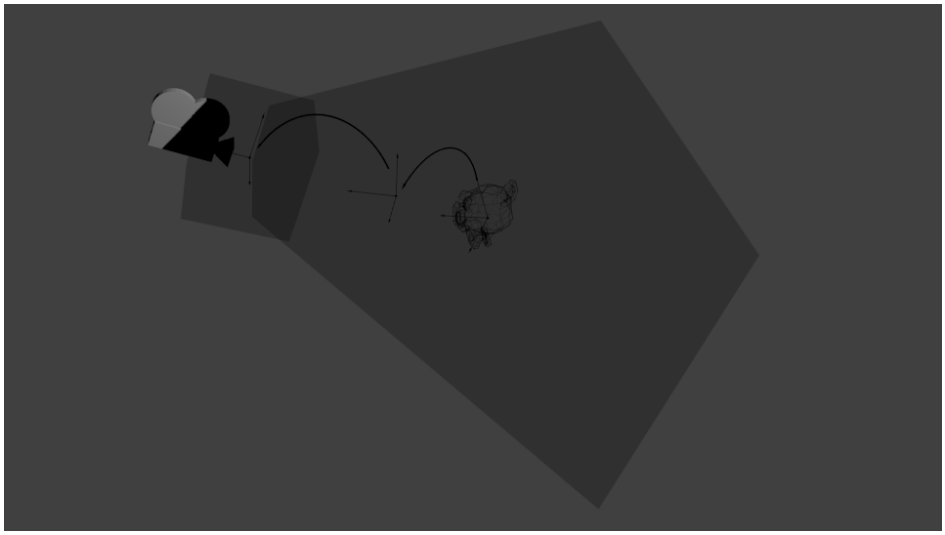

Figure 1.26: Perspective projection transformation (Figure taken from http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/).

The projection transformation puts coordinates from camera coordinates to homogeneous clipping coordinates. Instead of understanding the perspective projection as a view frustum, as in Figure 1.27 we can also see it as the unit cube (-1 to +1 on all axes), see Figure 1.28.

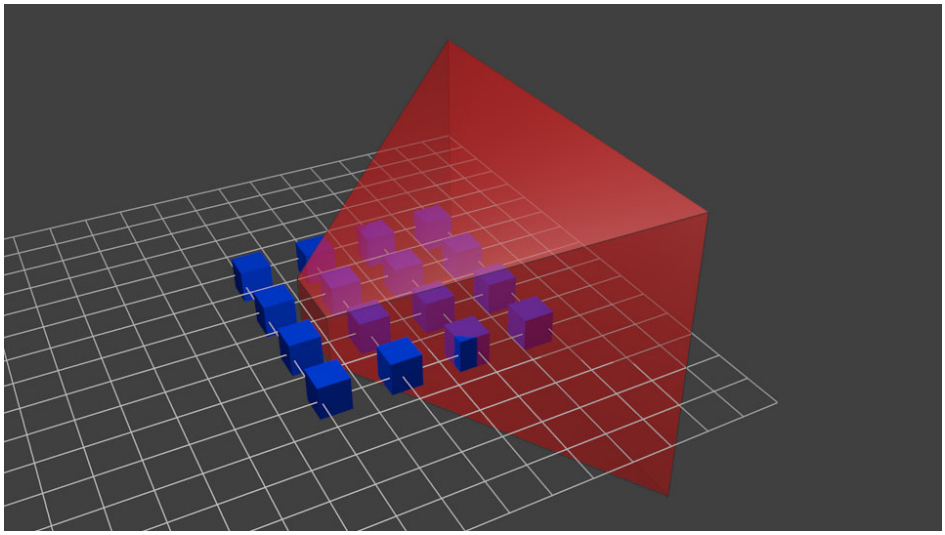

Figure 1.27: Frustum projection (Figure taken from http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/).

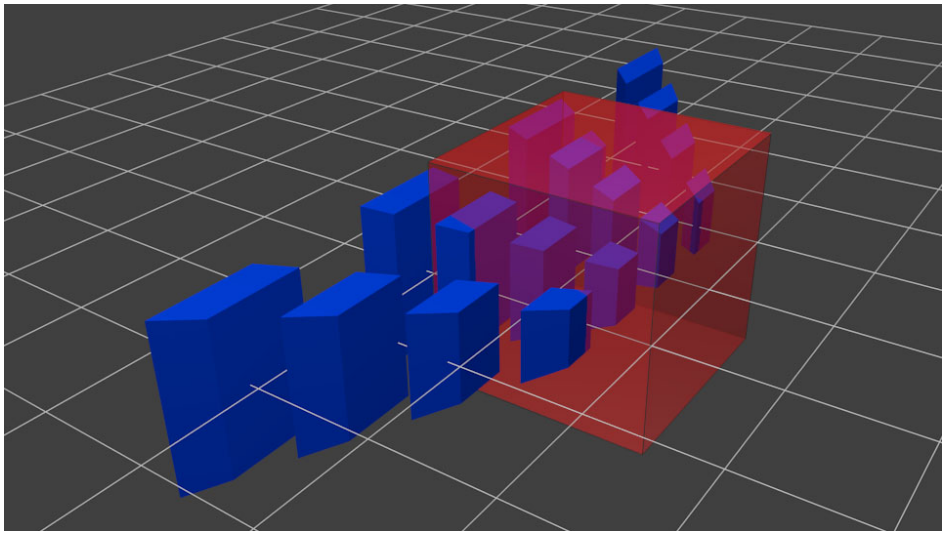

Figure 1.28: Unit cube projection (Figure taken from http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/).

Having the homogeneous coordinates we need to perform the perspective division to achieve the foreshortening, ending up in normalised device coordinates (NDC) in the range of [-1, +1] for all coordinates, see Figure 1.29.

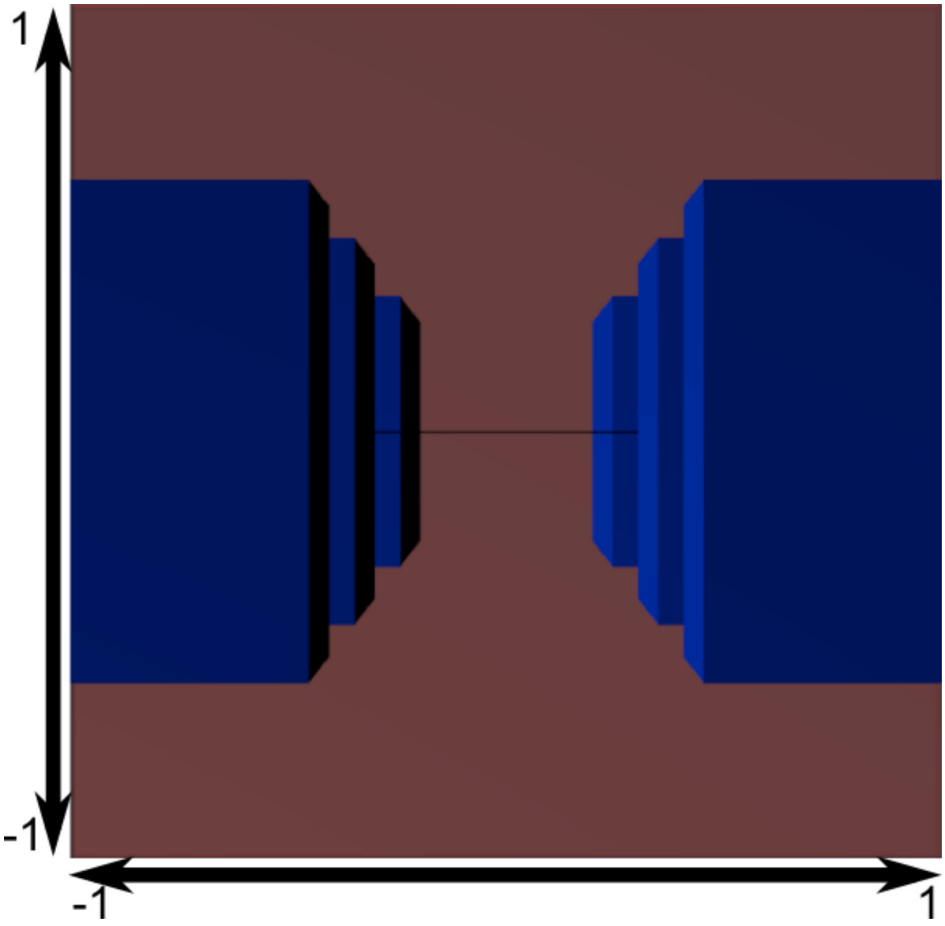


Figure 1.29: Perspective division (Figure taken from http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/).

Finally we perform the transform from NDC to the screen coordinates with the viewport transformation, see Figure 1.30.

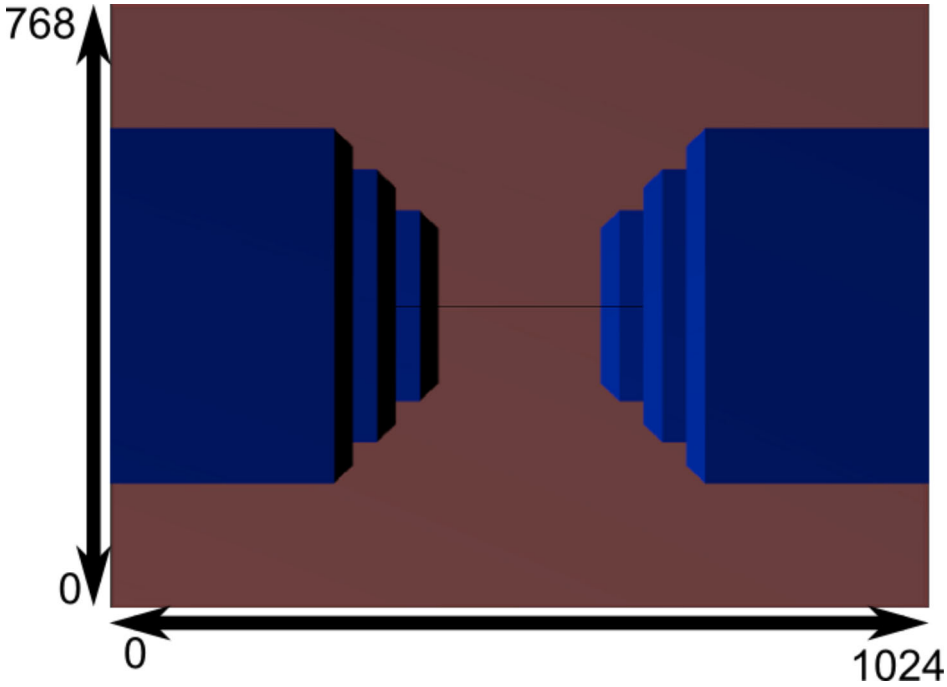


Figure 1.30: Viewport transformation to screen space (Figure taken from http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/).

Figure 1.31 summarises all the transformations the coordinates go through until they end up on the screen.

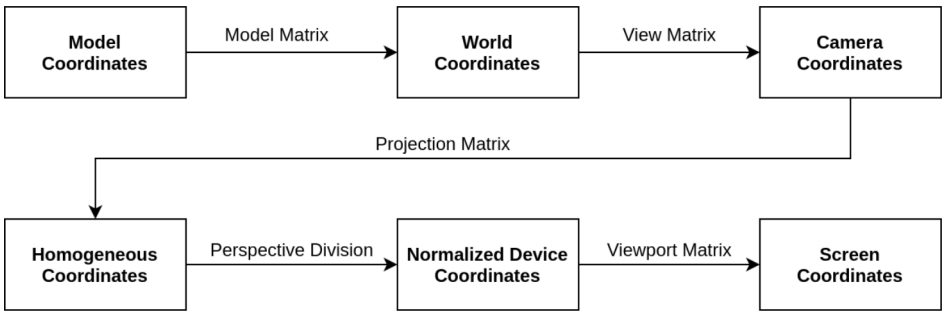


Figure 1.31: Transformations from Model Space to Screen Space.

More visually:

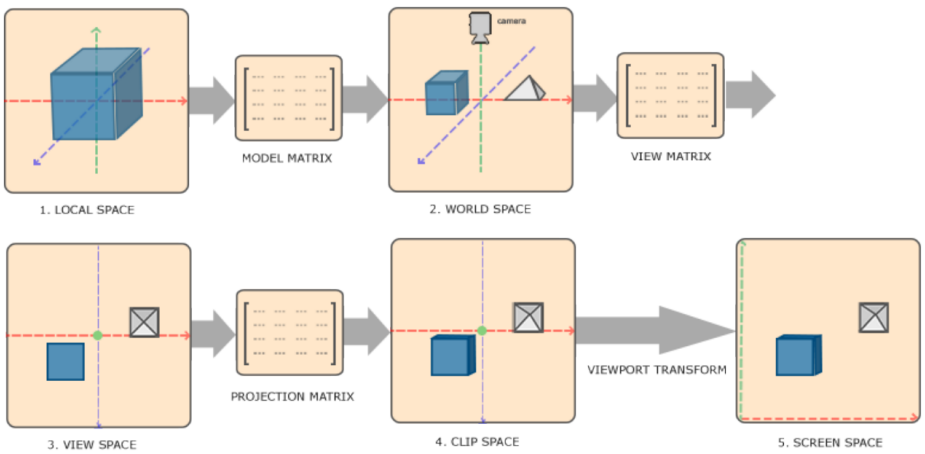


Figure 1.32: Transformations from model space to screen space (Figure from [29]).

### 1.5.1 Linear Transformations

Matrices and matrix multiplication are nothing more than a convenient mechanism for expressing linear transformations, which in turn are a useful way to do the coordinate manipulations needed for displaying models. The vital matrix mechanism is explained here, while interesting uses for it will come up in numerous places in subsequent discussions. First, a definition. A 4x4 matrix takes a four-component vector to another four-component vector through multiplication by the following rule:

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \rightarrow \begin{pmatrix} ax + by + cz + dw \\ ex + fy + gz + hw \\ ix + jy + kz + lw \\ mx + ny + oz + pw \end{pmatrix}$$

Figure 1.33: Matrix multiplication (Figure taken from [27]).

Each component of the new vector is a linear function of all the components of the old vector, hence the need for 16 values in the matrix. The multiplication always takes the vector (0, 0, 0, 0) to (0, 0, 0, 0). This is characteristic of linear transformations and shows that if this was a 3x3 matrix times a three-component vector, why translation (moving) can't be done with a matrix multiply. We'll see how translating a three-component vector becomes possible with a 4x4 matrix and homogeneous coordinates later on.

In our viewing models, we will want to take a vector *v* through multiple transformations, here expressed as matrix multiplications by matrices *A* and then *B*:

$$v' = Av$$
$$v'' = Bv' = B(Av) = (BA)v$$

To be efficiently able to do this we can therefore compose the matrix transformation *B* and *A* into a single transform *C*, therefore requiring only a single matrix multiplication instead of two:

$$v'' = Cv$$
$$C = BA$$

It is important to note that matrix multiplication is not commutative, that is $AB \neq BA$ and therefore also multiplication with a vector $Av \neq vA$. However matrix multiplication is associative:

$$C(BA) = (CB)A = CBA$$

Therefore we can re-associate the accumulated matrix multiplications as:

$$C(B(Av)) = (CBA)v$$

## 1.5.2 Homogenous Coordinates

Three-dimensional data can be scaled and rotated with linear transformations of three-component vectors by multiplying by 3x3 matrices.

Unfortunately, translating (moving/sliding over) three-dimensional Cartesian coordinates cannot be done by multiplying with a 3x3 matrix. It requires an extra vector addition to move the point (0, 0, 0) somewhere else. This is a called an affine transformation, which is not a linear transformation. (Recall that any linear transformation maps (0, 0, 0) to (0, 0, 0)). Including that addition means the loss of the benefits of linear transformations, like the ability to compose multiple transformations into a single transformation. So, we want to find a way to translate with a linear transformation. Fortunately, by embedding our data in a four-coordinate space, affine transformations turn back into a simple linear transform (meaning you can move your model laterally using only multiplication by a 4x4 matrix). So the advantage of using homogenous coordinates is that 1) it allows to apply perspective and 2) it allows also to capture translation using only a linear transformation. This has the consequence that we are able to get all the rotations, translations, scaling, and projective transformations we need by doing matrix multiplication if we first move to a four-coordinate system.

For example, to move data by 0.3 in the y direction, assuming a fourth vector coordinate of 1.0:

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.3 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1.0 \end{pmatrix} \rightarrow \begin{pmatrix} x \\ y + 0.3 \\ z \\ 1.0 \end{pmatrix}$$

At the same time, we acquire the extra component needed to do perspective. A homogeneous coordinate has one extra component and does not change the point it represents when all its components are scaled by the same amount. For example, all these coordinates represent the same point:

$$\begin{pmatrix} 2.0 \\ 3.0 \\ 5.0 \\ 1.0 \end{pmatrix} \begin{pmatrix} 4.0 \\ 5.0 \\ 10.0 \\ 2.0 \end{pmatrix} \begin{pmatrix} 0.2 \\ 0.3 \\ 0.5 \\ 0.1 \end{pmatrix}$$

In this way, homogeneous coordinates act as directions instead of locations; scaling a direction leaves it pointing in the same direction. Standing at (0, 0), the homogeneous points (1, 2), (2, 4), and others along that line appear in the same place. When projected onto the 1D space, they all become the point 2, see Figure 1.34.

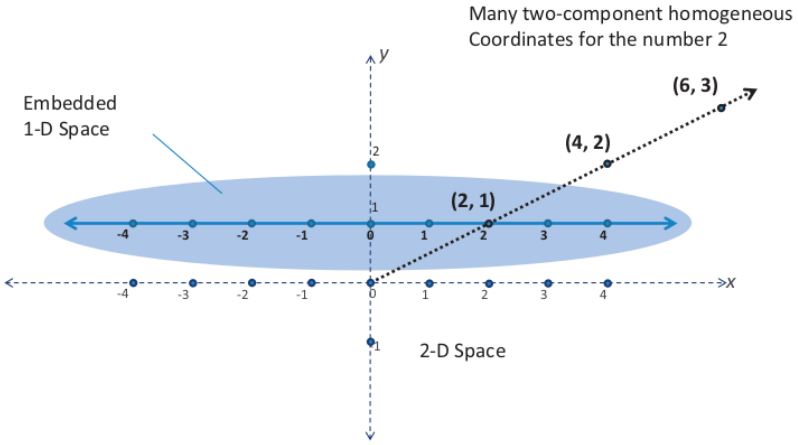


Figure 1.34: One-dimensional homogeneous space which shows how to embed the 1D space into two dimensions, at the location y = 1, to get homogeneous coordinates (Figure taken from [27]).

The desire is to translate points in the 1D space with a linear transform. This is impossible within the 1D space, as the point 0 needs to move - something 1D linear transformations cannot do. However, employing a skewing transformation allows to translate the embedded 1D space as shown in Figure 1.34 while preserving the location of (0,0) in the 2d space (all linear transforms keep (0, 0) fixed), see Figure 1.35.

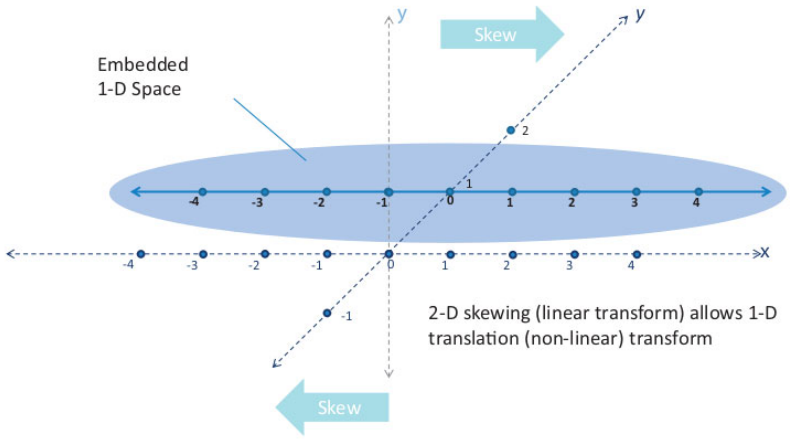


Figure 1.35: Translation by skewing (Figure taken from [27]).

If the last component of an homogeneous coordinate is 0, it implies a *point at infinity*. The 1D space has only two such points at infinity, one in the positive direction and one in the negative direction. However, the 3D space, embedded in a four-coordinate homogeneous space, has a point at infinity for any direction you can point. These points can model the perspective point where two parallel lines (e.g., sides of a building or railroad tracks) would

appear to meet. The perspective effects we care about, though, will become visible without needing to specifically think about this.

We will move to homogeneous coordinates by adding a fourth w component of 1.0:

$$\begin{pmatrix} 3.0 \\ 4.0 \\ 5.0 \end{pmatrix} \rightarrow \begin{pmatrix} 3.0 \\ 4.0 \\ 5.0 \\ 1.0 \end{pmatrix}$$

If we want to go back to cartesian coordinates later, we can do this by dividing all components by the fourth component and dropping the fourth component:

$$\begin{pmatrix} 4.0 \\ 6.0 \\ 10.0 \\ 2.0 \end{pmatrix} \xrightarrow{divide\ by\ w} \begin{pmatrix} 2.0 \\ 3.0 \\ 5.0 \\ 1.0 \end{pmatrix} \xrightarrow{drop\ w} \begin{pmatrix} 2.0 \\ 3.0 \\ 5.0 \end{pmatrix}$$

Perspective transforms modify w components to values other than 1.0. Making w larger can make coordinates appear further away. When it's time to display geometry, OpenGL will transform homogeneous coordinates back to the three-dimensional Cartesian coordinates by dividing their first three components by the last component. This will make the objects farther away (now having a larger w) have smaller Cartesian coordinates, hence getting drawn on a smaller scale. A w of 0.0 implies (x, y) coordinates at infinity (the object got so close to the viewpoint that its perspective view got infinitely large). This can lead to undefined results.

## 1.5.3 Transformations

We start our task of mapping into device coordinates by adding a fourth component to our three-dimensional Cartesian coordinates, with a value of 1.0, to make homogeneous coordinates. These coordinates are then ready to be multiplied by one or more 4x4 matrices that rotate, scale, translate, and apply perspective. Examples of how to use each of these transforms are given below. The summary is that each of these transformations can be made through multiplication by a 4x4 matrix, and a series of such transformations can be composed into a single 4x4 matrix, once, that can then be used on multiple vertices.

### 1.5.3.1 Translation

Translating an object takes advantage of the fourth component we just added to our model

coordinates and of the fourth column of a 4x4 transformation matrix. We want a matrix T to multiply all our object's vertices v by to get translated vertices v'.

Each component can be translated by a different amount by putting those amounts in the fourth column of T. For example, to translate by 2.5 in the positive x direction, and not at all in the y or z directions:

$$T = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 2.5 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

and multiplying a vector v = (x,y,z,1) gives

$$\begin{pmatrix} x + 2.5 \\ y \\ z \\ 1.0 \end{pmatrix} = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 2.5 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1.0 \end{pmatrix}$$

Therefore, the general form of a translation matrix and its inverse is:

$$T = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad T^{-1} = \begin{bmatrix} 1 & 0 & 0 & -x \\ 0 & 1 & 0 & -y \\ 0 & 0 & 1 & -z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

See Figure 1.36 for a translation transformation:

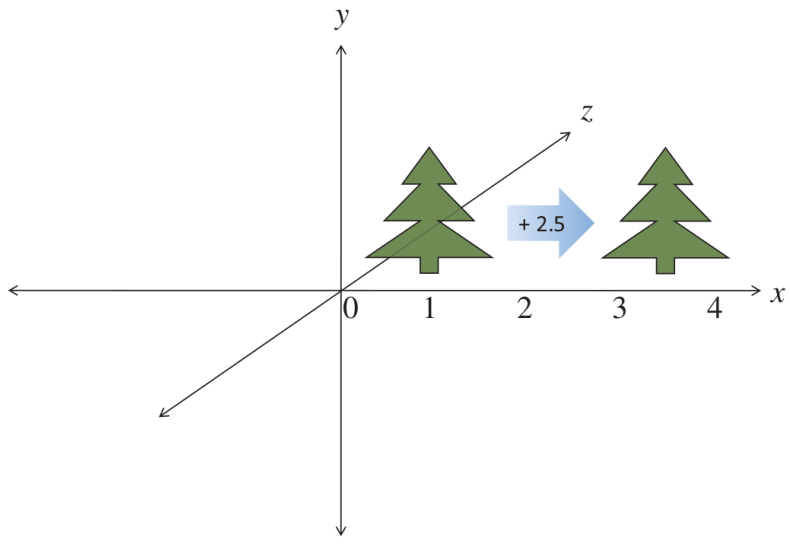


Figure 1.36: Translating an object 2.5 units in the x direction (Figure taken from [27]).

### 1.5.3.2 Scaling

Growing or shrinking an object can be done by putting the desired scaling factor on the first three diagonal components of the matrix. Making a scaling matrix S, which applied to all vertices v in an object, would change its size. Also nonisomorphic scaling is possible, for example to mirror around the y axis, simply set a negative scaling factor. The following transformation makes an object 3 times lager:

$$S = \begin{bmatrix} 3.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 3.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 3.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

$$\begin{pmatrix} 3x \\ 3y \\ 3z \\ 1.0 \end{pmatrix} = \begin{bmatrix} 3.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 3.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 3.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1.0 \end{pmatrix}$$

Therefore, the general form of a scaling matrix and its inverse is:

$$S = \begin{bmatrix} x & 0 & 0 & 1 \\ 0 & y & 0 & 1 \\ 0 & 0 & z & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad S^{-1} = \begin{bmatrix} \frac{1}{x} & 0 & 0 & 1 \\ 0 & \frac{1}{y} & 0 & 1 \\ 0 & 0 & \frac{1}{z} & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

See Figure 1.37 for a scaling transformation.

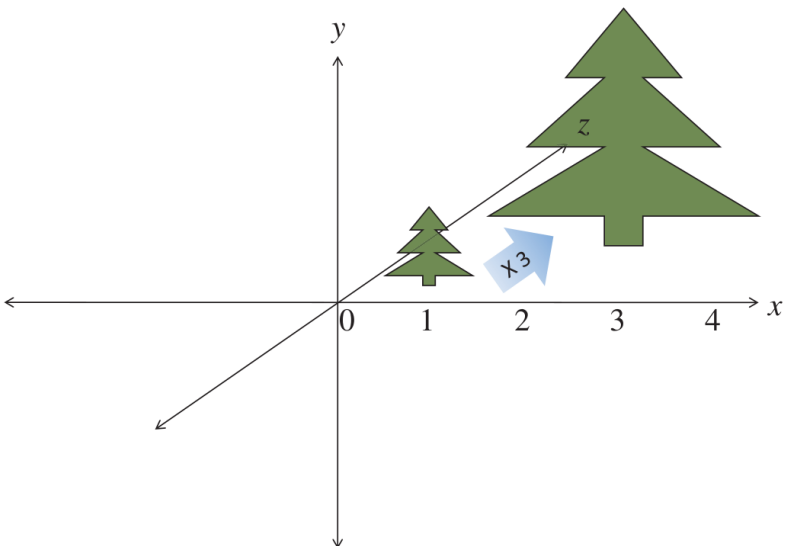


Figure 1.37: Scaling an object to three times its size. Note that if the object is off the center, this also moves its center three times further from (0,0,0) (Figure taken from [27]).

If the object being scaled is not centered at (0, 0, 0), the simple matrix above will also move it further or closer to (0, 0, 0) by the scaling amount. Usually, it is easier to understand what

happens when scaling if you first center the object around (0, 0, 0). Then scaling leaves it in the same place while changing its size. If you want to change the size of an off-center object without moving it, first translate its center to (0, 0, 0), then scale it, and finally translate it back, see Figure 1.38.

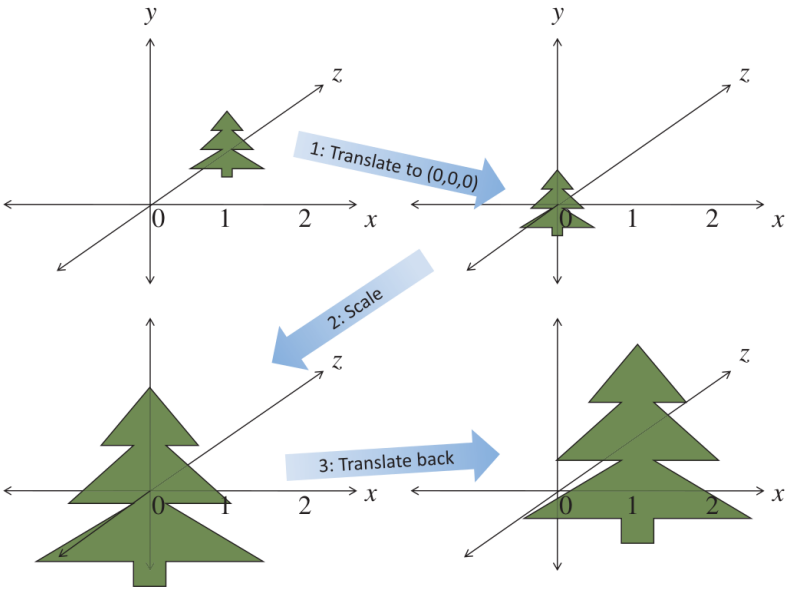


Figure 1.38: Scaling an object with respect to its center by moving to the origin, scaling and then moving back (Figure taken from [27]).

This would use three matrices, T, S and $T^{-1}$ for translate to (0, 0, 0), scale, and translate back, respectively. When each vertex v of the object is multiplied by each of these matrices in turn, the final effect is that the object would change size in place, yielding a new set of vertices v':

$$v' = T^{-1}(S(Tv))$$
$$v' = (T^{-1}ST)v)$$

which allows for pre-multiplication of the three matrices into a single matrix.

$$M = T^{-1}ST$$
$$v' = Mv$$

M now does the complete job of scaling an off-center object.

### 1.5.3.3 Rotation

Rotating an object follows a similar scheme. We want a matrix R that when applied to all vertices v in an object will rotate it. The following example, shown in Figure 1.39 rotates 50 degrees counterclockwise around the z axis.

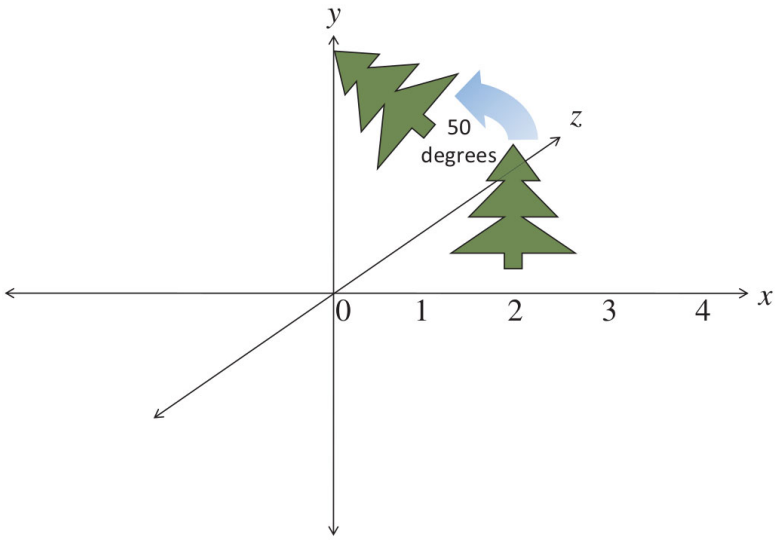


Figure 1.39: Rotating an object 50 degrees in the xy plane, around the z axis: only the x- and y-coordinates of the object change and the z stay constant. Note that if the object is off center, it also revolves the object around the point (0,0,0) (Figure taken from [27]).

The corresponding rotation matrix is:

$$R = \begin{bmatrix} cos\,50 & -sin\,50 & 0.0 & 0.0 \\ sin\,50 & cos\,50 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

$$\begin{pmatrix} cos\,50\,x - sin\,50\,y \\ sin\,50\,x + cos\,50\,y \\ z \\ 1.0 \end{pmatrix} = \begin{bmatrix} cos\,50 & -sin\,50 & 0.0 & 0.0 \\ sin\,50 & cos\,50 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1.0 \end{pmatrix}$$

When rotating around the z axis above, the vertices in the object keep their z values the same, rotating in the xy plane. To rotate instead around the x axis by an amount $\theta$:

$$R_x = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & cos\,\theta & -sin\,\theta & 0.0 \\ 0.0 & sin\,\theta & cos\,\theta & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

To rotate around the y axis by an amount $\theta$:

$$R_y = \begin{bmatrix} cos\,\theta & 0.0 & -sin\,\theta & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ sin\,\theta & 0.0 & cos\,\theta & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

If the object being rotated is not centered at (0, 0, 0), the matrices above will also rotate the whole object around (0, 0, 0), changing its location. Again, as with scaling, it'll be easier to first center the object around (0, 0, 0). So, again, translate it to (0, 0, 0), transform it, and then translate it back. This could use three matrices, T, R, and $T^{-1}$, to translate to (0, 0, 0), rotate, and translate back, see Figure 1.40.

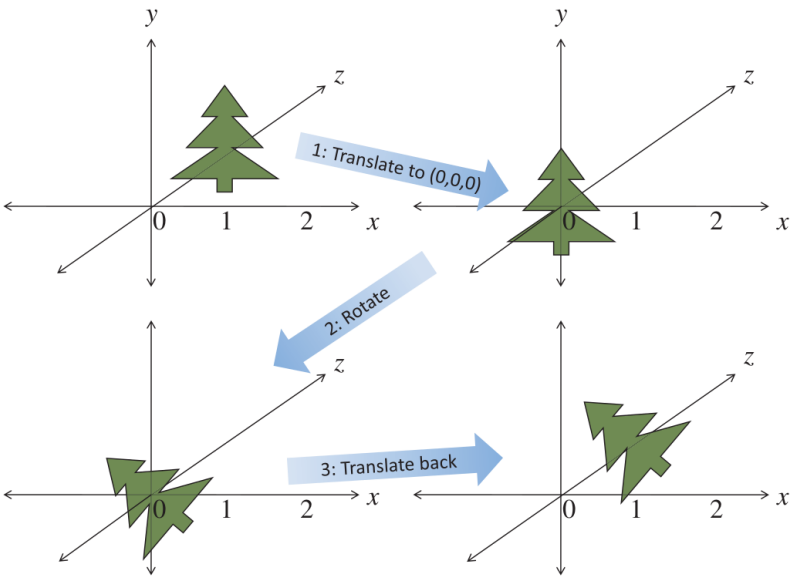


Figure 1.40: Rotating an object 50 degrees around its center, in place, by moving it to the origin, rotating and moving it back (Figure taken from [27]).

### 1.5.3.4 Perspective Projection

We now assume viewing and modeling transformations are completed, with larger z values meaning objects are further away. For all, the viewpoint is now at (0, 0, 0), looking generally toward the positive z direction. Let's consider an over-simplified (hypothetical) perspective projection.

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1.0 \end{pmatrix} \rightarrow \begin{pmatrix} x \\ y \\ z \\ z \end{pmatrix}$$

Note the last matrix row replaces the w (fourth) coordinate with the z coordinate. This will make objects with a larger z (further away) appear smaller when the division by w occurs, creating a perspective effect. However, this particular method has some shortcomings. For one, all z values will end up at 1.0, losing information about depth. We also didn't have much control over the cone we are projecting and the rectangle we are projecting onto. Finally, we didn't scale the result to the [−1.0, 1.0] range expected by the viewport transform. Using a proper view frustum projection takes this into account, see Figure 1.41.

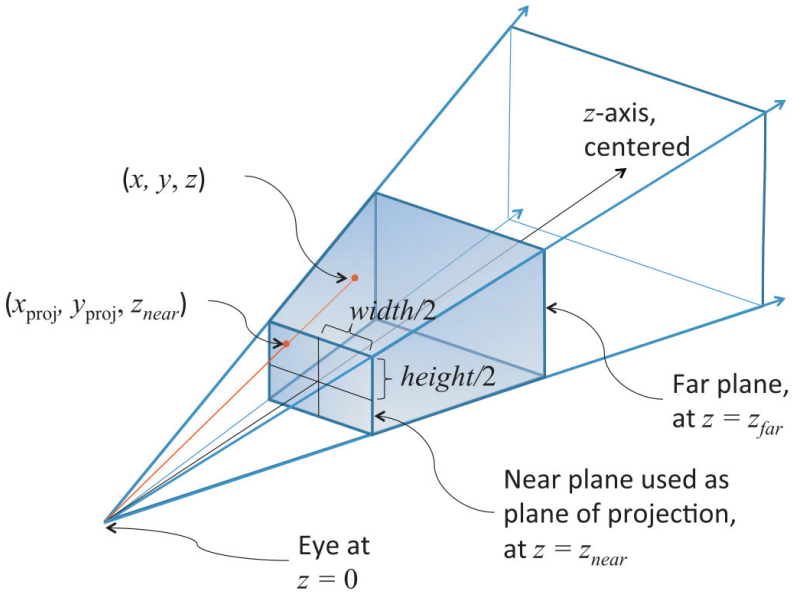


Figure 1.41: Frustum projection (Figure taken from [27]).

We want to project points in the frustum onto the near plane, directed along straight lines going toward (0, 0, 0). Any straight line emanating from (0, 0, 0) keeps the ratio if z to x the same for all its points, and similarly for the ratio of z to y. Thus, the $(x_{proj}, y_{proj})$ value of the projection on the near plane will keep the ratios of $\frac{z_{near}}{z} = \frac{x_{proj}}{x}$ and $\frac{z_{near}}{z} = \frac{y_{proj}}{y}$. We know there is an upcoming division by depth to eliminate homogeneous coordinates, so solving for $x_{proj}$ while still in the homogeneous space simply gives $x_{proj} = xz_{near}$. Similarly, $y_{proj} = yz_{near}$. If we then include a divide by the size of the near plane to scale the near plane to the range of [−1.0, 1.0], we end up with the requisite first two diagonal elements shown in the projection transformation matrix.

$$\begin{bmatrix} \frac{z_{near}}{width/2} & 0.0 & 0.0 & 0.0 \\ 0.0 & \frac{z_{near}}{height/2} & 0.0 & 0.0 \\ 0.0 & 0.0 & -\frac{z_{far}+z_{near}}{z_{far}-z_{near}} & \frac{2z_{far}+z_{near}}{2z_{far}-z_{near}} \\ 0.0 & 0.0 & -1.0 & 0.0 \end{bmatrix}$$

### 1.5.3.5 Perspective Division to Normalised Device Coordinates

After perspective projection, the resulting vectors, still having four coordinates, are the homogeneous coordinates expected by the OpenGL pipeline, also known as homogeneous clip coordinates. However, the perspective projection doesn't actually create the 3D effect; for that, we need to do something called the perspective divide. Each coordinate in OpenGL actually has four components, x, y, z and w. The projection matrix sets things up so that after multiplying with the projection matrix, each coordinate's w will increase the further away the object is. The further away something is, the more it will be pulled towards the

center of the screen, see Figure 1.42.

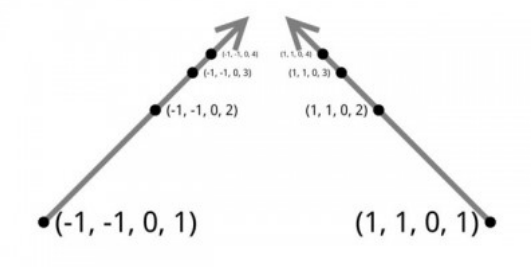


Figure 1.42: Perspective division (Figure taken from https://www.learnopengles.com/tag/perspective-divide/).

Therefore, the next step in projecting the perspective view onto the screen is to divide the (x, y, z) coordinates in v' by the w coordinate in v', for every vertex:

$$v' = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \rightarrow \begin{pmatrix} x/w \\ y/w \\ z/w \\ 1 \end{pmatrix}$$

This will transform the coordinates into so called Normalised Device Coordinates (NDC), which is a transformation of the coordinates into the unit cube which all axes are in the range of [-1,+1]. The advantage of having these coordinates is that they form a neutral basis for transforming to subsequent different resolutions of the screen space. See Figure 1.43 for normalised device coordinates.

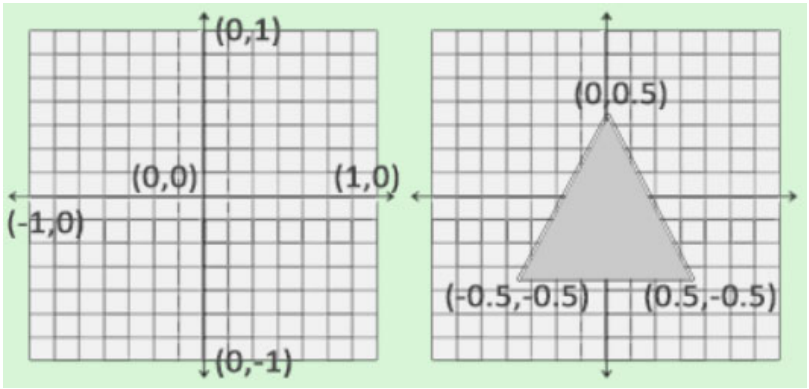


Figure 1.43: Normalised device coordinates (Figure taken from [29]).

### 1.5.3.6 Viewport Transformation

Finally, the viewport transformation is applied which transforms the 3 dimensional NDC into 2 dimensional screen coordinates. This is done by applying the NDC transform and then simply dropping the z and w component of the resulting vector.

## 1.5.4 Transforming Normals

Each 3D model has at least vertex coordinates of the triangles which make up the model. However, it is very useful to have also a normal vector (or *normals* for short) for each vertex, which are vectors that point in the direction perpendicular to a surface at some point. They are necessary for lighting, shading and other advanced rendering techniques. It is important to understand that normals are transformed different than vertices as they denote *directions* and not positions.

Normal vectors are typically only three-component vectors; not using homogeneous coordinates. For one thing, translating a surface does not change its normal, so normals don't care about translation, removing one of the reasons we used homogeneous coordinates. Since normals are mostly used for lighting, which we complete in a pre-perspective space, we remove the other reason we use homogeneous coordinates (projection).

Perhaps counterintuitively, normal vectors aren't transformed in the same way as vertices or position vectors are. Imagine a surface at an angle that gets stretched by a transformation. Stretching makes the angle of the surface shallower, which changes the perpendicular direction in the opposite way than applying the same stretching to the normal would. This would happen, for example, if you stretch a sphere to make an ellipse, see Figure 1.44 We need to come up with a different transformation matrix to transform normals than the one we used for vertices.

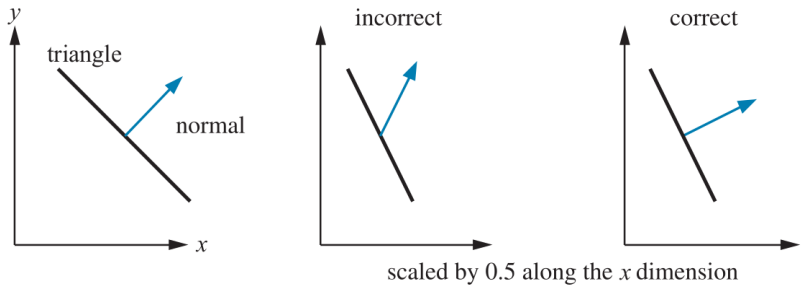


Figure 1.44: On the left is the original geometry, a triangle and its normal shown from the side. The middle illustration shows what happens if the model is scaled along the x- axis by 0.5 and the normal uses the same matrix. The right figure shows the proper transform of the normal. (Figure taken from [2]).

So, how do we transform normals? To start, let M be the 3x3 matrix that has all the rotations and scaling needed to transform your object from model coordinates to eye coordinates, before transforming for perspective. This would be the upper 3x3 block in your 4x4 transformation matrix, before compounding translation or projection transformations into it.

Then, to transform normals, use the following equation.

$$n' = M^{-1^T} n$$

That is, take the transpose of the inverse of M and use that to transform your normals. If all you did was rotation and isometric (nonshape changing) scaling, you could transform directions with just M. They'd be scaled by a different amount, but will no doubt have a normalize call in their future that will even that out.

Note that for this reason the w component of a normal in model space is always 0.0, because it is a direction. The w component of a vertex in model space however is always 1.0 because it is a position in space.

## 1.6 Conclusion

In this chapter we briefly introduced the graphics rendering pipeline and discussed the mathematics behind the transformations behind it. However, there is much more to this topic such as scan-line and flood-fill algorithms for rendering polygons, clipping algorithms, shadowing, per-pixel lighting, texture mapping,… We ignored these topics as they are clearly beyond the scope of this course and refer the interested students to [2] and [27].

## Sources

This chapter is based on material from [2], [15], [27] as well as the following additional online resources:

http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/

https://www.learnopengles.com/tag/perspective-divide/

## References

[2] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. 2019. *Real-time rendering*. Crc Press.

[15] Donald Hearn, M Pauline Baker, and others. 2004. *Computer graphics with opengl*. Upper Saddle River, NJ: Pearson Prentice Hall,

[25] Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2016. *Physically based rendering:*

*From theory to implementation*. Morgan Kaufmann.

[27] Dave Shreiner, Graham Sellers, John Kessenich, and Bill Licea-Kane. 2013. *OpenGL programming guide: The official guide to learning opengl, version 4.3*. Addison-Wesley.

[29] Joey de Vries. 2015. Learn opengl. *Licensed under CC BY* 4, (2015).

14. http://www.pbr-book.org/↩