1 Transformations

Earlier in the course we have seen that OpenGL uses matrices to compute transformations (scaling, translation, rotation with 'glScale()' & Co) and also to compute the projection of the camera with 'gluPersepective()'. This part will focus on explaining the mathematics behind the scene. Even though OpenGL seems to hide these calculus with the transformation functions like 'glRotate()', you will need a minimal understanding of the mechanisms in order to implement standard computer graphics algorithms. The goal of the chapter is not to understand perfectly the mathematics with rigorous and complicated demonstrations but rather gives one the means to use the mathematics as an efficient tool.

1.1 2D transformations

2D transformations are convenient to introduce the subject. They also has the advantage to be easier to illustrate. Moreover the analogy to the 3D space is almost painless.

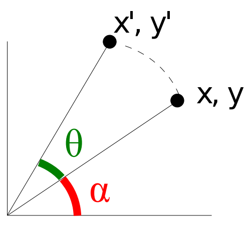
1.1.1 2D rotation

Here is a basic example of how to rotate a point $[x\,y]$ around the origin of a Cartesian coordinate system:

$$\begin{cases} x' = \cos(\theta).x - \sin(\theta).y \\ y' = \sin(\theta).x + \cos(\theta).y \end{cases}$$

The point $[x'\,y']$ is the rotation of $[x\,y]$ about the angle θ from the origin. This can actually be derived from the parametric equation of the circle and the trigonometric identity $\cos(a+b)$, and $\sin(a+b)$:

$$\begin{cases} x' = r.cos(\alpha+\theta) = \underbrace{r.cos(\alpha)}_{x}.\cos(\theta) - \underbrace{r.sin(\alpha)}_{y}.\sin(\theta) \\ y' = r.sin(\alpha+\theta) = r.cos(\alpha).\sin(\theta) + r.sin(\alpha).\cos(\theta) \end{cases}$$



But most importantly we can express this rotation as the product of a 2x2 matrix and the point $[x\,y]$:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} . \begin{bmatrix} x \\ y \end{bmatrix}$$

What is even more cooler is we can now store this matrix with four floats and rotate any point with about θ by multiplying it with the matrix.

Exercise:

1) Compute the rotation of $\theta = \frac{\pi}{2}$ of the points $[1;1]$ and $[0 \quad 1]$ with a 2x2 rotation matrix.

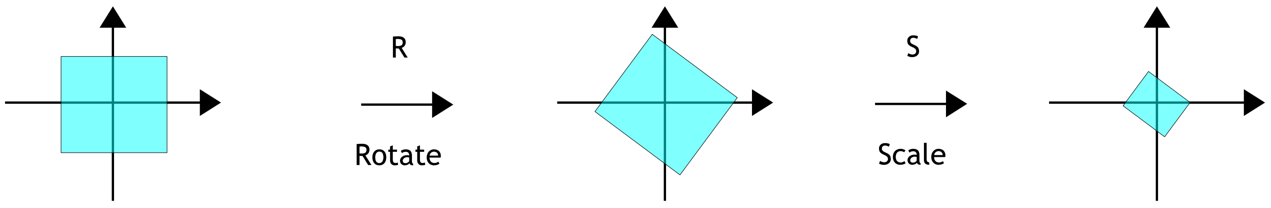
1.1.2 2D scaling (Homothetic transformation)

Can we use matrices for scaling?

$\begin{cases} x' = s_x . x \\ y' = s_y . y \end{cases}$ expressed with a matrix: $\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} . \begin{bmatrix} x \\ y \end{bmatrix}$

Yes, we can express an homothetic transformation (scaling) with a 2x2 matrix. Better the composition of transformation matrices result in the successive transformations. Multiplying a scaling matrix S with a rotation matrix R result in a matrix M which will rotate then scale a point P multiplied by it:

$$P' = M.P = S.R.P$$



1.1.3 2D translation and homogeneous coordinates

Can we translate a point with a 2x2? Answer is no, unfortunately:

$$\begin{cases} x' = x + t_x \\ y' = y + t_y \end{cases}$$

You can 't find a 2x2 matrix M translating a 2D point P into P' such as $P' = M.P$. But there is a workaround. Instead of representing points and vectors in a affine space with n Cartesian coordinates ($n=2$ for the 2D case) mathematicians have found we can represent these points and vectors with one more coordinate. As a result, one can express the translation and all affine transformations with a 3x3 matrix. Our 2D point or vector will now have a third coordinates w such as $P = [x\, y\, w]$. These coordinates $[x\, y\, w]$ are called homogeneous coordinates as opposed to Cartesian coordinates [x y]. The 3x3 matrix for translating the point P with homogeneous coordinates looks like this:

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

But how is this translating P anyway? As far as we know we end up with a point $P' = [x + w.t_x \quad y + w.t_y \quad w]$ and it can be hard to see the relation between this point and a more conventional point $[x\, y]$ in the affine space. So here it is, we need to convert the point in homogeneous coordinates to Cartesian coordinates. To do so, simply divide all components of the homogeneous point by its last component:

$$\overbrace{[x \quad y \quad w]}^{\text{homogenous point}} \rightarrow \overbrace{[x/w \quad y/w]}^{\text{Cartesian point}}$$

We also need to choose the value of w which is:

- ✔ $w = 0$ for vectors
- ✔ $w = 1$ for points

We can convert the point from the last example to Cartesian coordinates. We had in homogeneous coordinates:

$$P' = [x + w.t_x \quad y + w.t_y \quad w]$$

Which is now in Cartesian coordinates:

$$P' = [\frac{(x + 1.t_x)}{1} \quad \frac{(y + 1.t_y)}{1}] = [x + t_x \quad y + t_y]$$

Finally if our point was in fact a vector we would have:

$$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \end{bmatrix}$$

The vector stay the same. Obviously translating a vector gives the same vector. However it's a special cases to handle because we can't divide by $w = 0$. In this case the vector's Cartesian coordinates are simply the homogeneous coordinates and we omit the last component w.

To summaries instead of manipulating n Cartesian coordinates in a nD affine space we are going to use $n+1$ homogeneous coordinates in a nD projective space. This enables us to express transformations such as scaling translation rotation and even projection, with a matrix product. This is a unified representation of transformations which is extensively used in computer graphics.

Here is the homogeneous version of the rotation matrix and the scaling matrix in 2D:

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Exercise:

1) Do in this order the translation $[1 \ \ 3]$ and scaling $[0{,}5 \ \ 0{,}5]$ of the point $P=[2 \ \ 2]$ with homogeneous matrices.

2) With the homogeneous matrix $\begin{bmatrix} s_x & 0 & t_x \\ 0 & s_y & t_y \\ 0 & 0 & 1 \end{bmatrix}$ apply a translation and a scaling on point P (use the values in '1)'). The result is different why?

1.2 3D transformations

Finding the homogeneous matrices in 3D can be derived from the 2D examples. These derivations are not the point of this course so I won't go in too much details. You just need to know they exists and how to use them. For translation and scaling it is pretty straightforward:

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translation

Rotation matrices about the x, y and z are a little be more complex to infer. The easiest to derive is the rotation about the z axis, which is basically the same as the 2D rotation except that we don't change the 'z' component. The three matrices below compute the rotation of a 3D point about the origin for a rotation of magnitude θ:

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

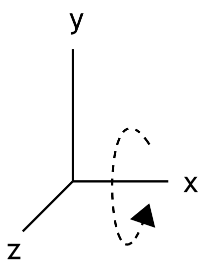
Rotation about the x axis

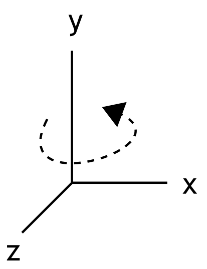
$$R_y = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

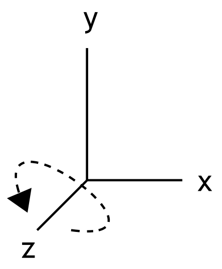
Rotation about the y axis

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

Rotation about the z axis

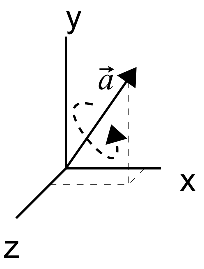






Usually we need a more general form of the rotation to rotate around an arbitrary axis. It is the matrix below (you can derive by composing R_x R_y R_z however you'll need to understand system coordinates changes presented in next section):

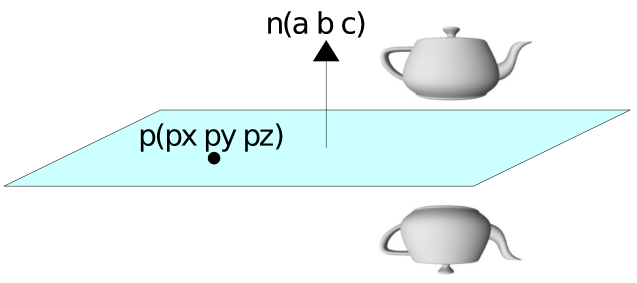
$$R_{xyz}=\begin{bmatrix} t.x^2+c & t.x.y-s.z & t.x.z+s.y & 0 \\ t.x.y+s.z & t.y^2+c & t.y.z-s.x & 0 \\ t.x.z-s.y & tyz+sx & t.z^2+c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Rotation about an arbitrary axis $\vec{a}=[x,y,z]$ from the origin with $c=\cos(\theta)$, $s=\sin(\theta)$, $t=1-\cos(\theta)$

Another useful transformation is the reflexion matrix. One can compute the reflexion of a point/vector according to a reflexion plane. (Finding the reflexion matrix looks a lot like the derivation of R_{xyz} and can be done with R_x R_y R_z and an inverse scaling):

$$\begin{bmatrix} 1-2a^2 & -2ab & -2ac & -2ad \\ -2ab & 1-2b^2 & -2bc & -2bd \\ -2ac & -2bc & 1-2c^2 & -2cd \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

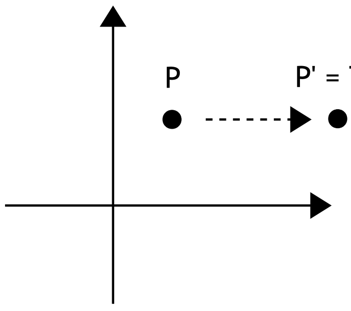


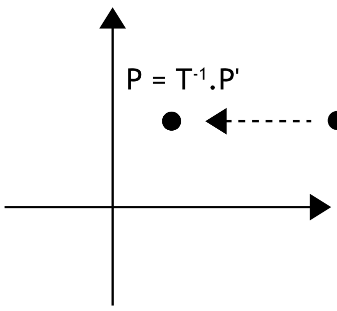
With $f(x,y,z)=ax+by+cz+d=0$ the equation of the plane where $\vec{n}=[a;b;c]$ is the normal to the plane. And d the shortest distance from the origin to the plane. N.B: $d=-\vec{n}.\vec{p}$ where $\vec{p}=[p_x;p_y;p_z]$ is a point on the plane.

If you want more details about how to derive these matrices you can look at the appendix.

1.3 Invert transformations

Sometime it is useful to find the inverse of a transformation. The inverse of a transformation A is another transformation A^{-1} that cancel the transformation A. For instance if we translate a 3D point with a homogeneous matrix T, we can translate it back to its previous position with the inverse transformation T^{-1}:





In the previous section, I have shown how we could merge different transformations just by multiplying them successively. So if we want to translate a point with the matrix T and then with T^{-1}we can do it with a single matrix M:

$$M=\begin{bmatrix}1 & 0 & 0 & t_x\\ 0 & 1 & 0 & t_y\\ 0 & 0 & 1 & t_z\\ 0 & 0 & 0 & 1\end{bmatrix}.\begin{bmatrix}1 & 0 & 0 & -t_x\\ 0 & 1 & 0 & -t_y\\ 0 & 0 & 1 & -t_z\\ 0 & 0 & 0 & 1\end{bmatrix}=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}=I$$

Here M is the matrix identity which means any point transformed by M will stay at the same position.

So for a general case if we transform a point by a homogeneous matrix A and we want to find the inverse transformation A^{-1}, the matrix A^{-1} has to satisfy this equality:

$$A.A^{-1}=I$$

What is worth noticing is this is how the inverse matrix is defined. Multiplying a matrix A by its inverse matrix A^{-1} equals the matrix identity. We now know how to compute the inverse transformation with matrices for <u>any</u> invertible transformations. We can find the inverse of a transformation A by inverting its matrix.

Lets see other examples. For the scaling transformation the inverse transformation can be found intuitively:

$$\begin{bmatrix}s_x & 0 & 0 & 0\\ 0 & s_y & 0 & 0\\ 0 & 0 & s_z & 0\\ 0 & 0 & 0 & 1\end{bmatrix}.\begin{bmatrix}1/s_x & 0 & 0 & 0\\ 0 & 1/s_y & 0 & 0\\ 0 & 0 & 1/s_z & 0\\ 0 & 0 & 0 & 1\end{bmatrix}=I$$

For the rotation about an angle θ we can rotate back by the opposite angle $-\theta$.

However intuition has its limits, sometimes you'll handle a matrix A whose transformations are unknown to you. In these cases you have to find numerically the inverse matrix A^{-1}. How to compute the inverse matrix is way out of the scope of this course so I'll be brief on the topic.

There is a lot ways you can compute the inverse of a matrix. You could resort to algorithms such as: Gauss-Jordan elimination, Gaussian elimination, or even a LU decomposition... Methods will highly differ depending on the size of your matrix (whether its an 4x4 or 1000x1000) and its density (if the majority of the coefficients are zero or not). Several libraries (solvers) implements various techniques to find the inverse of a matrix or solve linear system of equations. Each technique has its tradeoffs, some are faster for sparse matrix (matrix with a majority of null coefficients) other are more appropriate for dense matrix.

In computer graphics 4x4 matrices are extensively used, so we usually “hard code” in the source files the inversion for matrices of this particular size. Direct inversion of matrix that small is fast enough. So we can forget about the previously stated methods to solve the equation $A.A^{-1}=I$.

Here it is, how to compute the inverse of a general 4x4 matrix knowing its coefficients:

For $A=\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix}$ the inverse is $A^{-1}=\frac{1}{det}.\begin{bmatrix} c_0 & c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 & c_7 \\ c_8 & c_9 & c_{10} & c_{11} \\ c_{12} & c_{13} & c_{14} & c_{15} \end{bmatrix}$

With $det=a.c_0+b.c_4+c.c_8+d.c_{12}$ and:

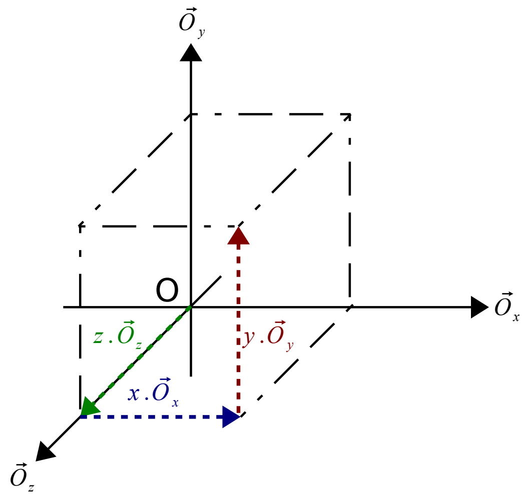
$$\left\{\begin{array}{rcl}
c0 & = & f.f_0-g.f_1+h.f_2 \\
c1 & = & -b.f_0+c.f_1-d.f_2 \\
c2 & = & b.f_3-c.f_4+d.f_5 \\
c3 & = & -b.f_6+c.f_7-d.f_8 \\
c4 & = & -e.f_0+g.f_9-h.f_{10} \\
c5 & = & a.f_0-c.f_9+d.f_{10} \\
c6 & = & -a.f_3+c.f_{11}-d.f_{12} \\
c7 & = & a.f_6-c.f_{13}+d.f_{14} \\
c8 & = & e.f_1-f.f_9+h.f_{15} \\
c9 & = & -a.f_1+b.f_9-d.f_{15} \\
c10 & = & a.f_4-b.f_{11}+d.f_{16} \\
c11 & = & -a.f_7+b.f_{13}-d.f_{17} \\
c12 & = & -e.f_2+f.f_{10}-g.f_{15} \\
c13 & = & a.f_2-b.f_{10}+c.f_{15} \\
c14 & = & -a.f_5+b.f_{12}-c.f_{16} \\
c15 & = & a.f_8-b.f_{14}+c.f_{17}
\end{array}\right.
\qquad
\left\{\begin{array}{c}
f_0=k.p-l.o \\
f_1=j.p-l.n \\
f_2=j.o-k.n \\
f_3=g.p-h.o \\
f_4=f.p-h.n \\
f_5=f.o-g.n \\
f_6=g.l-h.k \\
f_7=f.l-h.j \\
f_8=f.k-g.j \\
f_9=i.p-l.m \\
f_{10}=i.o-k.m \\
f_{11}=e.p-h.m \\
f_{12}=e.o-g.m \\
f_{13}=e.l-h.i \\
f_{14}=e.k-g.i \\
f_{15}=i.n-j.m \\
f_{16}=e.n-f.m \\
f_{17}=e.j-f.i
\end{array}\right.$$

Not all matrices are invertible so you'll have to check if the determinant '*det*' is not null. For those interested in the method used to determined the coefficients of A^{-1} you can search for techniques using the matrix of co-factor. In my case I just used a mathematical software (wxmaxima) which works like charm!

1.4 Coordinate systems

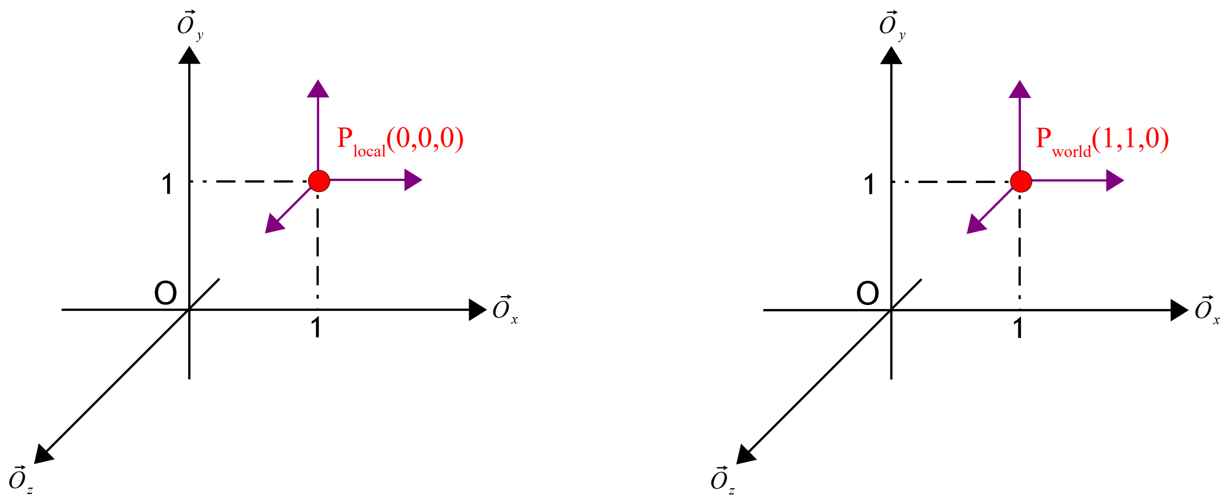
Its time talk a little more about coordinate systems, and especially how we can change from one coordinate system to another using matrices. But first lets define correctly what's a Cartesian coordinate system. The position of a 3D point with [x y z] Cartesian coordinates is defined as follow:

$$P = O + x.\vec{O}_x + y.\vec{O}_y + z.\vec{O}_z$$



The point O is the origin of the frame defined by the three orthogonal vectors $\vec{O}_x$, $\vec{O}_y$ and $\vec{O}_z$. Let call the world coordinates the point coordinates expressed with the frame $O=(0,0,0)$, $\vec{O}_x=(1,0,0)$, $\vec{O}_y=(0,1,0)$ and $\vec{O}_z=(0,0,1)$.

We can define another frame which origin and vectors coordinates are defined according to the world coordinates. I will call this frame the local frame. Point coordinates expressed within the local frame are called local coordinates:



In the above figure a point P can be either expressed in local coordinates (purple frame) or in the world coordinates (black frame). A common operation with matrices is to change a point coordinates from local to world coordinates. In the previous example to go from local coordinates to world coordinates we just have to do a translation:

$$P_{world}=\begin{bmatrix}1&0&0&1\\0&1&0&1\\0&0&1&0\\0&0&0&1\end{bmatrix}.P_{local}$$

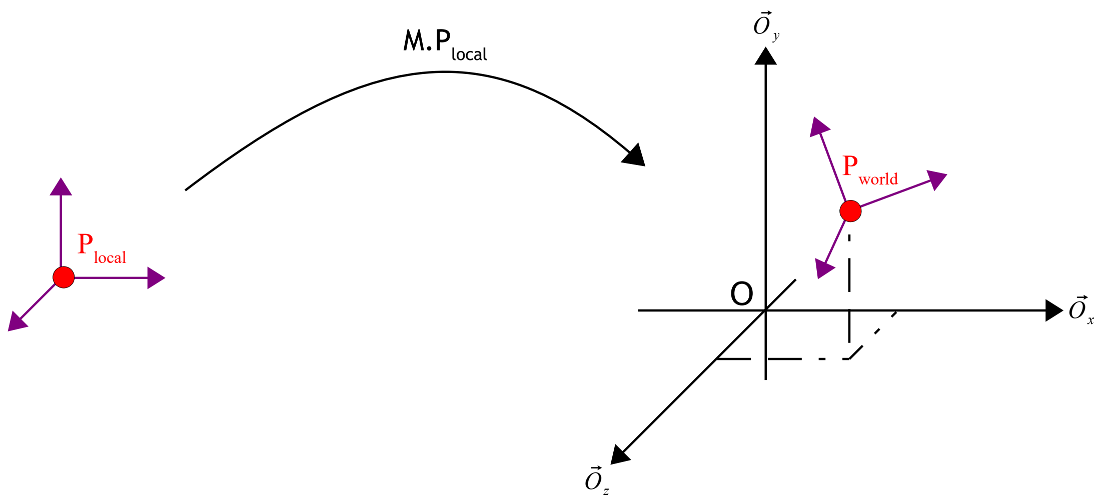
As we can see there is to possible interpretations with this matrix product. It's either the translation of a point, or a coordinate system change from local to world coordinates.

let O_l and L_x, L_y and L_z be respectively the origin and the vectors of a local frame expressed in the world coordinates. The point's $P_{local}(x,y,z)$ coordinates are defined according to this local frame. Finding P_{world} can be done as follow:

$$P_{world}=O_l+x.\vec{L}_x+y.\vec{L}_y+z.\vec{L}_z$$

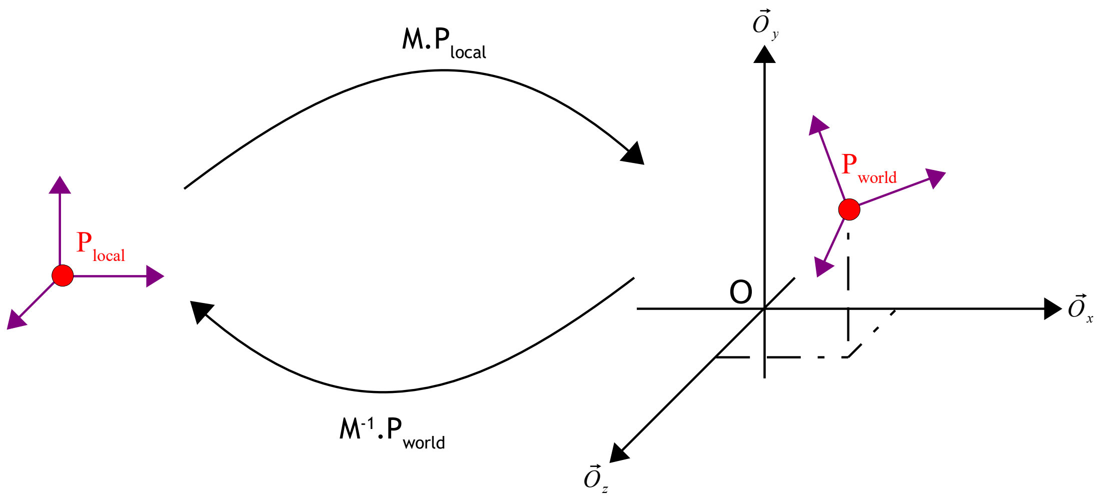
Which can be easily written with a matrix product:

$$P_{world}=M.P_{local}=\begin{bmatrix}L_{xx}&L_{yx}&L_{zx}&O_{lx}\\L_{xy}&L_{yy}&L_{zy}&O_{ly}\\L_{xz}&L_{yz}&L_{zz}&O_{lz}\\0&0&0&1\end{bmatrix}.P_{local}$$



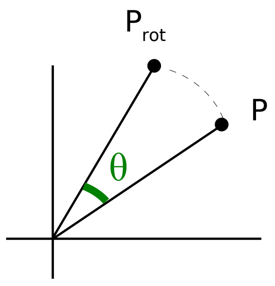
An interesting consequence with this interpretation is that we can go from world coordinates to local coordinates by inverting M:

$$P_{local} = M^{-1} . P_{world}$$



It's time to show why this coordinates system changes can be very convenient. I'll take an example in 2D to be more concise. So lets say we want to rotate a 2D point with a 3x3 homogeneous matrix. As you now know this is achieved with:

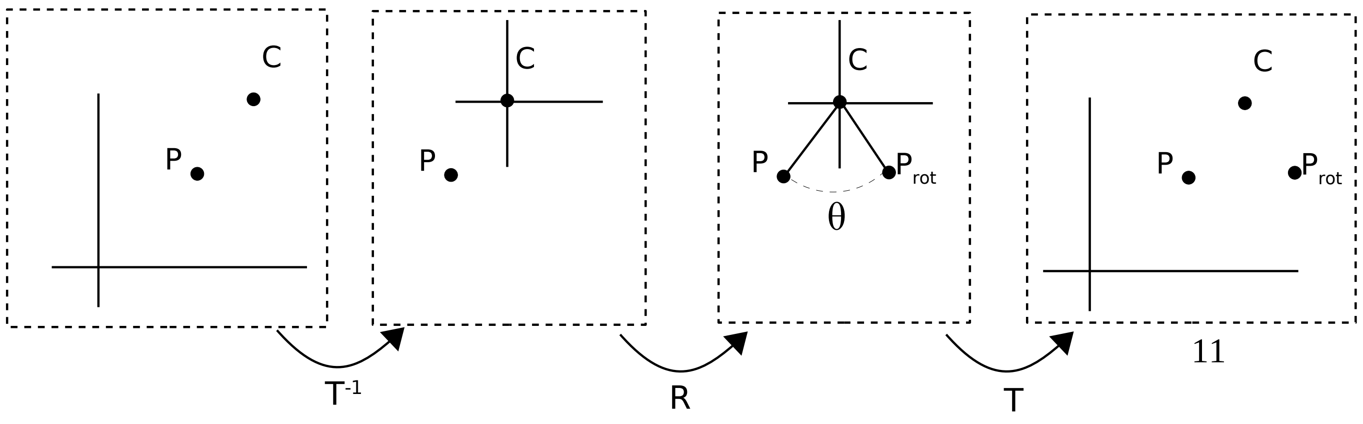
$$P_{rot} = R.P = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} . P$$



However this matrix R compute the rotation about the origin only. So if we want to rotate a point around another center of rotation we will have to change the coordinate system. Let C be the center of rotation doing a rotation around this point is done with:

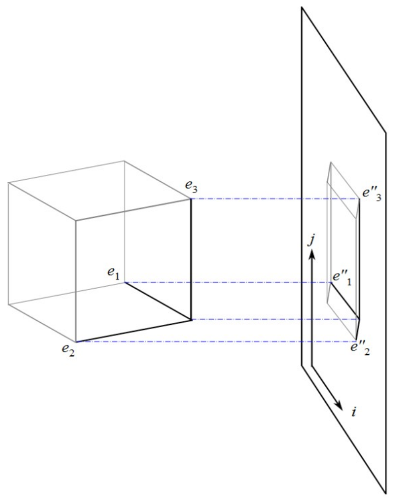
$$P_{rot} = (T.R.T^{-1}).P = \begin{bmatrix} 1 & 0 & C_x \\ 0 & 1 & C_y \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 1 & 0 & -C_x \\ 0 & 1 & -C_y \\ 0 & 0 & 1 \end{bmatrix} . P$$

So we translate in such way that C is now the center, achieve the rotation about the origin (which is C according to the world coordinates) and translate back. Otherwise said, we change the coordinate system to a local coordinate where C is the center, then rotate about the origin and go back to the world coordinate system:

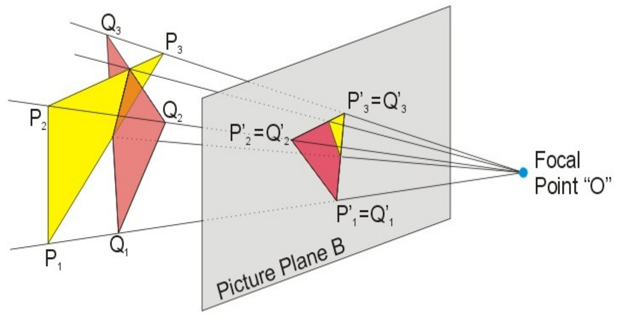


1.5 Projections

Now that we have seen all the mathematics for moving the objects lets see how we project them into the image plane. We are interested in two kind of projections orthographic and perspective:

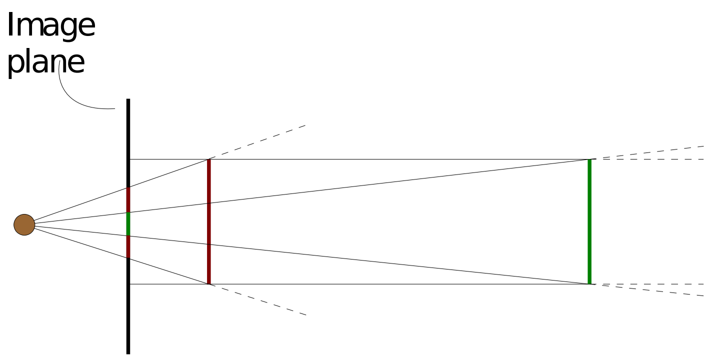


Orthographic

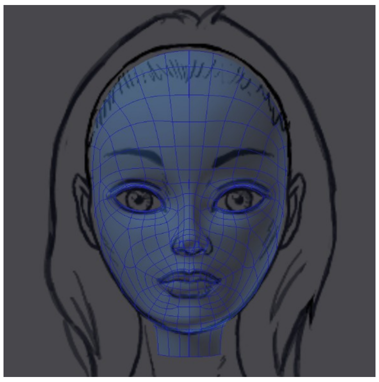


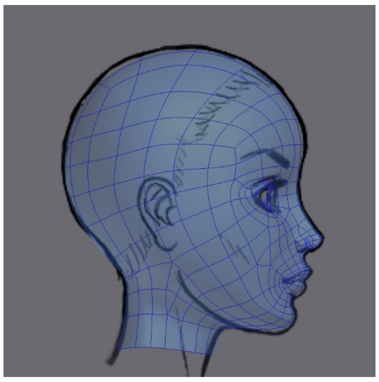
Perspective

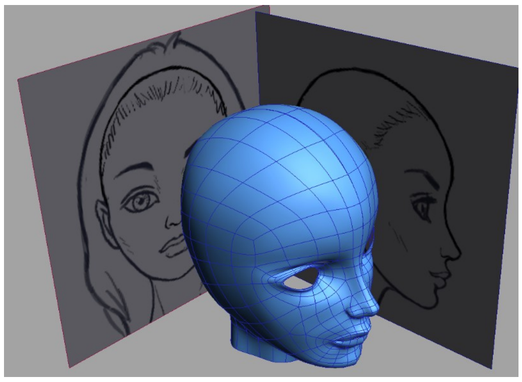
In orthographic projection all the projection lines are parallel between each other. This kind of projection is not very realistic, two identical objects at different depth will have the same size on the image plane. A human eye would see the furthest object smaller because the projection lines would meet at a single point:



Yet orthographic projection is very convenient for modeling 3D objects. A lot of modeler works with 2D guides to build their objects. These guides can only be used in an orthographic view:


Orthographic view: front


Orthographic view: left

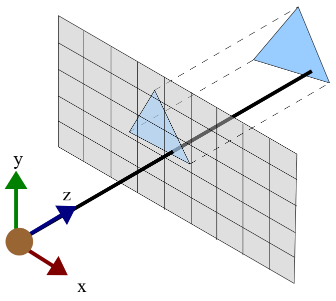

Perspective view

The user can iteratively go to front and left view and move the same point in 2D. The result is a correctly positioned point in 3D. This would not be possible if front end left views were a perspective projection.

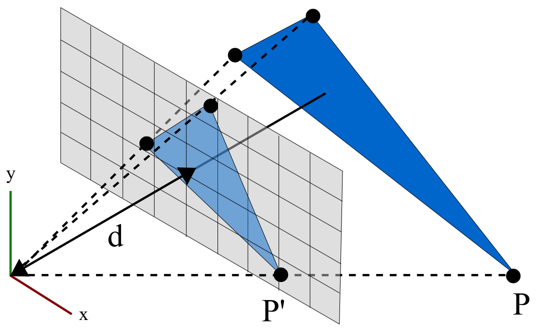
Finding matrices for perspective and orthogonal projection can be easier if we consider only special cases. For instance the orthogonal projection on an image plane parallel to the xy plane can be computed just by ignoring the z coordinate:

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

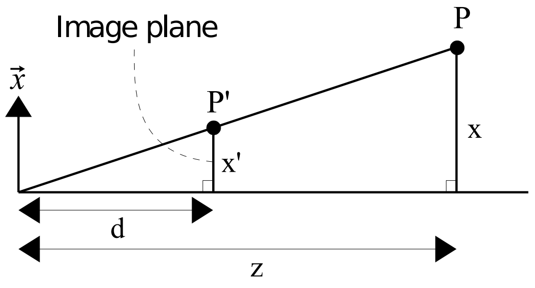
Projection matrix on the plane xy

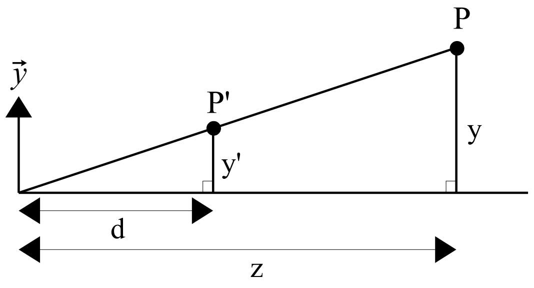


Finding the matrix for perspective projection involves a bit of trigonometry. Let's state the problem: we want to compute the perspective projection P' of a point P. As for the orthogonal case, we are going to project onto an image plane parallel to the plane xy and at a distance d:



Finding the position of $P'(x', y', z')$ according to $P(x, y, z)$ and d can be done with the Thales theorem. Consider the triangles in the *xz* plane and *yz* plane:





We deduce the following ratio equality:

$$\frac{d}{z}=\frac{x'}{x}, \frac{d}{z}=\frac{y'}{y}$$

This leads us to P' coordinates according to P and d:

$$\begin{cases} x'=x.(\frac{d}{z}) \\ y'=y.(\frac{d}{z}) \\ z'=d \end{cases} \text{ which can also be written } \begin{cases} x'=x/(\frac{z}{d}) \\ y'=y/(\frac{z}{d}) \\ z'=d \end{cases} \text{ (1)}$$

The last thing to do is to find a matrix $PProj_z$ which will do the projection of P. Here is the magic:

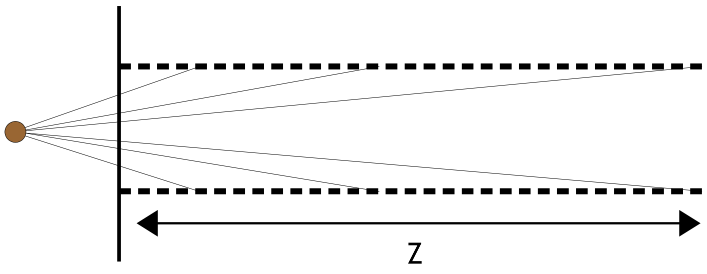
$$P'=PProj_z.P=\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}.\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}=\begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix}$$

Remember that to go from homogeneous coordinates to Cartesian we need to divide by the last component. Doing so in this case is called the perspective division and effectively project the point onto the image plane:

From homogeneous coordinates $P'=\begin{bmatrix} x \\ y \\ z \\ \frac{z}{d} \end{bmatrix}$ to Cartesian coordinates:

$$P'=\begin{bmatrix} x/(\frac{z}{d}) \\ y/(\frac{z}{d}) \\ z/(\frac{z}{d}) \end{bmatrix}=\begin{bmatrix} x/(\frac{z}{d}) \\ y/(\frac{z}{d}) \\ d \end{bmatrix}$$

This is another reason to use homogeneous coordinates. It enables us to compute a perspective, and represent points at an infinite distance from the camera. The further away a point the greater the homogeneous coordinates w. This implies that the further the object the smaller it will appear. Another consequence is that two parallel lines will seems to meet at a single point:



As the rail are going away there projection diminish.

In this course we will not need more general expression of the projections. We could try to find projection matrices according to an arbitrary image plane, but I'll leave that as an exercise. Actually OpenGL only uses the special cases presented here. As we are going to see, instead of projecting according to a camera position and viewing direction OpenGL will simply transform the entire scene to simulate the camera movements. The scene is always projected for a camera at $(0,0,0)$ looking towards $-z$.

1.6 Transformations in OpenGL

1.6.1 Pipeline

We have seen all the theory about transformations now its time to practice. In this section I'll give a program which draws a cube, then I'll describe how OpenGL transform the 3D vertex into 2D window coordinates. Here is the program:

(avec un exemple de prog)

comment marche opengl → (multiplication à chaque glTrans)

1.6.2 Stack of matrices

modélisation hierarchique (commutativité)

pipeline de transformation (pourquoi on clip de -w à w) ce qui explique ndc -1 1

mapping du z et utilisation glDepthRange. Lire le z buffer (autre chapitre peut être dans pipeline)

1 Texture Matrix (GL_TEXTURE)

Texture coordinates (s, t, r, q) are multiplied by GL_TEXTURE matrix before any texture mapping. By default it is the identity, so texture will be mapped to objects exactly where you assigned the texture coordinates. By modifying GL_TEXTURE, you can slide, rotate, stretch, and shrink the texture.

```
// rotate texture around X-axis

glMatrixMode(GL_TEXTURE);

glRotatef(angle, 1, 0, 0);
```

2 Color Matrix (GL_COLOR)

The color components (r, g, b, a) are multiplied by GL_COLOR matrix. It can be used for color space conversion and color component swaping. GL_COLOR matrix is not commonly used and is requiredGL_ARB_imaging extension.

3 Other Matrix Routines

glPushMatrix() :
push the current matrix into the current matrix stack.
glPopMatrix() :
pop the current matrix from the current matrix stack.
glLoadIdentity() :
set the current matrix to the identity matrix.
glLoadMatrix{fd}(m) :
replace the current matrix with the matrix m.
glLoadTransposeMatrix{fd}(m) :
replace the current matrix with the row-major ordered matrix m.
glMultMatrix{fd}(m) :
multiply the current matrix by the matrix m, and update the result to the current matrix.
glMultTransposeMatrix{fd}(m) :
multiply the current matrix by the row-major ordered matrix m, and update the result to the current matrix.
glGetFloatv(GL_MODELVIEW_MATRIX, m) :
return 16 values of GL_MODELVIEW matrix to m.