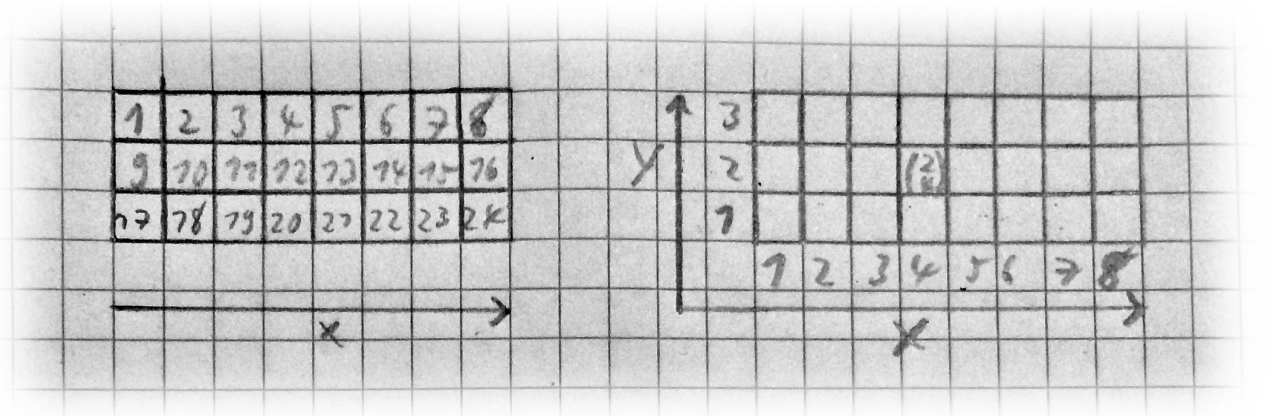
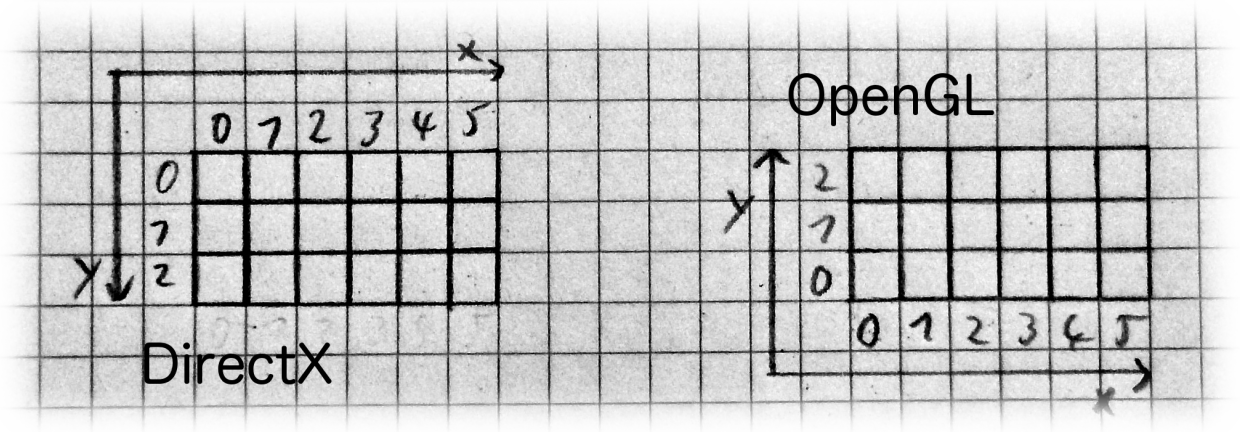
Coordinate Systems and other abstract things

Introduction
The following is a writeup of my thoughts, experience and research. It may seem kinda messy and some things can be wrong or incomplete, so feel free to correct me if you know better. While this topic is not that complicated, parts of it are quite hard to explain and information on some of the things I write about are really scattered on the internet, which is the reason for this.

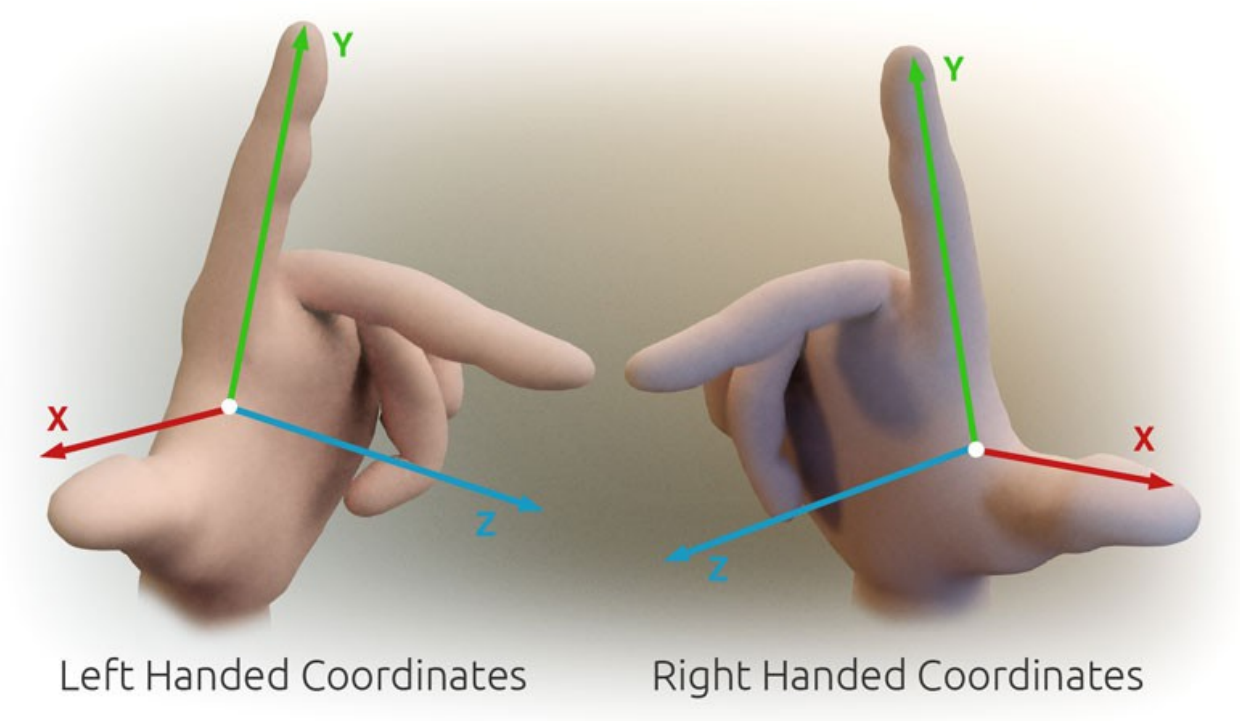
Basics
Since displays usually have something like pixels and in the end we usually want to assign a value to each of those pixels, it is a very good idea to align a coordinate system with those pixels.
This coordinate system could have just one axis, assuming all pixels are indexed as if they were on a line, but in reality the display has a 2D surface, so the more straight forward thought is to use 2 axes, where one addresses the columns and one the rows and this is what is usually done.
It does not matter how those axes are called, but it is common to call them x and y, and z if we need a third one, where it again does not matter which one represents which direction. The axes don´t even have to be in alignment with the hardware pixels and they don´t have to be orthogonal, although this makes things a lot easier and is what I am going to talk about. It is also not important where the origin of this coordinate system is.



As an example, when addressing a single pixel in OpenGL it expects an x value for the horizontal position and an y value for the vertical one, where x grows to the right, y from bottom to top and the origin is the bottom left corner.
In DirectX on the other hand, the origin is assumed to be in the top left corner and y is growing from top to bottom.
These assumptions about the graphics APIs are consistent (I am not completely sure about DirectX here...) and also like this when drawing 2D geometry, except that the origin is assumed to be in the center of the screen in both APIs.



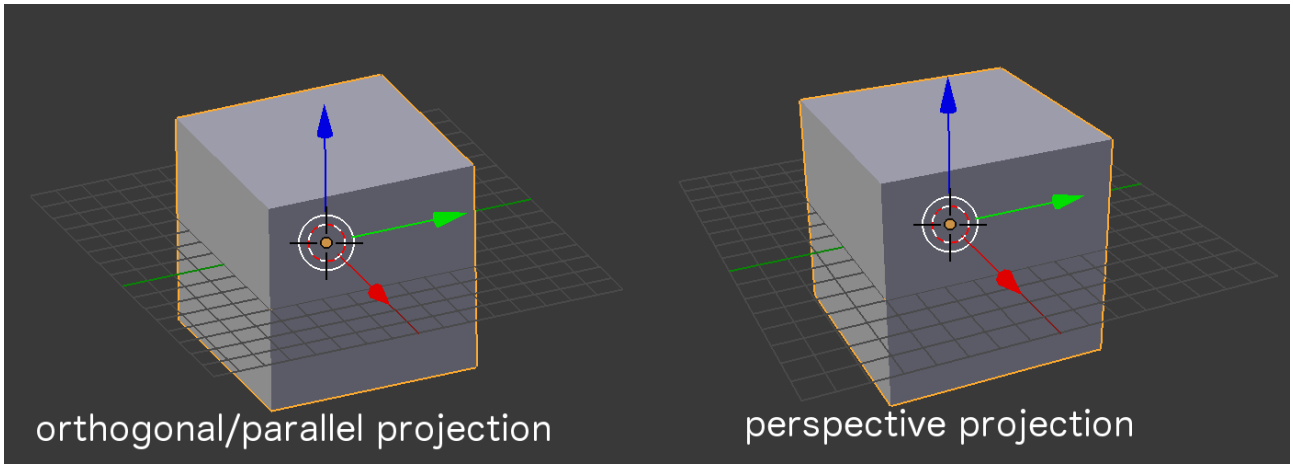
Handedness
But we want to draw 3D geometry which means, that we need a third axis. This new z axis is by default assumed to point from inside the screen towards the viewer in both, OpenGL and DirectX, although it only effects the value written in the depth buffer and thus the depth test function is what really defines the orientation of the z axis. Assuming the default setting of the depth test to fail if the new z is smaller than the old one, we can now just align our fingers with these x, y and z axes, where the thumb represents the x axis, the index finger the y axis and the middle finger the z axis and we will notice that the coordinate system assumed by DirectX can only be represented by our left hand, which makes it a so called left handed coordinate system and the OpenGL one can only be realized with the right hand which makes it a right handed coordinate system.
In the following I will focus on OpenGL, but the same ideas apply to DirectX.



(Source:
http://commons.wikimedia.org/wiki/File:3D_Cartesian_Coodinate_Handedness.jpg)

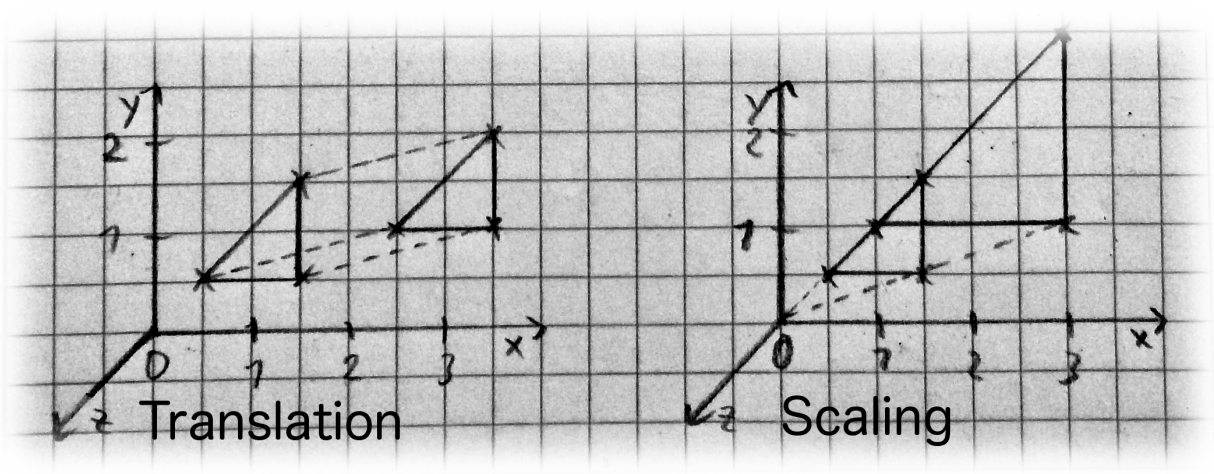
Projections
When rendering polygons through the hardware with pass through shaders, their vertex positions are expected to be in normalized device coordinates (NDC) with the

visible area ranging from -1 to 1 for all three axes. Everything outside those values is not visible and (0, 0, 0) represents the center of the screen. This also means that visible z values will range from 0 to -1.
With this knowledge, it is possible to push 3D meshes within those restrictions into the rendering pipeline. If the NDC ranges are too small for a task because you prefer to have your vertex positions specified in meters for example, you can just use a so called orthogonal projection matrix which does not do more than scaling the bigger values down. The problem with this is that objects far away will still look as if they were just in front of the camera because they don´t get smaller in the distance. This is sometimes wanted, but it looks wrong. To have far away objects appear smaller the further away they are, one can transform the geometry with a perspective projection matrix instead of an orthogonal one (for more information on projection matrices see for example http://www.songho.ca/opengl/gl_projectionmatrix.html). While the projection matrices could flip the axes and handedness, it is less confusing to just stick with the defaults, so that the handedness can be ignored from here on as everything else will just work.



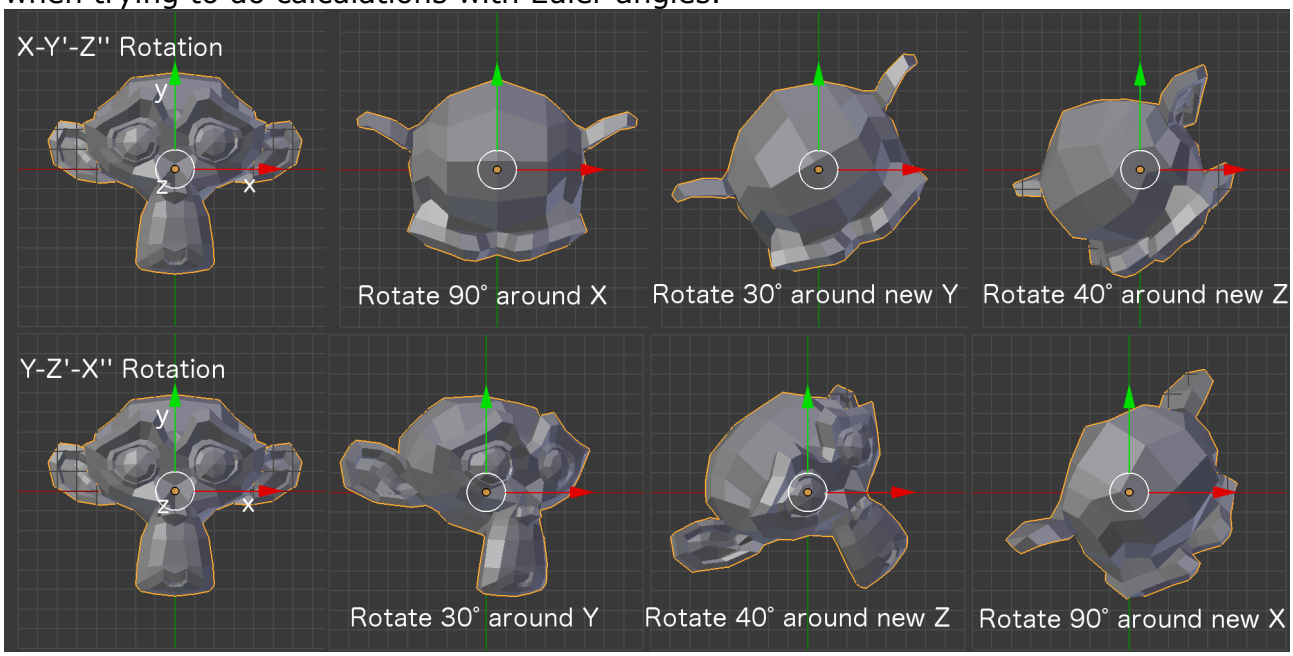
Basic Transformations
What we probably want as a user is to provide a couple of meshes, define their position, scale and orientation within a 3D world and see them from a camera with a position and orientation within this same world.

The easy part is the position. The meshes consist of a couple of vertices which are x, y and z values and to change the position of a mesh to for example (1, 2, 3) these values just have to be added to each of the vertices.
If we now want to move the camera to the same position, those values have to be subtracted from each vertices position and the result looks as if either the camera nor the mesh has been moved, because relative to each other they haven´t.
Scaling can be done by multiplying each vertex position with some factor.

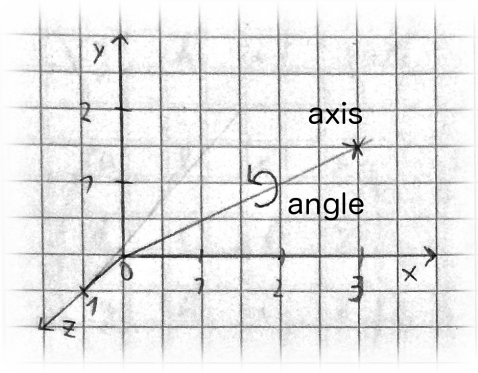


While the previous ideas are straight forward and not really something to think about (but there are of course different ways to represent position information than just as an x, y and z offset), things get quite complicated with the orientations.

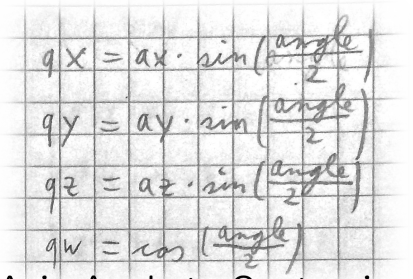
Euler angles
There are Euler angles, Tait-Bryan angles, angle-axis/axis-angle, quaternions, matrices, look-at and up vector and probably many more ways to represent orientations. Euler angles describe a series of rotations around the objects local coordinate system axes as for example x-y'-x'' where the ' signals that it is the resulting axis from the previous part of the rotation. Other sequences are very well possible and always describe different orientations for the same values.
This is very easy to use and usually behaves as expected, except when it does not, like when changing all three values at once, where the sequential character might feel strange to some people. The bigger problem is the so called gimbal lock, which is a situation where the rotated local axis for the next part of the rotation falls on the exact same axis as the first one. This means that one of the rotation values has no effect at all. As a result, Euler angles are working great for first person shooters, where you usually just turn around and look up and down, but are a really bad choice for space games where you need all 6 degrees of freedom, which will get really ugly when trying to do calculations with Euler angles.



Axis-Angle
Axis-angle representation, which uses a single angle describing a rotation around an arbitrary axis, has the disadvantage of 4 values that need to be stored for each object instead of 3 compared to Euler angles, but they work great for representing orientations and are kinda useless if you want to do any calculations with them...



Quaternions
The solution are quaternions, which are complex numbers with three imaginary parts and one real part. The fact that they are mathematical numbers results in rules for adding, subtracting, multiplying and dividing them with each other.
The three imaginary parts are usually stored as x, y and z while the real part is stored as w. Those values are best compared with those of the angle-axis representation because the quaternions w part is the Cosine of the angle and the x, y and z parts describe the orientation axis combined with the sinus of the angle, which makes the transformation between angle-axis and quaternion representation trivial.



Axis-Angle to Quaternion

A quaternion multiplication means the composition of the two rotations. Also A*B is different to B*A. A division is the composition of the first rotation with the inverse of the second rotation.
Another nice feature of quaternions is the so called SLERP function, which allows to interpolate between two quaternions in a very clean way (http://en.wikipedia.org/wiki/Slerp).
Because of all this, quaternions are actually a very good choice to represent orientations in 3 dimensions, but because they are also very abstract it is often useful to create them from angle-axis, as shown above or from Euler angles. The Euler angle part is tricky because there are so many different ways to express the same orientation through Euler angles, so in the end you should probably focus on one good working representation for the use case.

Matrices
Now I need to get a bit into matrices which can also be used to represent rotations. A 3D rotation matrix however needs 3*3 values, which is a disadvantage to quaternions.

On the other hand they can be multiplied very well, but needing more processing power than quaternions. Creating a rotation matrix from a quaternion is trivial and they can also be created from angle-axis and thus also from Euler angles by multiplying the rotation matrices for the angle around each of the world axes together. In the latter case the order of the multiplication defines the meaning of the Euler angles values.

$$R(q) = \begin{pmatrix} 1-2qy^2-2qz^2 & 2qx\,qy-2qz\,qw & 2qx\,qz+2qy\,qw \\ 2qx\,qy+2qz\,qw & 1-2qx^2-2qz^2 & 2qy\,qz-2qx\,qw \\ 2qx\,qz-2qy\,qw & 2qy\,qz+2qx\,qw & 1-2qx^2-2qy^2 \end{pmatrix}$$

Quaternion to rotation matrix

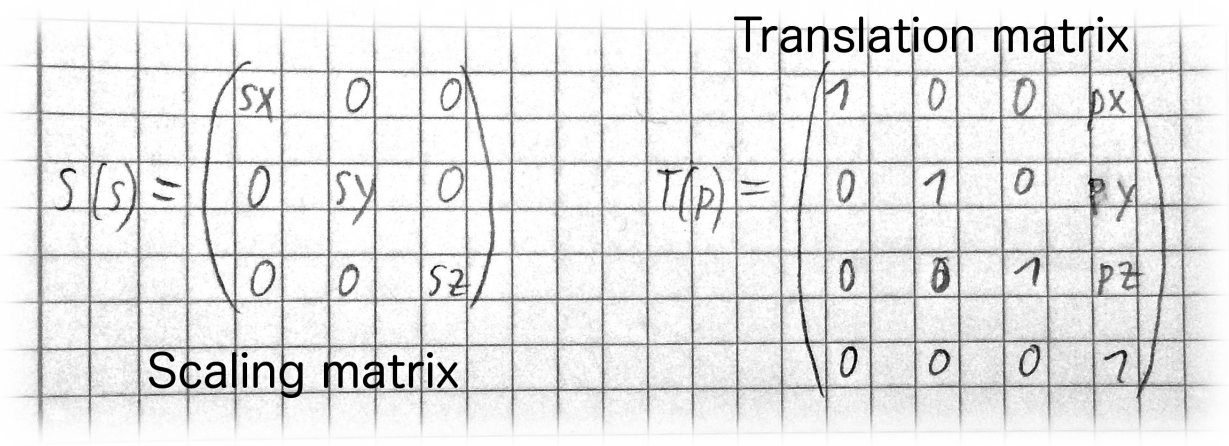
$$R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix} \quad R_y(\beta) = \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix}$$

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

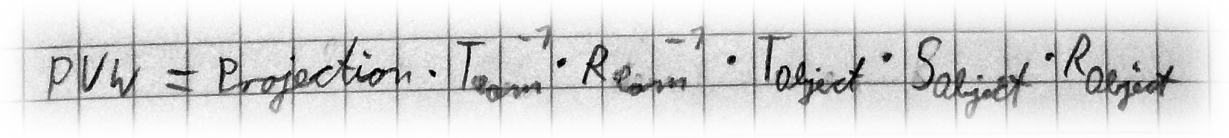
Basic rotation matrices around x, y and z axis

Putting everything together

As a last step everything has to be put together. And since graphics hardware is optimized on vector and matrix math and because a matrix can represent an objects position, scale and orientation as well as the position and orientation of the camera and the projection matrix and if needed even more things in just one matrix, a matrix is what we need to create from the data stored in the vectors and quaternions.



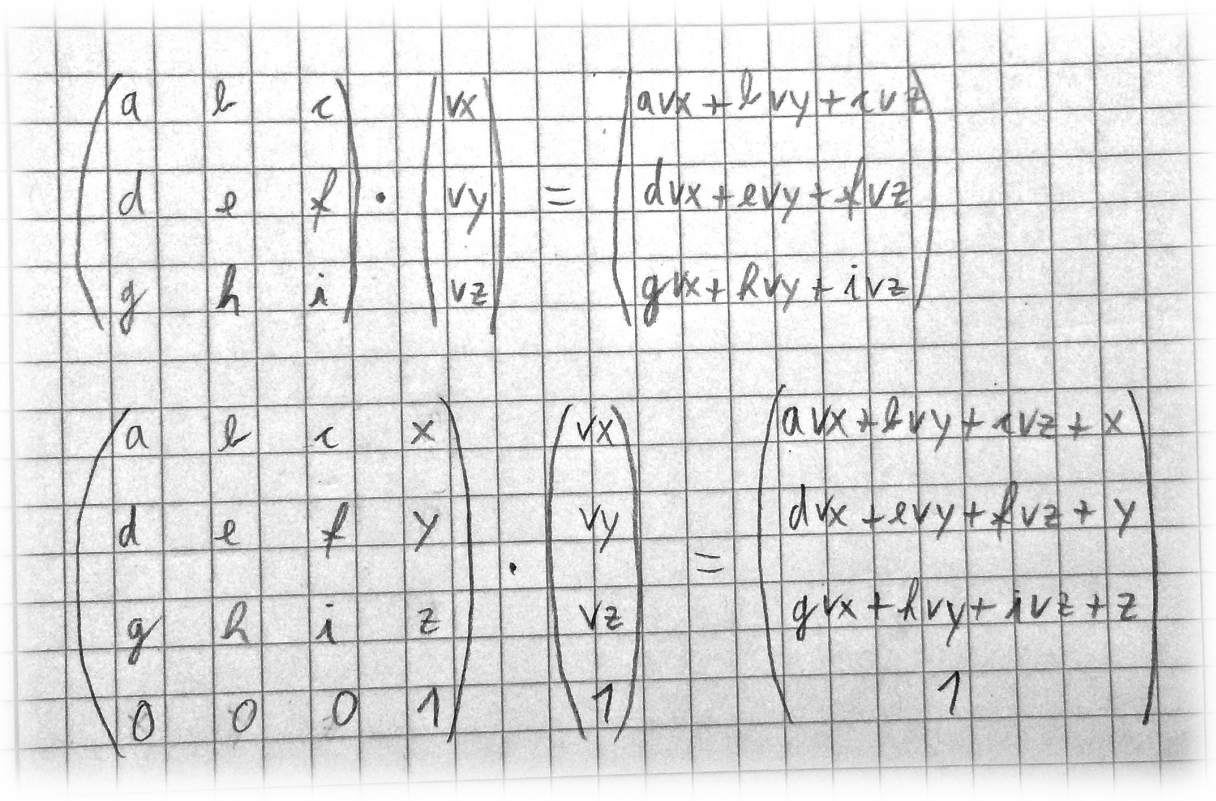
So all the data can be transformed to matrices which now have to be combined in the correct order.
Usually an object is supposed to be rotated around its own center, not the worlds center, which is the same if no other transformation is applied, so this is a good matrix to start with. Next up is the scaling, but it usually does not matter if you scale or rotate first. After this the object should be moved to its position with the translation matrix and because the cameras translation is still not in the result, we now want everything to rotate around the camera in the opposite direction of the camera orientation and then multiply with the cameras inverse translation matrix moving everything in the opposite direction of the camera position. The last transform is the projection matrix.



The resulting matrix can then be send to the vertex shader together with the raw mesh data where the matrix transformation will then be applied and the result is displayed on screen.

Notes
General notes
Translation, rotation and scale do not care about handedness, they will just work the way they should within their coordinate systems.
The only reason to think about the handedness within your coordinate system is to find out if a positive z is in front or behind the screen.
And while the whole internet seems to be discussing about the non existing of quaternion handedness, the handedness just does not matter and is a totally overrated and confusing property of a coordinate system.

Some more notes on matrices
Transforming a 3D vector with a 3x3 matrix cannot do any translation. To solve this, a higher dimensional vector and matrix is used.



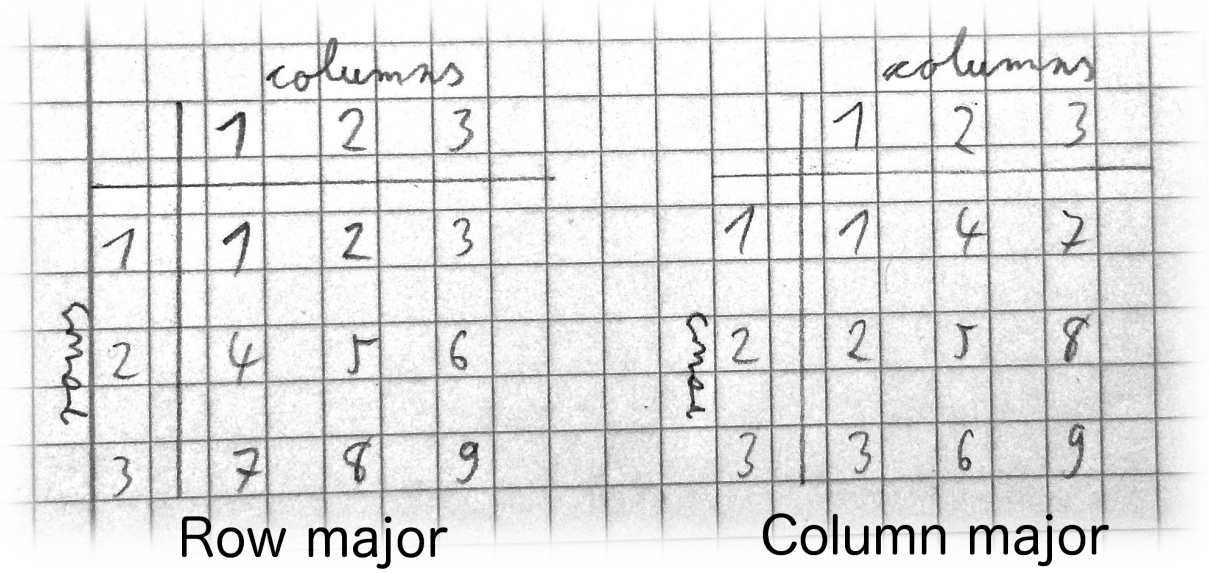
Rotation matrices are so called orthogonal matrices, meaning their inverse is the same as the transposed matrix.
This won´t work for translation and scaling matrices which are however trivial to invert.

$$R = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \qquad R^{-1} = R^{T} = \begin{pmatrix} a & d & g \\ b & e & h \\ c & f & i \end{pmatrix}$$

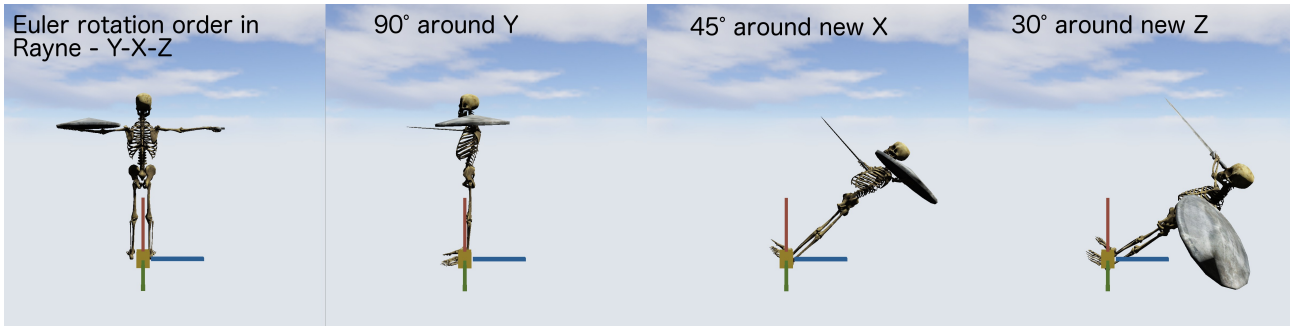
$$S = \begin{pmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & sz \end{pmatrix} \qquad S^{-1} = \begin{pmatrix} \frac{1}{sx} & 0 & 0 \\ 0 & \frac{1}{sy} & 0 \\ 0 & 0 & \frac{1}{sz} \end{pmatrix}$$

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

DirectX expects matrices to be row major, while OpenGL wants them column major.



Some more notes on Euler angle - quaternion conversion
There are so many different ways to do this, but I decided for the rotation around the y axis (yaw) first, followed by the rotation around the x axis (pitch) and in the end the rotation around the z axis (roll) and here the order of the multiplication and the way the matrices are stored (column or row major) matter a lot :P



The best way to determine how to transform Euler angles to quaternions is to create axis angle quaternions for each of the axes and multiply them in the wanted order. This can be optimized be removing unneeded parts of the multiplication due to zeros (http://www.euclideanspace.com/maths/geometry/rotations/conversions/eulerToQuaternion/Euler%20to%20quat.pdf).

The best way to determine how to transform quaternions to Euler angles seems to be, to multiply the base rotation matrices in the wanted order, find a formula to extract the angles from the matrix values and link the result with the quaternion to matrix conversion. In the end, some singularities need to be handled (http://www.euclideanspace.com/maths/geometry/rotations/conversions/quaternionToEuler/quat_2_euler_paper_ver2-1.pdf).