Exploring Applications of Linear Algebra in Computer Graphics

Michal Kurek, Hamzeh Hamdan

July 13, 2023

COVER LETTER

As we developed our project, we split the research paper into several sections. While Michal researched quaternions and transformations in three-dimensional space, I looked into two-dimensional transformations and perspective projections. However, we continually checked each other's sections and proofs, giving feedback on each other's mathematics and writing style. Additionally, Michal created the original graphics we used in our paper.

After receiving feedback from our TF Kevin Lin, we redefined the purpose of our paper. Kevin pointed out that our paper didn't have a lot of mathematical content, and some of the mathematical definitions we used were imprecise. We decided to go back and refine our math, by giving more precise definitions of the concepts used throughout the paper. We also decided to visually separate definitions and theorems from other parts of the paper by enclosing them in boxes.

Moreover, following Kevin's advice, we've added multiple theorems and definitions, which were missing in our project draft. We have also clarified some of the language in sections 3 and 5, as well as included figures and examples to help the reader build the intuition behind the concepts being introduced.

Additionally, the feedback we received from our fellow students was very helpful in declaring what content was already written at a Math 22A student level and what content needed to be further clarified. This included some of the mathematical concepts and examples we talk about in our paper.

One of the peer reviewers advised us to clarify some of the inconsistencies in our explanations. For example, in the draft we did not explain why the order of matrix multiplication in section 2 matters. We have since added relevant paragraphs explaining and/or correcting those inconsistencies.

Following our second peer reviewer's advice, we decided to delve more deeply into the topics which we covered briefly in the project draft: quaternions. We've included relevant definitions and theorems that

We additionally decided to exclude an example we had included in the shadow mapping section of our draft; in this example, we defined shadow mapping through an equation. We decided not to delve into the mathematics of shadow mapping, since the section would add a completely new mathematical field and much complexity beyond the time scope of the paper. Instead, we decided to give an introduction to the topic, with relevant figures to make the concept of shadow mapping more approachable and easier to understand.

Contents

1 Introduction

Computer graphics are fundamental many aspects of technology as we know it. From UX animations in smartphones to characters in games, every animation in computer graphics relies on a combination of different transformations. Opening an application on a mobile device, for example, commonly results with the icon on the home screen expanding to fit the screen; this animation is a combination of scale and translation, two transformations we will discuss with more detail later.

Beyond the simple transformations of objects in two-dimensional space, computer graphics has evolved to include simulated virtual and gaming environments. These projections commonly include many different complex transformations, but for the purposes of this paper, we will focus on three-dimensional rotations and real time three-dimensional shadow mapping.

In our paper, we will present two-dimensional transformations and build upon them to expand into transformations in the third dimension. With an additional dimension, transformations tend to become more complicated and their applications expand. In our paper, we will additionally expand into quaternions, an easier way of representing three-dimensional rotations, and perspective projections, the process by which three-dimensional environments are displayed onto a plane.

2 Transformations

Definition 1. *A transformation is a bijection of a set to itself, that is, it is an injective function such that an inverse exists.*

In this paper, when we are referring to transformations we are referring primarily to **geometric transformations** — transformations, where the domain and range are sets of points, most often both $\mathbb{R}^2$ or both $\mathbb{R}^3$.

2.1 Transformations in 2D space

Two-dimensional transformations lie at the core of computer graphics and geometric transformations. In this section, we will consider translation, rotation, scaling, reflection, and shear transformations.

Let us first consider one of the most basic transformations: a translation. In the context of our game, a translation involves changing the position of an object in space by moving each of the points comprising the object in the same direction by the same amount.

Definition 2. *A **translation** is a geometric transformation that moves each point of a figure, shape or object by some constant vector.*

Translations are transformations in which a point $A = \begin{bmatrix} x \\ y \end{bmatrix}$ is moved to another point $A' = \begin{bmatrix} x' \\ y' \end{bmatrix}$

by a given vector $\vec{t} = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$. Thus, $x' = x + t_x$ and $y' = y + t_y$, and

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}.$$

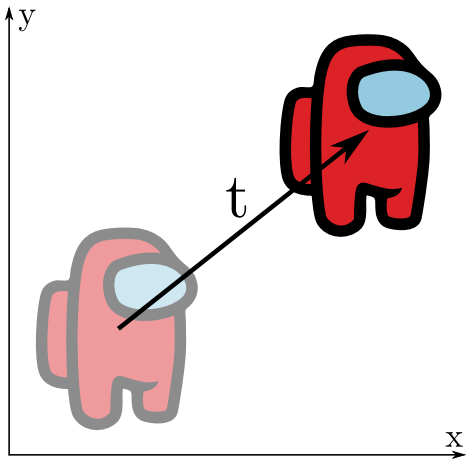

Figure 1: A simple translation in 2D space (simplified)

In our game, the vector $\vec{t}$ by which we translate the object's position could represent for example the object's velocity vector, defining the direction and distance to move the object by each frame.

What if we wanted our player character to be able to turn, or rotate, towards a direction the player specifies?

Definition 2. *A **rotation** is a transformation that maintains the distance between a point and the origin while changing its location relative to the origin.*

When we rotate an object, most of the times we only know the angle of the desired rotation as well as the point around which we rotate. A question arises: knowing the 2D coordinates of a point, how would we find the coordinates of that point after performing the rotation?

Theorem 1. *The transformation matrix used to perform a counterclockwise rotation by an angle θ with respect to the x axis is given by the matrix*

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}.$$

Proof. Consider some point $A = \begin{bmatrix} x \\ y \end{bmatrix}$ being rotated about the origin to another point $A' = \begin{bmatrix} x' \\ y' \end{bmatrix}$. We prove the theorem by utilizing polar coordinates.

Let r be the distance of the point A from the origin of the coordinate system. Let ϕ be the angle between the vector starting in the origin and ending in A and the x axis, and θ be the angle we are rotating through. Since rotations preserve the distance of points from the origin of rotation, we can write:

$$A = \begin{bmatrix} r\cos\phi \\ r\sin\phi \end{bmatrix}, \quad A' = \begin{bmatrix} r\cos(\phi+\theta) \\ r\sin(\phi+\theta) \end{bmatrix}$$

Thus,

$$A' = \begin{bmatrix} r\cos\phi\cos\theta - r\sin\phi\sin\theta \\ r\sin\phi\cos\theta + r\cos\phi\sin\theta \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix}$$

And hence

$$A' = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}.$$

■

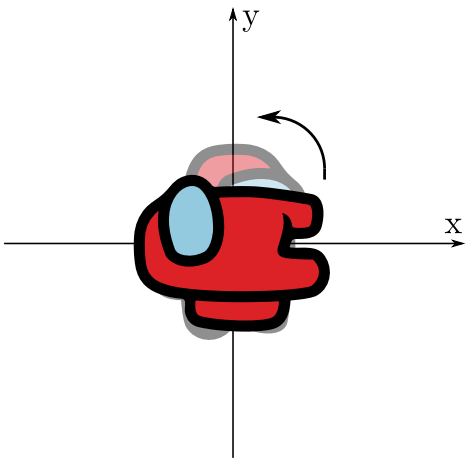

Figure 2: A simple rotation in 2D space

Theorem 2. *The transformation matrix used to perform a* ***clockwise*** *rotation by an angle θ with respect to the x axis is given by the matrix*

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}.$$

Proof. Note, that for a **clockwise** rotation, θ changes sign, and hence the rotation matrix is

$$\begin{bmatrix} \cos(-\theta) & -\sin(-\theta) \\ \sin(-\theta) & \cos(-\theta) \end{bmatrix}$$

Using the sine and cosine properties, we get

$$\begin{bmatrix} \cos(-\theta) & -\sin(-\theta) \\ \sin(-\theta) & \cos(-\theta) \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}.$$

■

Scaling is a transformation that stretches or compresses every point coordinate by some factor. Different factors can be used for different axes. For example, let us denote S_x as the scaling factor for the x axis, and similarly S_y for the y axis. For a point $A = \begin{bmatrix} x \\ y \end{bmatrix}$, The following are true: $x' = S_x \cdot x$ and $y' = S_y \cdot y$. Thus,

$$A' = \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}.$$

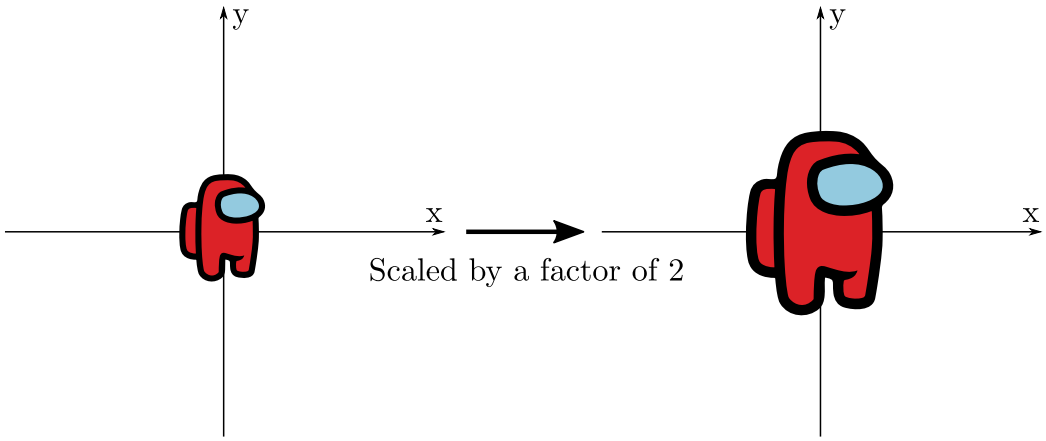

Figure 3: A simple scaling in 2D space

Scalings are useful in 2D graphics, since if we want to make an object appear closer to the viewer, we simply make it appear bigger on the screen.

Reflections are special cases of scaling in which either $S_x = -1$ and $S_y = 1$ representing a reflection about the y axis, or $S_x = 1$ and $S_y = -1$ representing a reflection about the x axis.

Shear is a transformation that skews an object with respect to the x or y axis. Consider a case in which we are skewing by the angle α with respect to the x axis and by the angle β with respect

to the y axis. It follows that $x' = x + y\tan\beta$ and $y' = y + x\tan\alpha$. This can be represented as:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & \tan\beta \\ \tan\alpha & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}.$$

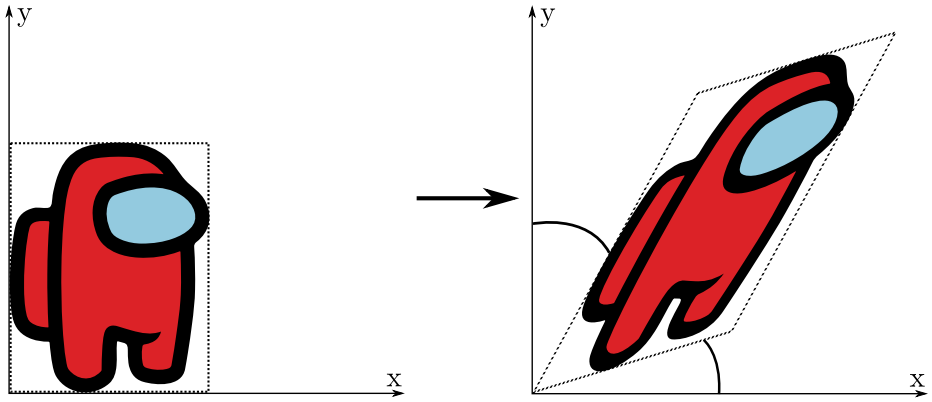

Figure 4: A simple sheer in 2D space

2.2 Homogeneous Coordinate Transformations

Imagine we want to transform a point by rotating it by some angle and translating by some vector:

1. First, we rotate the point using the rotation matrix (matrix multiplication)
2. We perform the translation using matrix addition (matrix addition)

Now, imagine we want to do the same to an object: we apply the same operations to all of the points comprising the object.

If we could somehow carry out the entire transformation using solely matrix multiplication, we could simplify the process of separately rotating and translating all of the points. Fortunately, **homogeneous coordinates** come to the rescue. Suppose we have a point (x, y) in the Cartesian coordinates. We introduce a third coordinate w. We call that coordinate the **homogeneous coordinate**. Writing our point as (x, y, w), we have now represented our point in the homogeneous coordinate system.

$$\underset{Homogeneous}{(x, y, w)} \iff \underset{Cartesian}{\left(\frac{x}{w}, \frac{y}{w}\right)}$$

One of the advantages of representing 2D points using this system is that it allows for easier computation of transformations. Namely, the homogeneous coordinate system allows us to represent transformations using solely matrix multiplication.

Let us give some examples. For simplicity's sake, we take $w = 1$ (since $\left(\frac{x}{1}, \frac{y}{1}\right) = (x, y)$) Take vector translation as an example:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} \longrightarrow \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

where the vector with coordinates t_x, t_y represents the initial translation vector. We can use a similar approach to represent rotation, scaling and sheering. We call the matrices representing those transformations in our new coordinate system homogeneous. Thus, the homogeneous scaling matrix is

$$\begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

since

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} S_x \cdot x \\ S_y \cdot y \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix},$$

the homogeneous rotation matrix

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

since

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} r\cos(\theta+\phi) \\ r\sin(\theta+\phi) \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix},$$

and the homogeneous sheer matrix

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & \tan\beta & 0 \\ \tan\alpha & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}.$$

To simulate "body motion" in 2D, we can now simply concatenate the matrix multiplication by first scaling, then rotation and finally translating any given vector in 2D space. Let us give an example.

Suppose we had the following player sprite represented in 2D space. Moreover, suppose we want to translate it by some vector, scale it, and rotate it. We can now do so using the homogeneous coordinate system by simply carrying out the matrix multiplication as so:

$$T' = M_{\text{translation}} \cdot M_{\text{rotation}} \cdot M_{\text{scaling}} \cdot T$$

where $M_{\text{translation}}, M_{\text{rotation}}, M_{\text{scaling}}$ represent the respective transformation matrices, and T represents some initial point T. The order of matrix multiplication matters, since matrix multiplication is not commutative. Scaling an object which has been moved away from the origin produces a result different than scaling an object centered at the origin: think of scaling as not simply stretching or shrinking objects, but moving points away from or towards the origin. Any object not centered at the origin is going to be moved away from $(0, 0)$ or towards $(0, 0)$. Similarly for rotating objects, rotating objects not centered at the origin will result in a different final orientation of the object.

Because of this, for rotation and scaling, we want the transformation to be performed with the object being centered at the origin, and hence we perform the translation last.

2.3 Transformations in 3D space

Taking a similar approach as we did with the homogeneous coordinates in 2D, by introducing a fourth coordinate we can represent and chain different transformations in 3D. Extending our approach to 3D is quite easy. Consider for example the following homogeneous scaling matrix

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Secondly, the translation matrix by a vector v is

$$\begin{bmatrix} 1 & 0 & 0 & v_1 \\ 0 & 1 & 0 & v_2 \\ 0 & 0 & 1 & v_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

since

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} x + v_x \\ y + v_y \\ z + v_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Rotations in 3D are a bit more tricky than in 2D. Let's start with an easy one: a rotation around the z-axis. In the 2D rotation about the origin we have covered, the origin remains in its original place. In a 3D rotation about an axis, all of the points on that axis will remain fixed. Thus, we can take the same approach as in a 2D transformation while fixing the z coordinate, which results in the

matrix

$$M_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Similarly, by fixing the x and y axes, we can derive

$$M_y(\theta) = \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and

$$M_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If we treat a rotation in 3D space as a composite of rotations about the three axes, we can denote the rotation transformation as

$$M(\theta_x, \theta_y, \theta_z) = M_z(\theta_z)M_y(\theta_y)M_x(\theta_x)$$

while the rotation matrix can be found by carrying out the matrix multiplication, and is equal to

$$\begin{bmatrix} \cos\theta_y\cos\theta_z & -\cos\theta_x\sin\theta_z - \sin\theta_x\sin\theta_y\cos\theta_z & \sin\theta_x\sin\theta_z - \cos\theta_x\sin\theta_y\cos\theta_z & 0 \\ \cos\theta_y\sin\theta_z & \cos\theta_x\cos\theta_z - \sin\theta_x\sin\theta_y\sin\theta_z & -\sin\theta_x\cos\theta_z - \cos\theta_x\sin\theta_y\sin\theta_z & 0 \\ \sin\theta_y & \sin\theta_x\cos\theta_y & \cos\theta_x\cos\theta_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The relevant angles for those rotations $(\theta_x, \theta_y, \theta_z)$ are called the **Euler angles**. In aviation and aerodynamics for example, we refer to those angles as the yaw, pitch, and roll.

What if we wanted to find an axis of rotation for some rotation R knowing its rotation matrix? To answer this question, we have to solve

$$Rx = (1)x$$

that is, we are looking for an eigenvector corresponding to the eigenvalue of 1 for this particular rotation matrix. Such a vector is unaffected by the transformation, since it stays fixed on the same line that it spans and it's magnitude does not change.

Consider as an example a simple rotation about the z axis. The rotation matrix is given by

$$M_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We solve

$$M_z(\theta)x = (1)x$$

$$(M_z(\theta) - I)\, x = 0$$

or

$$\begin{bmatrix} \cos\theta - 1 & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta - 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Row reducing the matrix on the left hand side gives

$$\begin{bmatrix} 1 & -\frac{\sin\theta}{\cos\theta - 1} & 0 & 0 \\ \sin\theta & \cos\theta - 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Hence

$$1 \cdot x = 0$$

$$1 \cdot y = 0$$

$$0 \cdot z = 0$$

We see that any vector with coordinates $x = 0$, $y = 0$, and an arbitrary z coordinate is an eigenvector and lays on the axis of rotation. Thus, the axis of rotation is the z axis, which is what we expected.

3 Quaternions

Representing rotations in $\mathbb{R}^3$ using rotation matrices can get quite cumbersome.

As we will see, there exists a different (and in some ways better) way of representing rotations in 3D space: **quaternions**.

Definition 3. *A **quaternion** is a number of the form*

$$a + bi + cj + dk$$

where $a, b, c, d \in \mathbb{R}$, while i, j, k are related in such a way that

$$i^2 = j^2 = k^2 = ijk = -1$$

and

$$ij = k,\ ji = -k,\ jk = i,\ kj = -i,\ ki = j,\ ik = -j$$

We call i, j, k the basic quaternions or the fundamental quaternion units.

We can think of quaternions as an extension of complex numbers; whereas when dealing with complex numbers we considered the real dimension and the imaginary dimension, when working with quaternions we will consider 4 dimensions: one real dimension and three imaginary dimensions.

Similarly to how complex numbers can represent rotations in 2D space, quaternions help us represent rotations in 3D space.

A commonly used notation for quaternions is $q = (s, v)$ where s represents the real part, and v represents the imaginary vector part of q, that is

$$s = a$$
$$v = bi + cj + dk = (b, c, d)$$

3.1 Quaternion norm, conjugate, and inverse. Versors.

Definition 4. *The magnitude $|q|$ of a given quaternion $q = a + bi + cj + dk$ is a scalar value equal to*

$$|q| = \sqrt{a^2 + b^2 + c^2 + d^2} = \sqrt{s^2 + |v^2|} = \sqrt{q\overline{q}}$$

*where $\overline{q}$ is the **conjugate** of q defined as*

$$\overline{q} = (s, -v)$$

Definition 5. *A **unit quaternion** is a quaternion of norm equal to 1, that is*

$$|q| = 1 \quad or \quad a^2 + b^2 + c^2 + d^2 = 1$$

*Unit quaternions are also referred to as **rotation quaternions** or **versors**.*

Rotation quaternions are commonly represented in their **axis-angle representation** form, that is, we can represent a versor $q = a + bi + cj + dk$ as

$$q = \cos\left(\frac{\theta}{2}\right) + b\sin\left(\frac{\theta}{2}\right)i + c\sin\left(\frac{\theta}{2}\right)j + d\sin\left(\frac{\theta}{2}\right)k$$

or

$$q = \cos\left(\frac{\theta}{2}\right) + \mathbf{v}\sin\left(\frac{\theta}{2}\right)$$

where θ is the angle of rotation around the axis defined by the vector $\mathbf{v}$.

To explain how the quaternion rotation actually works, we need a couple more definitions

Definition 6. *The inverse of a quaternion q is defined as*

$$q^{-1} = \frac{\overline{q}}{|q|^2}.$$

Note, that for unit quaternions, since $|q| = 1$, we have that the inverse is equal to the conjugate of that quaternion

$$q^{-1} = \overline{q}$$

Theorem 3. *The result of multiplying two quaternions $r = (r_0, r_v)$ and $s = (s_0, s_v)$ is equal to*

$$rs = (r_0, r_v)(s_0, s_v) = (r_0 s_0 - r_v \cdot s_v,\ \ r_0 s_v + s_0 r_v + r_v \times s_v)$$

where $r_v \cdot s_v$ is the dot product of vectors r_v and s_v, and $r_v \times s_v$ is the cross product of r_v and s_v.

Proof. Let $r = (r_0, r_v)$ and $s = (s_0, s_v)$. Carrying out the multiplication and expanding, we get

$$\begin{aligned} rs = (r_0, r_v)(s_0, s_v) = \\ = r_0 s_0 + r_0 s_v + s_0 r_v + r_v s_v \end{aligned}$$

Let us compute $r_v s_v$. Let $r_v = (ui + vj + wk)$ and $s_v = (xi + yj + zk)$. We get

$$\begin{aligned} r_v s_v &= (ui + vj + wk)(xi + yj + zk) = \\ &= uixi + uiyj + uizk + vjxi + vjyj + vjzk + wkxi + wkyj + wkzk = \\ &= -ux + uyk - uzj - vxk - vy + vzi + wxj - wyi - wz = \\ &= -ux - vy - wz + vzi - wyi + wxj - uzj + uyk - vxk \\ &= -(ux + vy + wz) + (vz - wy)i + (wx - uz)j + (uy - vx)k \\ &= -r_v \cdot s_v + r_v \times s_v \end{aligned}$$

where $r_v \cdot s_v$ is the dot product of r_v and s_v, and $r_v \times s_v$ is the cross product of r_v and s_v. Plugging this result back into the original equation, we get

$$\begin{aligned} rs &= r_0 s_0 + r_0 s_v + s_0 r_v - r_v \cdot s_v + r_v \times s_v = \\ &= r_0 s_0 - r_v \cdot s_v + r_0 s_v + s_0 r_v + r_v \times s_v \end{aligned}$$

where $r_0 s_0 - r_v \cdot s_v$ is the real part, and $r_0 s_v + s_0 r_v + r_v \times s_v$ is the imaginary part. Thus, we can write

$$rs = (r_0, r_v)(s_0, s_v) = (r_0 s_0 - r_v \cdot s_v, \;\; r_0 s_v + s_0 r_v + r_v \times s_v)$$

■

One important property of rotations is that they don't change the scaling of the objects. Luckily, quaternion multiplication has a convenient property: multiplying quaternions preserves their norms.

To prove that claim, first we will prove the following lemma.

Lemma 1. *For any two quaternions p and q, we have*

$$\overline{q}\;\overline{p} = \overline{pq}$$

Proof. Carrying out the multiplication, we see that indeed

$$\begin{aligned} \overline{q}\;\overline{p} &= (q_0, \; -q_v)(p_0, \; -p_v) \\ &= (q_0 p_0 - (-q_v \cdot (-p_v)), \;\; q_0(-p_v) + p_0(-q_v) + (-q_v) \times (-p_v)) \\ &= (p_0 q_0 - (-p_v \cdot (-q_v)), \;\; p_0(-q_v) + q_0(-p_v) + (-p_v) \times (-q_v)) \\ &= (p_0 q_0 - p_v \cdot q_v, \;\; -p_0 q_v - q_0 p_v - p_v \times q_v) \\ &= \overline{pq} \end{aligned}$$

■

Now, we can prove the following theorem

Theorem 4. *Quaternion norm is multiplicative. Namely, for any two quaternions p and q we have:*

$$|p||q| = |pq|$$

Proof. We use the definition of a quaternion's norm. Let $|p| = \sqrt{p\overline{p}}$ and $|q| = \sqrt{q\overline{q}}$. Hence

$$|p||q| = \sqrt{(p\overline{p})(q\overline{q})}$$

Since $q\overline{q}$ is a scalar, and scalars commute with every quaternion, we can write

$$\sqrt{(p\overline{p})(q\overline{q})} = \sqrt{p(q\overline{q})\overline{p}}$$

By Lemma 1, we write

$$\sqrt{p(q\overline{q})\overline{p}} = \sqrt{pq\overline{q}\,\overline{p}} = \sqrt{pq\overline{pq}}$$

and hence, by the definition of a quaternion's norm

$$\sqrt{pq\overline{pq}} = |pq|$$

■

It follows, that when we multiply some given quaternion by a unit quaternion, the norm of the resulting quaternion is the same as of the original one.

Moreover, the norm of the inverse of a quaternion is the same as the norm of the quaternion itself.

3.2 Representing rotations using quaternions

To perform a rotation of a point $\mathbf{P} = (x, y, z)$ by a quaternion $\mathbf{q}$, we

1. Constructing a quaternion p from the point P, by taking P's coordinates as p's imaginary components, and we set the quaternion's real part to be equal to zero (quaternions without the real part are called **pure** quaternions and are used to represent vectors in three-dimensional space):
$$p = 0 + xi + yj + zk$$

2. We perform the rotation by carrying out the quaternion multiplication
$$p' = qpq^{-1}$$

3. Since $|q| = |q^{-1}|$, we see that $|p'| = |p|$. The components of the resulting pure quaternion p' can be used to extract the coordinates of the rotated point P':
$$p' = 0 + x'i + y'j + z'k \quad \longrightarrow \quad P' = (x', y', z')$$

For a given $q = \cos\left(\frac{\theta}{2}\right) + \mathbf{v}\sin\left(\frac{\theta}{2}\right)$, this rotation can be interpreted as a rotation about the axis $\mathbf{v}$ by an angle θ. Since the norm of p' is equal to the norm of p, it is easy to see that the distance between point P and the origin has been preserved after the rotation.

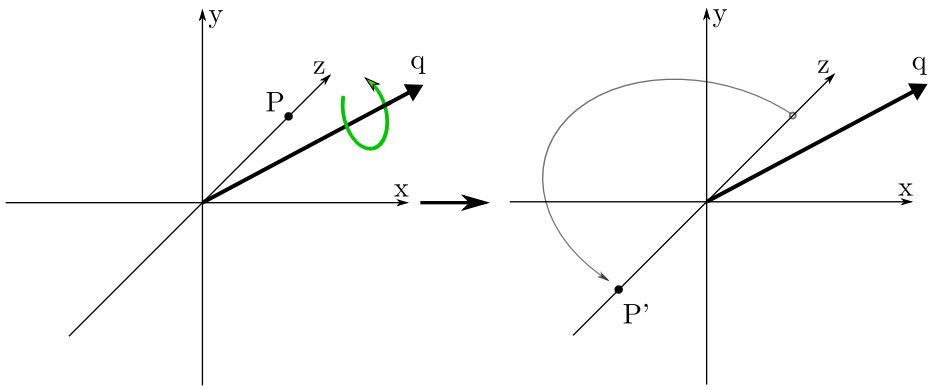

Figure 5: A simple quaternion rotation

3.3 Compound rotations

Quaternions provide us with an easy way of compounding multiple rotations. Take for example two quaternions q_1 and q_2 representing two different rotations. Let P be some point in space we are going to rotate. Using the approach from the previous subsection, we first construct a pure quaternion p, using the coordinates of P. To perform the two rotations, we then simply calculate

$$p' = q_2 \left(q_1 p q_1^{-1}\right) q_2^{-1}$$

The above can be represented (using quaternion multiplication) as

$$p' = q_3 p q_3^{-1} \quad \text{where} \quad q_3 = q_2 q_1$$

3.4 Benefits over using matrices

Compared to rotation matrices, quaternions can be more efficient and easier to work with. Since quaternions are 4-tuples, they provide us with a more concise way of representing a rotation than a $3x3$ matrix does.

Furthermore, the angle of rotation and rotation axis can be trivially recovered, since for a versor q we have

$$q = a + bi + cj + dk = \cos\left(\frac{\theta}{2}\right) + \mathbf{v}\sin\left(\frac{\theta}{2}\right)$$

and thus

$$\theta = 2\arccos(a)$$

and the axis of rotation

$$\mathbf{v} = \left(\frac{b}{\sin\left(\frac{\theta}{2}\right)}, \frac{c}{\sin\left(\frac{\theta}{2}\right)}, \frac{d}{\sin\left(\frac{\theta}{2}\right)}\right)$$

which is equivalent to the unit vector

$$\left(\frac{b}{\sin\left(\frac{\theta}{2}\right)}\right)i + \left(\frac{c}{\sin\left(\frac{\theta}{2}\right)}\right)j + \left(\frac{d}{\sin\left(\frac{\theta}{2}\right)}\right)k$$

Another benefit of using quaternions over matrices is compound rotations. For example, to carry out a composition of two rotations using rotation matrices, we have to work out the product of the two corresponding matrices, which would require 27 multiplies and 18 additions (Wiki article). Using quaternions however, we only have to work out 16 multiplies and 12 additions to achieve the same rotation chain. This has big implications for computer graphics, since it allows us to save time on performing calculations and results in major performance gains.

One area where quaternions excel is interpolation. To animate a rotating object inside of 3D space, we need to know its initial orientation, final orientation, as well as all of the intermediate orientations; we can think of those as intermediate steps, or frames of an animation (see Figure 6).

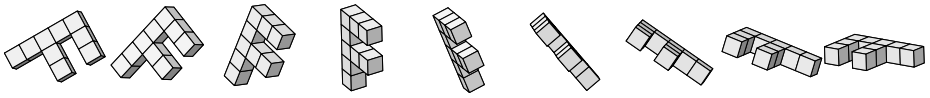

Figure 6: Intermediate steps of an interpolation (adapted from Rotation Splines)

Using algorithms like Slerp (shorthand for spherical linear interpolation), which utilize quaternion algebra behind the scenes, we can calculate those intermediate orientations. This is especially useful for animating movements like rotations, as well as inverse kinematics systems.

4 Perspective Projections

Consider now a three-dimensional environment, maybe a battle field in a game or a simulation of a bedroom. How can we render a three-dimensional environment through a the perspective of a certain point? The process of doing so is called perspective projection, and it requires a projection matrix.

Before we define what a projection matrix is, we must first introduce certain concepts and standard practices in computer graphics.

Definition 7. *The **camera space** is a three-dimensional space with the coordinate system from the camera's point of view, that is, the camera is located at the origin, usually looking down along the -Z axis.*[a]

[a] AUTODESK: Camera Space, Object Space, and World Space

Definition 8. ***Normalized device coordinate (NDC) Space*** *is a screen independent display coordinate system that encompasses a cube where the x, y, and z components range from -1 to 1.*[a]

[a] Science Direct: Device Coordinate

Essentially, the camera space is the space in which the camera exists, and the NDC space is the space that defines the view volume.

Definition 9. *Consider a three dimensional point P represented as a row vector using homogeneous coordinate in camera space. A* ***projection matrix*** *is a $4x4$ matrix M_{proj} such that*

$$P \cdot M_{proj} = P',$$

where P' is the projected version of P onto the canvas in NDC space.

Projection transformations are used to project vertices of three dimensional objects onto the screen in order to create images of these objects that follow the rules of perspective. Thus, we use projection matrices to transform vertices or three dimensional points, not vectors. Additionally, these points must be represented using homogeneous coordinates; recall that a projection matrix is a $4x4$ matrix, and thus any point to which the projection transformation is applied to must be a [1x4] vector.

4.1 Orthographic and perspective projections.

There are two types of projections: orthographic projections and perspective projections. The differences among these projections relies on a new concept: the viewing frustum.

Definition 10. *The* ***viewing frustum*** *is the region of space in the modeled world that may appear on the screen. It is a truncated pyramid that extends from the camera, where the apex is the camera position, the lines extending from the camera are all of equal length, and the sides of the pyramid are noted as image planes.* [a]

[a] Essentials of Interactive Computer Graphics: Concepts and Implementation (Page 14)

Definition 11. *The* ***angle of view*** *is the vertical angle that subtends the height of the near (image) plane from the camera position.* [a]

[a] Essentials of Interactive Computer Graphics: Concepts and Implementation (Page 14)

Perspective projections are projections of an object onto a canvas plane in the viewing frustum that is parallel to the far (image) plane. As the canvas plane moves towards or away from the camera, the objects size changes.

Now consider the case in which the angle of view approaches zero. The four lines defining the frustum's pyramid will become parallel to each other and the frustum pyramid becomes a rectangular prism. Thus, an object size in the image stays constant since the canvas dimensions remain the same regardless of its distance to the camera. This type of projection is called an orthographic projection.

4.2 Constructing a perspective projection matrix.

Consider a three dimensional point A in a viewing frustum, with $Range(x) \in [l, r]$, $Range(y) \in [b, t]$, and $Range(x) \in [-n, -f]$. In order words, the truncated frustum's nearest plane to the origin is defined by the following points: $(l, t, -n)$, $(r, t, -n)$, $(r, b, -n)$, and $(l, b, -n)$, listed from clockwise from the top left corner. Our projection matrix P is an nxn matrix that transforms the truncated frustum into an NDC with $Range(x), Range(y), Range(z) \in [-1, 1]$.

Note that a camera space uses a right-handed coordinate system while NDC uses a left-handed coordinate system; while the camera at the origin looks along the -Z axis in the camera space, it looks along the +Z axis in NDC. Thus, we must negate the near and far planes when we create the matrix.

The three dimensional point A has coordinates (x, y, z) (and will be represented through the homogeneous coordinate system as (x, y, z, w) from now on) in the camera space. Let $A' = AP$ such that A' is the projection of A onto the projection plane (the near plane of the viewing frustum, denoted as n). The coordinates of A' thus are $(x', y', -n, w')$. Note that A and A' both exist along the same vector extending from the origin and both exist in the viewing frustum. Thus, there exists some scalar a such that:

$$\frac{x'}{w'} = a\frac{x}{w}, \frac{y'}{w'} = a\frac{y}{w}, \text{ and } \frac{-n}{w'} = a\frac{z}{w}.$$

In other words,

$$a\frac{w'}{w} = \frac{x'}{x} = \frac{y'}{y} = \frac{-n}{z}.$$

Using these ratios, we can calculate x' and y' as the following:

$$x' = \frac{-nx}{z} = \frac{nx}{-z} \text{ and } y' = \frac{-ny}{z} = \frac{ny}{-z}.$$

Note that x' and y' are written in homogeneous form. Thus, we now know that after A is multiplied by P, the resulting coordinates are still homogeneous coordinates, with $w = -z$. We can transform these coordinates into their respective NDC coordinates by dividing by the x, y, and z coordinates by the w coordinate.

Thus, we get the following equation:

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = P \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}, \begin{bmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \end{bmatrix} = \begin{bmatrix} x'/w' \\ y'/w' \\ z'/w' \end{bmatrix}$$

We can now set the w-component of A' as $-z$, giving us the following:

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}, \quad \therefore w' = -z.$$

Next, we map x' to x_{ndc} and y' to y_{ndc} with linear relationship $[l, r] \implies [-1, 1]$ and $[b, t] \implies [-1, 1]$. We will first consider the mapping from x' to x_{ndc} and then y' to y_{ndc}.

Consider the mapping of x' to x_{ndc}, in which we have a vertical axis x_{ndc} and a horizontal axis x'. Recall that $x' \in [l, r]$ and $x_{ndc} \in [-1, 1]$. The mapping looks like this:

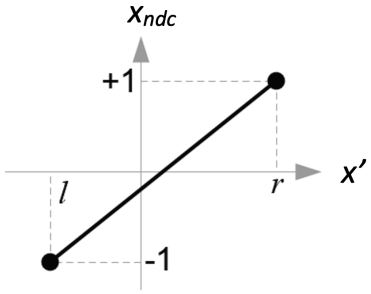

Figure 7: Adapted from OpenGL Projection Matrix.

We can calculate the mapping from $(l, -1)$ to $(r, 1)$ as the following:

$$x_{ndc} = \frac{1 - (-1)}{r - l} \cdot x' + B.$$

Substituting $(r, 1)$ into (x', x_{ndc}), we get the following two equations:

$$1 = \frac{2r}{r-l} + B$$
$$B = 1 - \frac{2r}{r-l} = \frac{r-l-2r}{r-l} = \frac{r+l}{r-l}$$
$$\therefore x_{ndc} = \frac{2x'}{r-l} - \frac{r+l}{r-l}.$$

Let us now consider the mapping of y' to y_{ndc}, in which we have a vertical axis y_{ndc} and a horizontal axis y'. Recall that $y' \in [b, t]$ and $y_{ndc} \in [-1, 1]$. The mapping looks like this:

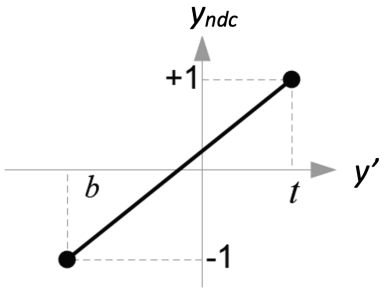

Figure 8: Adapted from OpenGL Projection Matrix.

We can calculate the mapping from $(b, -1)$ to $(t, 1)$ as the following:

$$y_{ndc} = \frac{1-(-1)}{t-b} \cdot y' + B.$$

Substituting $(t, 1)$ into (y', y_{ndc}), we get the following two equations:

$$1 = \frac{2t}{t-b} + B$$
$$B = 1 - \frac{2t}{t-b} = \frac{t-b-2t}{t-b} = \frac{t+b}{t-b}$$
$$\therefore y_{ndc} = \frac{2y'}{t-b} - \frac{t+b}{t-b}.$$

Recall that we've already calculated the values of x' and y' as the following:

$$x' = \frac{-nx}{z} = \frac{nx}{-z} \text{ and } y' = \frac{-ny}{z} = \frac{ny}{-z}.$$

Plugging these values into the equations we've just derived, we get the following equation for x_{ndc}:

$$x_{ndc} = \frac{2x'}{r-l} - \frac{r+l}{r-l} = \frac{2 \cdot \frac{nx}{-z}}{r-l} - \frac{r+l}{r-l} = \frac{2nx}{(r-l)(-z)} - \frac{r+l}{r-l}$$
$$= \frac{\frac{2n}{r-l}x}{-z} + \frac{\frac{r+l}{r-l}z}{-z} = \frac{\frac{2n}{r-l} \cdot x + \frac{r+l}{r-l} \cdot z}{-z}.$$

Plugging these values into the equations we've just derived, we get the following equation for y_{ndc}:

$$y_{ndc} = \frac{2y'}{t-b} - \frac{t+b}{t-b} = \frac{2 \cdot \frac{ny}{-z}}{t-b} - \frac{t+b}{t-b} = \frac{2ny}{(t-b)(-z)} - \frac{t+b}{t-b}$$
$$= \frac{\frac{2n}{t-b}y}{-z} + \frac{\frac{t+b}{t-b}z}{-z} = \frac{\frac{2n}{t-b} \cdot x + \frac{t+b}{t-b} \cdot z}{-z}.$$

Note: Recall that these values are homogeneous coordinates. Thus, when entering these values in P, we must not include the division by $-z$.

Our equation for A' and the projection P can thus be updated as the following:

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ . & . & . & . \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}, \quad \therefore w' = -z.$$

The calculation of the final row z in our matrix is a little different. We know that z is the projection plane's location and does not depend on the x or y coordinates. Thus, we use the w-component to find the relationship between z_{ndc} and z.

We rewrite $z_{ndc} = z'/w' = (Az + Bw)/-z$. Recall that w is the w-coordinate of the camera space, and thus $w = 1$. Thus, we rewrite $z_{ndc} = (Az+B)/-z$. We know that the following (z, z_{ndc}) coordinates exist: $(-n, -1)$ and $(-f, 1)$. Plugging these in, we get the following:

$$\begin{cases} \frac{-An+B}{n} = -1 \\ \frac{-Af+B}{f} = 1 \end{cases}$$

We rewrite the equations as $B = An - n$ and $-Af + B = f$. Plugging the first into the second, we get the following:

$$-Af + (An - n) = f$$
$$A = -\frac{f+n}{f-n}.$$

Plugging this value of A back into B, we get the following:

$$(\frac{f+n}{f-n})n + B = -n$$
$$B = -\frac{2fn}{f-n}.$$

Plugging these values back into the original equation $z_{ndc} = (Az + B)/-z$, we get the following relation:

$$z_{ndc} = \frac{-\frac{f+n}{f-n}z - \frac{2fn}{f-n}}{-z}.$$

Thus, our projection matrix is calculated as the following:

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

5 Shadow Mapping

A shadow is a dark area where light from a light source is occluded by some object. In computer graphics, shadows are a great way of adding depth and realism to a scene.

One of the easier ways of implementing shadows is **shadow mapping**. The idea behind shadow mapping is pretty simple: shadows exist in areas that aren't reached by light, thus we can map shadows to a scene by rendering the scene from the light's point of view. Every point (or pixel) visible from the light source is going to be considered lit, while every point not visible from the light source (i.e. occluded by some object) is going to be considered to be in shadow (Figure 9).

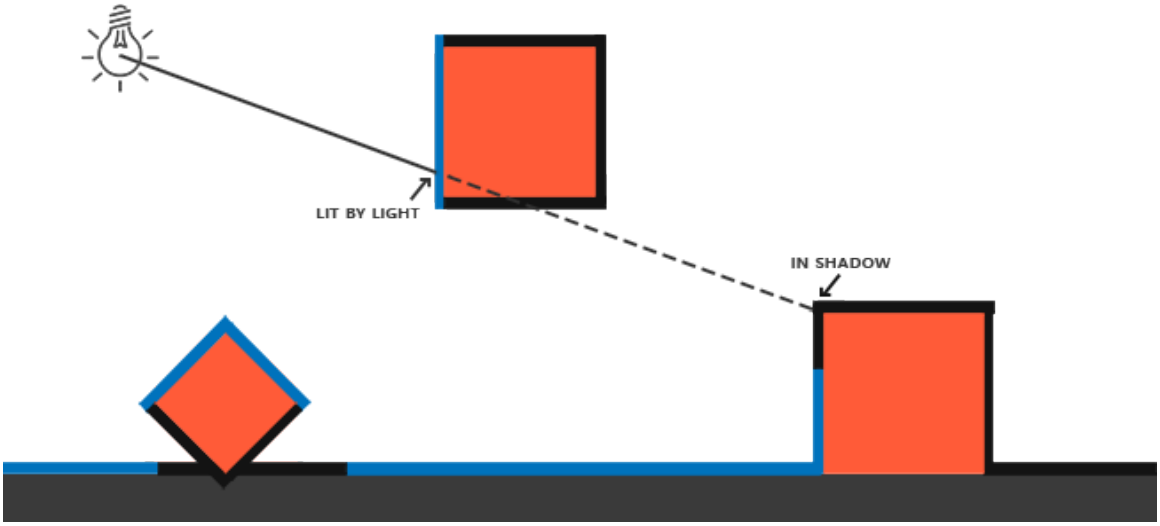

Figure 9: Adapted from LearnOpenGL

Consider the following scene

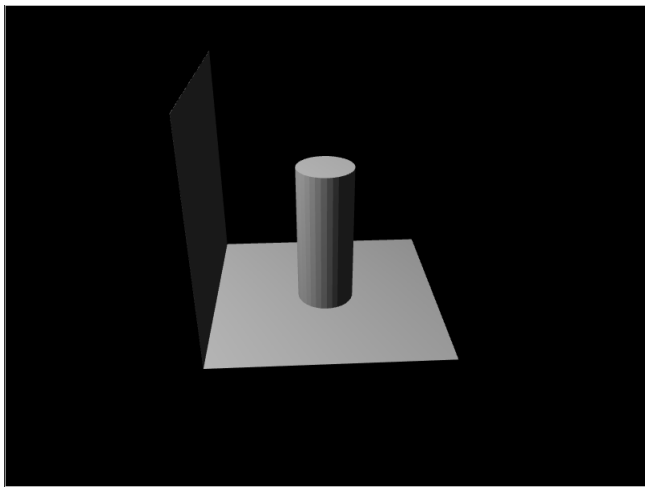

Figure 10: Adapted from GPU Shader Tutorial

The first step to shadow mapping involves generating a **shadow map**. A shadow map is a texture storing the **depth**, or the distance from the light source to a pixel, for each pixel seen from the light's point of view. The depth values (distances) get "clamped" to the range $[0, 1]$, where the pixels closest to the light source (where the distance is the smallest) are represented as the darkest,

and the pixels furthest away as the brightest.

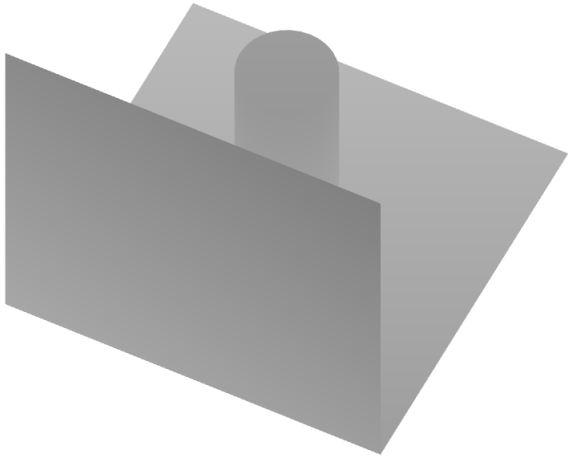

Figure 11: A simple shadow map (adapted from GPU Shader Tutorial)

Looking at the scene from the camera's point of view, we now check each pixel against the generated depth map: if the distance of the pixel is greater than the value stored in the shadow map, the pixel is in shadow. Otherwise, the pixel is illuminated. Consider for example point P in the following figure. Since P's distance to light is greater than the depth value stored in the shadow map, we know that P is in a shadow.

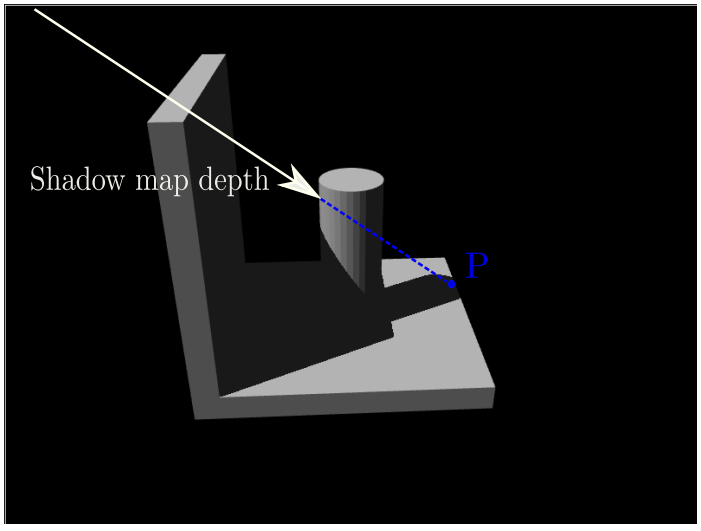

Figure 12: Adapted from GPU Shader Tutorial

6 Conclusion

Transformations are essential to computer graphics. The ability to define a mathematical function that can be applied to an object to create the desirable effect is central to computer animations. However, this is not limited to linear algebra. In fact, as we saw with quaternions, there are some transformations that are much more efficiently computed without the use of linear algebra. On the other hand, some transformations, like perspective projections, are fundamentally dependent on linear algebra. This is because some transformations are, by nature, more directly linear than others.

The unique applicability of fields of mathematics is why we study them in the first place. We enter a field of mathematics with a new way of thinking, marvelling at the beauties it reveals about our world. This is one reason we decided to expand into quaternions; we wanted to show the importance of diversifying mathematical knowledge in research.

In the future, our paper can be expanded upon to include a more detailed analysis of different shadow mapping techniques, exploring ray tracing, percentage closer filtering, perspective shadow maps, cascaded shadow maps, and variance shadow maps. Additionally, another paper might apply three dimensional rotations using linear algebra, and better demonstrate the true benefit of quaternions.

7 References

References

[1] Lay, David C. Linear Algebra and Its Applications / David C. Lay. Addison-Wesley, 1997.

[2] *Vector Math for 3D Computer Graphics*, Bradley Kjell

[3] *3D Projection* Wikipedia, Wikimedia Foundation, 27 June 2021

[4] Vince, John. Quaternions for Computer Graphics. Springer London, 2011.

[5] "Quaternion." Wikipedia, Wikimedia Foundation, 3 Nov. 2021, https://en.wikipedia.org/wiki/Quaternion.

[6] Lončarić, Nataša & Kraljić, Marko. (2018). Matrices in computer graphics. Tehnički glasnik. 12. 120-123. 10.31803/tg-20180119143651.

[7] "Shadow Mapping." Wikipedia, Wikimedia Foundation, 7 Nov. 2021, https://en.wikipedia.org/wiki/Shadow_mapping.

[8] Eberly, David. "A Linear Algebraic Approach to Quaternions." September 16, 2002, https://www.geometrictools.com/Documentation/LinearAlgebraicQuaternions.pdf

[9] "Quaternions and Spatial Rotation." Wikipedia, Wikimedia Foundation, 22 Nov. 2021, https://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation.

[10] Joel L. Weiner and George R. Wilkens, "Quaternions and Rotations in E^4.", The American Mathematical Monthly, Vol. 112, No. 1 (Jan., 2005), pp. 69-76

[11] Viro Oleg, "Lecture 5. Quaternions," http://www.math.stonybrook.edu/ oleg/courses/mat150-spr16/lecture-5.pdf

[12] "Spherical Linear Interpolation (SLERP)§." Spherical Linear Interpolation (Slerp) - Splines, Version 0.1.0-43-g0faf87f, 22 Nov. 2021, https://splines.readthedocs.io/en/latest/rotation/slerp.html.

[13] "Shader Advanced - Shadow Mapping." GPU Shader Tutorial, https://shadertutorial.dev/advanced/shadow-mapping/.

[14] "Shadow Mapping." LearnOpenGL, https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping.

[15] "Among Us Video Game SVG." Freesvgplanet, 16 Feb. 2021, https://freesvgplanet.com/among-us-svg-free-layered-among-us-svg-video-game-svg-instant-download-silhouette-cameo-shirt-design-among-svg-png-dxf-0984/.

[16] "8.3 - Perspective Projections." LearnWebGL, http://learnwebgl.brown37.net/08_projections/projections_perspective.html.

[17] Ahn, Song Ho. OpenGL Transformation, http://www.songho.ca/opengl/gl_transform.htmlprojection.

[18] Alamia, Marco. "Article - World, View and Projection Transformation Matrices." Coding Labs: World, View and Projection Transformation Matrices, http://www.codinglabs.net/article_world_view_projection_matrix.aspx.

[19] Dobrushkin, Vladimir. "Computer Graphics." Fluids at Brown, https://www.cfm.brown.edu/people/dobrush/cs52/Mathematica/Part7/graphics.html.

[20] Scratchapixel. "The Perspective and Orthographic Projection Matrix." Scratchapixel, 15 Aug. 2014, https://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix/projection-matrix-introduction.