AD

Transformations

We now know how to create objects, color them and/or give them a detailed appearance using textures, but they're still not that interesting since they're all static objects. We could try and make them move by changing their vertices and re-configuring their buffers each frame, but that's cumbersome and costs quite some processing power. There are much better ways to transform an object and that's by using (multiple) matrix objects. This doesn't mean we're going to talk about Kung Fu and a large digital artificial world.

Matrices are very powerful mathematical constructs that seem scary at first, but once you'll grow accustomed to them they'll prove extremely useful. When discussing matrices, we'll have to make a small dive into some mathematics and for the more mathematically inclined readers I'll post additional resources for further reading.

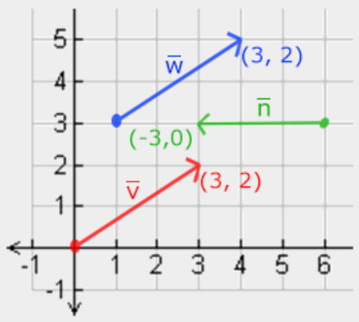
However, to fully understand transformations we first have to delve a bit deeper into vectors before discussing matrices. The focus of this chapter is to give you a basic mathematical background in topics we will require later on. If the subjects are difficult, try to understand them as much as you can and come back to this chapter later to review the concepts whenever you need them.

AD

Vectors

In its most basic definition, vectors are directions and nothing more. A vector has a direction and a magnitude (also known as its strength or length). You can think of vectors like directions on a treasure map: 'go left 10 steps, now go north 3 steps and go right 5 steps'; here 'left' is the direction and '10 steps' is the magnitude of the vector. The directions for the treasure map thus contains 3 vectors. Vectors can have any dimension, but we usually work with dimensions of 2 to 4. If a vector has 2 dimensions it represents a direction on a plane (think of 2D graphs) and when it has 3 dimensions it can represent any direction in a 3D world.

Below you'll see 3 vectors where each vector is represented with `(x,y)` as arrows in a 2D graph. Because it is more intuitive to display vectors in 2D (rather than 3D) you can think of the 2D vectors as 3D vectors with a `z` coordinate of `0`. Since vectors represent directions, the origin of the vector does not change its value. In the graph below we can see that the vectors $\bar{v}$ and $\bar{w}$ are equal even though their origin is different:



When describing vectors mathematicians generally prefer to describe vectors as character symbols with a little bar over their head like $\bar{v}$. Also, when displaying vectors in formulas they are generally displayed as follows:

$$\bar{v} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Because vectors are specified as directions it is sometimes hard to visualize them as positions. If we want to visualize vectors as positions we can imagine the origin of the direction vector to be `(0,0,0)` and then point towards a certain direction that specifies the point, making it a position vector (we could also specify a different origin and then say: 'this vector points to that point in space from this origin'). The position vector `(3,5)` would then point to `(3,5)` on the graph with an origin of `(0,0)`. Using vectors we can thus describe directions **and** positions in 2D and 3D space.

Just like with normal numbers we can also define several operations on vectors (some of which you've already seen).

Scalar vector operations

A scalar is a single digit. When adding/subtracting/multiplying or dividing a vector with a scalar we simply add/subtract/multiply or divide each element of the vector by the scalar. For addition it would look like this:

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} + x \rightarrow \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} + \begin{pmatrix} x \\ x \\ x \end{pmatrix} = \begin{pmatrix} 1+x \\ 2+x \\ 3+x \end{pmatrix}$$

Where $+$ can be $+,-,\cdot$ or $\div$ where $\cdot$ is the multiplication operator.

Vector negation

Negating a vector results in a vector in the reversed direction. A vector pointing north-east would point south-west after negation. To negate a vector we add a minus-sign to each component (you can also represent it as a scalar-vector multiplication with a scalar value of `-1`):

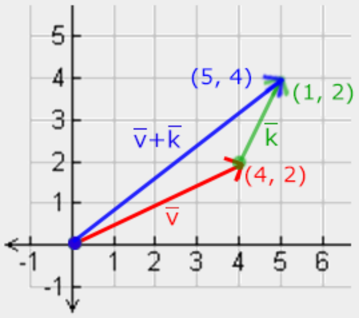
$$-\bar{v} = -\begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = \begin{pmatrix} -v_x \\ -v_y \\ -v_z \end{pmatrix}$$

Addition and subtraction

Addition of two vectors is defined as component-wise addition, that is each component of one vector is added to the same component of the other vector like so:

$$\bar{v} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \bar{k} = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} \rightarrow \bar{v} + \bar{k} = \begin{pmatrix} 1+4 \\ 2+5 \\ 3+6 \end{pmatrix} = \begin{pmatrix} 5 \\ 7 \\ 9 \end{pmatrix}$$

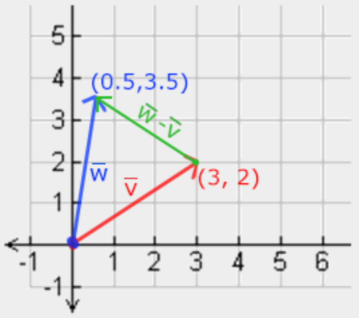
Visually, it looks like this on vectors `v=(4,2)` and `k=(1,2)`, where the second vector is added on top of the first vector's end to find the end point of the resulting vector (head-to-tail method):



Just like normal addition and subtraction, vector subtraction is the same as addition with a negated second vector:

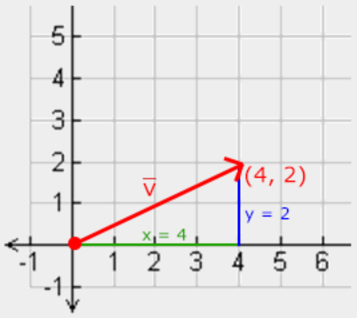
$$\bar{v} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \bar{k} = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} \rightarrow \bar{v} + -\bar{k} = \begin{pmatrix} 1+(-4) \\ 2+(-5) \\ 3+(-6) \end{pmatrix} = \begin{pmatrix} -3 \\ -3 \\ -3 \end{pmatrix}$$

Subtracting two vectors from each other results in a vector that's the difference of the positions both vectors are pointing at. This proves useful in certain cases where we need to retrieve a vector that's the difference between two points.



Length

To retrieve the length/magnitude of a vector we use the Pythagoras theorem that you may remember from your math classes. A vector forms a triangle when you visualize its individual `x` and `y` component as two sides of a triangle:



AD

known and we want to know the length of the tilted side $\bar{v}$ we
m as:

$$||\bar{v}|| = \sqrt{x^2 + y^2}$$

$||\bar{v}||$ r $\bar{v}$. This is easily extended to 3D by adding z^2 to the equation.

als:

$$||\bar{v}|| = \sqrt{4^2 + 2^2} = \sqrt{16 + 4} = \sqrt{20} = 4.47$$

Which is `4.47`.

There is also a special type of vector that we call a unit vector. A unit vector has one extra property and that is that its length is exactly 1. We can calculate a unit vector $\hat{n}$ from any vector by dividing each of the vector's components by its length:

$$\hat{n} = \frac{\bar{v}}{||\bar{v}||}$$

We call this normalizing a vector. Unit vectors are displayed with a little roof over their head and are generally easier to work with, especially when we only care about their directions (the direction does not change if we change a vector's length).

Vector-vector multiplication

Multiplying two vectors is a bit of a weird case. Normal multiplication isn't really defined on vectors since it has no visual meaning, but we have two specific cases that we could choose from when multiplying: one is the dot product denoted as $\bar{v} \cdot \bar{k}$ and the other is the cross product denoted as $\bar{v} \times \bar{k}$.

Dot product

The dot product of two vectors is equal to the scalar product of their lengths times the cosine of the angle between them. If this sounds confusing take a look at its formula:

$$\bar{v} \cdot \bar{k} = ||\bar{v}|| \cdot ||\bar{k}|| \cdot \cos\theta$$

Where the angle between them is represented as theta (θ). Why is this interesting? Well, imagine if $\bar{v}$ and $\bar{k}$ are unit vectors then their length would be equal to 1. This would effectively reduce the formula to:

$$\hat{v} \cdot \hat{k} = 1 \cdot 1 \cdot \cos\theta = \cos\theta$$

Now the dot product **only** defines the angle between both vectors. You may remember that the cosine or cos function becomes `0` when the angle is 90 degrees or `1` when the angle is 0. This allows us to easily test if the two vectors are orthogonal or parallel to each other using the dot product (orthogonal means the vectors are at a right-angle to each other). In case you want to know more about the `sin` or the `cos` functions I'd suggest the following Khan Academy videos about basic trigonometry.

You can also calculate the angle between two non-unit vectors, but then you'd have to divide the lengths of both vectors from the result to be left with $cos\theta$.

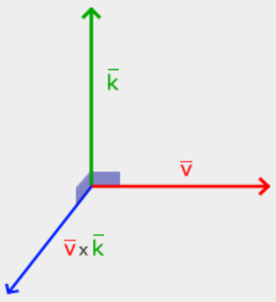
So how do we calculate the dot product? The dot product is a component-wise multiplication where we add the results together. It looks like this with two unit vectors (you can verify that both their lengths are exactly `1`):

$$\begin{pmatrix} 0.6 \\ -0.8 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = (0.6 * 0) + (-0.8 * 1) + (0 * 0) = -0.8$$

To calculate the degree between both these unit vectors we use the inverse of the cosine function cos^{-1} and this results in `143.1` degrees. We now effectively calculated the angle between these two vectors. The dot product proves very useful when doing lighting calculations later on.

Cross product

The cross product is only defined in 3D space and takes two non-parallel vectors as input and produces a third vector that is orthogonal to both the input vectors. If both the input vectors are orthogonal to each other as well, a cross product would result in 3 orthogonal vectors; this will prove useful in the upcoming chapters. The following image shows what this looks like in 3D space:



Unlike the other operations, the cross product isn't really intuitive without delving into linear algebra so it's best to just memorize the formula and you'll be fine (or don't, you'll probably be fine as well). Below you'll see the cross product between two orthogonal vectors A and B:

$$\begin{pmatrix} A_x \\ A_y \\ A_z \end{pmatrix} \times \begin{pmatrix} B_x \\ B_y \\ B_z \end{pmatrix} = \begin{pmatrix} A_y \cdot B_z - A_z \cdot B_y \\ A_z \cdot B_x - A_x \cdot B_z \\ A_x \cdot B_y - A_y \cdot B_x \end{pmatrix}$$

As you can see, it doesn't really seem to make sense. However, if you just follow these steps you'll get another vector that is orthogonal to your input vectors.

Matrices

Now that we've discussed almost all there is to vectors it is time to enter the matrix! A matrix is a rectangular array of numbers, symbols and/or mathematical expressions. Each individual item in a matrix is called an element of the matrix. An example of a 2x3 matrix is shown below:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Matrices are indexed by `(i,j)` where `i` is the row and `j` is the column, that is why the above matrix is called a 2x3 matrix (3 columns and 2 rows, also known as the dimensions of the matrix). This is the opposite of what you're used to when indexing 2D graphs as `(x,y)`. To retrieve the value 4 we would index it as `(2,1)` (second row, first column).

Matrices are basically nothing more than that, just rectangular arrays of mathematical expressions. They do have a very nice set of mathematical properties and just like vectors we can define several operations on matrices, namely: addition, subtraction and multiplication.

Addition and subtraction

Matrix addition and subtraction between two matrices is done on a per-element basis. So the same general rules apply that we're familiar with for normal numbers, but done on the elements of both matrices with the same index. This does mean that addition and subtraction is only defined for matrices of the same dimensions. A 3x2 matrix and a 2x3 matrix (or a 3x3 matrix and a 4x4 matrix) cannot be added or subtracted together. Let's see how matrix addition works on two 2x2 matrices:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1+5 & 2+6 \\ 3+7 & 4+8 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$$

The same rules apply for matrix subtraction:

$$\begin{bmatrix} 4 & 2 \\ 1 & 6 \end{bmatrix} - \begin{bmatrix} 2 & 4 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 4-2 & 2-4 \\ 1-0 & 6-1 \end{bmatrix} = \begin{bmatrix} 2 & -2 \\ 1 & 5 \end{bmatrix}$$

Matrix-scalar products

A matrix-scalar product multiples each element of the matrix by a scalar. The following example illustrates the multiplication:

$$2 \cdot \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 2 \cdot 1 & 2 \cdot 2 \\ 2 \cdot 3 & 2 \cdot 4 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

Now it also makes sense as to why those single numbers are called scalars. A scalar basically *scales* all the elements of the matrix by its value. In the previous example, all elements were scaled by 2.

So far so good, all of our cases weren't really too complicated. That is, until we start on matrix-matrix multiplication.

Matrix-matrix multiplication

Multiplying matrices is not necessarily complex, but rather difficult to get comfortable with. Matrix multiplication basically means to follow a set of pre-defined rules when multiplying. There are a few restrictions though:

1. You can only multiply two matrices if the number of columns on the left-hand side matrix is equal to the number of rows on the right-hand side matrix.
2. Matrix multiplication is not commutative that is $A \cdot B \neq B \cdot A$.

Let's get started with an example of a matrix multiplication of 2 `2x2` matrices:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1 \cdot 5 + 2 \cdot 7 & 1 \cdot 6 + 2 \cdot 8 \\ 3 \cdot 5 + 4 \cdot 7 & 3 \cdot 6 + 4 \cdot 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

Right now you're probably trying to figure out what the hell just happened? Matrix multiplication is a combination of normal multiplication and addition using the left-matrix's rows with the right-matrix's columns. Let's try discussing this with the following image:

$$\begin{bmatrix} \boxed{1 \quad 2} \\ \boxed{3 \quad 4} \end{bmatrix} \cdot \begin{bmatrix} \boxed{\begin{matrix} 5 \\ 7 \end{matrix}} & \boxed{\begin{matrix} 6 \\ 8 \end{matrix}} \end{bmatrix} = \begin{bmatrix} \boxed{1 \cdot 5 + 2 \cdot 7} & 1 \cdot 6 + 2 \cdot 8 \\ 3 \cdot 5 + 4 \cdot 7 & \boxed{3 \cdot 6 + 4 \cdot 8} \end{bmatrix} = \begin{bmatrix} \boxed{19} & 22 \\ 43 & \boxed{50} \end{bmatrix}$$

We first take the upper row of the left matrix and then take a column from the right matrix. The row and column that we picked decides which output value of the resulting `2x2` matrix we're going to calculate. If we take the first row of the left matrix the resulting value will end up in the first row of the result matrix, then we pick a column and if it's the first column the result value will end up in the first column of the result matrix. This is exactly the case of the red pathway. To calculate the bottom-right result we take the bottom row of the first matrix and the rightmost column of the second matrix.

To calculate the resulting value we multiply the first element of the row and column together using normal multiplication, we do the same for the second elements, third, fourth etc. The results of the individual multiplications are then summed up and we have our result. Now it also makes sense that one of the requirements is that the size of the left-matrix's columns and the right-matrix's rows are equal, otherwise we can't finish the operations!

The result is then a matrix that has dimensions of (`n,m`) where `n` is equal to the number of rows of the left-hand side matrix and `m` is equal to the columns of the right-hand side matrix.

Don't worry if you have difficulties imagining the multiplications inside your head. Just keep trying to do the calculations by hand and return to this page whenever you have difficulties. Over time, matrix multiplication becomes second nature to you.

Let's finish the discussion of matrix-matrix multiplication with a larger example. Try to visualize the pattern using the colors. As a useful exercise, see if you can come up with your own answer of the multiplication and then compare them with the resulting matrix (once you try to do a matrix multiplication by hand you'll quickly get the grasp of them).

$$\begin{bmatrix} 4 & 2 & 0 \\ 0 & 8 & 1 \\ 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 4 & 2 & 1 \\ 2 & 0 & 4 \\ 9 & 4 & 2 \end{bmatrix} = \begin{bmatrix} 4 \cdot 4 + 2 \cdot 2 + 0 \cdot 9 & 4 \cdot 2 + 2 \cdot 0 + 0 \cdot 4 & 4 \cdot 1 + 2 \cdot 4 + 0 \cdot 2 \\ 0 \cdot 4 + 8 \cdot 2 + 1 \cdot 9 & 0 \cdot 2 + 8 \cdot 0 + 1 \cdot 4 & 0 \cdot 1 + 8 \cdot 4 + 1 \cdot 2 \\ 0 \cdot 4 + 1 \cdot 2 + 0 \cdot 9 & 0 \cdot 2 + 1 \cdot 0 + 0 \cdot 4 & 0 \cdot 1 + 1 \cdot 4 + 0 \cdot 2 \end{bmatrix}$$
$$= \begin{bmatrix} 20 & 8 & 12 \\ 25 & 4 & 34 \\ 2 & 0 & 4 \end{bmatrix}$$

As you can see, matrix-matrix multiplication is quite a cumbersome process and very prone to errors (which is why we usually let computers do this) and this gets problematic real quick when the matrices become larger. If you're still thirsty for more and you're curious about some more of the mathematical properties of matrices I strongly suggest you take a look at these Khan Academy videos about matrices.

Anyways, now that we know how to multiply matrices together, we can start getting to the good stuff.

Matrix-Vector multiplication

Up until now we've had our fair share of vectors. We used them to represent positions, colors and even texture coordinates. Let's move a bit further down the rabbit hole and tell you that a vector is basically a `Nx1` matrix where `N` is the vector's number of components (also known as an N-dimensional vector). If you think about it, it makes a lot of sense. Vectors are just like matrices an array of numbers, but with only 1 column. So, how does this new piece of information help us? Well, if we have a `MxN` matrix we can multiply this matrix with our `Nx1` vector, since the columns of the matrix are equal to the number of rows of the vector, thus matrix multiplication is defined.

But why do we care if we can multiply matrices with a vector? Well, it just so happens that there are lots of interesting 2D/3D transformations we can place inside a matrix, and multiplying that matrix with a vector then *transforms* that vector. In case you're still a bit confused, let's start with a few examples and you'll soon see what we mean.

Identity matrix

In OpenGL we usually work with `4x4` transformation matrices for several reasons and one of them is that most of the vectors are of size 4. The most simple transformation matrix that we can think of is the identity matrix. The identity matrix is an `NxN` matrix with only 0s except on the diagonal. As you'll see, this transformation matrix leaves a vector completely unharmed:

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

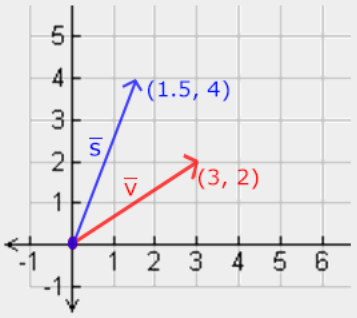
The vector is completely untouched. This becomes obvious from the rules of multiplication: the first result element is each individual element of the first row of the matrix multiplied with each element of the vector. Since each of the row's elements are 0 except the first one, we get: $1 \cdot 1 + 0 \cdot 2 + 0 \cdot 3 + 0 \cdot 4 = 1$ and the same applies for the other 3 elements of the vector.

You may be wondering what the use is of a transformation matrix that does not transform? The identity matrix is usually a starting point for generating other transformation matrices and if we dig even deeper into linear algebra, a very useful matrix for proving theorems and solving linear equations.

Scaling

When we're scaling a vector we are increasing the length of the arrow by the amount we'd like to scale, keeping its direction the same. Since we're working in either 2 or 3 dimensions we can define scaling by a vector of 2 or 3 scaling variables, each scaling one axis (`x`, `y` or `z`).

Let's try scaling the vector $\bar{v} = (3, 2)$. We will scale the vector along the x-axis by `0.5`, thus making it twice as narrow; and we'll scale the vector by `2` along the y-axis, making it twice as high. Let's see what it looks like if we scale the vector by `(0.5,2)` as $\bar{s}$:



Keep in mind that OpenGL usually operates in 3D space so for this 2D case we could set the z-axis scale to `1`, leaving it unharmed. The scaling operation we just performed is a non-uniform scale, because the scaling factor is not the same for each axis. If the scalar would be equal on all axes it would be called a uniform scale.

Let's start building a transformation matrix that does the scaling for us. We saw from the identity matrix that each of the diagonal elements were multiplied with its corresponding vector element. What if we were to change the `1`s in the identity matrix to `3`s? In that case, we would be multiplying each of the vector elements by a value of `3` and thus effectively uniformly scale the vector by 3. If we represent the scaling variables as (S_1, S_2, S_3) we can define a scaling matrix on any vector (x, y, z) as:

$$\begin{bmatrix} S_1 & 0 & 0 & 0 \\ 0 & S_2 & 0 & 0 \\ 0 & 0 & S_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} S_1 \cdot x \\ S_2 \cdot y \\ S_3 \cdot z \\ 1 \end{pmatrix}$$

Note that value 1. The w component is used for other purposes as we'll see later on.

Transla

Translation g another vector on top of the original vector to return a new vector with a
different p e vector based on a translation vector. We've already discussed vector
addition s ew.

Just like th re several locations on a 4-by-4 matrix that we can use to perform certain
operations se are the top-3 values of the 4th column. If we represent the translation
vector as (T_x, T_y, T_z) ine the translation matrix by:

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{pmatrix}$$

This works slation values are multiplied by the vector's w column and added to the
vector's or the matrix-multiplication rules). This wouldn't have been possible with a 3-
by-3 matri

Hom
The w is also known as a homogeneous coordinate. To get the 3D vector from a homogeneous vector we divide the x, y and z coordinate by its w coordinate. We usually do not notice this since the w component is 1.0 most of the time. Using homogeneous coordinates has several advantages: it allows us to do matrix translations on 3D vectors (without a w component we can't translate vectors) and in the next chapter we'll use the w value to create 3D perspective.

Also, whenever the homogeneous coordinate is equal to 0, the vector is specifically known as a direction vector since a vector with a w coordinate of 0 cannot be translated.

With a translation matrix we can move objects in any of the 3 axis directions (x, y, z), making it a very useful transformation matrix for our transformation toolkit.

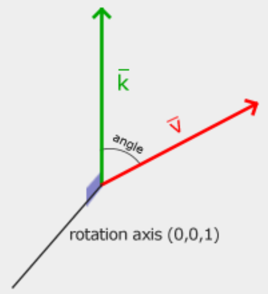
Rotation

The last few transformations were relatively easy to understand and visualize in 2D or 3D space, but rotations are a bit trickier. If you want to know exactly how these matrices are constructed I'd recommend that you watch the rotation items of Khan Academy's linear algebra videos.

First let's define what a rotation of a vector actually is. A rotation in 2D or 3D is represented with an angle. An angle could be in degrees or radians where a whole circle has 360 degrees or 2 PI radians. I prefer explaining rotations using degrees as we're generally more accustomed to them.

Most rotation functions require an angle in radians, but luckily degrees are easily converted to radians:
`angle in degrees = angle in radians * (180 / PI)`
`angle in radians = angle in degrees * (PI / 180)`
Where `PI` equals (rounded) `3.14159265359`.

Rotating half a circle rotates us 360/2 = 180 degrees and rotating 1/5th to the right means we rotate 360/5 = 72 degrees to the right. This is demonstrated for a basic 2D vector where $\bar{v}$ is rotated 72 degrees to the right, or clockwise, from $\bar{k}$:



Rotations in 3D are specified with an angle **and** a rotation axis. The angle specified will rotate the object along the rotation axis given. Try to visualize this by spinning your head a certain degree while continually looking down a single rotation axis. When rotating 2D vectors in a 3D world for example, we set the rotation axis to the z-axis (try to visualize this).

Using trigonometry it is possible to transform vectors to newly rotated vectors given an angle. This is usually done via a smart combination of the `sine` and `cosine` functions (commonly abbreviated to `sin` and `cos`). A discussion of how the rotation matrices are generated is out of the scope of this chapter.

A rotation matrix is defined for each unit axis in 3D space where the angle is represented as the theta symbol θ.

AD

Rotation around the X-axis:

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

Rotation around the Y-axis:

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

Rotation around the Z-axis:

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

Using the rotation matrices we can transform our position vectors around one of the three unit axes. To rotate around an arbitrary 3D axis we can combine all 3 them by first rotating around the X-axis, then Y and then Z for example. However, this quickly introduces a problem called Gimbal lock. We won't discuss the details, but a better solution is to rotate around an arbitrary unit axis e.g. `(0.662,0.2,0.722)` (note that this is a unit vector) right away instead of combining the rotation matrices. Such a (verbose) matrix exists and is given below with (R_x, R_y, R_z) as the arbitrary rotation axis:

$$\begin{bmatrix} \cos\theta + R_x{}^2(1-\cos\theta) & R_xR_y(1-\cos\theta) - R_z\sin\theta & R_xR_z(1-\cos\theta) + R_y\sin\theta & 0 \\ R_yR_x(1-\cos\theta) + R_z\sin\theta & \cos\theta + R_y{}^2(1-\cos\theta) & R_yR_z(1-\cos\theta) - R_x\sin\theta & 0 \\ R_zR_x(1-\cos\theta) - R_y\sin\theta & R_zR_y(1-\cos\theta) + R_x\sin\theta & \cos\theta + R_z{}^2(1-\cos\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A mathematical discussion of generating such a matrix is out of the scope of this chapter. Keep in mind that even this matrix does not completely prevent gimbal lock (although it gets a lot harder). To truly prevent Gimbal locks we have to represent rotations using quaternions, that are not only safer, but also more computationally friendly. However, a discussion of quaternions is out of this chapter's scope.

Combining matrices

The true power from using matrices for transformations is that we can combine multiple transformations in a single matrix thanks to matrix-matrix multiplication. Let's see if we can generate a transformation matrix that combines several transformations. Say we have a vector `(x,y,z)` and we want to scale it by 2 and then translate it by `(1,2,3)`. We need a translation and a scaling matrix for our required steps. The resulting transformation matrix would then look like:

$$Trans.Scale = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 2 & 0 & 2 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Note that we first do a translation and then a scale transformation when multiplying matrices. Matrix multiplication is not commutative, which means their order is important. When multiplying matrices the right-most matrix is first multiplied with the vector so you should read the multiplications from right to left. It is advised to first do scaling operations, then rotations and lastly translations when combining matrices otherwise they may (negatively) affect each other. For example, if you would first do a translation and then scale, the translation vector would also scale!

Running the final transformation matrix on our vector results in the following vector:

$$\begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 2 & 0 & 2 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 2x+1 \\ 2y+2 \\ 2z+3 \\ 1 \end{bmatrix}$$

Great! The vector is first scaled by two and then translated by `(1,2,3)`.

In practice

Now that we've explained all the theory behind transformations, it's time to see how we can actually use this knowledge to our advantage. OpenGL does not have any form of matrix or vector knowledge built in, so we have to define our own mathematics classes and functions. In this book we'd rather abstract from all the tiny mathematical details and simply use pre-made mathematics libraries. Luckily, there is an easy-to-use and tailored-for-OpenGL mathematics library called GLM.

GLM

GLM stands for Open**GL** **M**athematics and is a *header-only* library, which means that we only have to include the proper header files and we're done; no linking and compiling necessary. GLM can be downloaded from their website. Copy the root directory of the header files into your *includes* folder and let's get rolling.

Most of GLM's functionality that we need can be found in 3 headers files that we'll include as follows:

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
```

Let's see if we can put our transformation knowledge to good use by translating a vector of `(1,0,0)` by `(1,1,0)` (note that we define it as a `glm::vec4` with its homogeneous coordinate set to `1.0`:

```
glm::vec4 vec(1.0f, 0.0f, 0.0f, 1.0f);
glm::mat4 trans = glm::mat4(1.0f);
trans = glm::translate(trans, glm::vec3(1.0f, 1.0f, 0.0f));
vec = trans * vec;
std::cout << vec.x << vec.y << vec.z << std::endl;
```

We first define a vector named `vec` using GLM's built-in vector class. Next we define a `mat4` and explicitly initialize it to the identity matrix by initializing the matrix's diagonals to `1.0`; if we do not initialize it to the identity matrix the matrix would be a null matrix (all elements `0`) and all subsequent matrix operations would end up a null matrix as well.

The next step is to create a transformation matrix by passing our identity matrix to the `glm::translate` function, together with a translation vector (the given matrix is then multiplied with a translation matrix and the resulting matrix is returned).
Then we multiply our vector by the transformation matrix and output the result. If we still remember how matrix translation works then the resulting vector should be `(1+1,0+1,0+0)` which is `(2,1,0)`. This snippet of code outputs `210` so the translation matrix did its job.

Let's do something more interesting and scale and rotate the container object from the previous chapter:

```
glm::mat4 trans = glm::mat4(1.0f);
trans = glm::rotate(trans, glm::radians(90.0f), glm::vec3(0.0, 0.0, 1.0));
trans = glm::scale(trans, glm::vec3(0.5, 0.5, 0.5));
```

First we scale the container by `0.5` on each axis and then rotate the container `90` degrees around the Z-axis. GLM expects its angles in radians so we convert the degrees to radians using `glm::radians`. Note that the textured rectangle is on the XY plane so we want to rotate around the Z-axis. Keep in mind that the axis that we rotate around should be a unit vector, so be sure to normalize the vector first if you're not rotating around the X, Y, or Z axis. Because we pass the matrix to each of GLM's functions, GLM automatically multiples the matrices together, resulting in a transformation matrix that combines all the transformations.

The next big question is: how do we get the transformation matrix to the shaders? We shortly mentioned before that GLSL also has a `mat4` type. So we'll adapt the vertex shader to accept a `mat4` uniform variable and multiply the position vector by the matrix uniform:

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec2 aTexCoord;

out vec2 TexCoord;

uniform mat4 transform;

void main()
{
    gl_Position = transform * vec4(aPos, 1.0f);
    TexCoord = vec2(aTexCoord.x, aTexCoord.y);
}
```

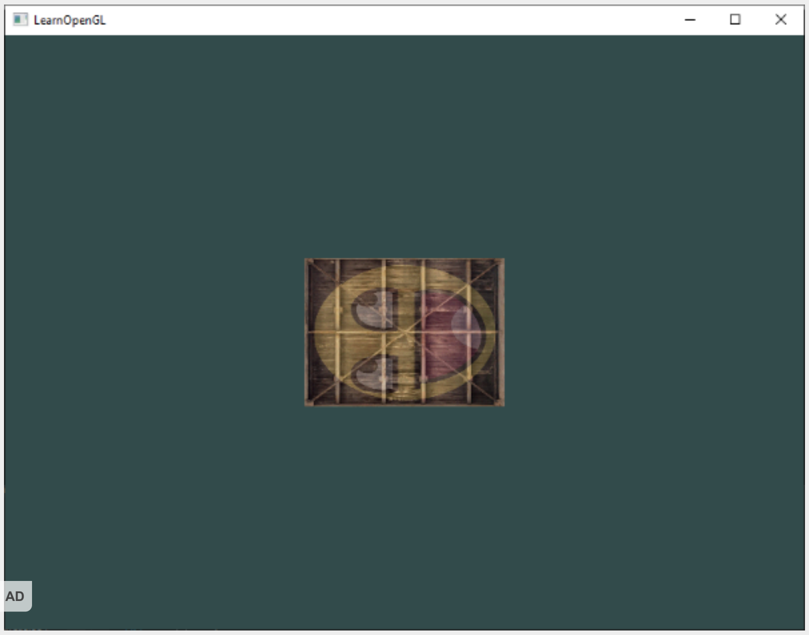
GLSL also has `mat2` and `mat3` types that allow for swizzling-like operations just like vectors. All the aforementioned math operations (like scalar-matrix multiplication, matrix-vector multiplication and matrix-matrix multiplication) are allowed on the matrix types. Wherever special matrix operations are used we'll be sure to explain what's happening.

We added the uniform and multiplied the position vector with the transformation matrix before passing it to `gl_Position`. Our container should now be twice as small and rotated `90` degrees (tilted to the left). We still need to pass the transformation matrix to the shader though:

```
unsigned int transformLoc = glGetUniformLocation(ourShader.ID, "transform");
glUniformMatrix4fv(transformLoc, 1, GL_FALSE, glm::value_ptr(trans));
```

We first query the location of the uniform variable and then send the matrix data to the shaders using `glUniform` with `Matrix4fv` as its postfix. The first argument should be familiar by now which is the uniform's location. The second argument tells OpenGL how many matrices we'd like to send, which is `1`. The third argument asks us if we want to transpose our matrix, that is to swap the columns and rows. OpenGL developers often use an internal matrix layout called column-major ordering which is the default matrix layout in GLM so there is no need to transpose the matrices; we can keep it at `GL_FALSE`. The last parameter is the actual matrix data, but GLM stores their matrices' data in a way that doesn't always match OpenGL's expectations so we first convert the data with GLM's built-in function `value_ptr`.

We created a transformation matrix, declared a uniform in the vertex shader and sent the matrix to the shaders where we transform our vertex coordinates. The result should look something like this:



Perfect! Our container is indeed tilted to the left and twice as small so the transformation was successful. Let's get a little more funky and see if we can rotate the container over time, and for fun we'll also reposition the container at the bottom-right side of the window. To rotate the container over time we have to update the transforma er loop because it needs to update each frame. We use GLFW's time function to get an a

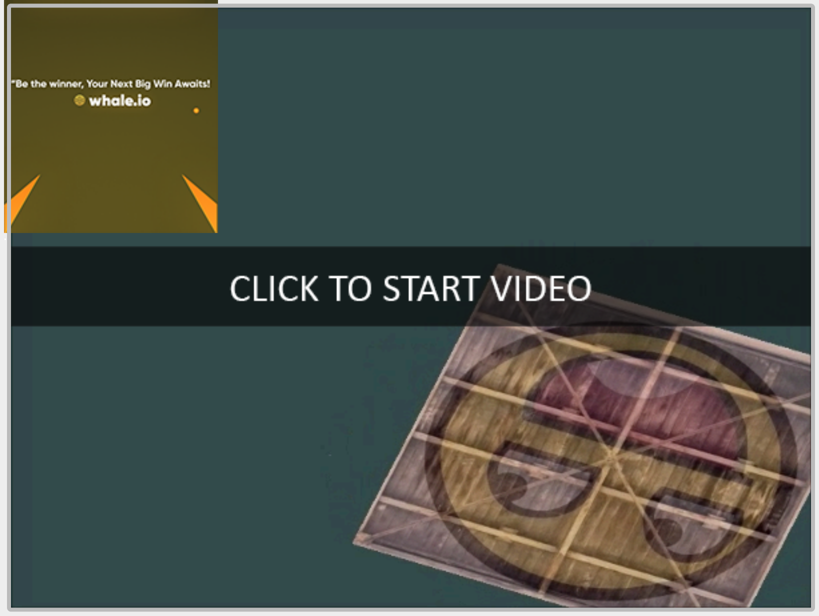
```
glm::ma                 4(1.0f);
trans =                 ns, glm::vec3(0.5f, -0.5f, 0.0f));
trans =                 (float)glfwGetTime(), glm::vec3(0.0f, 0.0f, 1.0f));
```

@Whale
Community Driven Gambling
Trusted by Millions...
Trustpilot
Resume Play
"Be the winner, Your Next Big Win Awaits!
whale.io

Keep in mi case we could declare the transformation matrix anywhere, but now we have to create it nuously update the rotation. This means we have to re-create the transforma ation of the render loop. Usually when rendering scenes we have several transforma re-created with new values each frame.

Here we fi around the origin `(0,0,0)` and once it's rotated, we translate its rotated version to of the screen. Remember that the actual transformation order should be read in rev de we first translate and then later rotate, the actual transformations first apply a ro ation. Understanding all these combinations of transformations and how they apply understand. Try and experiment with transformations like these and you'll quickly get

If you did t get the following result:



And there you have it. A translated container that's rotated over time, all done by a single transformation matrix! Now you can see why matrices are such a powerful construct in graphics land. We can define an infinite amount of transformations and combine them all in a single matrix that we can re-use as often as we'd like. Using transformations like this in the vertex shader saves us the effort of re-defining the vertex data and saves us some processing time as well, since we don't have to re-send our data all the time (which is quite slow); all we need to do is update the transformation uniform.

If you didn't get the right result or you're stuck somewhere else, take a look at the source code and the updated shader class.

In the next chapter we'll discuss how we can use matrices to define different coordinate spaces for our vertices. This will be our first step into 3D graphics!

Further reading

- Essence of Linear Algebra: great video tutorial series by Grant Sanderson about the underlying mathematics of transformations and linear algebra.
- Matrix Multiplication XYZ: check out this amazing interactive visual tool for showcasing matrix multiplication. Trying a few of these should help solidify your understanding.

Exercises

- Using the last transformation on the container, try switching the order around by first rotating and then translating. See what happens and try to reason why this happens: solution.
- Try drawing a second container with another call to `glDrawElements` but place it at a different position using transformations **only**. Make sure this second container is placed at the top-left of the window and instead of rotating, scale it over time (using the `sin` function is useful here; note that using `sin` will cause the object to invert as soon as a negative scale is applied): solution.