EE 267 Virtual Reality
Course Notes: A Brief Overview of the Graphics Pipeline

Gordon Wetzstein
gordon.wetzstein@stanford.edu

This document serves as a supplement to the material discussed in lecture 2. The document is meant as a brief overview of the mathematical operations involved in the graphics pipeline. For a detailed discussion of this topic, please refer to the textbook [Marschner and Shirley 2016] which is freely available online to Stanford students in the Stanford library (see course website for the link).

1 Basic Definitions

Computer graphics is the art and science of modeling, editing, animating, and rendering complex virtual environments (VE) using a formal mathematical description of everything in the VE. This includes 3D object shapes, material properties, lights, dynamic behavior of objects, physical properties, and many other aspects. Professional game and VR developers and also visual effects specialists in the film industry have been using programs like Unity, Unreal, 3D Studio Max, Maya, or Blender for years or even decades to model, edit, animate, and render scenes. Most VR applications today are probably made with either the Unity or Unreal game engines. Unity is available for free as a personal edition and it has a huge amount of community support and tutorials online. You can use Unity for your course project if you like, but we won't spend much time on it during the course.

The goal of the first part of the course, and also these notes, is to get you familiar with the basic mathematical concepts that makes all of these programs, computer graphics in general, and virtual reality work. Once you have a basic understanding of these concepts, you can easily learn Unity or other game engines on your own because you'll know how they work.

To get started, we will focus on the math behind turning a description of a 3D scene into a 2D image. This process is called rendering. At the most basic level, a 3D scene is composed of a number of 3D objects. Each object consists of many graphics primitives, such as 3D points or *vertices*, surface normals, and faces. We only consider triangles as the primitives for faces, but other types of surface descriptions exist, including quads or other types of polygons and parametric surface representations. Each triange consists of 3 vertices. A set of connected triangles is called *mesh* and typically models a 3D object or parts of it. A mesh is therefore a piecewise planar approximation of a continuous surface. To make it look more smooth, typically a surface normal is associated with each vertex. The normal is a unit-length vector that corresponds to the gradient of the continuous surface – it will help make the mesh look smooth after the lighting and shading calculations.

To create a mesh, you can either use one of the 3D modeling programs listed above to make it from scratch or use a 3D scanner to digitize a physical object. We won't spend much time on model generation in this course, so we will simply assume that somebody already created objects for us and we just compose these into a scene and render them. To render a scene, we need to transform all its 3D vertices and surface normals into 2D screen, i.e. pixel, coordinates. The rest of these notes go through this process in detail and you will implement these transformations in your first assignment using JavaScript.

2 Coordinate System

First things first, let's define the coordinate system we will be working with throughout the course. Basically, there are two options: left-handed and right-handed coordinate systems. OpenGL, WebGL, and many other graphics

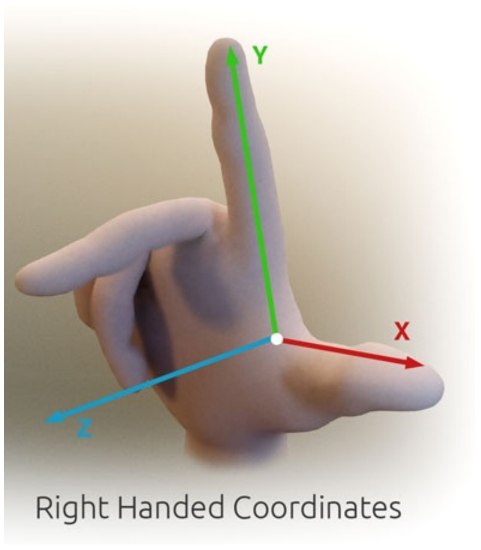

Figure 1: *Right-handed coordinate system used in this course.*

systems use a right-handed coordinate system, which is shown in Figure 1. We will be using this coordinate system throughout this course as well.

To make this coordinate system intuitive, let's do a quick exercise. As shown in Figure 1, raise your right hand, turn it so your thumb points to the right, your index finger points up, and your middle finger points towards you. In that order, we have the x-axis (positive x along the thumb), the y-axis (positive y along index finger), and the z-axis (positive z along the middle finger). Hold this pose for a bit and remember it. It seems silly but will be incredibly useful!

Note that some graphics systems, including Unity and the Unreal game engine, use a left-handed coordinate system. This can be confusing, but since we won't be working with Unity or Unreal until very late in the course, or never, let's not worry about it for now.

3 A Trip Down the Graphics Pipeline

At the most basic level, the graphics pipeline describes a series of matrix-vector multiplications that allow us to transform a 3D point into a 2D pixel coordinate on the screen. We start by representing our 3D point, or *vertex*, $\mathbf{p}$ as a homogeneous vector with four elements:

$$\mathbf{p} = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{1}$$

Several transforms are then applied to this point, each one represented as a 4×4 matrix-vector multiplication. These matrices will transform the point from one *space* into another, as illustrated in Figure 2 (from [Fernando and Kilgard 2003]).

3.1 Model Transform

Our vertex is originally defined in *object space*. For example, if you load a 3D model from a file, each vertex of this model will be defined with respect to some local coordinate origin, which could be in the middle of the object. Say you are modeling a room with several identical chairs in it. You only need a single 3D model of a chair and then you

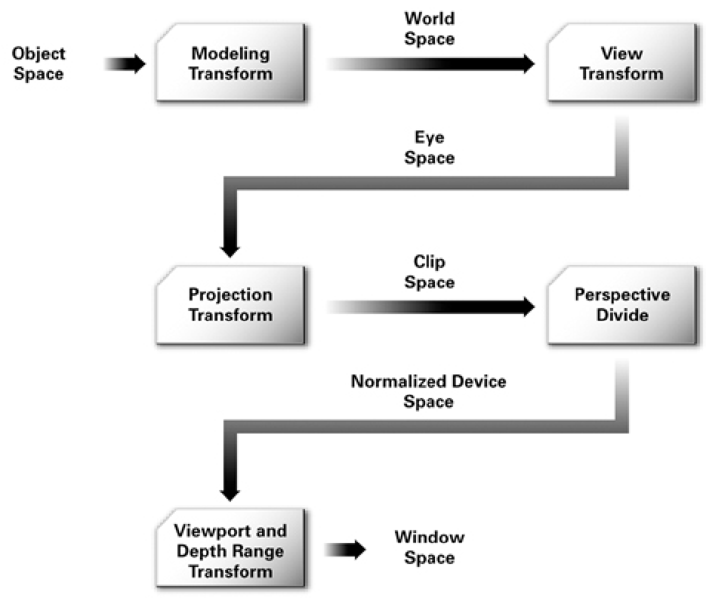

Figure 2: *Illustration of the graphics pipeline along with its transforms and spaces.*

create several copies of that and place them in the room. The 3D chair model will have its own origin and all points in the file will be defined with respect to that. This is object space.

To place each chair at a different position in the room, which now has another coordinate origin, e.g. the corner or center of the room, we need to *translate*, *rotate*, and *scale* each chair so that it goes where it belongs in the room. We can think of this sequence of rotation, translation, and scale operations as transforming the vertices of the chair from object space into *world space* (e.g., the coordinate system of the room). Mathematically, this is expressed by multiplying the vertex with the 4×4 model matrix $\mathbf{M}$ as

$$\mathbf{p}_{world} = \mathbf{M} \cdot \mathbf{p} \tag{2}$$

As discussed in lecture 2, $\mathbf{M}$ is usually a combination of rotation, translation, and scale matrices. Model transform matrices are combine by multiplying them. For example, $\mathbf{M} = \mathbf{T} \cdot \mathbf{R}_x$ is a combination of translation and rotation around the x axis. Note that the order of these multiplications matters! In this case, the rotation is first applied to $\mathbf{p}$ followed by the translation. So you have to read more complicated model matrix combinations from right to left to get the order correctly.

Below is a summary of the most commonly used model transforms: translations, rotations, and scaling.

3.1.1 Translation Matrices

A homogeneous translation matrix is created as

$$T(d) = \begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{3}$$

where d_x, d_y, d_z are the translation distances along the x, y, z axes, respectively.

3.1.2 Rotation Matrices

Homogeneous rotation matrices are often given for a rotation around a specific axis. This is referred to as rotation using Euler Angles. We will discuss more complicated types of rotations, for example using quaternions, later in the course. For now, let's assume that all 3D rotations can be expressed as successive rotations around the x, y, z axes.

A homogeneous matrix modeling a rotation around the x axis by angle θ given in radians is

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4}$$

A homogeneous matrix modeling a rotation around the y axis by angle θ given in radians is

$$R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{5}$$

A homogeneous matrix modeling a rotation around the z axis by angle θ given in radians is

$$R_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{6}$$

3.1.3 Scaling Matrices

A homogeneous scaling matrix is created as

$$S(s) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{7}$$

where s_x, s_y, s_z are the scale factors along the x, y, z axes, respectively.

3.2 View Transform

The view transform is described by another 4×4 matrix $\mathbf{V}$, i.e. the view matrix, and transforms the point from world to camera or view or eye space. In camera space, the camera is in the origin and it looks down the negative z-axis. We can construct the view matrix using a few parameters of our virtual camera, including its position, the look-at point, and the up vector. Please refer to the lecture slides for the exact formula to compute $\mathbf{V}$ from these parameters. Once $\mathbf{V}$ is constructed, we transform our point as

$$\mathbf{p}_{camera} = \mathbf{V} \cdot \mathbf{p}_{world} = \mathbf{V} \cdot \mathbf{M} \cdot \mathbf{p} \tag{8}$$

Note that the model and view matrices can be combined into a single *modelview* matrix by simply multiplying them as $\mathbf{V} \cdot \mathbf{M}$.

3.2.1 Constructing a View Matrix

Constructing a 4×4 view matrix is a bit more complicated than translations, rotations, and scaling, but it is certainly not super difficult. Probably the most common way to define a view matrix is to imagine that you have a virtual camera in the scene and that you want to see the scene from this camera's viewpoint. To specify this virtual camera, we need its 3D position in world coordinates (let's call that $eye = (eye_x, eye_y, eye_z)$), a point (not a vector!) in world coordinates that this camera is looking at $center = (center_x, center_y, center_z)$, and a normalized up vector that defines the roll angle of the camera $up = (up_x, up_y, up_z)$. This definition can lead to confusion, because eye and $center$ are 3D points in world space whereas up is a 3D vector.

Given these virtual camera parameters, we calculate a few intermediate vectors:

$$z^c = \frac{eye - center}{\|eye - center\|}, \quad x^c = \frac{up \times z^c}{\|up \times z^c\|}, \quad y^c = z^c \times x^c. \tag{9}$$

Here, $\times$ is the cross product of two vectors. You can interpret what the view matrix does as a combination of rotation and translation. First, it translates the entire world such that the camera is in the origin. Then, it rotates the world around this new origin to make sure that the camera looks down the negative z axis. Formally, the view matrix is created as

$$V = R \cdot T(-eye) \begin{pmatrix} x^c_x & x^c_y & x^c_z & 0 \\ y^c_x & y^c_y & y^c_z & 0 \\ z^c_x & z^c_y & z^c_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & -eye_x \\ 0 & 1 & 0 & -eye_y \\ 0 & 0 & 1 & -eye_z \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{10}$$

Note that the subscripts refer to the x, y, z component of the corresponding vector, so don't get confused with this notation.

ATTENTION: while some graphics APIs, like OpenGL, specify a function *glLookat*, that directly computes this view matrix from the parameters above, Three.js does have a lookat function but it only implements the rotation part of the view matrix. So best compute it yourself!

3.3 Projection Transform

Now we apply the projection transform. Intuitively, this transform models the field of view and aspect ratio of the camera lens. Moreover, the projection matrix also defines if we use a *perspective projection* or an *orthographic projection*. A perspective projection models a camera with a single center of projection, i.e. the camera center, and thus includes effects like perspective foreshortening and the change of the relative object size with distance. This model is similar to the human eye or a real camera. An orthographic projection models a theoretical camera that has no center of projection, so each pixel corresponds to a light ray that is parallel to all the other rays (see Fig. 3).

A 4×4 projection matrix $\mathbf{P}$ can be constructed as discussed in the lecture slides. A point is then transformed from camera space to *clip* space as

$$\mathbf{p}_{clip} = \mathbf{P} \cdot \mathbf{p}_{camera} = \mathbf{P} \cdot \mathbf{V} \cdot \mathbf{M} \cdot \mathbf{p}. \tag{11}$$

Again, we could combine the model, view, and projection matrices into a single *modelview-projection matrix* by simply multiplying them as $\mathbf{P} \cdot \mathbf{V} \cdot \mathbf{M}$.

There are several different types of projection matrices that are commonly used: orthographic projection matrices as well as symmetric and asymmetric perspective projection matrices. Please refer to the lecture slides for the specific form of these matrices.

3.4 Perspective Divide

Remember that our point $\mathbf{p}$ had four elements. The first three are simply x, y, z and the last one is called the homogeneous coordinate w. Whereas $w = 1$ for the point in object space, it is probably not going to be 1 anymore in clip space, especially after applying a perspective projection. At this stage, we want to divide out the homogeneous coordinate to get a vector with only three elements in what is known as *normalized device space* or normalized device

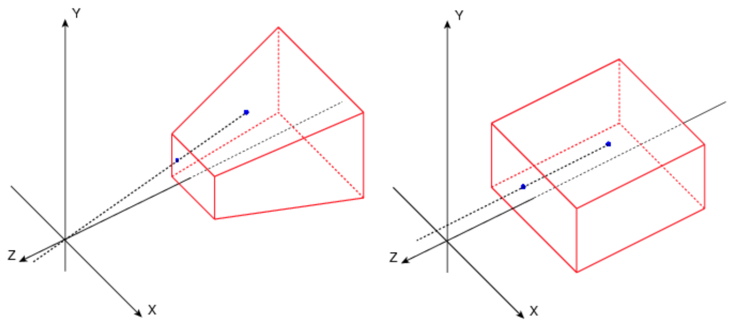

Figure 3: *View frustra of two types of projection transforms: perspective projection (left) and orthographic projection (right).*

coordinates (ndc):

$$\begin{pmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \\ 1 \end{pmatrix} = \begin{pmatrix} x_{clip}/w_{clip} \\ y_{clip}/w_{clip} \\ z_{clip}/w_{clip} \\ w_{clip}/w_{clip} \end{pmatrix}. \tag{12}$$

3.5 Window Transform

Finally, we get the pixel coordinates of our point in the window with $w \times h$ pixels as

$$\begin{pmatrix} x_{window} \\ y_{window} \\ z_{window} \end{pmatrix} = \begin{pmatrix} \frac{w}{2}(x_{ndc}+1) \\ \frac{h}{2}(y_{ndc}+1) \\ \frac{1}{2}(z_{ndc}+1) \end{pmatrix}. \tag{13}$$

3.6 Transforming Vectors / Surface Normals

Vectors can be transformed with a very similar pipeline as vertices, but there are a few differences. Let's consider surface normals as one of the most fundamental type of a vector that we will be working with. A surface normal $\mathbf{n}$ is usually associated with each vertex. In homogeneous coordinates, the normal is given as

$$\mathbf{n} = \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix} \tag{14}$$

Note that the homogeneous coordinate of this vector is 0 whereas that of a vertex is 1. This indicates that a translation and other transforms have no effect on the vector, because only the direction it is pointing at matters. Surface normals are always assumed to be normalized, i.e. $\sqrt{x^2+y^2+z^2} = 1$.

Applying the modelview matrix $\mathbf{V} \cdot \mathbf{M}$ to a vector or normal requires a bit of attention. First, to avoid non-uniform scaling, we cannot use the modelview matrix directly but need to use the inverse transpose of the modelview matrix instead (see discussion in lecture slides). So instead of multiplying $\mathbf{V} \cdot \mathbf{M} \cdot \mathbf{n}$ to get the normal in view space, we need to compute $\left((\mathbf{V} \cdot \mathbf{M})^{-1}\right)^T \cdot \mathbf{n}$. Second, for transforming the vector we will directly use the upper left 3×3 part of the modelview matrix and only invert-and-transpose this before multiplying it on the x, y, z component of the normal. Alternatively, we could invert-and-transpose the full 4×4 modelview matrix and then discard the w component of the resulting matrix-vector product. But the latter approach would be more computationally expensive.

Note that there is no perspective divide or window transformation for the normal. Typically, we only transform it into view space and then do all of our lighting and shading calculations in view space. We will discuss lighting and shading next week.

4 There and Back Again

Oftentimes, it is useful to go back from the window coordinates of a vertex to its ndc, clip, camera, or world coordinates. We can calculate these by inverting the individual transforms as described in the following.

4.1 Inverting the Window Transform

Given the spatial locations $x_{window} \in [0,\, w]$, $y_{window} \in [0,\, h]$ and depth $z_{window} \in [0,\, 1]$ of a point in window coordinates, we can calculate the normalized device coordinates as follows:

$$\begin{pmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \end{pmatrix} = \begin{pmatrix} \frac{2 \cdot x_{window}}{w} - 1 \\ \frac{2 \cdot y_{window}}{h} - 1 \\ (2 \cdot z_{window}) - 1 \end{pmatrix}. \tag{15}$$

4.2 Inverting the Perspective Divide

Getting the clip coordinates from the ndc coordinates is a bit more tricky. It actually requires us to know the projection matrix that was used for the projection transform and we also need to make another assumption: the modelview matrix will not affect the homogeneous coordinate of the point, i.e. $w_{camera} = 1$. This assumption is true as long as the modelview matrix is only composed of translation, rotation, and scale operations, which is usually true. Using the formula for computing the perspective projection matrix, we then get

$$\begin{pmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \\ 1 \end{pmatrix} = \begin{pmatrix} x_{clip}/w_{clip} \\ y_{clip}/w_{clip} \\ z_{clip}/w_{clip} \\ w_{clip}/w_{clip} \end{pmatrix}, \qquad \begin{pmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{pmatrix} = \begin{pmatrix} P_{00} & 0 & P_{02} & 0 \\ 0 & P_{11} & P_{12} & 0 \\ 0 & 0 & P_{22} & P_{23} \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x_{camera} \\ y_{camera} \\ z_{camera} \\ 1 \end{pmatrix} \tag{16}$$

For now, let's not worry about what the specific values of P_{ij} are but we do need to know which elements in $\mathbf{P}$ are zero and we will use the fact that $P_{33} = -1$. Interestingly, this specific structure implies that $w_{clip} = -z_{camera}$ and $z_{clip} = P_{22} z_{camera} + P_{23}$. Therefore, we can compute w_{clip} as

$$z_{ndc} = \frac{z_{clip}}{w_{clip}} = -\frac{P_{22} z_{camera} + P_{23}}{z_{camera}} \quad \Leftrightarrow \quad w_{clip} = -z_{camera} = \frac{P_{23}}{z_{ndc} + P_{22}} \tag{17}$$

Knowing w_{clip}, we can invert the perspective divide as $x_{clip} = x_{ndc}\, w_{clip}$, $y_{clip} = y_{ndc}\, w_{clip}$, and $z_{clip} = z_{ndc}\, w_{clip}$.

4.3 Inverting the Remaining Transforms

With the clip coordinates in hand, we can invert the rest of the transforms using the inverse matrices. Specifically

$$\mathbf{p}_{camera} = \mathbf{P}^{-1} \cdot \mathbf{p}_{clip}, \tag{18}$$

$$\mathbf{p}_{world} = \mathbf{V}^{-1} \cdot \mathbf{p}_{camera} = \mathbf{V}^{-1} \cdot \mathbf{P}^{-1} \cdot \mathbf{p}_{clip}, \tag{19}$$

$$\mathbf{p} = \mathbf{M}^{-1} \cdot \mathbf{p}_{world} = \mathbf{M}^{-1} \cdot \mathbf{V}^{-1} \cdot \mathbf{P}^{-1} \cdot \mathbf{p}_{clip}, \tag{20}$$

4.4 Application of Inverse Transforms: Calculating the Metric Depth of a Fragment

Oftentimes, you render a scene to get an image + depth map and then you want to do some calculations in screen space using the metric depth of each fragment. We will encounter this scenario in lecture 6, where we first render the scene and then we want to apply a depth of field (DOF) shader to simulate the DOF of the human eye. Given window coordinates x, y, z_{window} for each fragment, we need to first calculate the metric distance of each fragment to the camera. To this end, we follow the steps outlined in the previous subsections to convert x, y, z_{window} back to camera coordinates x, y, z_{camera}. Then then metric distance is simply $\sqrt{x_{camera}^2 + y_{camera}^2 + z_{camera}^2}$ because the camera is actually in the origin, so we just need the length of the vector pointing from the origin to a 3d point in camera coordinates.

We do not necessarily compute the inverse 4×4 projection matrix if we know the matrix elements. Assuming the projection matrix has the form outlined in Equation 16, for example, we can calculate the camera coordinates directly from the clip coordinates as $z_{camera} = -w_{clip}$ and

$$x_{clip} = P_{00} x_{camera} + P_{02} z_{camera} \quad \Rightarrow \quad x_{camera} = \frac{x_{clip} - P_{02} z_{camera}}{P_{00}} \tag{21}$$

$$y_{clip} = P_{11} y_{camera} + P_{12} z_{camera} \quad \Rightarrow \quad y_{camera} = \frac{y_{clip} - P_{12} z_{camera}}{P_{11}} \tag{22}$$

Again, given the camera coordinates, we simply normalize them to get the metric distance of a fragment to the camera.

References

FERNANDO, R., AND KILGARD, M. J. 2003. *The Cg Tutorial.* Addison-Wesley Professional.

MARSCHNER, S., AND SHIRLEY, P. 2016. *Fundamentals of Computer Graphics.* CRC Press.