Notes related to OpenGL and graphics programming in general. Primary source for now is learnopengl.com

CLion things, markdown shortcuts etc

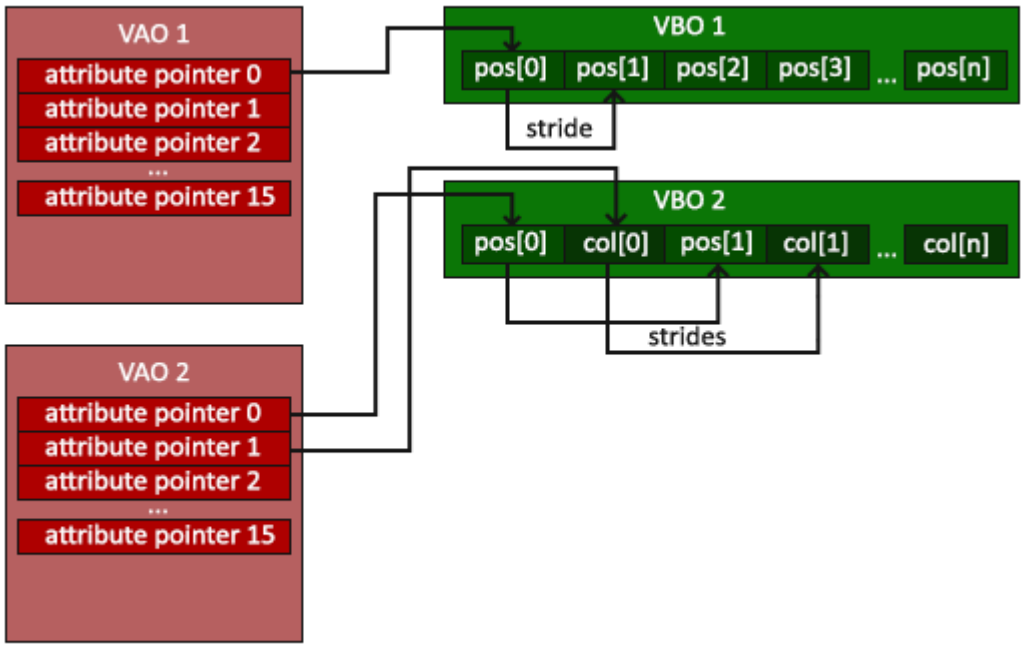
`Ctrl + B:` Bold
`Code (like this):` Ctrl + Shift + C
`Insert image:` Ctrl + U
`Insert link:` Ctrl + Shift + U
TeX and LaTeX syntax math supported with dollarsigns

To get relative paths to work: run -> edit configurations and set the working directory to project root

Vertex Array Objects (VAO)

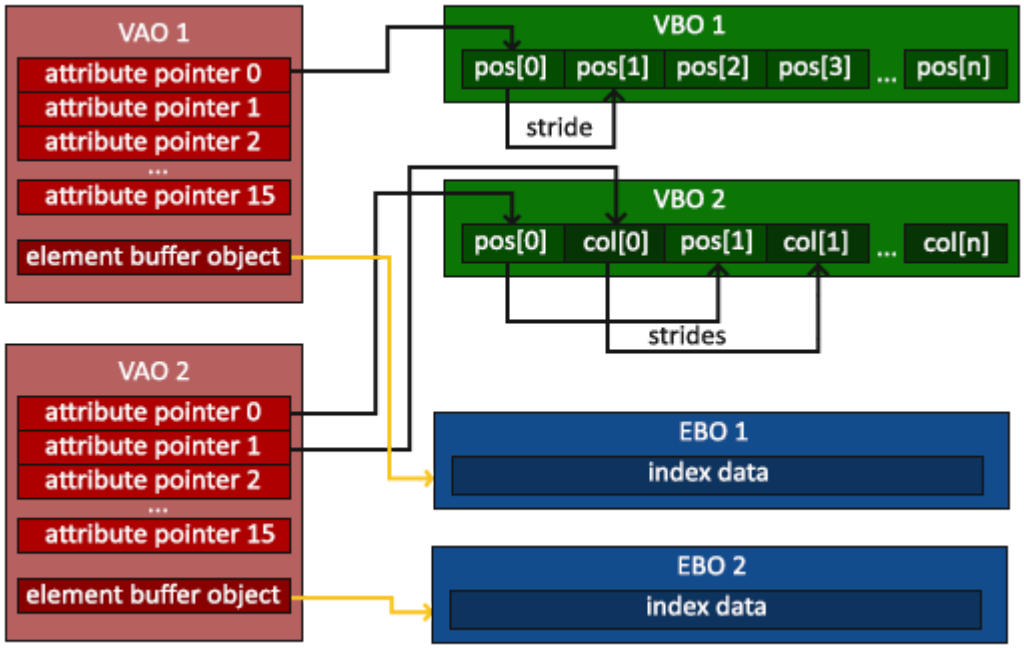
To use a VAO all you have to do is bind the VAO using `glBindVertexArray`. From that point on we should bind/configure the corresponding VBO(s) and attribute pointer(s) and then unbind the VAO for later use.

VAO stores our vertex attribute configuration and which VBO to use. Usually when you have multiple objects you want to draw, you first generate/configure all the VAOs (and thus the required VBO and attribute pointers) and store those for later use. The moment we want to draw one of our objects, we take the corresponding VAO, bind it, then draw the object and unbind the VAO again.



Element Buffer Objects (EBO)

Indexed drawing, using indices and only need to specify unique vertices.
The last element buffer object that gets bound while a VAO is bound is stored as the VAO's element buffer object. Binding to a VAO then also automatically binds that EBO.



Rasterization and Fragment Interpolation

During rasterization, the graphics pipeline converts vector graphics (triangles) into a raster image (pixels/fragments).

The rasterization stage usually results in a lot more fragments than vertices specified. The rasterizer determines the position of those fragments based on where they reside on the triangle. Based on this position, it interpolates all the fragment shader's input variables.

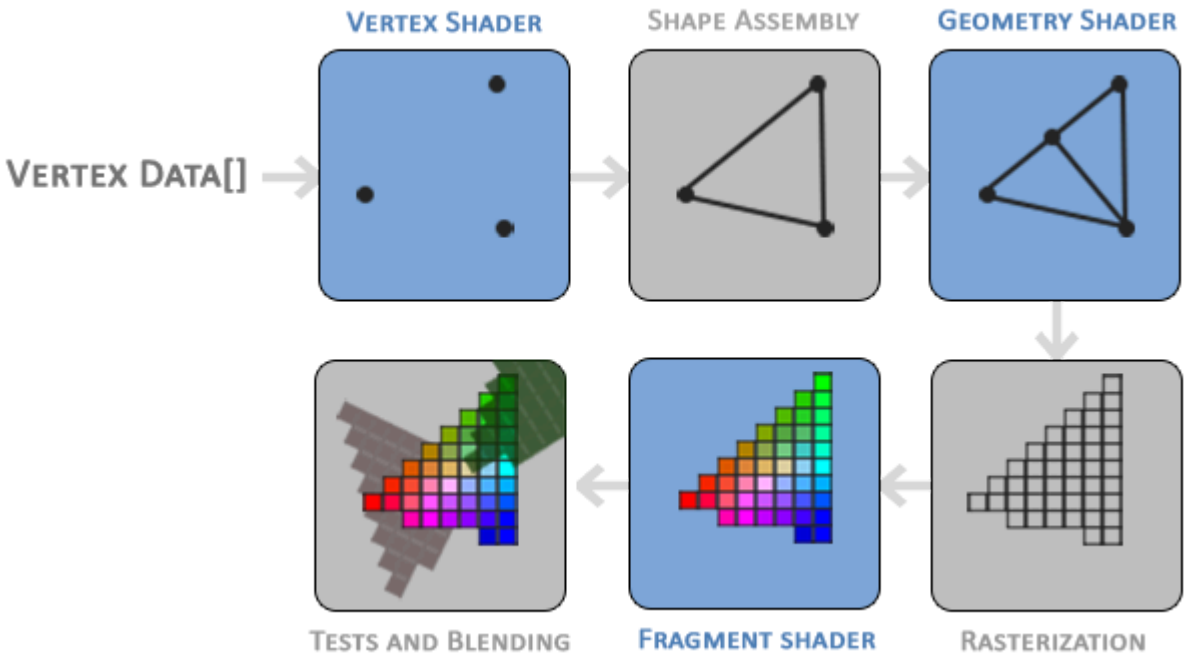
Interpolation is the act of calculating smooth, intermediate values (data points) between known points based on their relative positions. In graphics, it involves determining the value of a specific attribute (like color, texture coordinates, or normals) for each fragment within a primitive by mathematically blending the values from the primitive's vertices

Interpolation in OpenGL occurs after the vertex shader and before the fragment shader, during the rasterization stage. When you pass data from the vertex shader to the fragment shader, like texture coordinates, OpenGL performs interpolation based on the barycentric coordinates of the fragments.

determines the pixels covered by a primitive (e.g. a triangle) and interpolates the output parameters of the vertex shader

When OpenGL rasterizes a triangle, it divides the triangle into fragments (potential pixels), and for each fragment, it needs to interpolate the vertex attributes (like texture coordinates or colors) between the three vertices of the triangle

For example:
We have a line where the upper point has a green color and the lower point a blue color. If the fragment shader is run at a fragment that resides around a position at 70% of the line, its resulting color input attribute would then be a linear combination of green and blue. To be more precise: 30% blue and 70% green.



Geometry shader is completely optional. Tesselation is also optional.

Sampling

Retrieving texture color by using interpolated texture coordinates. In 2D texture images, texture coordinates range from 0 to 1 in both the y and x-axis, with `(0, 0)` being the bottom-left corner.
Pass 3 texture coordinates as a vertex attribute to the vertex shader. The output of these then get interpolated (if you dont change something with interpolation qualifier) during rasterization and the fragment shader receives the interpolated texture coordinates.

GLSL has built-in function **texture** function to sample a color from a texture. Takes a texture sampler (like sampler2D) as first argument and corresponding texture coordinates as second.

Example:

```
void main() {
    FragColor = texture(ourTexture, TexCoord);
}
```

Texture Wrapping

- `s` , `t` **axis = x, y (width, height)**

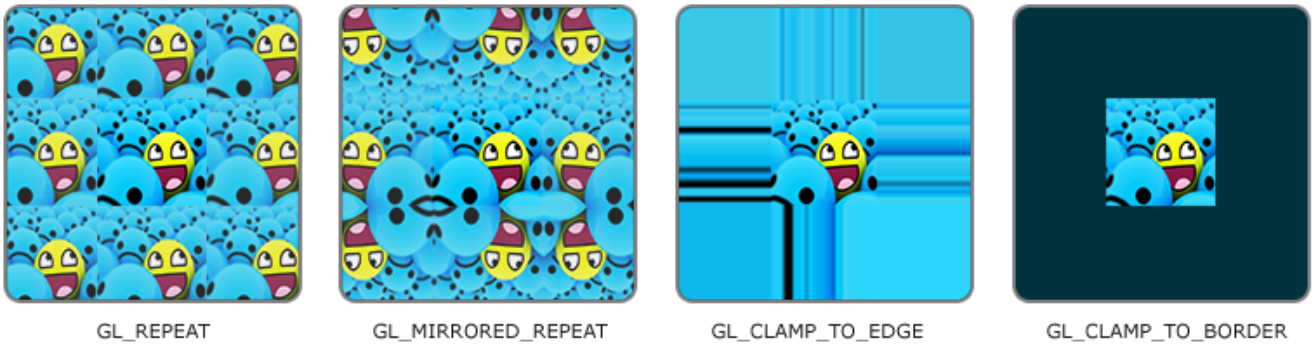
- `r` **(z)** if using 3D texture.

If we specify coordinates for the texture outside the range **(0, 0)** to **(1, 1)** the texture will wrap. The result of this depends on the wrapping method specified for the texture, for example with `glTexParameteri`

Example:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

`GL_REPEAT` is the default behavior for textures.

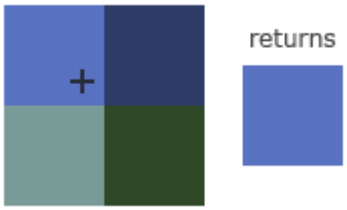


Texture Filtering

Texture coordinates do not depend on resolution but can be any floating point value. This means that OpenGL has to figure out which texture pixel (**Texel**) to map the (interpolated) texture coordinate to. This is especially important with very large objects with low-resolution textures. This is where **texture filtering** comes in.

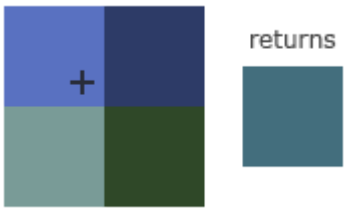
Options:

- **GL_NEAREST / Nearest Neighbour / Point Filtering:**
  Default. Selects the texel whose center is closest to the texture coordinate.



- **GL_LINEAR / Bilinear Filtering:**
  Takes an interpolated value from the texture coordinate's neighboring texels, approximating a color between the texels. The

closer a texel's center is to the coordinate, the more that texel's color contributes.



GL_NEAREST

GL_LINEAR

We can set texture filtering for **magnifying** and **minifying** operations (when scaling up or downwards). In this example we use nearest neighbour for downscaling and linear for upscaling.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, NEAREST)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

Texture Units

With `glUniform1i` we can assign a **location** value to the texture sampler so we can set multiple textures at once in a fragment shader. The **location** of a texture, can think like a container, is more commonly known as a **Texture Unit**. Default texture unit is 0 which is the default active texture. Not all graphics drivers assign a default texture.

Main purpose of **Texture Units** is to allow us to use more than one texture in a shader. By assigning **Texture Units** (locations) to the samplers, we can bind to multiple textures at once, as long as we activate the corresponding texture unit first.

Can activate **Texture Units** like this:

```
glActiveTexture(GL_TEXTURE0); // activate the texture unit first before binding texture
glBindTexture(GL_TEXTURE_2D, texture);
```

`GL_TEXTURE0` is always active by default. OpenGL should have 16 **Texture Units** we can use. If we want to add another texture we need to add 1 more sampler in the fragment shader.

```
#version 460 core
...

uniform sampler2D texture1;
uniform sampler2D texture2;

void main()
{
    FragColor = mix(texture(texture1, TexCoord), texture(texture2, TexCoord), 0.2);
}
```

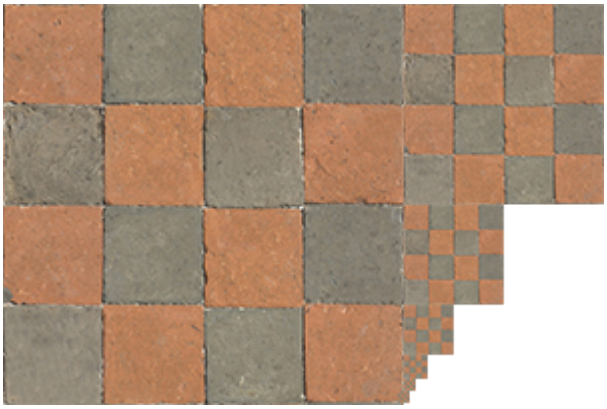
mix is a built-in GLSL function to mix textures. Takes two textures with texture coordinate and **linearly interpolates** between them based on the third argument. The third argument decide the specific values. If last value is 0.2, the final color will be a mix of **80%** of the first value and **20%** of the second. If its 1, it will be **0%** of the first value and **100%** of the second.

Mipmaps

Example Issue:

Objects far away have the same high-resolution texture as objects close by. Fragments span a larger portion of the texture, leading to difficulty picking a texture color and thus retrieving the right color value. This produces visible artifacts on small objects and unnecessarily uses memory bandwidth by applying high-resolution textures on small objects.

Solution: Mipmaps



Mipmaps are a collection of texture images where each subsequent texture is **half the size** of the previous one. The further away an object is, the smaller the texture that should be used. Mipmap level `0` is the original texture, `1` is the next largest and so on.

When switching between mipmap levels during rendering, OpenGL can produce some artifacts like sharp edges between two mipmap layers. Just as with texture filtering we can filter between mipmap levels using **NEAREST** and **LINEAR**

Common mistake is to set one of the mipmap filtering options as the magnification filter, doesnt have any effect since mipmaps are primarily used when textures are downscaled. Texture magnification doesnt use mipmaps and giving it a mipmap filtering option will generate `GL_INVALID_ENUM` error.

To load an image we use stb_image.h. We later use this data loaded in `glTexImage2D` .

```
int width, height, nrChannels;
unsigned char *data = stbi_load("container.jpg", &width, &height, &nrChannels, 0);
```

After generating a texture with glGenTextures binding with glBindTexture and setting parameters we can call glTexImage2D. After this the currently bound texture object now has the texture image attached to it. It only has the base-level of the texture attached though, so if we don't want to specify all the different images (mipmap levels) manually we can call `glGenerateMipmap(GL_TEXTURE_2D);` This automatically generates all the required mipmaps for the currently bound texture.

Good practice to free image memory after generating texture and mipmaps

```
stbi_image_free(data);
```

Applying textures

As with other vertex attribute, adding to vertices (with the new texture coordinates), we need to notify OpenGL of the new attribute and format with `glVertexAttribPointer` , and not forget to adjust stride length. Activate attribute with `glEnableVertexAttribArray` . Also need to adjust vertex shader to accept texture coordinates as a vertex attribute with for example

```
layout (location = 2) in vec2 aTexCoord;
out vec2 TexCoord;
```

Fragment shader should then accept TexCoord output variable as an input variable. Fragment shader should also have access to the texture object we created. GLSL has built-in data-type for texture objects called sampler, postfix of specific type (1D, 2D or 3D). Can then add a texture to the fragment shader by declaring

```
uniform sampler2D ourTexture;
```

then pass this in to GLSL function. Example:

```
#version 460 core
out vec4 FragColor;

in vec3 ourColor;
in vec2 TexCoord;

uniform sampler2D ourTexture;

void main() {
    FragColor = texture(ourTexture, TexCoord);
}
```

Texture function in GLSL samples the corresponding color value using the set texture parameters. Output of the fragment shader is then the **filtered** color at the **interpolated** texture coordinate

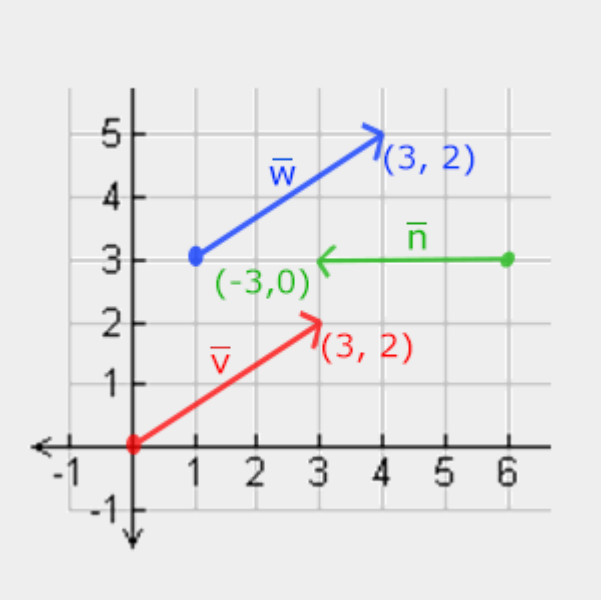
Example:

```
glBindVertexArray(VAO);
glBindTexture(GL_TEXTURE_2D, texture);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

Some math

Vectors

A vector has a direction and a magnitude (also known as its strength or length). A 2D vector exists in a flat plane, like a computer screen or a piece of paper.



3D vectors are representations of three-dimensional space and we add a Z-axis. In formulas it is common for vectors to be displayed as:

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

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.

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:

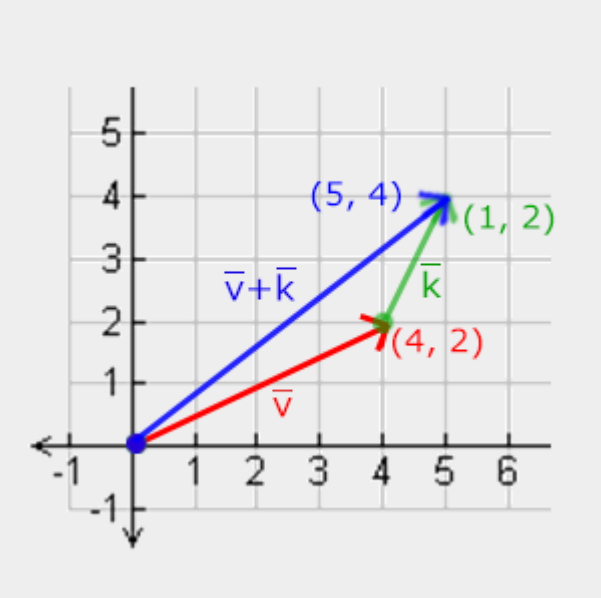
$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} + x \rightarrow \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} + \begin{pmatrix} x \\ x \\ x \end{pmatrix} = \begin{pmatrix} 1 + x \\ 2 + x \\ 3 + x \end{pmatrix}$$

Where + can be +,−,· or ÷ where · is the multiplication operator.

Addition of two vectors:

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

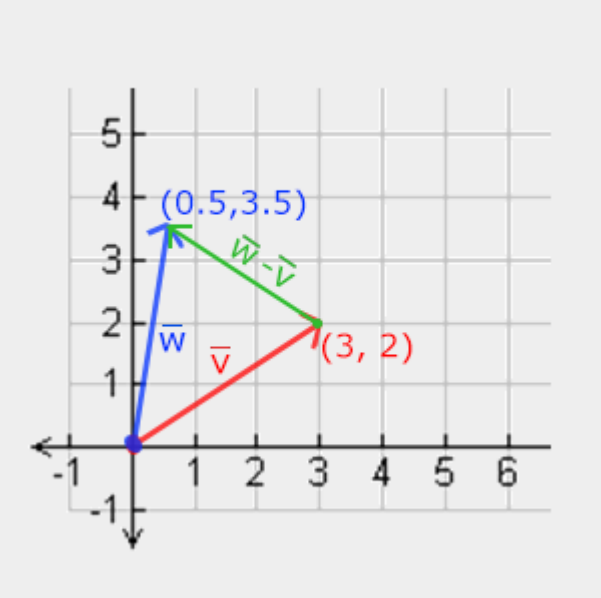
Addition of vectors v=(4,2) and k=(1,2)



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

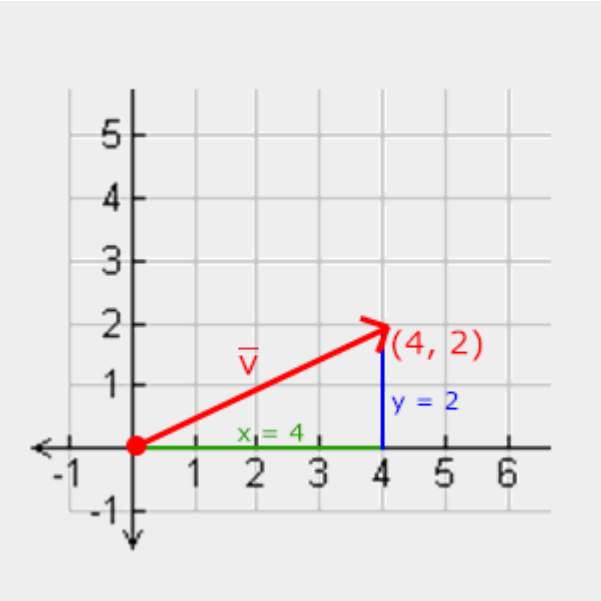
$$\bar{v} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \bar{k} = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} \rightarrow \bar{v} + -\bar{k} = \begin{pmatrix} 1+(-4) \\ 2+(-5) \\ 3+(-6) \end{pmatrix} = \begin{pmatrix} -3 \\ -3 \\ -3 \end{pmatrix}$$

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/Magnitude of vector

The length of a vector is always positive. To retrieve the **length/magnitude** of a vector we use **Pythagoras theorem**:



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

$\|\bar{v}\|$ is denoted as the length of vector $\bar{v}$. Can extend to 3D by adding z^2 to the equation.

a **Unit Vector** is a vector with a length of 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).

Dot product (skalärprodukt) of two vectors:

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

where $\cos\theta$ is the angle between the vectors. If the two vectors were **Unit Vectors** we would have

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

Now the **Dot Product** only defines the angle between the two vectors.

$\cos(90°)$ is equal to 0 and $\cos(0°)$ is equal to 1. This allows us to 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 (right angle is exactly $90°$). Dot product becomes very useful when doing lighting calculations.

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:

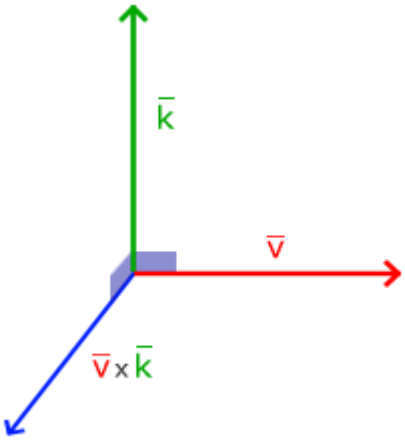
$$\begin{pmatrix} 0.6 \\ -0.8 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = (0.6 * 0) + (-0.8 * 1) + (0 * 0) = -0.8$$

To get the degree between the two unit vectors we use the inverse of the cosine function (arccos).

$$cos^{-1}(-0.8) = 143°$$

Crossproduct

Crossproduct 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, then a cross product would result in 3 orthogonal vectors. How it looks (v x k):



cross product between two orthogonal vectors A and B:

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

Matrices

Rectangular array of numbers, symbols and/or mathematical expressions. Each individual item in a matrix is called an **element** of the matrix. Example, 2x3 matrix:

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

Matrices are indexed by (i,j) where i is the row and j is the column. 2x3 is the ***dimensions*** of the matrix, 2 rows and 3 columns.

Addition and **Subtraction** is only defined for matrices of the same dimensions. Example:

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

Same rules for subtraction.

Matrix-Matrix multiplication

Rules:

- 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
- Matrix multiplication is not commutative that is A·B≠B·A.

Example:

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

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

Matrix multiplication is a combination of normal multiplication and addition using the left-matrix's rows with the right-matrix's columns.

Matrix-Vector multiplication

A vector is basically a **Nx1** matrix where **N** is the vector's number of **components** (also known as an **N-dimensional vector**). Vectors are just like matrices an array of numbers, but with only 1 column.

If we have a **M**x**N** matrix we can multiply this with the vector **N**x**1**, since the columns on the left is equal to the rows on the right.

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.

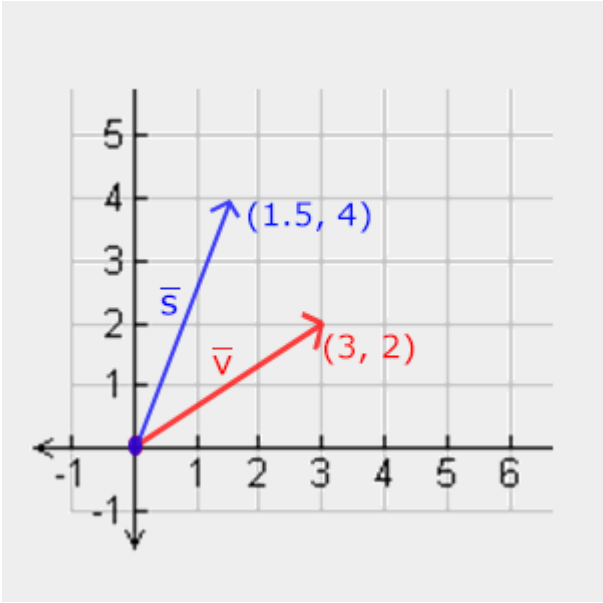
Identity matrix

In OpenGL we usually work with **4x4** transformation matrices for several reasons, of them being that most vectors are of size 4 (vec4). The most simple **transformation matrix** is the **identity matrix**. Leaves vector unharmed.

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

Even if it doesnt **transform**, the **identity matrix** is usually a starting point for generating other transformation matrices. Deeper into linear algebra it is very useful for proving theorems and solving linear equations.

Scaling



Scaling the vector $\bar{v} = (3, 2).$ by (0.5, 2.0). This scaling operation is a **non-uniform** scale, since the scaling factor is not the same for each axis. If it was, it would be called a **uniform-scale**. OpenGL usually operates in 3D space, we could set z-axis scale to 1, leaving it unharmed.

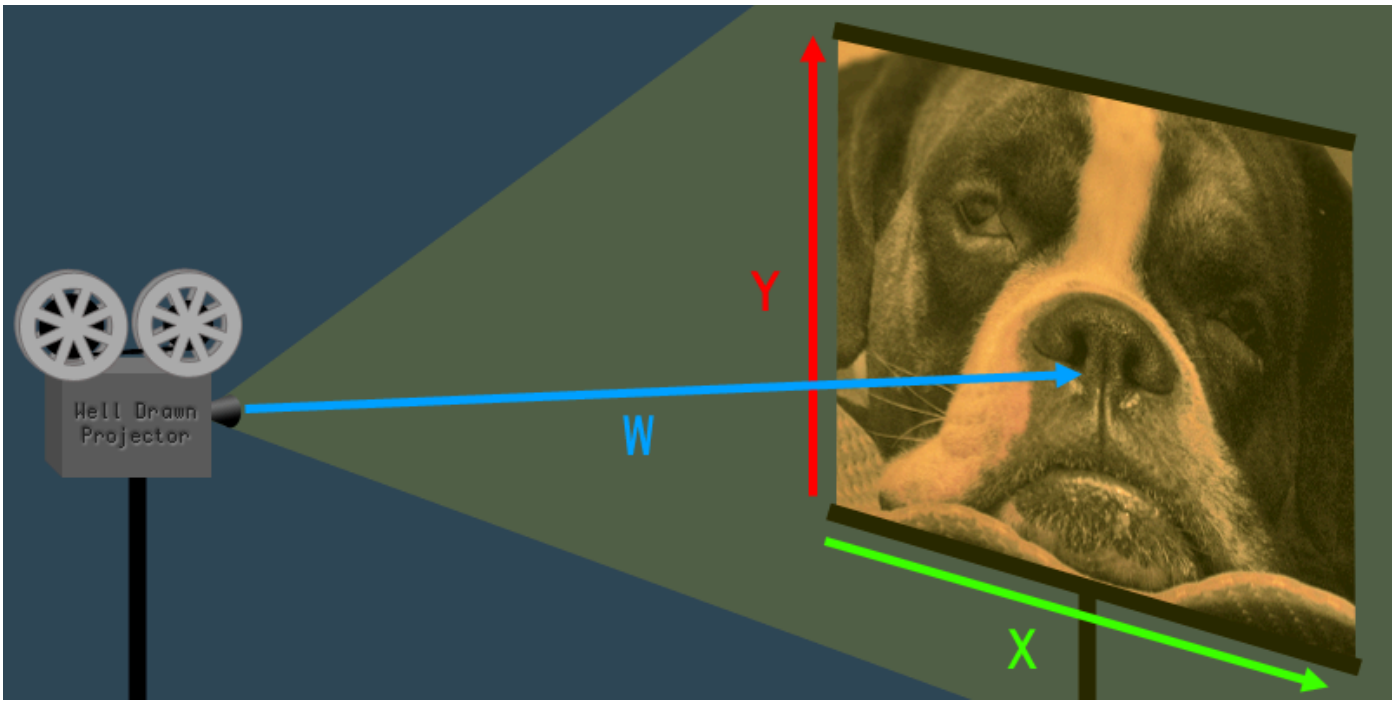
If we represent the scaling variables as (S_1, S_2, S_3) we can define a scaling matrix on any vector (*x,y,z*) as:

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

4th scaling value is set to 1 for now, the **w** component.

W component of a vector is known as a **Homogeneous coordinate**. Homogeneous coordinates add an extra dimension called **W**, which scales the X, Y, and Z dimensions. Matrices for translation and perspective projection transformations can only be applied to homogeneous coordinates, which is why they are so common in 3D computer graphics. Value of **w** affects the size (aka scale) of the image. To get the 3D vector from a homogeneous vector we divide the x, y and z coordinate by its **w** coordinate. This is called **perspective division**.

A lot of the time the initial value of **w** is set to 1, so we don't notice it.



Translation

Translation is the process of adding another vector on top of the original vector to return a new vector with a **different position**, thus moving the vector based on a **translation vector**. Just like the scaling matrix there are several locations on a 4-by-4 matrix that we can use to perform certain operations and for translation those are the top-3 values of the 4th column.

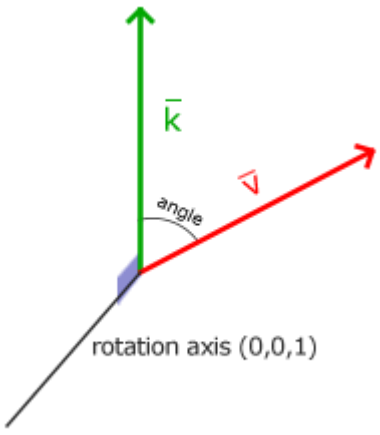
If we represent the translation vector as (T_x, T_y, T_z) we can define the translation matrix by:

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

Rotation

A rotation in 2D or 3D is represented with an angle, either in degrees or in radians.
Rotations in 3D are specified with an angle and a ***rotation axis***. Angle specified will rotate the object along the rotation axis given.

Using trigonometry it is possible to transform vectors to newly rotated vectors given an angle



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

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

With these rotation matrices we can transform our position vectors around one of the three unit axes. To rotate around arbitary 3D axis we can combine them by rotating around each one, but this introduces a problem called **Gimbal lock**. **Gimbal lock** means you lose a degree of freedom and rotation becomes constrained.

It is better to rotate around an arbitary unit axis, for example (0.662,0.2,0.722), which is a unit vector, right away instead of combining the rotation matrices:

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

To truly prevent **Gimbal locks** we have to represent rotations using **quaternions**, that except from being safer are more computionally friendly.

Combining matrices

True power from using matrices for transformations is that we can combine multiple transformations in single matrix thanks to **Matrix-Matrix** multiplication.

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:

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

Note that we first have the **translation** matrix and then the **scale transformation** matrix when multiplying matrices. As **Matrix multiplication** is **NOT** commutative, the order is important. This is because of the fact that when multiplying matrices the **right-most matrix** is applied/multiplied to the vector/point **FIRST** so you should read the multiplications from **right to left**. In this example scaling is on the right so it is applied first and then translation.

$$AB = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

$$BA = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$$

Important

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:

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

Resulting vector is first scaled by two and then translated by (1,2,3).

Here we first rotate the container around the origin (0,0,0) and once it's rotated, we translate its rotated version to the bottom-right corner of the screen. Remember that the actual transformation order should be read in reverse: even though in code we first translate and then later rotate, the actual transformations first apply a rotation and then a translation.Here we first rotate the container around the origin (0,0,0) and once it's rotated, we translate its rotated version to the bottom-right corner of the screen. Remember that the actual transformation order should be read in reverse: even though in code we first translate and then later rotate, the actual transformations first apply a rotation and then a translation.

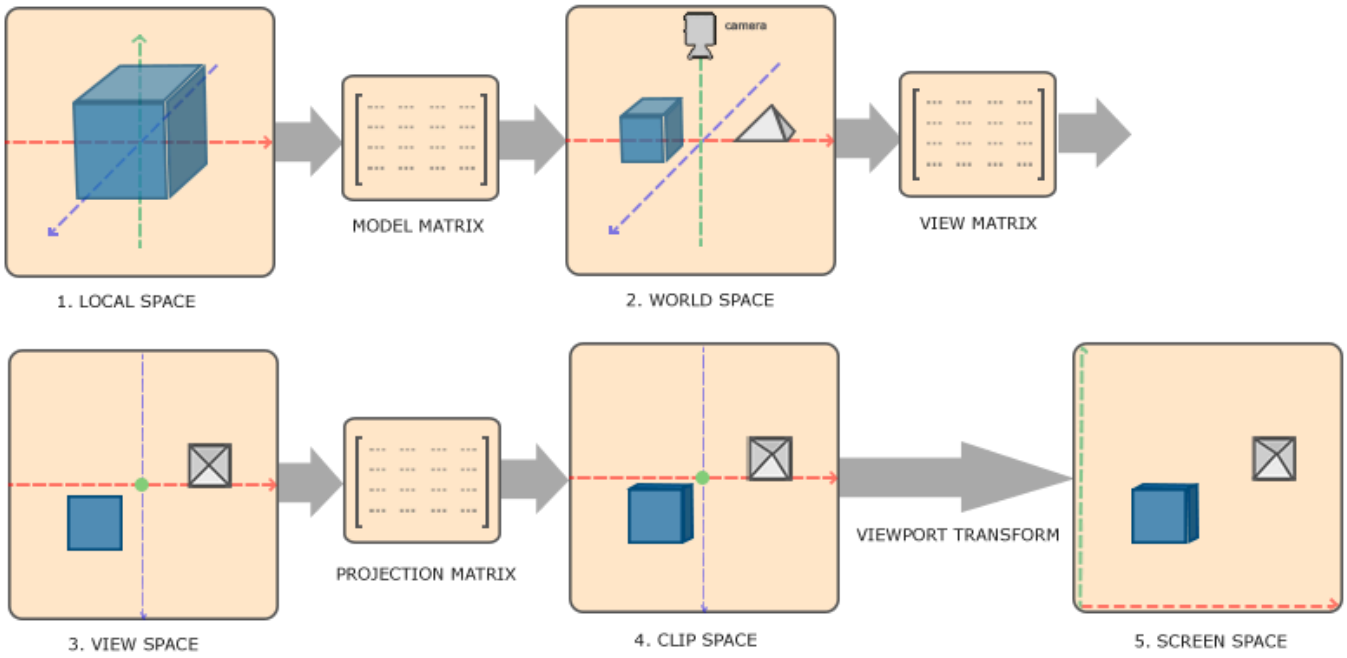
```
glm::mat4 trans = glm::mat4(1.0f);
trans = glm::translate(trans, glm::vec3(0.5f, -0.5f, 0.0f));
trans = glm::rotate(trans, static_cast<float>(glfwGetTime()), glm::vec3(0.0f, 0.0f, 1.0f));
GLuint transformLocation = glGetUniformLocation(shaderProgram2.getId(), "transform");
glUniformMatrix4fv(transformLocation, 1, GL_FALSE, glm::value_ptr(trans));
```

Coordinate systems

After each vertex shader run OpenGL expects all vertices that we want to become visible to be in **Normalized Device Coordinates**. X, y, and z-coordinates for each vertex should be between -1 and 1. Usually we specify the coordinates in a range, or space, and then in the vertex shader transform these to **NDC**. The **Normalized Device Coordinates** are then given to the rasterizer to transform them to 2D coordinates/pixels/fragments on the screen, and then to fragment shader for further processing (shading, texturing etc).

Vertices are transformed at different stages:

- Local space/object space/model space
- World space
- View space/camera space/eye space
- Clip space
- Screen space/2D coordinates



Local space

If you create a cube in blender, its initial position is probably (0, 0, 0). Coordinates are local to object, all the vertices are in local space.

World space

We want to define a position for each object to position them inside a larger world. Coordinates of all your vertices relative to a (game) world. This is for example the coordinate space where want objects transformed in such a way that they are scattered around the place, preferably in a realistic fashion.

Objects get transformed from **Local Space** to **World Space** with the help of the **Model** matrix. the **Model** matrix is a transformation matrix that translates, scales and/or rotates the object to place it in the world.

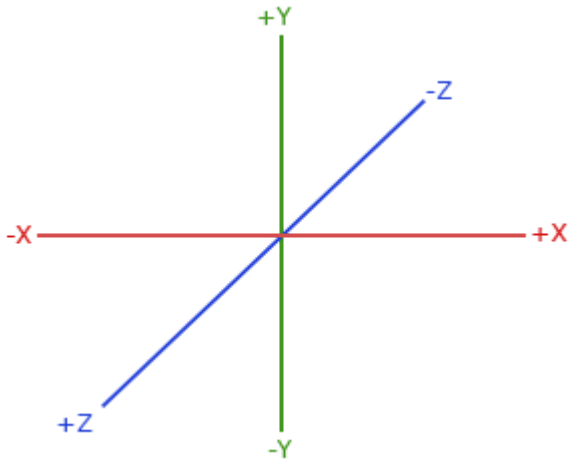
The **model matrix** consists of translations, scaling and/or rotations we'd like to apply to **transform** all object's vertices to the global world space.

Camera space

Also known as **View Space or Eye Space** this is where **World Space** coordinates get transformed to coordinates that are in front of the user's view. The combined transformations of this is stored inside a **View Matrix** that transforms world coordinates to view space.

To move a camera backwards, is the same as moving the entire scene forward.

That is exactly what a **view matrix** does, we move the entire scene around inversed to where we want the camera to move. Because we want to move backwards and since OpenGL is a right-handed system we have to move in the positive z-axis. We do this by translating the scene towards the negative z-axis. This gives the impression that we are moving backwards.



Clip space

At the end of each vertex shader run, OpenGL excepts coordinates to be within a certain range. If they are ***NOT*** within this range, they get **clipped**, discarded.

To transform vertex coordinates from **camera space** to **clip space** we define a **projection matrix** that specifies a range of coordinates, e.g. -1000 and 1000 in each dimension.

Within this specified range the projection matrix then converts the coordinates to **normalized device coordinates**, with **perspective divsion** between, dividing with **w**. Coordinates outside this range will not be mapped between -1.0 and 1.0 and therefore **clipped**.

For example, if the specified range is (-1000, 1000), a coordinate with position (1250, 750, 500) would **NOT** be visible. If only part of a primitive , for example a triangle is outside the **clipping volume**, OpenGL reconstruct the triangle as one or more triangles to fit within the clipping range.

Projection matrix creates a **viewing box** that is called a **frustum**, and each coordinate that ends up within this **frustum**, will end up visible on the users screen. This entire process to convert coordinates within a specified range to **NDC** is called **projection**, since the **projection matrix** projects 3D coordinates to the easy-to-map-to-2D **normalized device coordinates**.

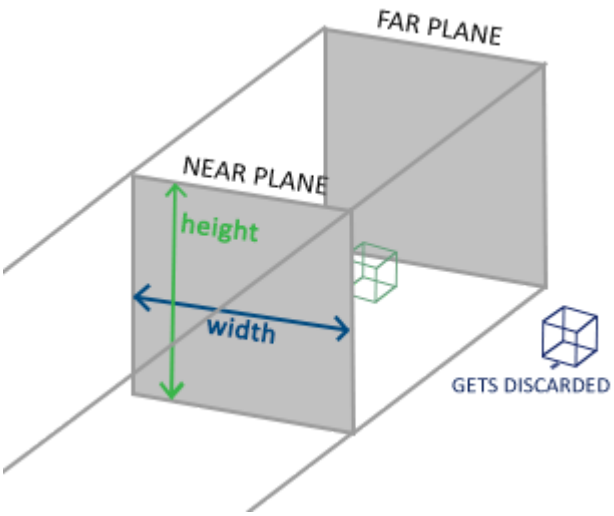
Perspective divison transforms the 4D clip space coordinates to 3D **normalized device coordinates**. This step is performed automatically at the end of the vertex shader step.

After this the resulting coordinates are mapped to screen coordinates using the settings of **glViewport** and turned into fragments.

Orthographic projection

The projection matrix to transform view coordinates to clip coordinates usually takes two different forms, where each form defines its own unique **frustum**. We can either create an **orthographic projection matrix** or a **perspective projection matrix**

Defines a cube-like frustum box that defines a clipping space where each vertex outside this box is clipped. When creating an **orthographic projection**, we specify the width, height and length of the visible **frustum**.



The frustum defines the visible coordinates and is specified by a **width**, a **height** and a **near** and **far** plane.

The **orthographic frustum** directly maps all coordinates inside the frustum to **normalized device coordinates** without any special side effects since it won't touch the w component of the transformed vector; if the w component remains equal to 1.0 perspective division won't change the coordinates.

To create an orthographic projection matrix we make use of GLM's built-in function glm::ortho

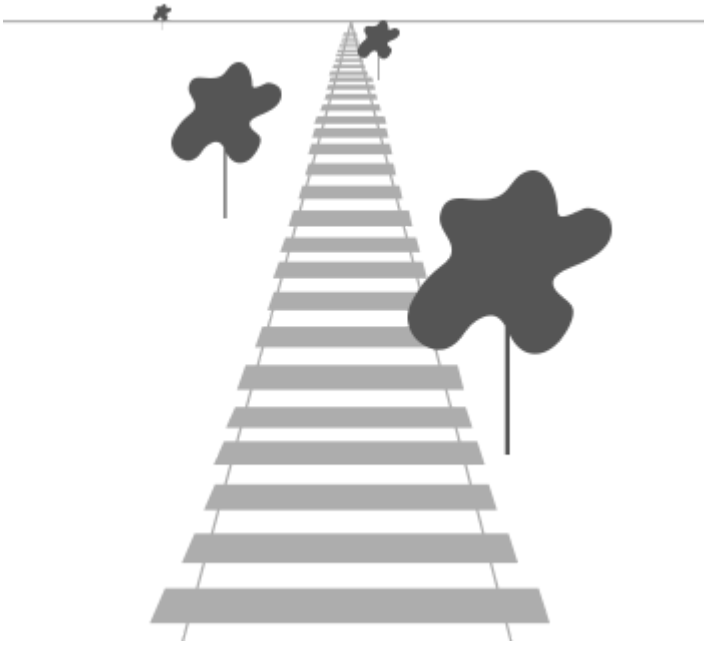
```
glm::ortho(0.0f, 800.0f, 0.0f, 600.0f, 0.1f, 100.0f);
```

The first two parameters specify the **left and right coordinate** of the frustum and the **third and fourth parameter specify the bottom and top part** of the frustum. With those **4 points we've defined the size of the near and far planes** and the **5th and 6th parameter then define the distances between the near and far plane**. This specific projection matrix transforms all coordinates between these x, y and z range values to normalized device coordinates.

An **orthographic projection** matrix **directly maps** coordinates to the 2D plane that is your screen, but in reality a direct projection produces unrealistic results since the projection doesn't take **perspective** into account. That is something the **perspective projection matrix** fixes for us.

Perspective projection

The effect of objects further away appearing smaller = **perspective**.



A **perspective projection matrix** maps a given **frustum** range to **clip space**, but also manipulates the **w** value of each vertex coordinate in such a way that the further away an object is from the viewer, the higher this **w** component becomes.

Once coordinates are transformed to **clip space** they are in the range **-w** to **w**, anything outside is clipped.

Face culling: Discarding triangle primitives based on their winding order relative to the viewport. The order of a triangle's vertices is called the winding order

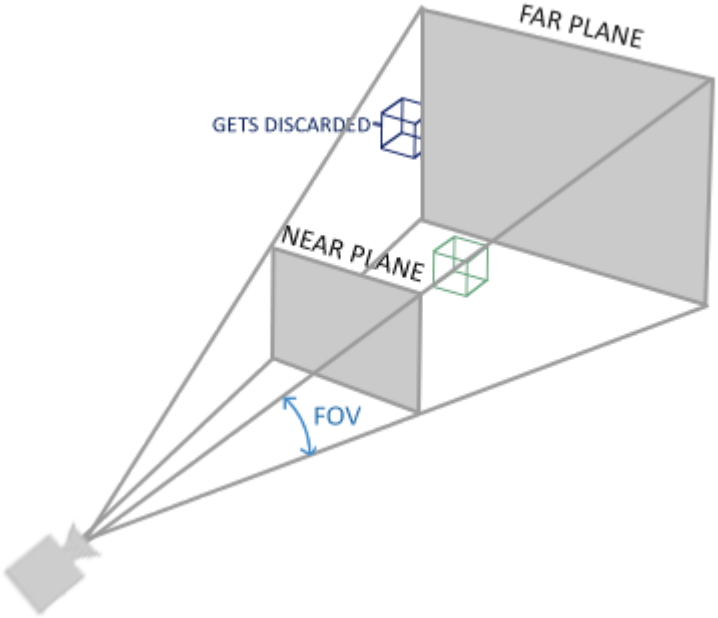
Once the coordinates are in **clip space**, **perspective division** is applied to them. By dividing with the **w** value, we get smaller vertex coordinates the further an object is from the viewer. The resulting coordinates are then in **normalized device space**.

A perspective projection matrix can be created in GLM as follows:

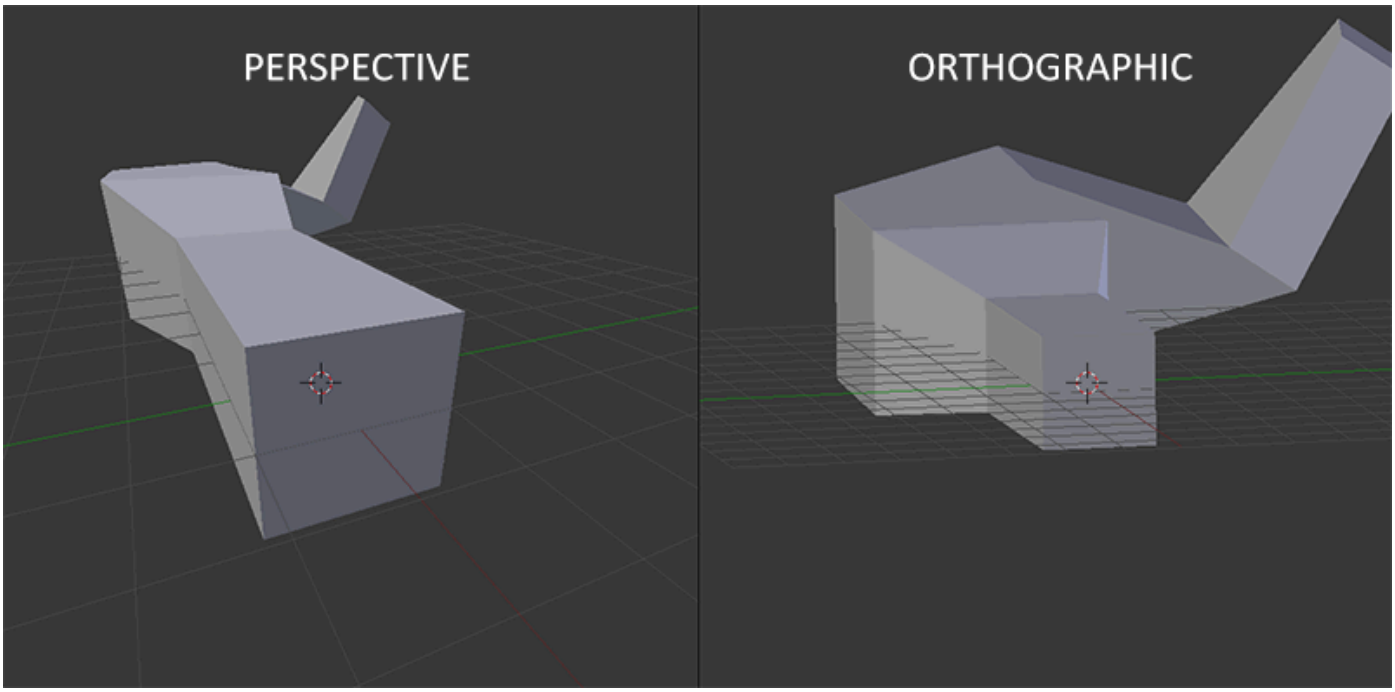
```
glm::mat4 proj = glm::perspective(glm::radians(45.0f), (float)width/(float)height, 0.1f, 100.0f);
```

1. First parameter defines the **Field of View**, sets how large the viewspace is. Usually set to 45 degrees for a realistic view.
2. Second parameter sets aspect ratio, calculated by dividing viewports width by its height.
3. Third and fourth parameters set the **near** and **far** plane of the **frustum**. Usually near distance is 0.1 and far distance 100. All vertices between the near and far plane and inside the frustum will be rendered.

Perspective frustum can be visualized as a non-uniformly shaped box from where each coordinate inside this box will be mapped to a point in clip space Perspective frustum:



Whenever the near value of your perspective matrix is set too high (like 10.0), OpenGL will clip all coordinates close to the camera (between 0.0 and 10.0), which can give a visual result you maybe have seen before in videogames where you could see through certain objects when moving uncomfortably close to them.



Because the **orthographic projection doesn't use perspective projection**, objects farther away **do not** seem smaller, which produces a weird visual output. For this reason the orthographic projection is mainly used for 2D renderings and for some architectural or engineering applications where we'd rather not have vertices distorted by perspective

So, we create a transformation matrix for each step: **Model**, **View** and **Projection** matrix. A vertex coordinate is then **transformed** to clip coordinates as follows:

$$V_{clip} = M_{projection} \cdot M_{view} \cdot M_{model} \cdot V_{local}$$

Order of matrix multiplication needs to be in reverse as explained earlier.

The resulting vertex should then be assigned to `gl_Position` in the vertex shader and OpenGL will then automatically perform **perspective Division** and **clipping**.