# The Math Behind 3D Computer Graphics

NAT501
Group 50a

Martin Østergaard Villumsen
CPR: 040387-2687

Jesper Thingholm
CPR: 090282-1627

Tino Didriksen
CPR: 180982-2393

Peter Nielsen
CPR: 250291-3185

Henrik Edelmann
CPR: 100591-1083

Supervisor:
Rolf Fagerberg

Deliverables online at:
`http://tetris.pjj.cc/`

May 30, 2012

# Abstract

First we introduce the basics of 3D geometry and the applicable Cartesian and cylindrical coordinate systems. We explain that three points form a plane in 3D space and that points in 3D space are represented as vectors. We define that objects are built from such planes and that each object has its own coordinate system.

Then we explain the various operations one can do on points in 3D space, and why matrices are so very useful in this field. We also cover how operations described using matrices can naturally be combined to fewer operations. We then introduce the matrices and matrix math required to perform the translation, rotation, scaling, and projection operations. These operations are further detailed in turn, where we specify what each operation is and why it is relevant to 3D graphics.

We then cover how rendering works by going through the rendering equation. After that, we describe how lighting is calculated and used with the standard lighting model used in OpenGL. We further detail the contributions of lighting model - emissive, specular, diffuse, and ambient - and how they all come together. Lastly, we explain the standard light types and attenuation.

Towards the end we give a quick outline of the work put into the practical application - our 3D Tetris game - and what hurdles and quirks we observed and how we overcame them.

Finally, we conclude that while the math is academically interesting, there is not much practical need to know any of it when working on computer applications that utilize 3D graphics.

# Contents

# 1 Introduction

The field of 3D computer graphics is wide and varied, covering topics ranging from computer games through data visualization to medical imaging and wind or water simulations. We have chosen to focus on the parts of 3D graphics that are directly applicable to computer games, and use that knowledge to construct a concrete product, namely a 3D Tetris clone. But even such a drastically limited scope provides ample opportunity to learn a slew of new mathematical concepts and fresh applications of well known (very) old concepts.

Of key interest to anyone wanting to delve into the math behind 3D graphics will be linear algebra's staple diet of vectors and matrices, though limited to four dimensions. This may sound like one dimension more than most would expect to need, but there are some challenges in the math that cannot trivially be solved in three dimensions, so this is a case where adding another dimension actually simplifies the overall algebra.

Common to all modern 3D computer games is the need for shuffling around a large number of points in 3D space, which is where matrix operations come in handy. But one does not want to rely on brute force application of multiple matrix operations for every point - as that would stall even the most powerful graphics processors on the market - requiring math to step in to simplify these operations into fewer combined operations that yield the same end result.

But linear algebra is not the only old math to find modern use: Classical Euler angles are a natural part of rotations. And while we won't cover them, it is worth mentioning that quaternions, largely forgotten for over a hundred years, have had new life breathed into them because they solve certain tasks for 3D graphics even better than linear algebra or Euler angles. So fields of mathematics that are at least 150 years old power the core of all of our generation's favorite pass-time.

## 1.1 Goals and Objectives

We want to understand the math behind the process of rotating, scaling, translating (moving), projecting, and lighting objects in 3D space onto a 2D viewpoint.

To do this, we intend to produce a 3D Tetris clone with a cylindric playing field where the blocks have 360° movement around the edges of the cylinder.

The Tetris game itself will be made in Java using the JOGL OpenGL bindings maintained by Jogamp. If we have time left over, we plan on expanding the game by adding features such as textures, shadows, physics, block ghosting, transparency, hold block, etc.

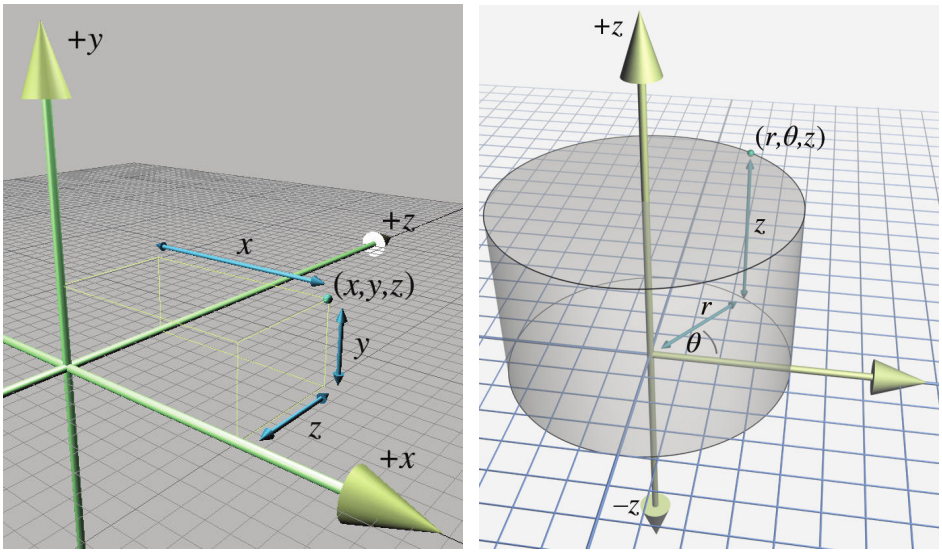
# 2 3D Geometry

## 2.1 Cartesian coordinates

A 2D Cartesian coordinate system consists of two axes; the $x$-axis and the $y$-axis, where one is perpendicular to the other.

When working in 3D Cartesian space, the 2D coordinate system is extended with a $z$-axis, which is perpendicular to the two other axes. This gives us three axes, where the $x$-axis is perpendicular to the $yz$-plane, the $y$-axis perpendicular to the $xz$-plane and the $z$ axis perpendicular to the $xy$-plane - the $x$-, $y$-, and $z$-axes being perpendicular to the three planes is known as the term mutually perpendicular.

When working with 3D graphics in the 3D Cartesian space, a point is specified by three numbers, $x$, $y$ and $z$, which are the signed distances to the $yz$, $xz$ and $xy$ planes, where the distance is measured along a line parallel to the given axis. Figure 1a gives an example of how to locate a point in the 3D Cartesian space.




(a) Locating a point in the 3D Cartesian space ([2] p. 14).

(b) Cylindrical coordinate system ([2] p. 203).

Figure 1: Cartesian & Cylindrical Coordinate Systems

## 2.2 Cylindrical coordinates

Cartesian coordinates is one way to represent points in 3D space - another way is to use cylindrical coordinates, which can be quite convenient when working in a cylinder-shaped environment or describing a cylinder-shaped object (i.e. a set of points), such as in our cylindrical 3D Tetris game.

Points represented by cylindrical coordinates are described by the distance to the origin $r$, the angle $\Theta$, and the height $z$. So instead of using $(x, y, z)$ we use $(r, \Theta, z)$, where the height $z$ is the same as in Cartesian coordinates. While being convenient in some situations, it is also natural to think in terms of distance and direction (for example, SDU lies 5 km. south-east of Odense City). An example of a cylindrical coordinate system is shown in Figure 1b.

## 2.3 Points as vectors

An important tool when working with 3D graphics are vectors, which are used to represent points in space, such as vertices of a triangle or the location of an object. Note that points and vectors are conceptually distinct, but mathematically equivalent.

By taking the cross product of two vectors, $\overrightarrow{p_1}$ and $\overrightarrow{p_2}$, you get a vector, $\overrightarrow{n}$, which is perpendicular to the plane spanned by $\overrightarrow{p_1}$ and $\overrightarrow{p_2}$. This vector is referred to as the normal vector $\overrightarrow{n}$. The normal vector is used to calculate lighting and reflections (see section 4).

## 2.4 Planes defined by 3 points

In 3D graphics all objects are (usually) built from triangles, where each triangle consists of three points, which are connected by three lines. A plane in 3D space is defined by three non-collinear (i.e. not on a line) points. No matter how each point is moved (as long they all are non-collinear) it will always be a plane. This would not be true if you worked with squares (i.e. four points), which is why all objects in 3D graphics are usually built from triangles.

## 2.5 Objects in 3D space

When operating on objects in 3D space it is important to perform the same operations on all planes that is part of that object. Objects are represented as vectors consisting of three sets of points, which allows us to apply an operation on all points in the object at the same time. An example of a 3D object in space is given in Figure 2.

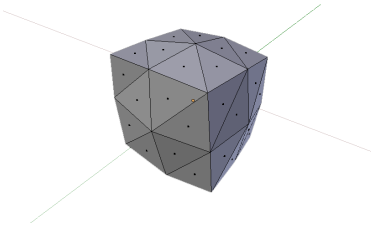


Figure 2: 3D object

For example, given three points, the plane spanned by these points would be represented as one vector $\overrightarrow{V}$:

$$\overrightarrow{v_1} := (x_1, y_1, z_1)$$
$$\overrightarrow{v_2} := (x_2, y_2, z_2)$$
$$\overrightarrow{v_3} := (x_3, y_3, z_3)$$
$$\overrightarrow{V} := (v_1, v_2, v_3)$$

Which makes sure all three points are moved equally if the plane $\overrightarrow{V}$ is moved.

### 2.6 Multiple coordinate systems

When working with games and graphics it is convenient to work with multiple coordinate systems. In theory, we can establish a coordinate space anywhere we want by picking an arbitrary point as the origin and decide how the axes should be oriented. However, in practice the coordinates of the origin is not an arbitrary chosen point, since coordinate spaces are created for specific reasons [1] p. 79-80.

The idea of multiple coordinate systems is to have one coordinate space for the camera, one for each object, and one world space. The world coordinate space establishes the global frame for everything and can express the position of all other coordinate spaces. When an object is moved or rotated the object space is moved/rotated along with it, and also changes orientation. Given is a real life example of the distinction between world space and an object space ([1] p. 83):

> When someone gives you driving directions, sometimes you will be told to "turn left" and other times you will be told to "go east." "Turn left" is a concept that is expressed in object space, and "go east" is expressed in world space.

Which gives a clear understanding of the distinction between world space and object space.

The camera space is actually an object space and is quite important. It is associated with the viewpoint used for rendering and the camera is placed at the origin in the camera space. Note that it is important to distinguish between the camera space (which is 3D) and the screen space (which is 2D) [1] p. 83-84. The mapping from camera space to screen space involves an operation known as projection (see section 3.7).

## 3 Operations on 3D points

In computer graphics, and the math behind it, it is necessary to be able to rotate, move, scale, etc., sets of points which lies within 3D space. This can be done with operations: An operation is a function that either moves, rotates, projects or applies another function on a set of points or even a single point. Our operations are represented as matrix functions. In the following chapter we will deduce the different operations for moving, rotating, scaling and projecting.

## 3.1 Why use matrices

The reason for using matrices is that they allow handling of multiple functions on different coordinates, and are therefore practical for doing computer graphics. This representation makes it possible to combine all the different operations into one matrix.

## 3.2 Combining operations

The matter in which one combines operations represented as matrices is by multiplying them. However, when dealing with matrices there are a few rules for multiplication: The first and most important part is the matter of dimensions, the rule is that the matrix on the right side of the multiply sign must have the same number of rows as the left side matrix has columns. The following example will show exactly how 2 matrices are multiplied: [1] p. 118.

$$\begin{pmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{pmatrix} \cdot \begin{pmatrix} d_1 & d_2 & d_3 \\ e_1 & e_2 & e_3 \\ f_1 & f_2 & f_3 \end{pmatrix}$$
$$= \begin{pmatrix} a_1d_1 + a_2e_1 + a_3f_1 & a_1d_2 + a_2e_2 + a_3f_2 & a_1d_3 + a_2e_3 + a_3f_3 \\ b_1d_1 + b_2e_1 + b_3f_1 & b_1d_2 + b_2e_2 + b_3f_2 & b_1d_3 + b_2e_3 + b_3f_3 \\ c_1d_1 + c_2e_1 + c_3f_1 & c_1d_2 + c_2e_2 + c_3f_2 & c_1d_3 + c_2e_3 + c_3f_3 \end{pmatrix}$$

The order of multiplication is also important when combining operations meaning that if you have 3 matrices $M$, $R$ and $F$, $(MR)F$ is not the same as $(FR)M$ [1] p. 120. This concludes how operations are combined.

## 3.3 Homogenous coordinates

[1] p. 176 Homogenous coordinates are represented as an extra coordinate $w$ on points in $n$-space. If we were to look at the 2D case a point would normally be written as $(x, y)$, as a homogenous coordinate set it is written as $(x, y, w)$. In 3D (2D homogenous space) space $w = 1$ represents a 2D plane for which all points can be projected to, by the following transformation $(x/w, y/w)$. If $w = 0$ the division is undefined and therefore defined as a direction vector towards a point at infinity and not transformed.

For any given point $(x, y)$ in the homogenous plane there are an infinite amount of corresponding points which can be written as $(kx, ky, k)$ given that $k \neq 0$. These points form a line through the (homogenous) origin. This can be expanded to the 3D case and by adding the homogenous coordinate it will become 4D. The projection transformation to a 3D point on the plane given by $w$'s value is then as follows $(x/w, y/w, z/w)$. One of the reasons for using the 4D case is explained in section 3.4.6. Another reason is that when using the proper value for $w$, a perspective projection is performed.

## 3.4 Translation

### 3.4.1 What is a translation

A translation is the operation of moving a point, or a set of points, by a given distance in a given direction.

In Cartesian coordinate space this can be done by altering the $x$-, $y$-, and $z$-values of the vector representing each point.

If all points of a set is moved in the same direction, by the same distance, their relative positions are unchanged. If e.g. two points both are translated by $\Delta x$ units in the $x$-direction, by $\Delta y$ in the $y$ direction, and by $\Delta z$ in the $z$-direction, then they will have the same position relative to each other.

### 3.4.2 Why is it relevant in 3D applications

In games and other 3D applications, translation is necessary to move objects and cameras around in the scene.

It is also necessary to temporarily move sets of points to the origin in order to perform other operations on them.

### 3.4.3 Translation by addition

One simple way of translating a point is by adding a vector consisting of the amounts of desired translation on each of the axes.

$$\overrightarrow{v} := (x, y, z)\,,\ \overrightarrow{t}_1 := (\Delta x, \Delta y, \Delta z)$$
$$\overrightarrow{v}' := \overrightarrow{v} + \overrightarrow{t}_1$$
$$\overrightarrow{v}' = (x + \Delta x, y + \Delta y, z + \Delta z)$$

### 3.4.4 Translation by addition is not a linear transformation

Translation by addition is not a linear transformation [1] p. 130. Therefore, it is not possible to translate a point by multiplying it with a 3x3 matrix.

This is because the result of the zero vector multiplied with a matrix will always result in the zero vector itself. This means that a point located on one of the axes cannot be moved away from that axis by use of translation.

$$(0,0,0)\begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix} = (0,0,0), \forall m_{ij} \in \mathbb{R}$$

### 3.4.5 Why translate by use of linear transformation

When performing operations on many points it is desirable to be able to concatenate as many functions as possible [1] p. 153. If e.g. a set of points has to be multiplied with a matrix $R \in \mathbb{R}^{3x3}$ and a matrix $S \in \mathbb{R}^{3x3}$ it requires fewer computations to first multiply $R$ and $S$, and then multiply the result with the points, rather than multiplying the points with $R$, and then multiplying the result with $S$, because $R$ and $S$ can be multiplied once for the entire set of points.

$$((x, y, z) \cdot R) \cdot S = (x, y, z) \cdot (RS)$$

If, on top of that, one wishes to translate the points by adding a vector, that operation cannot be concatenated with the rest.

$$((x, y, z) \cdot R) \cdot S + \overrightarrow{t} = (x, y, z) \cdot (RS) + \overrightarrow{t}$$

This gives a multiplication and an addition operation on each point. If, however, translation can be represented as a square matrix with dimensions equal to other transformation matrices, such as rotation, scaling, etc., then it too can be concatenated into one matrix leading to only one multiplication operation on each point.

### 3.4.6 4D translation matrix

This can be done by changing all transformation matrices into 4x4 matrices, and likewise change the points into 4D points by adding a fourth value, $w$ to the vector representation [1] p. 178.

$$(x, y, z, 1) \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \Delta x & \Delta y & \Delta z & 1 \end{pmatrix} = (x + \Delta x, y + \Delta y, z + \Delta z, 1)$$

Now, translation in 3D can be performed by multiplying points with matrices. All other transformation matrices also need to be changed into 4x4 since they have to be multiplied with a 4D vector. By doing this the translation operation can be concatenated with other transformations so that each point is accessed fewer times.

The translation matrix can be inversed. This will give a matrix that translates points the same distance, but in the opposite direction. The inverse of the translation matrix is:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \Delta x & \Delta y & \Delta z & 1 \end{pmatrix}^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\Delta x & -\Delta y & -\Delta z & 1 \end{pmatrix}$$

This is true, because the two matrices multiplied give the identity matrix.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \Delta x & \Delta y & \Delta z & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\Delta x & -\Delta y & -\Delta z & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

While this matrix translates the points in 3D, it is technically not a translation matrix. What the matrix does when multiplied with a set of 4D points is to shear the object in 4D [1] p. 179. This shearing of the four dimensional object moves the object's 3D hypersurface in 3D space.

Shearing is a transformation, that skews an object by dragging parts of it. This stretches the object in a non-uniform matter distorting angles [1] p. 152.

While four dimensional space is challenging to display, the concept can be illustrated by translating two-dimensional points.

$$v_1 := (0,0), v_2 := (1,0), v_3 := (1,1)$$

Add a third homogeneous coordinate to each vector:

$$v_1 \to (0,0,1), v_2 \to (1,0,1), v_3 \to (1,1,1)$$

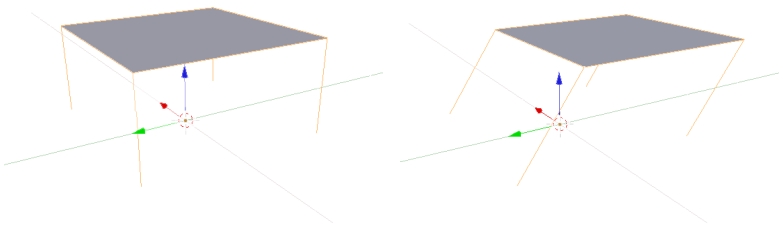
$$T := \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \Delta x & \Delta y & 1 \end{pmatrix}$$

$$v_1 \cdot T = (0 + \Delta x, 0 + \Delta y, 1)$$

$$v_2 \cdot T = (1 + \Delta x, 0 + \Delta y, 1)$$

$$v_3 \cdot T = (1 + \Delta x, 1 + \Delta y, 1)$$

The matrix $T$, which is capable of translating 2D points, can shear a 3D model, thereby translating its 2D surface.



(a) A 2D plane in its original position

(b) The plane translated by shearing 3D space

Figure 3: Translation by shearing

## 3.5 Rotation

Rotation is one of the cornerstones within computer graphics. It allows a rather simple handling of realistic movements, that otherwise would be very complex. Say e.g. you want to make a triangle in the $xy$-plane, viewed from the $z$-axis, and rotate it clockwise in 3D space. This could be done with a normal translation matrix, but would prove a challenge to get a smooth movement, as you would always have to go back and move the most recent points. This means you would have to design a function for translating that for the first application takes the original points, and moves them in the correct way for a clockwise rotation. After the first points, the function would then have to take its own result and move in a different manner, in order to maintain the position of the point that the triangle is rotating about. The rotation matrix can solve this problem, simply by using the original set of points and angles.

### 3.5.1 Rotation around cardinal axes

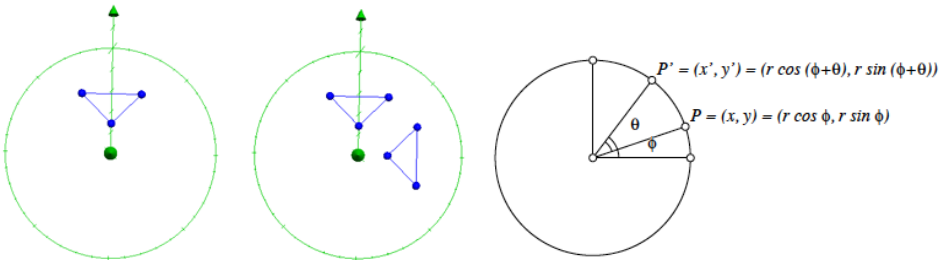
There are many different models for rotating points in 3D space. An important thing to understand, is that rotation is not only limited to computational graphics, but is a mathematical field. The first method of rotating points in 3D space we will explore, is rotation by rotating a point around the 3 cardinal axes with directions in our coordinate system:

$$x = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \; y = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \; z = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

The derivation of the rotational matrices [4] p.3

If we again look at our triangle as a single point $P$ in the $xy$-plane, looking down from a positive $z$-direction, the position of $P$ can be described as a polar coordinate set. This gives us a simple way to describe $P$ with a distance and an angle (see figure 4c). As we are in the $xy$-plane all the $z$-coordinates will remain the same throughout the rotation, hence we can write the coordinates without taking $z$ into account e.g. $P = (x, y)$. As we have a point in Cartesian coordinates we use the transformation to polar coordinate space given by the equations below. Our goal, however, is to find an equation which rotates Cartesian coordinates around the $z$-axis, as the figures 4a and 4b indicates.

$$x = r\cos(\theta), y = r\sin(\theta), r = \sqrt{x^2 + y^2}$$




(a) Non rotated triangle

(b) Rotated triangle

(c) Polar representation of $P$ and $P'$ [4] p. 3

Figure 4: Rotation examples

Since we want to find the equation for rotating $P$ in the plane, we will let $P' = (x', y')$ describe the rotated point which we wish to find. When polar coordinates are used any new point with the same $r$ can be found by addition or subtraction of another angle. Because we wish to find the equation for pure rotation, we give that $r$ is set and therefore:

$$P' = (x', y') = (r\cos(\theta + \Theta), r\sin(\theta + \Theta)$$

Using the general addition formulas for sine and cosine, the following derivation of the rotation equation can be done.

$$\begin{aligned}(x', y') &= (r\cos(\theta + \Theta), r\sin(\theta + \Theta)) \\ &= (r\cos(\theta)\cos(\Theta) - r\sin(\theta)\sin(\Theta), r\cos(\theta)\sin(\Theta) + r\sin(\theta)cos(\Theta)) \\ &= (x\cos(\Theta) - y\sin(\Theta), x\sin(\Theta) + y\cos(\Theta))\end{aligned}$$

This gives a manner of describing a rotation in the $xy$-plane as a coordinate set. However, as previously stated this is also the rotation about the $z$-axis with origin as the center of the circle. This gives the following equation, which is the foundation for the rotation matrix around the $z$-axis in 3D space.

$$R_{z,\Theta}(x, y, z) = (x\cos(\Theta) - y\sin(\Theta), x\sin(\Theta) + y\cos(\Theta), z)$$

If you wish to describe this formula as a matrix it would be as follows:

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

The following equations for rotation about $x$ and $y$ can derived in a similar manner:

$$\begin{aligned}R_x(x, y, z) &= (x, y\cos(\Theta) - z\sin(\Theta), y\sin(\Theta) + z\cos(\Theta)) \\ R_y(x, y, z) &= (x\cos(\Theta) + z\sin(\Theta), y, -x\sin(\Theta) + z\cos(\Theta))\end{aligned}$$

Taking the fourth dimension trick into account our three rotation matrices are:

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

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

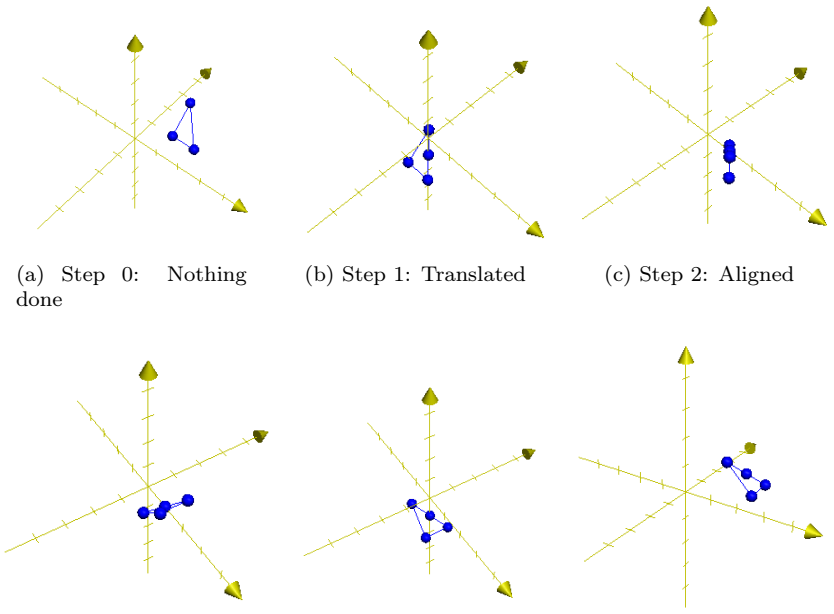
$$R_z = \begin{pmatrix} \cos(\Theta) & -\sin(\Theta) & 0 & 0 \\ \sin(\Theta) & \cos(\Theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

An important note to remember when applying these matrices is that they are derived within the polar coordinate system, and therefore all angles are presented as a function of $\pi$, so that 90 degrees will be $\frac{\pi}{2}$. Since our coordinate system is derived upon is a right-handed system, a positive angle gives an clockwise rotation.

### 3.5.2 Rotation in general

This entire section originates from [4] p. 4. If you were to rotate a figure any amount around a point that is not on one of the cardinal axes - henceforth, this will be referred to as the point being not aligned with one of the cardinal axes - you would have to align the point with one axis in order to rotate the figure in the desired manner. In the following explanations all operations made on the point, is also done on the figure so that their relative position remains the same.

The way one aligns a point with an axis is to rotate the point around the other axes so that the value of the other axes is zero. e.g $P = (x, 0, 0)$ is aligned with the x-axis. The method of aligning the point with one axis involves multiple operations and therefore we will need to use the combining of operations as previously discussed.



(a) Step 0: Nothing done

(b) Step 1: Translated

(c) Step 2: Aligned

(d) Step 3: Rotated

(e) Step 4: Unaligned

(f) Step 5: Translated back, rotation done

Figure 5: Rotation examples

Firstly, you would move the point to the origin, but since you cannot move the point and figure non-uniformly without distorting the figure, you move it so that the value of the axis you want to align the point with is 0.

Hereafter you rotate your point and figure around one of the other axes; when doing so, the value of that specific axis remains the same. Since you have chosen the 1st axis to align with, the only axis value left in the point to set to zero is the 3rd axis.

After this rotation is done you rotate the point and figure (points) around the 3rd axis so that the 2nd axis value of the point you align is zero. Remembering that the value of an axis does not change during a rotation about that axis, the 2nd axis value in the point remains 0.

So when those two steps are done, the point and thereby the figure is aligned with the 1st axis, and you can now rotate the desired amount around the aligned axis, to achieve the right rotation. However the point we wanted to rotate about is still placed on the 1st axis, and not where we wanted originally. In order to do so we will have to undo the rotations we did earlier and in the opposite order, so if we rotated in the following order: 3rd, 2nd, 1st (aligned rotation) the order of undoing rotations are to be: 2nd, 3rd.

If we look at our functions within our matrices they are all functions of sine and cosine and therefore the opposite rotation can be written as minus the angle used to do the rotation e.g. if the rotation angle is $\Theta$ the angle to undo the

rotation is $-\Theta$. The following example will show this connection:

$$\begin{aligned}\begin{pmatrix} x & y & z & 1 \end{pmatrix} \cdot R_x(\Theta) \cdot R_x(-\Theta) &= \begin{pmatrix} x & y & z & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} x & y & z & 1 \end{pmatrix}\end{aligned}$$

This shows that a rotation can be undone by using the same rotation matrix just with the negative angle - henceforth this will be known as the inverse rotation, written as $R_x^{-1}$. Therefore if we return to the general plane the order of rotations so far is: $T_{x,y,z} \cdot R_{3rd} \cdot R_{2nd} \cdot R_{1st}$.

In order to undo the rotations so that we can translate the figure back into the correct rotated position, we need to apply the inverse rotations in the order described above. So our new order of rotations becomes:

$$T_{x,y,z} \cdot R_{3rd} \cdot R_{2nd} \cdot R_{1st} \cdot R_{2nd}^{-1} \cdot R_{3rd}^{-1}$$

This undoes the rotations done to align the point with the 1st axis and the last step in the process is to undo the uniform movement done by using the translation matrix. The manner which you undo the movement is by using the inverse translation matrix $T^{-1}$. This moves our rotated figured back, so that the point which we wanted to rotate about is at its original position. Figure 5 is a step by step example of this process.

This is denoted as the point in a row vector form, and the matrices in the correct order which would look like the following equation. For this example the translation moves the point $P$ you want to rotate about to $x = 0$.

$$\begin{pmatrix} x & y & z & w \end{pmatrix} \cdot T \cdot R_z \cdot R_y \cdot R_x \cdot R_y^{-1} \cdot R_z^{-1} \cdot T^{-1}$$

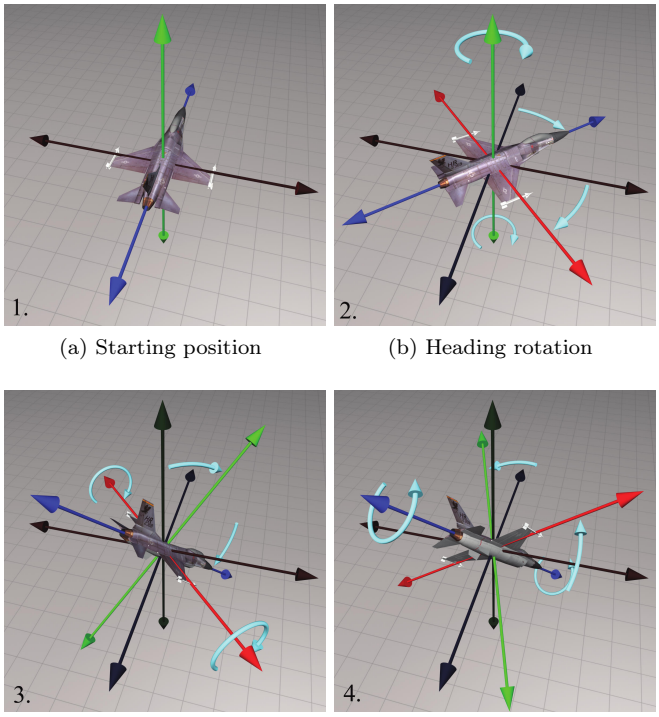
### 3.5.3 Euler angles

Euler angles is a different method of rotation: Simply put, each figure has its own coordinate system with axes along it self, and by rotating the figure around these 3 axes, any rotated point can be reached. The point you want to rotate the figure about is the origin for the figure's coordinate system and the axes are mutually perpendicular. See section 2.6.

There are multiple variations of terminologies for Euler angles: We will in this report go into the term, "heading-pitch-bank" as presented in [1] p. 230. The coordinate system for this terminology is based around a left handed coordinate system with the $y$-axis upwards, the $z$-axis going into the screen (forward), and the $x$-axis to the right. In this terminology, the rotation about the object's $y$-axis is a heading rotation, a rotation about the $x$-axis is a pitch rotation, and the $z$-axis gives a bank rotation.

It can be somewhat difficult to understand what the different rotations does, but if you picture an airplane, a heading rotation will change the way the aircraft points in the plane it is in. Pitch is then how much the nose points

upwards or downwards, where a positive angle will give a downwards rotation due to the left-handedness. Finally, bank controls how much the plane rolls to either side, where a positive angle gives a counterclockwise rotation (due to left-handedness).




(a) Starting position (b) Heading rotation

(c) Pitch rotation (d) Bank rotation

Figure 6: "Heading-Pitch-Bank" [1] p. 230-231

A problem with the Euler system is that every angle can be described with a very large number of combinations of angles. Therefore a conical system can be introduced in order to have every angle only described by 1 set of angles [1] p. 238 . The way one does this is by limiting the angles of heading to the interval $[-180; 180]$ degrees, bank and pitch to $[-90; 90]$ degrees. If pitch in this canonical system is $\pm 90°$, the bank angle is always 0.

As most computers calculate Euler angles in the opposite order of what we've listed them, this is also the matrix we will look at in this report. If the order of axis rotations to be applied is different, the matrices can be found here: [5].

We are interested in the bank ($z$), pitch ($x$) heading ($y$) order. Since the matrices are presented for a right handed system, all the signs of sinus functions are reversed. In order to save space the following notations will be used: $c$ will be $\cos(\Theta)$, the angle according to which axis will be noted as a subscript. The sine

function will likewise be denoted with a $s$. [5]

$$R = Z_1 X_2 Y_3$$
$$= \begin{pmatrix} c_1c_3 + s_1s_2s_3 & c_2s_1 & -c_1s_3 - c_3s_1s_2 \\ -c_3s_1 - c_1s_2s_3 & c_1c_2 & -s_1s_3 + c_1c_3s_2) \\ c_2s_3) & -s_2 & c_2c_3 \end{pmatrix}$$

These angles are not a function of $\pi$ and therefore any real number within the constraints earlier defined, in order to avoid duality, will work.

The advantages of using Euler angles is that they are very intuitive to use, and are therefore easily applied by humans. Another advantage is that they are very easily stored, due to the low amount of information needed compared to matrices. Most programming libraries have an built-in function that handles Euler angle rotations. This allows all rotations to be stored as a set of three numbers. [1] p. 241.

## 3.6 Scaling

Scaling is one of the simpler operations which can be performed when working with math behind 3D graphics. When scaling an object, the object is made larger or smaller by multiplying by a constant $k$. When $k > 1$ the object is augmented and when $0 < k < 1$ the object is diminished. If $k < 0$ the object is also reflected across the plane perpendicular to the given axis. If $k = -1$ for one of the coordinates, the size is not changed but the object is reflected along the axis [4] p. 4-5.

If $k = 0$ for one of the coordinates an orthographic projection is performed, which is elaborated in section 3.7.3.

By applying the constant to the entire object, a uniform scale is performed and the angles and proportions are preserved. By multiplying part of the object by a constant $k$, a nonuniform scale is performed, altering the proportions and eventually the angles of the object [1] p. 144.

### 3.6.1 Scaling along cardinal axes

In order to multiply each coordinate with a constant, a scaling matrix is used to perform the scaling. Such a matrix looks like this:

$$S(k_x, k_y, k_z) = \begin{pmatrix} k_x & 0 & 0 \\ 0 & k_y & 0 \\ 0 & 0 & k_z \end{pmatrix}$$

This matrix multiplies each coordinate with a constant. In order to leave one (or two) of the coordinates unchanged, (e.g. if you only want to scale the object in the $x$-direction and perform a nonuniform scale) the coordinate(s) you do not want to change is just multiplied by 1.

For reasons explained earlier (section 3.4.6), we use 4x4 matrices for all operations in 3D space, thus we expand the scaling matrix to a 4x4 matrix:

$$S(k_x, k_y, k_z) = \begin{pmatrix} k_x & 0 & 0 & 0 \\ 0 & k_y & 0 & 0 \\ 0 & 0 & k_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Note that when scaling by simply multiplying with a scaling matrix, the scaling performed is about the origin.

### 3.6.2 Scaling in general

In order to scale about the object instead of the origin, we can use the same trick as with rotation: Translate the object to the origin, apply the scaling, and translate the object back to the original position. This, however, only works when scaling along the cardinal axes. In order to scale in directions not aligned with the cardinal axes, we can use the same ideas previously used: Translate the object to the origin, align with one of the cardinal axes, apply the scaling, undo the alignment, and translate the object back to the original position.

As with translation and rotation, the scaling function also has an inverse function as long as all three values are non-zero. The inverse scaling function would not make much sense if $k = 0$ for one of the coordinates, since that basically would be a projection onto a 2D plane, which cannot be undone. The inverse scaling function is defined as follows [4] p. 5:

$$S^-1(k_x, k_y, k_z) = S\left(\frac{1}{k_x}, \frac{1}{k_y}, \frac{1}{k_z}\right)$$

Given a point $\overrightarrow{P} = \begin{pmatrix} x & y & z & 1 \end{pmatrix}$, we can show that applying a scaling matrix followed by the inverse scaling matrix results in the starting point:

$$\begin{aligned} \overrightarrow{P} \cdot S(k_x, k_y, k_z) \cdot S^-1(k_x, k_y, k_z) &= \overrightarrow{P} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ &= \overrightarrow{P} \end{aligned}$$

## 3.7 Projection

### 3.7.1 Intro to projection

Projection is the operation of mapping 3D points onto a 2-dimensional plane. This can e.g. be to project 3D points to the screen, or to some plane in the 3D world. The plane being projected to is called the projection plane [1] p. 184.

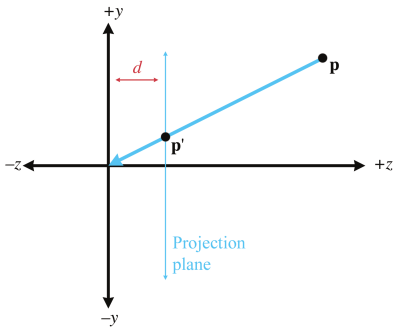
When projecting to the monitor, the operation of projection is used to determine where on the screen each vertex should be painted, and thereby, in turn also the color of each pixel on the screen.

Projections can be orthographic or perspective.

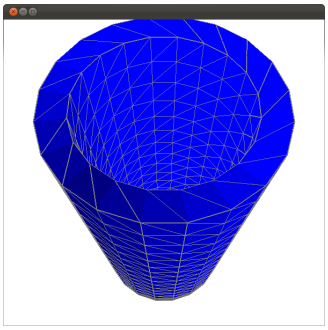
### 3.7.2 Perspective projection

Perspective projection is a form of projection, where the lines of projection all intersect in a single point, called the center of projection [1] p. 185. This results in objects further away from the projection plane look smaller, while objects closer to the plane look larger.

Perspective projection operates with the concept of field of view (fov), which is basically how wide the area being projected is. In 3D space there is a field of view for both the horizontal, and vertical axes on the projection plane. The zoom level is closely linked to the field of view [1] p. 367.




(a) The effect of perspective projection on the projection plane. Source: [1]



(b) Perspective projection

Figure 7: Perspective projection

### 3.7.3 Orthographic projection

An alternative way of projecting is orthographic projection. This form of projection contains no perspective, and therefore objects appear the same size regardless of how far they are from the projection plane. Furthermore lines that are parallel in 3D space will also be parallel in orthographic projection. The reason for this is the fact that the lines of projection are parallel, in contrast to perspective projection where they intersect in a single point [1] p. 368.

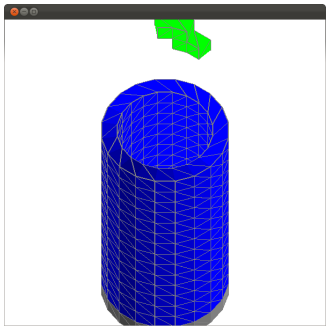


Figure 8: Orthographic projection

### 3.7.4 View frustum

The view frustum is a set of planes that define the space that is potentially visible to the camera. The view frustum is determined by six clip planes. Anything outside the view frustum is removed from the equation. This is also known as clipping [1] p. 451.

The far clip plane determines how far away the camera can see. This is done for two reasons. Firstly it limits the amount of vertices being rendered by clipping anything farther away. Secondly it is nessesary to determine how far vertices are from the camera. If $n$-bit numbers are used to represent the distance from the camera to the vertex, only $2^n$ different values can be stored. If e.g. integers are used then distances from zero to $2^n - 1$ can be stored, and any distance greater than $2^n - 1$ would would not be distinguishable from one another [1] p. 364-365.

The top, bottom, and sides of the frustum represent the top, bottom, and sides of the output plane. This means that all points located e.g. on the top clip plane will be projected onto the top of the projection plane, where it intersects with the top clip plane. This means that the top, bottom, and side planes determine the projection's field of view [1] p. 365.

In this way, those four planes determine the field of view, and zoom level. The top, and bottom planes determine the vertical field of view. Similarly the side planes determine the vertical field of view. By increasing the angle between two of those planes one can increase the field of view, either horizontally or vertically. Similarly, by decreasing the angle, the field of view is decreased.

The view frustum, that in perspective projection graphically can be interpreted as a frustum, is in orthographic projection a cube [2] p. 122.

The zoom level means something different in orthographic projection and perspective projection. Where in perspective projection zoom is related to the angle between the view frustum's clip planes, in orthographic projection it is directly related to the size of the view frustum [1] p. 369.

There is an inverse relationship between field of view, and zoom level [1] p. 367. When the field of view is increased, the zoom level is decreased. This is caused by the fact that increased field of view means that the space inside the view frustum, needed to be projected to the same plane is increased. Thereby there is less space to project each object onto.

The relationship between the zoom level and the field of view in perspective projection:

$$\text{zoom} = \frac{1}{\tan\left(\frac{\text{fov}}{2}\right)}$$

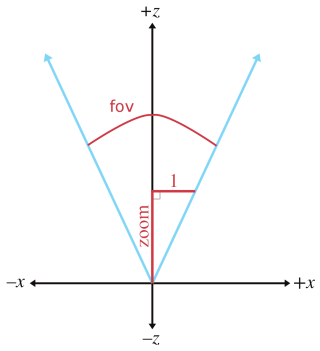



Figure 9: The relationship between zoom and field of view source: [1] p. 366

Below can be seen the relationship between the zoom level, and the size of the view frustum in orthographic projection [1] p. 369.

$$\text{zoom} = \frac{2}{\text{size}}$$

That gives one vertical zoom level, and one horizontal.

### 3.7.5 Coordinate Spaces and the clip matrix

The vertices of a model are, to begin with, represented in model space. In order to project them one needs to convert them into world space, and then into camera space. Camera space is a coordinate system where one axis points in the direction the camera is facing. This is typically the $z$-axis [1] p. 370.

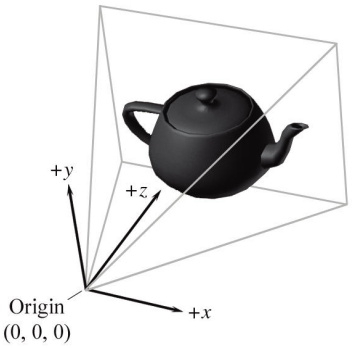



Figure 10: Camera space source: [1] p. 84

After the vertices are converted to camera space, they are converted into clip space, using the aptly named clip matrix. When using the clip matrix, the fourth homogenous coordinate comes into action. The correct values for the $x$- and $y$-values can be found by division with a correct $w$. For perspective projection the value to be divided by is equal to $z$ [1] p. 371.

$$(x, y, z, w)\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} = (x, y, z, z)$$

When projecting orthogonaly, the $w$ value is found using a sligtly different matrix multiplication.

$$(x, y, z, w)\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = (x, y, z, z)$$

In orthogonal projection the multiplication is with the identity matrix, and results in the vertex being unaltered. Geometricaly speaking this is because the $z$ value is not relevant for determining where on the projection plane, each vertex should be projected to.

The clip matrix also alters the vertices inside the view frustum, so that the view frustum, even for perspective projection geometrically becomes a cube [2] p. 122. In this way it is easier to check if vertices are outside the view frustum or not. Because of the differences in the view frustum between perspective and orthographic projection, there is a clip matrix for each type of projection.

The clip matrix for perspective projection looks as follows [1] p. 375:

$$\begin{pmatrix} \text{zoom}_x & 0 & 0 & 0 \\ 0 & \text{zoom}_y & 0 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & 1 \\ 0 & 0 & \frac{-2nf}{f-n} & 0 \end{pmatrix}$$

While the clip matrix for orthographic projection looks as follows [1] p. 376:

$$\begin{pmatrix} \text{zoom}_x & 0 & 0 & 0 \\ 0 & \text{zoom}_y & 0 & 0 \\ 0 & 0 & \frac{2}{f-n} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & 1 \end{pmatrix}$$

Where $\text{zoom}_x$ is the horizontal zoom value, $\text{zoom}_y$ is the vertical zoom value. $f$ and $n$ are the distances to the near and far clipping plane respectively. Different Application Programming Interfaces (API) use different conventions, e.g. column vectors instead of row vectors. This can result in matrices that look sligthly different, and the clip matrix is no exception [1] p. 375-377.

As a 4x4 matrix, the clip matrix can be concatenated with other transformation matrices, to reduce the number of calculations.

# 4 Lighting and Rendering

Rendering is the act of creating a picture where each pixel has the information of which color it is going to have with the use of computer programs. So for the computer to draw anything on the screen we need to perform rendering.

In the real world, we see each pixel as emitted or reflected light which the section on the standard lighting model will explain. It consist of ways to mathematically calculate the brightness and reflections to get the correct color of the light which is heading towards the camera.

In the latter part, we have light sources which make light without any emitting surfaces. This is done because it in some cases is more controllable and easier to just have light without having to build something emitting it. We will go over the standard light types and light attenuation.

## 4.1 Rendering

The basic idea about rendering is to specify what color of light is approaching the camera for that given pixel on the screen. We can be looking directly at a surface emitting light or light from a light source bounced from one or more places before going in the direction of the camera. Therefore we can simplify all this and say that we have to find the surface closest to the camera in the direction of each pixel and tell what light it is emitting or reflecting with the direction to that pixel.

One of the solutions is the use of raytracing, where we trace the rays backwards, meaning that we send out a ray from the middle of the given pixel and find the first object it strikes. We can then calculate the color that is being reflected or emitted from that point on the object and color the pixel.

Finding the color of the surface is rather easy if the surface emits its own light. Then the pixel will get the color of the light emitted without taking any reflections into account. Most of the time a surface will not emit its own light and we then have to find out what color the surface reflects in the direction of that pixel. The answer is given by the bidirectional reflectance distribution function (BRDF).

$f(x, \hat{w}_{in}, \hat{w}_{out}, \lambda)$

Source [1] p. 351. In this function $x$ is a position on the surface, $\hat{w}_{in}$ is the direction of the incoming light and $\hat{w}_{out}$ is the direction the light will be reflected. The hats means they are unit vectors. $\lambda$ is the color of the light, because each color have its own reflectance distribution.

Before going any further with the BRDF we need to specify another thing: Radiance. The eye can see 3 colors and then mix them together to create all the colors we can see. However, each color has a wavelength, and in physics light is considered energy in the form of electromagnetic radiation. The more radiation, the brighter the light. Lambert's law describes the strength of a ray, the radiance, and it is defined as radiant power per unit projected area, per unit solid angle. So both the power of the ray, the size of the area and incoming angle determines the radiance. Back to the BDRF we can now use it to determine the rendering equation, which will help us find what light is reflected on the surface into our eyes. The rendering equation is as following:

$$\begin{aligned} & L_{out}(x, \hat{w}_{out}, \lambda) \\ =& L_{emis}(x, \hat{w}_{out}, \lambda) + \int_{\Omega} L_{in}(x, \hat{w}_{in}, \lambda) f(x, \hat{w}_{in}, \hat{w}_{out}, \lambda)(-\hat{w}_{in} \cdot \hat{n}) d\hat{w}_{in} \end{aligned}$$

Source [1] p. 359. $x$ is a point on a surface and $\lambda$ is a single wavelength/color channel, so it only takes one color channel and one point on the surface at a time. On the left side of the equation: $L_{out}(x, \hat{w}_{out}, \lambda)$ is "the radiance leaving the point in the certain outgoing direction", which is the color we are going to paint the pixel.

On the right side we are going to split it up in two parts: $L_{emis}(x, \hat{w}_{out}, \lambda)$ is the radiance emitted from the point $x$ in the given outgoing direction.

The other part is a little more advanced: $\int_{\Omega} L_{in}(x, \hat{w}_{in}, \lambda) f(x, \hat{w}_{in}, \hat{w}_{out}, \lambda)(-\hat{w}_{in} \cdot \hat{n}) d\hat{w}_{in}$, but to simplify it, it means the radiance reflected from point $x$ in the certain direction.

We will now break it up to make it easier to understand: $\int_{\Omega}$ means the sum of all possible incoming directions [1] p. 360. Furthermore, we have $(x, \hat{w}_{in}, \lambda)$ which takes the radiance at the point from a certain direction. $f(x, \hat{w}_{in}, \hat{w}_{out}, \lambda)$ is the BRDF which tells us how much radiance from the direction before that will be reflected in the direction we are interested in.

Lastly, $(-\hat{w}_{in} \cdot \hat{n})d\hat{w}_{in}$ will account for the fact that if light hits perpendicular on a surface, it will reflect more than if it is hit with a glancing angle. The $\hat{n}$ is a vector which will at the most be 1 when the light hits perpendicular to the surface, and it will be lower the more glancing the angle is.

## 4.2 The standard lighting model

A lighting model is a formula that can express a lot of BRDFs by adjusting numbers for the different materials. A normal 3D game can contain several lighting models, but many older games only contain one lighting model. This lighting model is so basic that both OpenGL and DirectX have it hardwired into their rendering pipeline, and the Nintendo Wii also supports it.

The basic idea with the model is to split the light that comes into our eyes into four different kinds, where each kind is calculated differently. The four kinds are:

**The emissive contribution** called $c_{emis}$ describes the amount of radiance emitted from the surface in the given direction.

**The specular contribution** called $c_{spec}$ describes the light that hits a surface directly from a light source and is reflected perfectly into the eyes. A good example of this is a mirror.

**The diffuse contribution** called $c_{diff}$ describes the light that hits a surface directly from a light source and is divided evenly in every direction.

**The ambient contribution** called $c_{amb}$ is a factor that will light up an entire scene. It is used instead of light that will be reflected more than one time.

### 4.2.1 The emissive contribution

To calculate the emissive light, we just need two things which is the color and intensity of the light the surface emits. It is specified as:

$$c_{emis} = m_{emis}$$

The material's emissive color, $m_{spec}$, controls the intensity and color of the light and the value will be higher for more reflective surfaces. So the color of the emissive contribution is what the material emits.

### 4.2.2 The specular contribution

To get a better understanding of the math, we have inserted this picture:

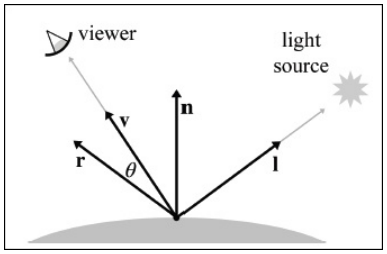



Figure 11: The Phong model for specular reflection. Source: [1]

Note we will stop using the hat to indicate vectors. In the picture, $n$ is the surface normal vector, $l$ is the unit vector with the direction of the light and $r$ is the unit vector of the reflected light created by the perfect bounce. $v$ is the unit vector with the direction to the eye and $\theta$ is the angle between $r$ and $v$.

The reflection vector, $r$, is computed like this:

$$r = 2(n \cdot l)n - l$$

Because we now know $r$ we can find the specular lighting by using the Phong model:

$$c_{spec} = (s_{spec} \otimes m_{spec})(cos\theta)^{m_{gls}} = (s_{spec} \otimes m_{spec})(v \cdot r)^{m_{gls}}$$

Source [1] p. 400. The product can't be negative. $\otimes$ means component-wise multiplication of colors. The $m_{gls}$ is the specular exponent and defines how large the hotspot is. The hotspot is e.g. the light circle on balls when the light bounces directly from a light bulb to your eyes. A smaller $m_{gls}$ produces at larger circle but less falloff then a high $m_{gls}$.

As in the emissive contribution the material's specular color, $m_{spec}$, controls the intensity and color. The $s_{spec}$ on the other hand is the intensity and color of the light. Both $m_{spec}$ and $m_{spec}$ is component-wise multiplied so both are taken into account when we compute the light.

An optimization of the Phong model has been found:

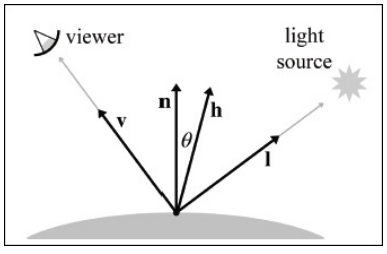



Figure 12: The Blinn model for specular reflection. Source: [1]

Here $r$ has been replaced by the halfway vector $h$: $h = \frac{v+l}{||v+l||}$. This way $r$ doesn't need to be calculated, and the specular light will therefore demand less processing power. The Blinn model therefore looks like this:

$$c_{spec} = (s_{spec} \otimes m_{spec})(cos\theta)^{m_{gls}} = (s_{spec} \otimes m_{spec})(n \cdot h)^{m_{gls}}$$

Source [1] p. 403.

### 4.2.3 The diffuse contribution

For diffused light, we don't need to know the viewer's location as all rays are divided in all directions. The direction of the light is however still important, since we know that if the light will hit the surface at a perpendicular angle then it will reflect more light. The diffused light is calculated like this:

$$c_{diff} = (s_{diff} \otimes m_{diff})(n \cdot l)$$

As in specular lighting, $n$ is the surface normal vector and $l$ is the unit vector pointing to the light. $s_{diff}$ is the color and intensity of the light and $m_{diff}$ is the color and intensity of the material. The product cannot be negative.

### 4.2.4 The ambient contribution

With ambient light we don't need any unit vector, camera or surface normal vector, because the light comes from everywhere and is reflected in all directions. The equation is as follows:

$$c_{amb} = g_{amb} \otimes m_{amb}$$

As before, $m_{amb}$ is the intensity and color of the material. $g_{amb}$ is the color and intensity of the light, and the $g$ stands for global because there is often only one value for an entire scene.

### 4.2.5 Putting it all together

When we put all together we have this:

$$\begin{aligned} c_{lit} =& c_{spec} + c_{diff} + c_{amb} + c_{emis} \\ =& (s_{spec} \otimes m_{spec}) max(n \cdot h, 0)^{m_{gls}} + \\ & (s_{diff} \otimes m_{diff}) max(n \cdot l, 0) + g_{amb} \otimes m_{amb} + m_{emis} \end{aligned}$$

Source [1] p. 407.
The max in $c_{spec}$ and $c_{diff}$ because none of them can be negative. In the following picture we can see all components in use except the emitting component:

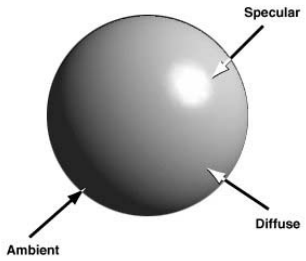



Figure 13: Ball with lighting effects. Source: [6]

The ambient lighting is covering the whole ball while the diffused is covering about half of it. Lastly we see that the specular is only using a small spot.

## 4.3 Light sources

This section will cover light sources without any emitting surfaces to make it easier and more controllable to use light.
First we are going to cover the standard light types which are supported by OpenGL and DirectX.

### 4.3.1 Standard light types

**A point light** is a light with a position and a color, including intensity that will emit light in all directions, and might have a falloff radius where the intensity of the light will reach 0.

**A spot light** on the other hand will have a position and send the light in a specified direction. They might also have a falloff distance.

- **A conical spot light** has a circle as bottom and the circle's peripheral is its falloff angle. Inside the falloff angle there is a hotspot angle, and the light inside it is more intense than outside it.
- **A rectangular spot light** forms a pyramid instead of a cone.

**A directional light** is light from a point so far away it can light up an entire scene. The sun uses this type.

Source [1] p. 414.

### 4.3.2 Light attenuation

Surfaces get less illumination from a light source the further away it is. To calculate this it is common to use a linear interpolation function like this:

$$f(n) = \begin{cases} 1 & \text{if } d \leq d_{min} \\ \frac{d_{max} - d}{d_{max} - d_{min}} & \text{if } d_{min} < d < d_{max} \\ 0 & \text{if } d \geq d_{max} \end{cases}$$

Source [1] p. 418.

$d$ is the distance. When the distance is smaller than $d_{min}$ the light is at full intensity and when the distance is larger than the $max$ it is zero light intensity. Between the $min$ and $max$ is a linear falloff that will make the intensity smaller the further away from $min$ and closer to $max$ that $d$ is. This function can be used with the standard light types to calculate the light in the falloff area.

# 5 Practical Application

As an example of a practical application, we chose to produce a Tetris clone. Since Tetris is fundamentally a 2D game, we had to come up with a way to utilize the 3rd dimension. One idea was to have the player fill out a cube, but that was rejected since it would get impossible to maintain any sort of overview of which spots were free or not. We instead went with a cylindrical shape where the player has to fill the outer wall and have full 360° movement.

## 5.1 Tech note

We chose to use OpenGL[7] via Java[8] for our application, using the JogAmp JOGL library[9]. This was the only real choice as we use a variety of operating systems and the majority of us had only programmed in Java. OpenGL is one of the major libraries[1] for communicating with the computer's graphics hardware; it abstracts away the vast majority of the complex calculations and mathematical details of getting 3D graphics onto the screen.

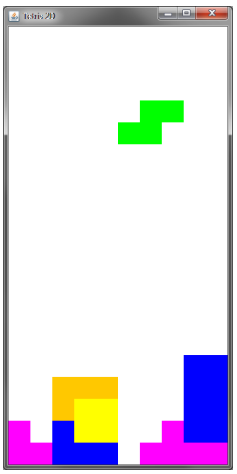


Figure 14: 2D reference implementation

## 5.2 Cylindrical Tetris

An interesting property of a cylinder wall is that we can model it exactly like a regular 2D Tetris game would work, and to simulate 360° movement we can allow the piece to move past the edges and appear on the other side.

[1] The other major contender is Microsoft's DirectX

This turned out to be problematic to handle, unfortunately, so we came up with a different model where instead of moving the Tetris piece left or right, we move the entire playing field right or left. To the player the effect is the same, but it is much easier to simulate.

But that merely reduced the core game to 2D, and even 2D Tetris has some tricky details. One of those is how to actually model removing a whole row which is made up of parts of several Tetris pieces, without mangling the pieces themselves. We chose to treat a Tetris piece as made up of individual colored square blocks. By also defining the playing field as a matrix of colored square blocks, placing a Tetris piece turned into a simple operation of copying the blocks of the piece onto the field, and removing a whole row no longer had to consider what a piece was.

So when should a piece be considered "placed"? When it can't move down any further is the obvious answer, but when is that? Tetris pieces are not uniform shapes, so detecting when a piece can't move any further in some direction required testing every block of the piece against the playing field. Some optimization could be done to test fewer blocks, but since the maximum number of collision tests per movement is 6, it was simply not worth it to complicate the code for such a tiny gain.

Rotating a piece required a similar collision test. Here we chose to actually perform the rotation, test whether the piece was colliding with the field, and then undo the rotation if it was. Since the vast majority of piece rotations are tried while there is plenty of space around the piece, this proved to be the most efficient way.

We produced a 2D Tetris game - playable at `http://tetris.pjj.cc/2d/` - using the above rules, including the simulated wrap-around of a cylinder, to get the bugs and behavior as we wanted it, then reused the same implementation for the 3D version.

## 5.3 Displaying in 3D

The first problem to hit the 3D implementation was the fact that the game rules are all about square blocks, but we want to display that as a cylinder wall. This required coming up with a way to curve the blocks, or approximate a curve. We chose to draw each block so that when viewed from above they are isosceles trapezoids with the large edge facing outwards.

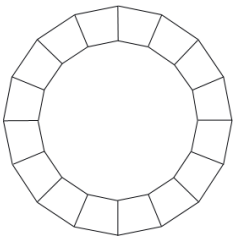
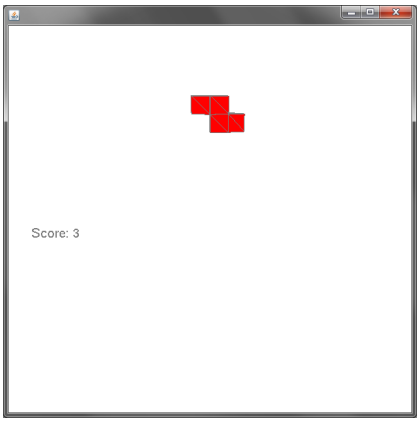
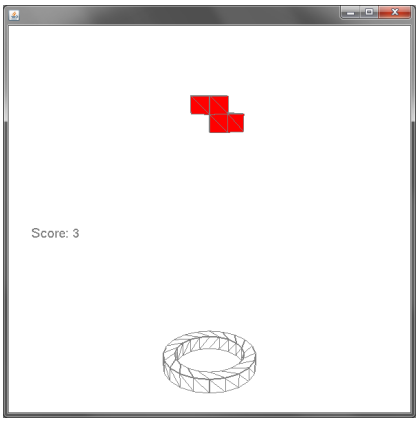


Figure 15: Ring of isosceles trapezoids

Another issue was that it was difficult to see where the bottom of the cylinder was. In the 2D version, the bottom is quite naturally the bottom of the window, but in 3D where the camera can move around and show the empty void from various directions, the concept of bottom loses meaning. To aid the player we added a ring of neutral blocks to denote where the bottom is.

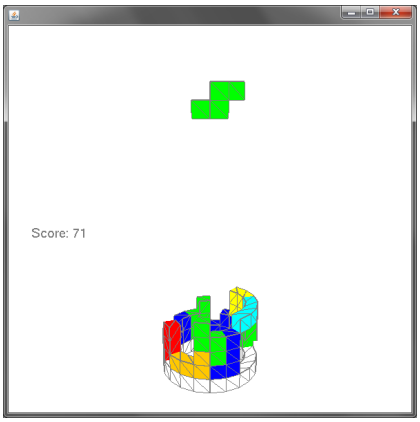



(a) Where is the bottom?

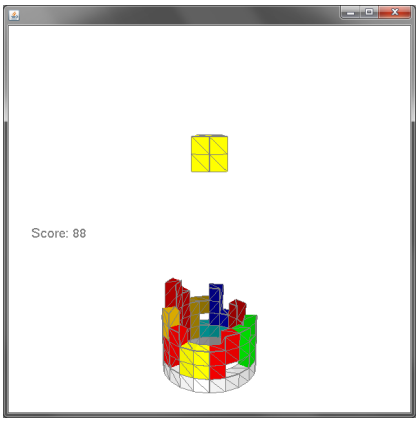



(b) There is the bottom.

We also found that the flat colors made it difficult to maintain an overview of which pieces were in front and which were in the back, so we added shading to dim the blocks in the background. This was done using a simple formula of lowering the color brightness the further from the center the block is. The effect is quite pronounced, especially in a moving game. It could have been done with proper lighting, but in computer science quick and dirty is sufficient in the vast majority of cases.




(c) Without shading




(d) Added shading

We experimented with showing the grid of the whole playing field, but this turned out to be rather confusing, so we dropped that.

The 3D version is playable at `http://tetris.pjj.cc/3d/`.

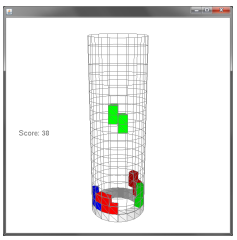


Figure 16: Visible grid

# 6 Conclusion

Through the work on the report, we learned a lot about the methods and theory behind 3D computer graphics, which are the underpinnings of modern popoular libraries such OpenGL and DirectX.

We feel that we have achieved everything we set out to do in our goals and objectives, although there are many more facets to the field of 3D graphics that we did not cover.

We produced a functional 3D Tetris game, and we found that knowing the math behind 3D graphics is not at all a requirement for making practical 3D applications. The programming libraries abstract away the vast majority of the math and lets one perform almost every operation by only giving the minimal amount of information.

# References


[1] Fletcher Dunn, Ian Parberry, 3D Math Primer for Graphics and Game Development, 2. edition, CRC Press.

[2] Eric Lengyel, Mathematics for 3D Game Programming & Computer Graphics, 2. edition, Charles River Media.

[3] http://www.land-of-kain.de/docs/jogl/

[4] Tom Davis, Homogeneous Coordinates and Computer Graphics, http://www.geometer.org/mathcircles/

[5] http://en.wikipedia.org/wiki/Euler_angles

[6] http://flylib.com/books/2/789/1/html/2/images/04fig01.jpg3

[7] http://khronos.org/opengl

[8] http://java.com/

[9] http://jogamp.org/jogl/