159.709 Computer Graphics

Lecture 03 - Linear Algebra & Transforms

Daniel Playne
d.p.playne@massey.ac.nz

Linear Algebra

A major portion of computer graphics depends on representing and manipulating objects in three-dimensional coordinates. A lot of the techniques used are fundamental concepts in linear algebra.

We will go through an introduction to some of the important and most useful concepts from linear algebra but this will most certainly not be comprehensive. Linear algebra as a subject has more than enough content to fill many courses.

Linear Algebra

If this is your first introduction to linear algebra (or you haven't used it in a long time) you may find some of the concepts or methods rather abstract - stick with it.

These methods will be put to good use for transforming objects and converting between the different coordinate spaces in the graphics pipeline.

Most of the time these transforms will be built using functions but it will be important to understand the methods used in order to correctly build and debug your programs.

Euclidean Spaces

A Euclidean space is an n-dimensional real space denoted by $\mathbb{R}^n$. Within this space we can have vectors denoted by $\vec{\mathbf{v}}$ that are n-tuples, that is a list of real numbers.

$$\vec{\mathbf{v}} \in \mathbb{R}^n = \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-1} \end{pmatrix}$$

For example, a vector in two-dimensional space can be written as: $\vec{\mathbf{v}} \in \mathbb{R}^2 = \begin{pmatrix} v_0 \\ v_1 \end{pmatrix}$.

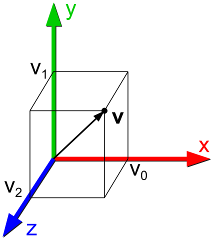
Euclidean Spaces

The real numbers that define a vector $\vec{\mathbf{v}} \in \mathbb{R}^n$ are expressed in a basis of the space. We will almost always stick to representing vectors in the standard basis, which in three-dimensions is:

$$\vec{\mathbf{x}} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad \vec{\mathbf{y}} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \quad \vec{\mathbf{z}} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

Euclidean Spaces

This gives us the interpretation of a vector $\vec{\mathbf{v}} = (v_0, v_1, v_2)$ as $\vec{\mathbf{v}} = v_0\vec{\mathbf{x}} + v_1\vec{\mathbf{y}} + v_2\vec{\mathbf{z}}$. Which can be visualised as.



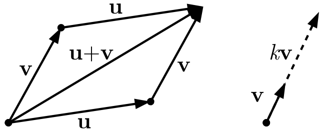
Vector Operators

Vectors in a Euclidean space have the operators ***addition*** and ***multiplication by a scalar***.

$$
\begin{aligned}
\vec{\mathbf{u}} + \vec{\mathbf{v}} &= \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{n-1} \end{pmatrix} + \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-1} \end{pmatrix} = \begin{pmatrix} u_0 + v_0 \\ u_1 + v_1 \\ \vdots \\ u_{n-1} + v_{n-1} \end{pmatrix} \in \mathbb{R}^n \\
k\vec{\mathbf{v}} &= k \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-1} \end{pmatrix} = \begin{pmatrix} kv_0 \\ kv_1 \\ \vdots \\ kv_{n-1} \end{pmatrix} \in \mathbb{R}^n
\end{aligned}
$$

Vector Operators

The addition and multiplication operators can be visualised (in two-dimensions) as:



Some important rules for Euclidean spaces and ***addition***:

$$\begin{aligned}(\vec{\mathbf{u}} + \vec{\mathbf{v}}) + \vec{\mathbf{w}} &= \vec{\mathbf{u}} + (\vec{\mathbf{v}} + \vec{\mathbf{w}}) && \text{(associativity)} \\ \vec{\mathbf{u}} + \vec{\mathbf{v}} &= \vec{\mathbf{v}} + \vec{\mathbf{u}} && \text{(commutativity)}\end{aligned}$$

Addition Rules

Two important vectors $\vec{\mathbf{0}} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$ and $\vec{-\mathbf{v}} = \begin{pmatrix} -v_0 \\ -v_1 \\ \vdots \\ -v_{n-1} \end{pmatrix}$

that give the following:

$$\begin{aligned} \vec{\mathbf{0}} + \vec{\mathbf{v}} &= \vec{\mathbf{v}} \quad \text{(zero identity)} \\ \vec{\mathbf{v}} + (\vec{-\mathbf{v}}) &= \vec{\mathbf{0}} \quad \text{(additive inverse)} \end{aligned}$$

Multiplication Rules

Some important rules for Euclidean spaces and ***multiplication***:

$$\begin{aligned}
(ab)\vec{\mathbf{u}} &= a(b\vec{\mathbf{u}}) \\
(a+b)\vec{\mathbf{u}} &= a\vec{\mathbf{u}} + b\vec{\mathbf{u}} \quad \text{(distributive law)} \\
a(\vec{\mathbf{u}}+\vec{\mathbf{v}}) &= a\vec{\mathbf{u}} + a\vec{\mathbf{v}} \quad \text{(distributive law)} \\
1\vec{\mathbf{u}} &= \vec{\mathbf{u}}
\end{aligned}$$

Length of a Vector

The length of a vector is written as $|\vec{\mathbf{v}}|$ simply the (scalar) number that represents how long the vector is. This can be calculated as follows:

$$|\vec{\mathbf{v}}| = \sqrt{v_0^2 + v_1^2 + ... + v_{n-1}^2}$$

A vector with length $|\vec{\mathbf{u}}| = 1$ is called a ***unit vector***. The unit vector $\vec{\mathbf{u}}$ with the same direction as a non-unit vector $\vec{\mathbf{v}}$ can be calculated with:

$$\vec{\mathbf{u}} = \frac{\vec{\mathbf{v}}}{|\vec{\mathbf{v}}|}$$

Dot Product

In a Euclidean space we can also compute the ***dot product*** (or inner product) of two vectors. This is denoted as $\vec{\mathbf{u}} \cdot \vec{\mathbf{v}}$ and is defined as:

$$\vec{\mathbf{u}} \cdot \vec{\mathbf{v}} = \Sigma_{i=0}^{n-1} u_i v_i = u_0 v_0 + u_1 v_1 + \ldots + u_{n-1} v_{n-1}$$

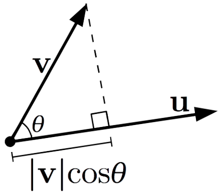
The dot product has the following rules ($\vec{\mathbf{u}} \perp \vec{\mathbf{v}}$ means that $\vec{\mathbf{u}}$ and $\vec{\mathbf{v}}$ are perpendicular):

$$\begin{aligned}
(\vec{\mathbf{u}} + \vec{\mathbf{v}}) \cdot \vec{\mathbf{w}} &= \vec{\mathbf{u}} \cdot \vec{\mathbf{w}} + \vec{\mathbf{v}} \cdot \vec{\mathbf{w}} \\
(a\vec{\mathbf{u}}) \cdot \vec{\mathbf{v}} &= a(\vec{\mathbf{u}} \cdot \vec{\mathbf{v}}) \\
\vec{\mathbf{u}} \cdot \vec{\mathbf{v}} &= \vec{\mathbf{v}} \cdot \vec{\mathbf{u}} \\
\vec{\mathbf{u}} \cdot \vec{\mathbf{v}} &= 0 \iff \vec{\mathbf{u}} \perp \vec{\mathbf{v}}
\end{aligned}$$

Dot Product

The dot product is remarkably useful as it has the following relationship to the angle between two vectors.

$$\vec{\mathbf{u}} \cdot \vec{\mathbf{v}} = |\vec{\mathbf{u}}||\vec{\mathbf{v}}| \cos(\theta)$$

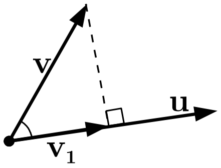


Dot Product

One very useful property of the dot product is that we can use it to 'project' a vector onto another vector. This can be calculated with the following:

$$\vec{\mathbf{v_1}} = \left(\frac{\vec{\mathbf{u}} \cdot \vec{\mathbf{v}}}{|\vec{\mathbf{u}}|^2} \right) \vec{\mathbf{u}} \tag{1}$$

Which can be visualised as:



Cross Product

The ***cross product*** or vector product is the product of two vectors and is normally denoted as $\vec{\mathbf{w}} = \vec{\mathbf{u}} \times \vec{\mathbf{v}}$.

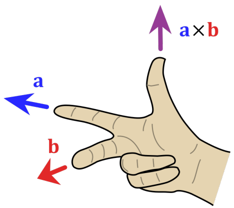
The cross product produces a vector $\vec{\mathbf{w}}$ with the following properties (θ is the smallest angle between $\vec{\mathbf{u}}$ and $\vec{\mathbf{v}}$):

- $|\vec{\mathbf{w}}| = |\vec{\mathbf{u}} \times \vec{\mathbf{v}}| = |\vec{\mathbf{u}}||\vec{\mathbf{v}}| \sin(\theta)$
- $\vec{\mathbf{w}} \perp \vec{\mathbf{u}}$ and $\vec{\mathbf{w}} \perp \vec{\mathbf{v}}$
- $\vec{\mathbf{u}}$, $\vec{\mathbf{v}}$ and $\vec{\mathbf{w}}$ form a right-handed system.

Cross Product

In three-dimensions the cross product can be calculated as:

$$\vec{\mathbf{w}} = \begin{pmatrix} w_x \\ w_y \\ w_z \end{pmatrix} = \vec{\mathbf{u}} \times \vec{\mathbf{v}} = \begin{pmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{pmatrix}$$



CC-BY-SA-3.0 Based on:

right hand rule cross product, User:Acdx, Wikimedia Commons

Matrices

Matrices are used heavily in computer graphics and are used to represent many types transforms and have many useful operations and properties. Many of the operations we will discuss are valid for arbitrarily sized matrices but we will focus on the 2×2, 3×3 and 4×4 matrices commonly found in computer graphics.

In general an $m \times n$ matrix $\mathbf{A}$ has m rows and n columns and is written as:

$$\mathbf{A} = \begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

Matrix-Matrix Addition

Two matrices $\mathbf{A}$ and $\mathbf{B}$ can be added together if they are the same size (both $m \times n$) to give a new matrix also of size $m \times n$.

$$\begin{aligned}
\mathbf{A}+\mathbf{B} &= \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} + \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \\ b_{30} & b_{31} & b_{32} & b_{33} \end{bmatrix} \\
&= \begin{bmatrix} a_{00}+b_{00} & a_{01}+b_{01} & a_{02}+b_{02} & a_{03}+b_{03} \\ a_{10}+b_{10} & a_{11}+b_{11} & a_{12}+b_{12} & a_{13}+b_{13} \\ a_{20}+b_{20} & a_{21}+b_{21} & a_{22}+b_{22} & a_{23}+b_{23} \\ a_{30}+b_{30} & a_{31}+b_{31} & a_{32}+b_{32} & a_{33}+b_{33} \end{bmatrix}
\end{aligned}$$

Scalar-Matrix Multiplication

A scalar c and a matrix $\mathbf{A}$ can be multiplied together to give a new matrix of the same size as $\mathbf{A}$.

$$\begin{aligned} c\mathbf{A} &= c\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \\ &= \begin{bmatrix} a_{00}\,c & a_{01}\,c & a_{02}\,c & a_{03}\,c \\ a_{10}\,c & a_{11}\,c & a_{12}\,c & a_{13}\,c \\ a_{20}\,c & a_{21}\,c & a_{22}\,c & a_{23}\,c \\ a_{30}\,c & a_{31}\,c & a_{32}\,c & a_{33}\,c \end{bmatrix} \end{aligned}$$

Matrix-Vector Multiplication

An $m \times n$ matrix $\mathbf{A}$ can be multiplied by an n-element vector $\vec{\mathbf{v}}$ as follows to produce a new n-element vector.

$$
\begin{aligned}
\mathbf{A}\vec{\mathbf{v}} &= \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{pmatrix} \\
&= \begin{pmatrix} \sum_{k=0}^{n-1} a_{0k} v_k \\ \sum_{k=0}^{n-1} a_{1k} v_k \\ \sum_{k=0}^{n-1} a_{2k} v_k \\ \sum_{k=0}^{n-1} a_{3k} v_k \end{pmatrix} \\
&= \begin{pmatrix} a_{00} v_0 + a_{01} v_1 + a_{02} v_2 + a_{03} v_3 \\ a_{10} v_0 + a_{11} v_1 + a_{12} v_2 + a_{13} v_3 \\ a_{20} v_0 + a_{21} v_1 + a_{22} v_2 + a_{23} v_3 \\ a_{30} v_0 + a_{31} v_1 + a_{32} v_2 + a_{33} v_3 \end{pmatrix}
\end{aligned}
$$

Matrix-Matrix Multiplication

An $m \times n$ matrix $\mathbf{A}$ can be multiplied by an $n \times p$ matrix $\mathbf{B}$ to give a $m \times p$ matrix.

$$
\begin{aligned}
\mathbf{AB} &= \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \\ b_{30} & b_{31} & b_{32} & b_{33} \end{bmatrix} \\
&= \begin{bmatrix} \sum_{k=0}^{n-1} a_{0k}b_{k0} & \sum_{k=0}^{n-1} a_{0k}b_{k1} & \sum_{k=0}^{n-1} a_{0k}b_{k2} & \sum_{k=0}^{n-1} a_{0k}b_{k3} \\ \sum_{k=0}^{n-1} a_{1k}b_{k0} & \sum_{k=0}^{n-1} a_{1k}b_{k1} & \sum_{k=0}^{n-1} a_{1k}b_{k2} & \sum_{k=0}^{n-1} a_{1k}b_{k3} \\ \sum_{k=0}^{n-1} a_{2k}b_{k0} & \sum_{k=0}^{n-1} a_{2k}b_{k1} & \sum_{k=0}^{n-1} a_{2k}b_{k2} & \sum_{k=0}^{n-1} a_{2k}b_{k3} \\ \sum_{k=0}^{n-1} a_{3k}b_{k0} & \sum_{k=0}^{n-1} a_{3k}b_{k1} & \sum_{k=0}^{n-1} a_{3k}b_{k2} & \sum_{k=0}^{n-1} a_{3k}b_{k3} \end{bmatrix}
\end{aligned}
$$

Matrix Multiplication Rules

Some important rules for matrix-matrix multiplication:

- $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$
- $(\mathbf{A} + \mathbf{B})\mathbf{C} = (\mathbf{AC}) + (\mathbf{BC})$
- $\mathbf{AI} = \mathbf{IA} = \mathbf{A}$

Where $\mathbf{I}$ is the identity matrix:

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matrix Multiplication Rules

It is very important to note that (in general) for matrix multiplication:

$$\mathbf{AB} \neq \mathbf{BA}$$

This will be an especially important rule to remember.

Matrix Transpose

The transpose of a matrix $\mathbf{A}$ is denoted $\mathbf{A}^T$ and is a matrix where $\mathbf{A} = [a_{ij}]$ and $\mathbf{A}^T = [a_{ji}]$ which means the columns of $\mathbf{A}$ become the rows of $\mathbf{A}^T$.

$$\mathbf{A}^T = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}^T = \begin{bmatrix} a_{00} & a_{10} & a_{20} & a_{30} \\ a_{01} & a_{11} & a_{21} & a_{31} \\ a_{02} & a_{12} & a_{22} & a_{32} \\ a_{03} & a_{13} & a_{23} & a_{33} \end{bmatrix}$$

Matrix Inverse

The inverse of a square matrix $\mathbf{A}$ is denoted $\mathbf{A}^{-1}$ and is a matrix where $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$.

Not all matrices have an inverse (a matrix which does not have an inverse is called a ***singular*** or ***degenerate*** matrix).

There are a number of methods that can be used to find the inverse of a matrix, we will not cover them in this course.

Transforms

A ***transform*** is an operation that takes points, vectors, colours etc and changes them in some way. Understanding how transforms work and becoming familiar with their use is extremely important in computer graphics.

They are used to position and animate objects within a scene, define lights and control cameras. They are used to create the effects of perspective and convert between different coordinate spaces.

Scale

The scale transform can be used to scale an entity in the x-, y- or z-dimensions by factors of s_x, s_y and s_z respectively. This can be represented with the following matrix $S(s_x, s_y, s_z)$.

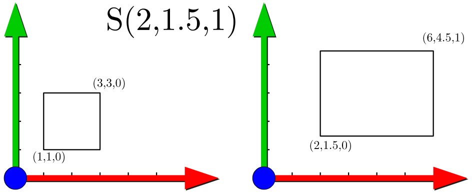
$$\begin{aligned} S(s_x, s_y, s_z) &= \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \\ S\vec{\mathbf{v}} &= \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = \begin{pmatrix} s_x v_x \\ s_y v_y \\ s_z v_z \end{pmatrix} \end{aligned}$$

For example:

$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \\ 3 \end{pmatrix}$$

Scale

Scaling the points in a shape will affect not only the size but also the position of the object.



If the intention is only to change the size of the object, it should be centred around the origin $(0, 0, 0)$ before the scaling transform is applied.

Shear

The ***shear*** transform can be used to distort an entire scene. In three-dimensions there are six basic shearing matrices $H_{xy}(s)$, $H_{xz}(s)$, $H_{yx}(s)$, $H_{yz}(s)$, $H_{zx}(s)$, $H_{zy}(s)$ where the first index denotes the coordinate being changed and the second denotes the coordinate used for shearing.

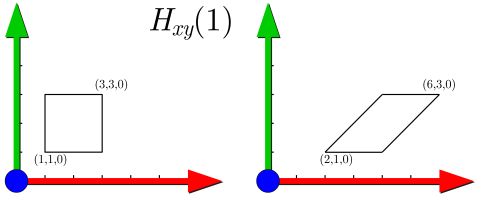
For example:

$$H_{xy}(s) = \begin{bmatrix} 1 & s & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Shear

The shear transform represents the addition of a one row or column multiplied by the parameter s to another.

The H_{xy} transform can be visualised as follows:



Rotation

Rotation transforms will rotate a vector/point by a certain angle around a given axis passing through the origin. This is a *rigid-body* transform which means the distances between points (and handedness) are preserved.

Rotations around the major axes are common and easy to represent with matrices $R_x(\phi)$, $R_y(\phi)$, $R_z(\phi)$, which rotate a vector/-point by an angle of ϕ around the x, y and z axes respectively.

Rotation

The rotation matrices $R_x(\phi)$, $R_y(\phi)$, $R_z(\phi)$ are:

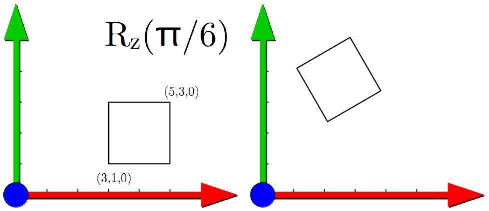
$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix}$$

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

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

Rotation

For example, the following transform shows a rotation of $\frac{\pi}{6}$ around the z-axis using $R_z(\phi)$.



Translation

While scaling and rotations (in three-dimensions) can be represented by 3×3 matrices, translations cannot.

A point describes a location in space while a vector represents a direction and has no location. Translations have no meaning for a vector, but are significant for points.

The usual solution to this problem used in computer graphics is to represent points and vectors in homogeneous notation.

Homogeneous Notation

Homogeneous notation is very heavily used in computer graphics to represent points and transformations, it is also rarely covered in linear algebra books.

Homogeneous notation augments matrices, vectors and points with an additional w dimension. Three-dimensional points/vectors become $\vec{\mathbf{v}} = (v_x, v_y, v_z, v_w)$ with $v_w = 1$ representing a point and $v_w = 0$ representing a vector.

Homogeneous Notation

The scaling, shear and rotation matrices can be augmented to give their homogeneous forms:

$$\begin{aligned}
S(s_x, s_y, s_z) &= \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
H_{xy}(s) &= \begin{bmatrix} 1 & s & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
R_x(\phi) &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned}$$

Translation

Translation transforms can be represented by a matrix $T(t_x, t_y, t_z)$ in this form as follows:

$$T(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

When multiplied by a point $\vec{\mathbf{p}} = (p_x, p_y, p_z, 1)$ gives:

$$T(t_x, t_y, t_z)\vec{\mathbf{p}} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x + t_x \\ p_y + t_y \\ p_z + t_z \\ 1 \end{pmatrix}$$

Translation

However, when a transform is applied to a vector $\vec{\mathbf{v}} = (v_x, v_y, v_z, 0)$ (representing a direction) there will be no effect.

$$T(t_x, t_y, t_z)\vec{\mathbf{v}} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \\ 0 \end{pmatrix} = \begin{pmatrix} v_x \\ v_y \\ v_z \\ 0 \end{pmatrix}$$

Transforms

Homogeneous notation allow the different transforms used in graphics applications to be defined in a consistent format.

Representing transforms as matrices allows a series of transforms to be combined together to a single matrix that represents the entire series. This is achieved by multiplying the matrices together.

Transform Order

Remember from matrix multiplication that:

$$\mathbf{AB} \neq \mathbf{BA}$$

This is especially important when combining transformations to position objects and cameras in 3D graphical scenes. Combining the same set of transforms in different orders will produce different transforms.

Transform Order

For example, consider the combination of the two transforms $R_z(\frac{\pi}{6})$ and $T(4, 0, 0)$, these will be represented as follows:

$$R_z(\phi) = \begin{bmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad T(4, 0, 0) = \begin{bmatrix} 1 & 0 & 0 & 4 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transform Order

For example, consider the combination of the two transforms $R_z(\frac{\pi}{6})$ and $T(4,0,0)$.

$$\begin{aligned}
R_z(\frac{\pi}{6})\,T(4,0,0) &= \begin{bmatrix}\cos\frac{\pi}{6} & -\sin\frac{\pi}{6} & 0 & 0\\ \sin\frac{\pi}{6} & \cos\frac{\pi}{6} & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0 & 4\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\\
&= \begin{bmatrix}\cos\frac{\pi}{6} & -\sin\frac{\pi}{6} & 0 & 4\cos\frac{\pi}{6}\\ \sin\frac{\pi}{6} & \cos\frac{\pi}{6} & 0 & 4\sin\frac{\pi}{6}\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}
\end{aligned}$$

Transform Order

When applied to a point $\vec{\mathbf{p}} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix}$ this gives:

$$\begin{aligned}
R_z(\frac{\pi}{6})\,T(4,0,0) &= \begin{bmatrix} \cos\frac{\pi}{6} & -\sin\frac{\pi}{6} & 0 & 4\cos\frac{\pi}{6} \\ \sin\frac{\pi}{6} & \cos\frac{\pi}{6} & 0 & 4\sin\frac{\pi}{6} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} \\
&= \begin{pmatrix} \cos\frac{\pi}{6} - \sin\frac{\pi}{6} + 4\cos\frac{\pi}{6} \\ \sin\frac{\pi}{6} + \cos\frac{\pi}{6} + 4\sin\frac{\pi}{6} \\ 0 \\ 1 \end{pmatrix} \\
&\approx \begin{pmatrix} 3.83 \\ 3.37 \\ 0 \\ 1 \end{pmatrix}
\end{aligned}$$

Transform Order

And in the opposite order:

$$
\begin{aligned}
T(4,0,0)R_z(\frac{\pi}{6}) &= \begin{bmatrix} 1 & 0 & 0 & 4 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\frac{\pi}{6} & -\sin\frac{\pi}{6} & 0 & 0 \\ \sin\frac{\pi}{6} & \cos\frac{\pi}{6} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} \cos\frac{\pi}{6} & -\sin\frac{\pi}{6} & 0 & 4 \\ \sin\frac{\pi}{6} & \cos\frac{\pi}{6} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned}
$$

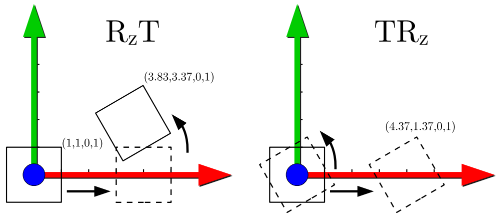
Transform Order

When applied to the same point $\vec{\mathbf{p}} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix}$ this gives:

$$\begin{aligned} T(4,0,0)R_z(\frac{\pi}{6})\vec{\mathbf{p}} &= \begin{bmatrix} \cos\frac{\pi}{6} & -\sin\frac{\pi}{6} & 0 & 4 \\ \sin\frac{\pi}{6} & \cos\frac{\pi}{6} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} \cos\frac{\pi}{6} - \sin\frac{\pi}{6} + 4 \\ \sin\frac{\pi}{6} + \cos\frac{\pi}{6} \\ 0 \\ 1 \end{pmatrix} \\ &\approx \begin{pmatrix} 4.37 \\ 1.37 \\ 0 \\ 1 \end{pmatrix} \end{aligned}$$

Matrix Multiplication Rules

The difference between these two transforms can be visualised as follows:



Note that the transforms are 'performed' in the order *right-to-left*. The transform R_zT performs the translation T first and then the rotation R_z.

Model-View-Projection Transforms

Most computer graphics applications use the ***Model-View-Projection*** approach to transforming the scene.

The transforms that must be applied are broken into three separate stages:

- ***Model*** - transforms the model to position in the world (***model-coordinates*** to ***world-coordinates***).
- ***View*** - transforms the scene to the *camera view* of the scene (***world-coordinates*** to ***eye-coordinates***).
- ***Projection*** - projects the scene onto a 2D screen (***eye-coordinates*** to ***clip-coordinates***).

Model Transform

The ***model*** transform is usually a simple combination of ***scale***, ***rotation*** and ***translation*** transforms to scale the object to the correct size, rotate it to the desired orientation and then translate it to its correct place in the scene.

While the exact order may vary based on the application, the usual order of these transforms is:

translation × ***rotation*** × ***scale***

View Transform

The ***view*** transform represents the position of a *virtual camera* that we can control and move about our 3D scene*.

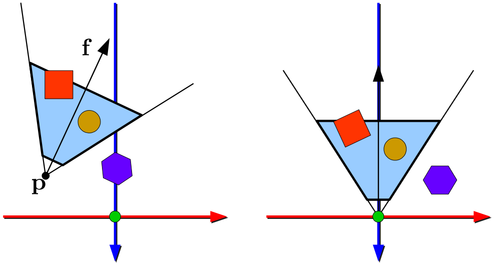
The *camera* will be represented by a position $\vec{\mathbf{p}}$, the direction $\vec{\mathbf{f}}$ that the camera is facing and a vector $\vec{\mathbf{u}}$ representing the up direction. The actual view transform will transform all of the objects in the scene (in ***world coordinates***) to the position they would appear as viewed by the camera (***eye-coordinates***).

* The camera actually always stays at the origin $(0, 0, 0)$ and we move the scene around it.

View Transform

The view transform takes the objects and transforms them to their position relative to the camera.

This can be visualised (using a perspective camera) as follows:



View Transform

The view transform will be built by making a rotation matrix V_R (the direction the camera is facing) and a translation matrix V_T (the position of the camera).

To calculate the rotation, the ***forward*** vector $\vec{\mathbf{f}}$ and the ***up*** vector $\vec{\mathbf{u}}$ are used to calculate a third vector - the ***right*** vector $\vec{\mathbf{r}}$ using the following cross product:

$$\vec{\mathbf{r}} = \vec{\mathbf{f}} \times \vec{\mathbf{u}}$$

(The vectors $\vec{\mathbf{f}}$ and $\vec{\mathbf{u}}$ should both be unit vectors.)

View Rotation

These three vectors can be then be used to create the view rotation matrix:

$$V_R = \begin{bmatrix} r_x & r_y & r_z & 0 \\ u_x & u_y & u_z & 0 \\ -f_x & -f_y & -f_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

View Translation

The translation matrix for the view transform is a standard translation matrix.

$$V_T = \begin{bmatrix} 1 & 0 & 0 & -p_x \\ 0 & 1 & 0 & -p_y \\ 0 & 0 & 1 & -p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(Note that the camera position is $\vec{\mathbf{p}}$ so we should translate all the other objects by $-\vec{\mathbf{p}}$.)

View Transform

Finally the ***view*** transform $\mathbf{V}$ can be constructed by calculating:

$$\mathbf{V} = \mathbf{V}_R \mathbf{V}_T$$

Given that transforms are performed in the order of *right-to-left* - this order might appear to be the wrong way around.

Remember that the view transform moves the scene relative to the camera, so we must first translate so the camera is at the origin and then rotate.

Projection Transform

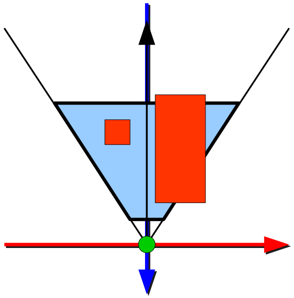
The ***projection*** transform takes objects in ***eye-coordinates*** and transforms the viewing frustum into a [-1,1] cube (***clip space***).

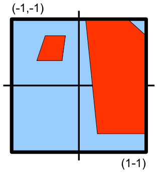
The ***perspective*** projection is most commonly used and creates the effects of foreshortening and vanishing points.

The other common projection transform is ***orthographic*** projection where the viewing frustum is a rectilinear box that represents a two-dimensional view of the world.

Perspective Projection

An example of the ***perspective*** transform (as viewed from above) can be visualised as follows:

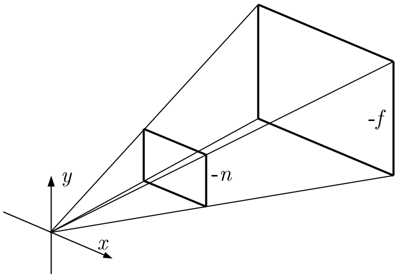




Perspective Projection

The perspective viewing frustum is defined with near and far clipping plans along the z-axis and a field of view defined by an aspect ratio and a vertical field of view (usually $65° - 70°$).

This perspective viewing frustum can be visualised as:



Perspective Projection Transform

The perspective projection transform P_P can be defined by the near and far clipping plans n and f, the aspect ratio $aspect$ and the vertical field of view $fovy$.

$$f = \frac{1}{\tan(\frac{fovy}{2})}$$

$$P_P = \begin{bmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

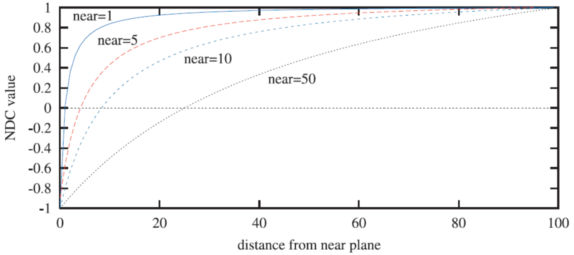
Perspective Projection Transform

A number of important things to note about this transform:

- Near & far clipping plans are specified as inverse z-coordinates (positive values).
- Inverts the z-coordinates.
- Non-linear mapping of z-coordinates for increased depth precision of objects near the camera.
- Sets $w = -z$ for perspective division.

Depth Buffer Mapping

The z-coordinates of the objects in the scene are skewed to give greater accuracy to objects that are near the viewer. Achieved by reducing the depth buffer accuracy at greater distances.



RTR 3.04.20 Depth Buffering - Copyright A.K. Peters Ltd. / Fair Use

Transforms in OpenGL

Transforms are usually implemented in OpenGL by constructing the matrices in the application stage and passing them into the shaders.

The standard ***model-view-projection*** transforms are usually computed in the ***vertex shader*** and are computed separately for each vertex.

When a transform is changed, the new matrix can be calculated and copied to the shader which will then affect how the objects in the scene are rendered.

Transforms in OpenGL

The matrices themselves are simply represented as an array of sixteen floating-point values. In OpenGL these matrices are stored in ***column-major*** order which means the indexes of the matrix entries are:

$$\begin{bmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{bmatrix}$$

Matrices can be defined as row-major order but must be transposed before they are copied into the graphics card (it is recommended that you always use column-major).

Transforms in OpenGL

Using this order we can define some functions to create the desired transforms - starting with the identity matrix:

```c
// Create an identity matrix
void identity(float I[16]) {
   I[0]  = 1.0f;  I[4]  = 0.0f;  I[8]  = 0.0f;  I[12] = 0.0f;
   I[1]  = 0.0f;  I[5]  = 1.0f;  I[9]  = 0.0f;  I[13] = 0.0f;
   I[2]  = 0.0f;  I[6]  = 0.0f;  I[10] = 1.0f;  I[14] = 0.0f;
   I[3]  = 0.0f;  I[7]  = 0.0f;  I[11] = 0.0f;  I[15] = 1.0f;
}
```

Transforms in OpenGL

We can similarly define translation and scale matrices.

```
// Create a translation matrix with (x,y,z)
void translate(float tx, float ty, float tz, float T[16]) {
   T[0]  = 1.0f;  T[4]  = 0.0f;  T[8]  = 0.0f;  T[12] = tx;
   T[1]  = 0.0f;  T[5]  = 1.0f;  T[9]  = 0.0f;  T[13] = ty;
   T[2]  = 0.0f;  T[6]  = 0.0f;  T[10] = 1.0f;  T[14] = tz;
   T[3]  = 0.0f;  T[7]  = 0.0f;  T[11] = 0.0f;  T[15] = 1.0f;
}

// Create a scale matrix
void scale(float sx, float sy, float sz, float S[16]) {
   S[0]  = sx;    S[4]  =  0.0f;  S[8]  = 0.0f;  S[12] = 0.0f;
   S[1]  = 0.0f;  S[5]  =  sy;    S[9]  = 0.0f;  S[13] = 0.0f;
   S[2]  = 0.0f;  S[6]  =  0.0f;  S[10] = sz;    S[14] = 0.0f;
   S[3]  = 0.0f;  S[7]  =  0.0f;  S[11] = 0.0f;  S[15] = 1.0f;
}
```

Transforms in OpenGL

Matrix operations are easy (if a little tedious) to write:

```
// Multiply matrix a * b to give c
void multiply44(float a[16], float b[16], float c[16]) {
   // Multiply each row of A with each column of B to give C
   c[0]  = a[0]*b[0]  + a[4]*b[1]  + a[8]*b[2]   + a[12]*b[3];
   c[4]  = a[0]*b[4]  + a[4]*b[5]  + a[8]*b[6]   + a[12]*b[7];
   c[8]  = a[0]*b[8]  + a[4]*b[9]  + a[8]*b[10]  + a[12]*b[11];
   c[12] = a[0]*b[12] + a[4]*b[13] + a[8]*b[14]  + a[12]*b[15];

   c[1]  = a[1]*b[0]  + a[5]*b[1]  + a[9]*b[2]   + a[13]*b[3];
   c[5]  = a[1]*b[4]  + a[5]*b[5]  + a[9]*b[6]   + a[13]*b[7];
   c[9]  = a[1]*b[8]  + a[5]*b[9]  + a[9]*b[10]  + a[13]*b[11];
   c[13] = a[1]*b[12] + a[5]*b[13] + a[9]*b[14]  + a[13]*b[15];

   c[2]  = a[2]*b[0]  + a[6]*b[1]  + a[10]*b[2]  + a[14]*b[3];
   c[6]  = a[2]*b[4]  + a[6]*b[5]  + a[10]*b[6]  + a[14]*b[7];
   c[10] = a[2]*b[8]  + a[6]*b[9]  + a[10]*b[10] + a[14]*b[11];
   c[14] = a[2]*b[12] + a[6]*b[13] + a[10]*b[14] + a[14]*b[15];

   c[3]  = a[3]*b[0]  + a[7]*b[1]  + a[11]*b[2]  + a[15]*b[3];
   c[7]  = a[3]*b[4]  + a[7]*b[5]  + a[11]*b[6]  + a[15]*b[7];
   c[11] = a[3]*b[8]  + a[7]*b[9]  + a[11]*b[10] + a[15]*b[11];
   c[15] = a[3]*b[12] + a[7]*b[13] + a[11]*b[14] + a[15]*b[15];
}
```

Transforms in OpenGL

This (and some other matrix operations) can be used to create the view matrix:

```c
// View Transform – forwards and up must be unit vectors
void view(float p[4], float f[3], float u[3], float V[16]) {
   // Calculate right vector
   float r[3];
   cross_product(f, u, r);

   // View rotation and translation
   float Vr[16], Vt[16];

   // Set view rotation
   Vr[0]  =  r[0];  Vr[4]  =  r[1];  Vr[8]  =  r[2];  Vr[12] =  0.0f;
   Vr[1]  =  u[0];  Vr[5]  =  u[1];  Vr[9]  =  u[2];  Vr[13] =  0.0f;
   Vr[2]  = -f[0];  Vr[6]  = -f[1];  Vr[10] = -f[2];  Vr[14] =  0.0f;
   Vr[3]  =  0.0f;  Vr[7]  =  0.0f;  Vr[11] =  0.0f;  Vr[15] =  1.0f;

   // Set view translation
   translate(-p[0], -p[1], -p[2], Vt);

   // Calculate view transform
   multiply44(Vr, Vt, V);
}
```

Transforms in OpenGL

Once our matrices have been defined in the application stage, we need a way to copy them into our shaders. For this we will use a `uniform` variable in our shaders.

`uniform` variables can be created in any shader as a way to copy information from the application into the rendering pipeline, these are read-only variables (from the shader perspective) that will have the same value for every shader instance.

Transforms in OpenGL

To define a new vertex shader that takes three separate transform matrices (`model`, `view` and `projection`) we can define three `uniform` variables of type `mat4` which is the GLSL type for a 4×4 matrix.

GLSL provides support for most matrix operations such as matrix-matrix multiplication, matrix-vector multiplication etc.

Transforms in OpenGL

Vertex Shader:

```glsl
// OpenGL 3.3
#version 330

// Input to Vertex Shader
in vec3 vert_Position;
in vec3 vert_Colour;

// Transform Matrices
uniform mat4 u_Model;
uniform mat4 u_View;
uniform mat4 u_Projection;

// Output to Fragment Shader
out vec3 frag_Colour;

void main() {
   // Vertex Colour
   frag_Colour = vert_Colour;

   //------------------------------------------------
   // Vertex Position
   //------------------------------------------------
   gl_Position = u_Projection * u_View * u_Model * vec4(vert_Position, 1.0f);
}
```

Transforms in OpenGL

We can copy transform matrices into these uniforms using code such as the following (example shown for the model matrix).

```
// ----------------------------------------
// Model Matrix
float modelMatrix[16];
translate(0.0f, 0.0f, 0.0f, modelMatrix);

// Get Model Matrix location
GLint modelLoc = glGetUniformLocation(program, "u_Model");

// Copy Rotation Matrix to Shader
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, modelMatrix);
// ----------------------------------------
```