Tutorials for OpenGL 3.3 and later

Tutorial 3 : Matrices

> The engines don't move the ship at all. The ship stays where it is and the engines move the universe around it.
> *Futurama*

This is the single most important tutorial of the whole set. Be sure to read it at least eight times.

Homogeneous coordinates

Until then, we only considered 3D vertices as a (x,y,z) triplet. Let's introduce w. We will now have (x,y,z,w) vectors.

This will be more clear soon, but for now, just remember this :

> If w == 1, then the vector (x,y,z,1) is a position in space.
> If w == 0, then the vector (x,y,z,0) is a direction.

(In fact, remember this forever.)

What difference does this make ? Well, for a rotation, it doesn't change anything. When you rotate a point or a direction, you get the same result. However, for a translation (when you move the point in a certain direction), things are different. What could mean "translate a direction" ? Not much.

Homogeneous coordinates allow us to use a single mathematical formula to deal with these two cases.

Transformation matrices

An introduction to matrices

Simply put, a matrix is an array of numbers with a predefined number of rows and colums. For instance, a 2×3 matrix can look like this :

$$\begin{bmatrix} 2 & 5 & 7 \\ 9 & 8 & 1 \end{bmatrix}$$

In 3D graphics we will only use 4×4 matrices. They will allow us to transform our (x,y,z,w) vertices. This is done by multiplying the vertex with the matrix :

Matrix x Vertex (in this order !!) = TransformedVertex

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} ax+by+cz+dw \\ ex+fy+gz+hw \\ ix+jy+kz+lw \\ mx+ny+oz+pw \end{bmatrix}$$

This isn't as scary as it looks. Put your left finger on the a, and your right finger on the x. This is *ax*. Move your left finger to the next number (b), and your right finger to the next number (y). You've got *by*. Once again : *cz*. Once again : *dw*. ax + by + cz + dw. You've got your new x ! Do the same for each line, and you'll get your new (x,y,z,w) vector.

Now this is quite boring to compute, an we will do this often, so let's ask the computer to do it instead.

In C++, with GLM:

```
glm::mat4 myMatrix;
glm::vec4 myVector;
// fill myMatrix and myVector somehow
glm::vec4 transformedVector = myMatrix * myVector; // Again, in this order ! this is
```

In GLSL :

```
mat4 myMatrix;
vec4 myVector;
// fill myMatrix and myVector somehow
vec4 transformedVector = myMatrix * myVector; // Yeah, it's pretty much the same thar
```

(have you cut'n pasted this in your code ? go on, try it)

Translation matrices

These are the most simple tranformation matrices to understand. A translation matrix look like this :

$$\begin{bmatrix} 1 & 0 & 0 & X \\ 0 & 1 & 0 & Y \\ 0 & 0 & 1 & Z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where X,Y,Z are the values that you want to add to your position.

So if we want to translate the vector (10,10,10,1) of 10 units in the X direction, we get :

$$\begin{bmatrix} 1 & 0 & 0 & 10 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 10 \\ 10 \\ 10 \\ 1 \end{bmatrix} = \begin{bmatrix} 1*10+0*10+0*10+10*1 \\ 0*10+1*10+0*10+0*1 \\ 0*10+0*10+1*10+0*1 \\ 1*10+0*10+0*10+0*1 \end{bmatrix} = \begin{bmatrix} 10+0+0+10 \\ 0+10+0+0 \\ 0+0+10+0 \\ 10+0+0+0 \end{bmatrix} = \begin{bmatrix} 20 \\ 10 \\ 10 \\ 1 \end{bmatrix}$$

(do it ! doooooo it)

… and we get a (20,10,10,1) homogeneous vector ! Remember, the 1 means that it is a position, not a direction. So our transformation didn't change the fact that we were dealing with a position, which is good.

Let's now see what happens to a vector that represents a direction towards the -z axis : (0,0,-1,0)

$$\begin{bmatrix}1 & 0 & 0 & 10\\0 & 1 & 0 & 0\\0 & 0 & 1 & 0\\0 & 0 & 0 & 1\end{bmatrix} \times \begin{bmatrix}0\\0\\-1\\0\end{bmatrix} = \begin{bmatrix}1*0+0*0+0*-1+10*0\\0*0+1*0+0*-1+0*0\\0*0+0*0+1*-1+0*0\\1*0+0*0+0*-1+0*0\end{bmatrix} = \begin{bmatrix}0+0+0+0\\0+0+0+0\\0+0-1+0\\0+0+0+0\end{bmatrix} = \begin{bmatrix}0\\0\\-1\\0\end{bmatrix}$$

… ie our original (0,0,-1,0) direction, which is great because as I said ealier, moving a direction does not make sense.

So, how does this translate to code ?

In C++, with GLM:

```
#include <glm/transform.hpp> // after <glm/glm.hpp>

glm::mat4 myMatrix = glm::translate(10,0,0);
glm::vec4 myVector(10,10,10,0);
glm::vec4 transformedVector = myMatrix * myVector; // guess the result
```

In GLSL : Well, in fact, you almost never do this. Most of the time, you use glm::translate() in C++ to compute your matrix, send it to GLSL, and do only the multiplication :

```
vec4 transformedVector = myMatrix * myVector;
```

The Identity matrix

This one is special. It doesn't do anything. But I mention it because it's as important as knowing that multiplying A by 1.0 gives A.

$$\begin{bmatrix}1 & 0 & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 1 & 0\\0 & 0 & 0 & 1\end{bmatrix} \times \begin{bmatrix}x\\y\\z\\w\end{bmatrix} = \begin{bmatrix}1*x+0*y+0*z+0*w\\0*x+1*y+0*z+0*w\\0*x+0*y+1*z+0*w\\0*x+0*y+0*z+1*w\end{bmatrix} = \begin{bmatrix}x+0+0+0\\0+y+0+0\\0+0+z+0\\0+0+0+w\end{bmatrix} = \begin{bmatrix}x\\y\\z\\w\end{bmatrix}$$

In C++ :

```
glm::mat4 myIdentityMatrix = glm::mat4(1.0);
```

Scaling matrices

Scaling matrices are quite easy too :

$$\begin{bmatrix}x & 0 & 0 & 0\\0 & y & 0 & 0\\0 & 0 & z & 0\\0 & 0 & 0 & 1\end{bmatrix}$$

So if you want to scale a vector (position or direction, it doesn't matter) by 2.0 in all directions :

$$\begin{bmatrix}2 & 0 & 0 & 0\\0 & 2 & 0 & 0\\0 & 0 & 2 & 0\\0 & 0 & 0 & 1\end{bmatrix} \times \begin{bmatrix}x\\y\\z\\w\end{bmatrix} = \begin{bmatrix}2*x+0*y+0*z+0*w\\0*x+2*y+0*z+0*w\\0*x+0*y+2*z+0*w\\0*x+0*y+0*z+1*w\end{bmatrix} = \begin{bmatrix}2*x+0+0+0\\0+2*y+0+0\\0+0+2*z+0\\0+0+0+1*w\end{bmatrix} = \begin{bmatrix}2*x\\2*y\\2*z\\w\end{bmatrix}$$

and the w still didn't change. You may ask : what is the meaning of "scaling a direction" ? Well, often, not much, so you usually don't do such a thing, but in some (rare) cases it can be handy.
(notice that the identity matrix is only a special case of scaling matrices, with (X,Y,Z) = (1,1,1). It's also a special case of translation matrix with (X,Y,Z)= (0,0,0), by the way)

In C++ :

```
// Use #include <glm/gtc/matrix_transform.hpp> and #include <glm/gtx/transform.hpp>
glm::mat4 myScalingMatrix = glm::gtx::transform::scale(2,2,2);
```

Rotation matrices

These are quite complicated. I'll skip the details here, as it's not important to know their exact layout for everyday use. For more information, please have a look to the Matrices and Quaternions FAQ (popular resource, probably available in your language as well).

In C++ :

```
// Use #include <glm/gtc/matrix_transform.hpp> and #include <glm/gtx/transform.hpp>
glm::vec3 myRotationAxis( ??, ??, ??);
glm::gtx::transform::rotate( angle_in_degrees, myRotationAxis );
```

Cumulating transformations

So now we know how to rotate, translate, and scale our vectors. It would be great to combine these transformations. This is done by multiplying the matrices together, for instance :

```
TransformedVector = TranslationMatrix * RotationMatrix * ScaleMatrix * OriginalVector;
```

!!! BEWARE !!! This lines actually performs the scaling FIRST, and THEN the rotation, and THEN the translation. This is how matrix multiplication works.

Writing the operations in another order wouldn't produce the same result. Try it yourself :
- make one step ahead (beware of your computer) and turn left;
- turn left, and make one step ahead

As a matter of fact, the order above is what you will usually need for game characters and other items : Scale it first if needed; then set its direction, then translate it. For instance, given a ship model (rotations have been removed for simplification) :

The wrong way :

- You translate the ship by (10,0,0). Its center is now at 10 units of the origin.
- You scale your ship by 2. Every coordinate is multiplied by 2 *relative to the origin*, which is far away… So you end up with a big ship, but centered at 2*10 = 20. Which you don't want.

The right way :

- You scale your ship by 2. You get a bug ship, centered on the origin.
- You translate your ship. It's still the same size, and at the right distance.

Matrix-matrix multiplication is very similar to matrix-vector multiplication, so I'll once again skip some details and redirect you the the Matrices and Quaternions FAQ if needed. For now, we'll simply ask the computer to do it :

in C++, with GLM :

```
glm::mat4 myModelMatrix = myTranslationMatrix * myRotationMatrix * myScaleMatrix;
glm::vec4 myTransformedVector = myModelMatrix * myOriginalVector;
```

in GLSL :

```
mat4 transform = mat2 * mat1;
vec4 out_vec = transform * in_vec;
```

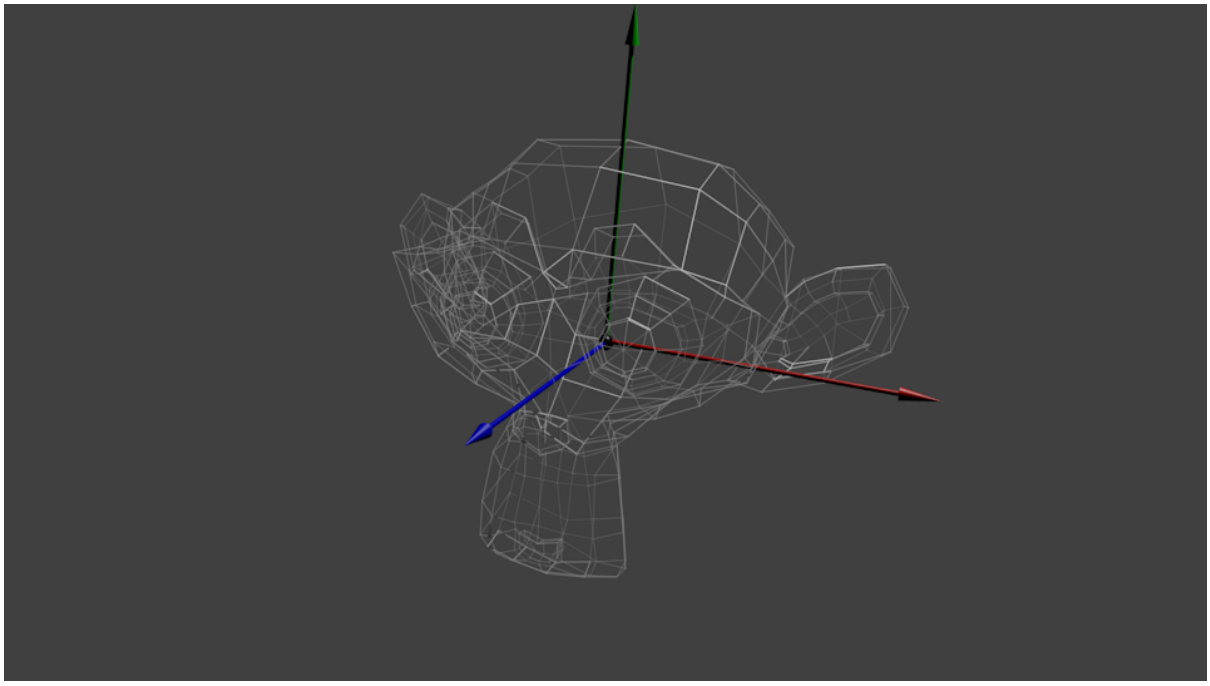
The Model, View and Projection matrices

For the rest of this tutorial, we will suppose that we know how to draw Blender's favourite 3d model : the monkey Suzanne.

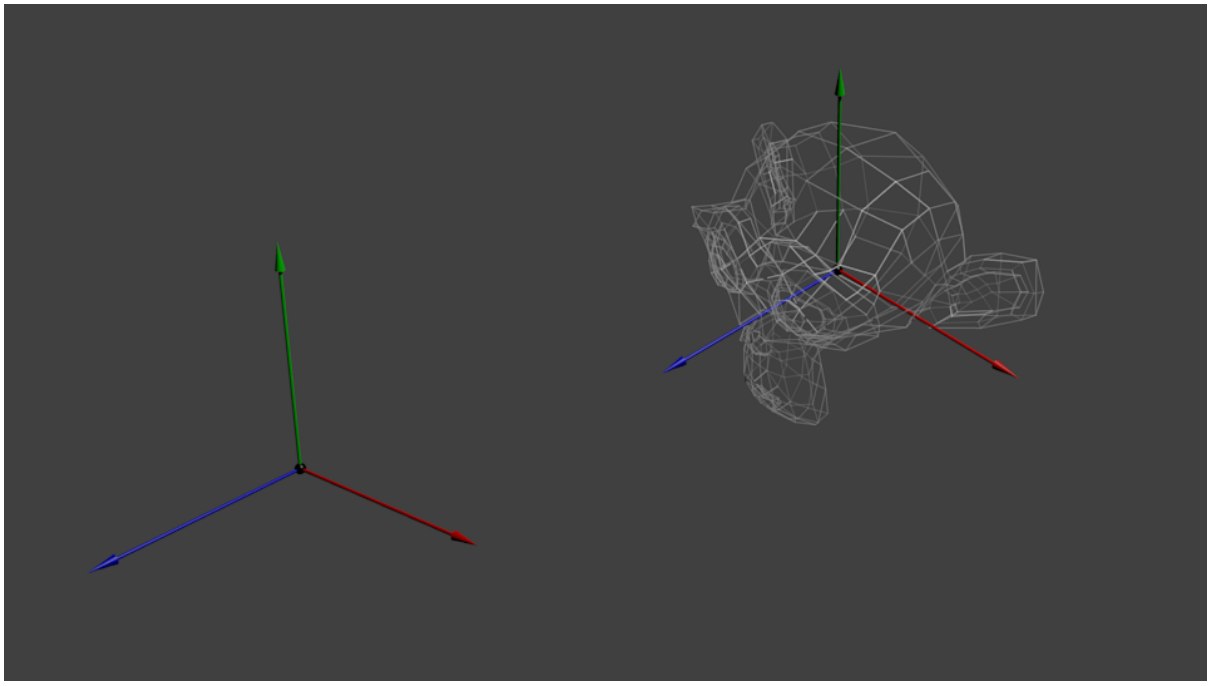
The Model, View and Projection matrices are a handy tool to separate transformations cleanly. You may not use this (after all, that's what we did in tutorials 1 and 2). But you should. This is the way everybody does, because it separates everything cleanly, as we'll see.

The Model matrix

This model, just as our beloved red triangle, is defined by a set of vertices. The X,Y,Z coordinates of these vertices are defined relative to the object's center : that is, if a vertex is at (0,0,0), it is at the center of the object.

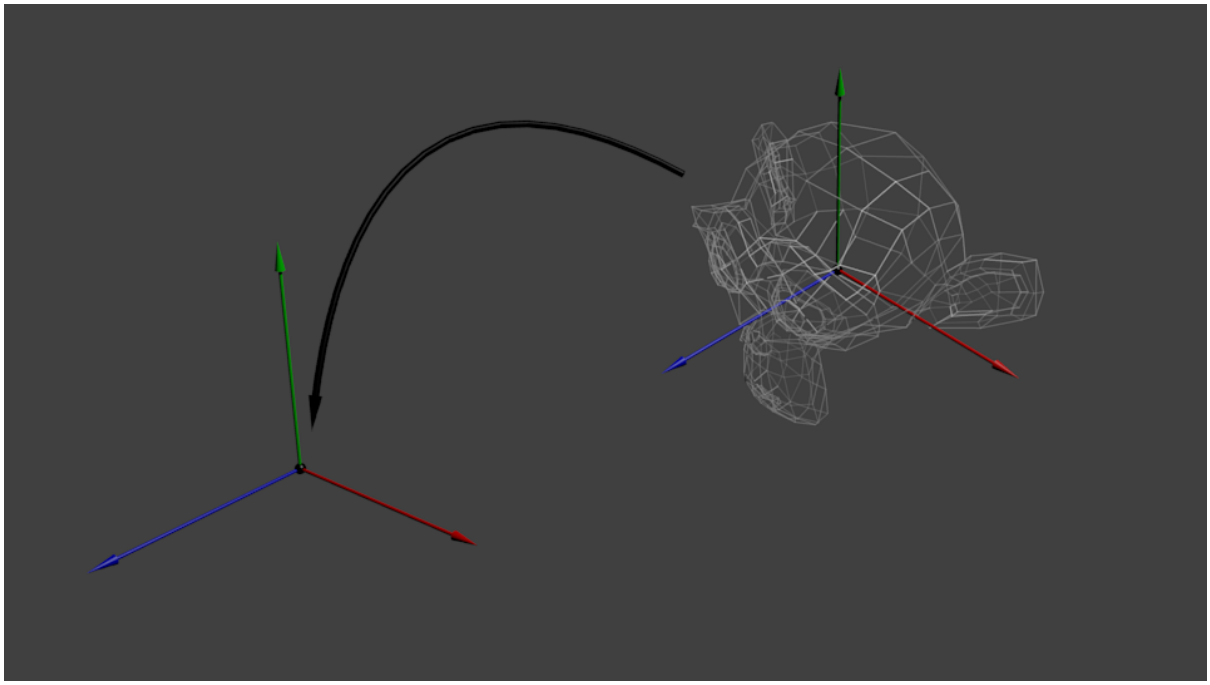


We'd like to be able to move this model, maybe because the player controls it with the keyboard and the mouse. Easy, you just learnt do do so : scale*rotation*translation, and done. You apply this matrix to all your vertices at each frame (in GLSL, not in C++!) and everything moves. Something that doesn't move will be at the _center of the world_.

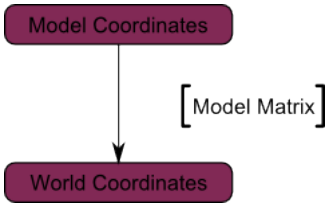


Your vertices are now in *World Space*. This is the meaning of the black arrow in the image below : *We went from Model Space (all vertices defined*

relatively to the center of the model) to World Space (all vertices defined relatively to the center of the world).



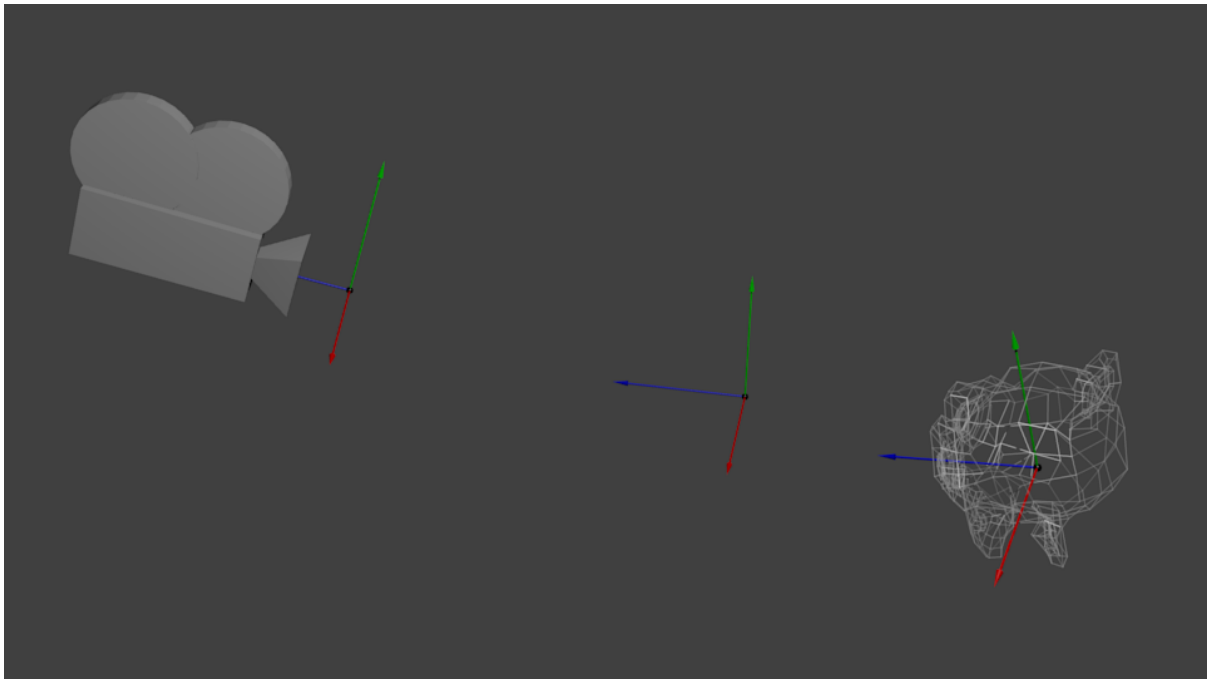
We can sum this up with the following diagram :



The View matrix

Let's quote Futurama again :

> The engines don't move the ship at all. The ship stays where it is and the engines move the universe around it.

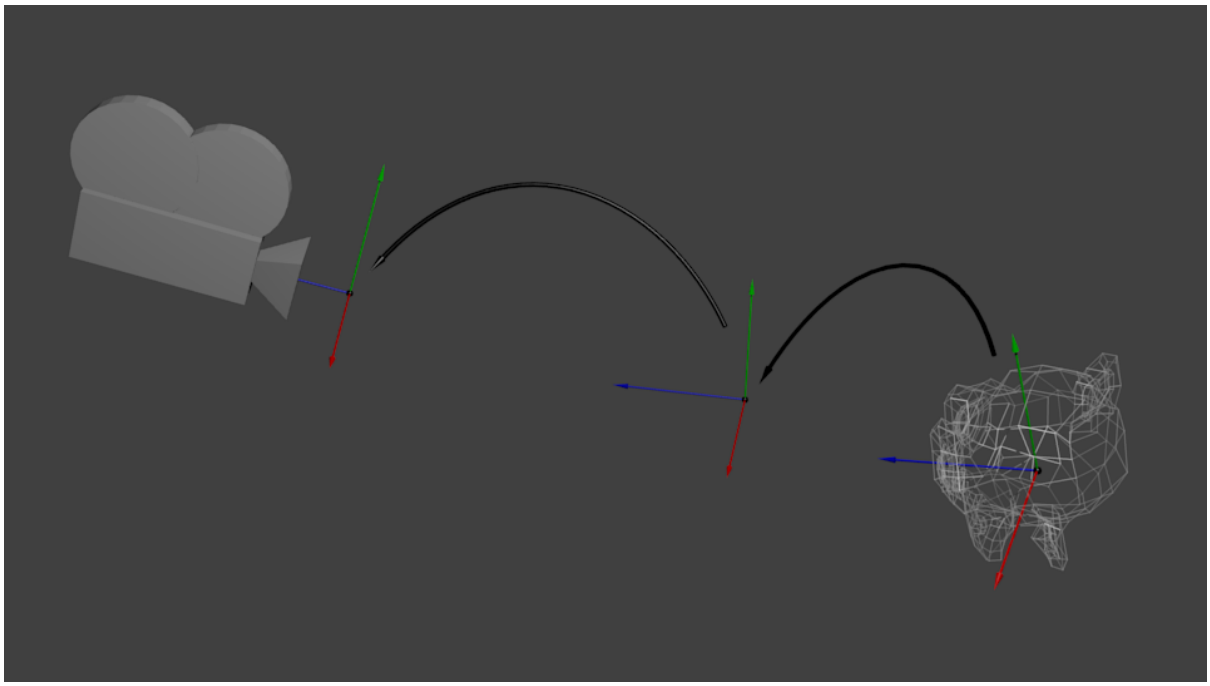


When you think about it, the same applies to cameras. It you want to view a moutain from another angle, you can either move the camera… or move the mountain. While not practical in real life, this is really simple and handy in Computer Graphics.

So initially your camera is at the origin of the World Space. In order to move the world, you simply introduce another matrix. Let's say you want to move your camera of 3 units to the right (+X). This is equivalent to moving your whole world (meshes included) 3 units to the LEFT ! (-X). While you brain melts, let's do it :

```
// Use #include <glm/gtc/matrix_transform.hpp> and #include <glm/gtx/transform.hpp>
glm::mat4 ViewMatrix = glm::gtx::transform::translate(-3,0,0);
```

Again, the image below illustrates this : *We went from World Space (all vertices defined relatively to the center of the world, as we made so in the previous section) to Camera Space (all vertices defined relatively to the camera).*

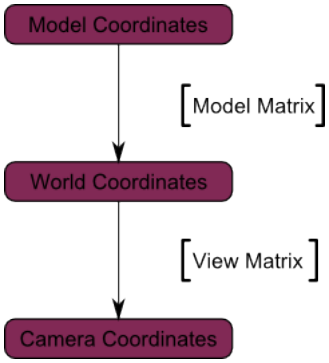


Before you head explodes from this, enjoy GLM's great glm::LookAt function:

```
glm::mat4 CameraMatrix = glm::LookAt(
    cameraPosition, // the position of your camera, in world space
    cameraTarget,   // where you want to look at, in world space
```

```
    upVector        // probably glm::vec3(0,1,0), but (0,-1,0) would make you looking
);
```

Here's the compulsory diagram :

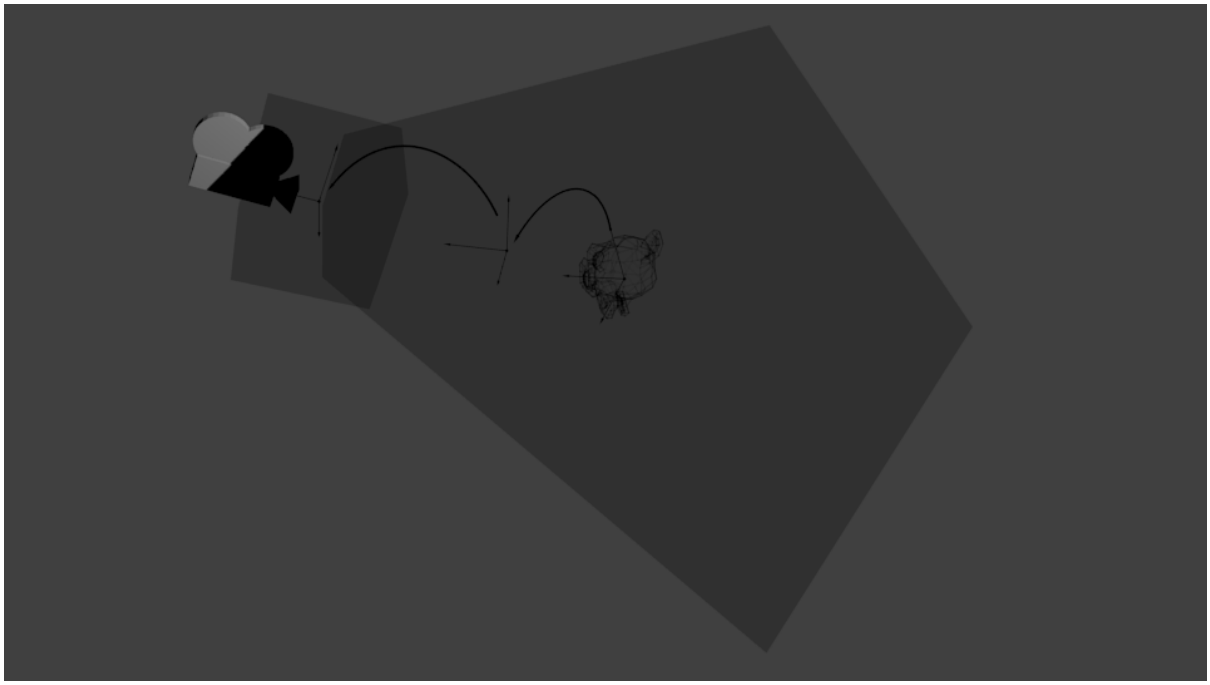


This is not over yet, though.

The Projection matrix

We're now in Camera Space. This means that after all theses transformations, a vertex that happens to have x==0 and y==0 should be rendered at the center of the screen. But we can't use only the x and y coordinates to determine where an object should be put on the screen : its distance to the camera (z) counts, too ! For two vertices with similar x and y coordinates, the vertex with the biggest z coordinate will be more on the center of the screen than the other.

This is called a perspective projection :



And luckily for us, a 4×4 matrix can represent this projection[1] :

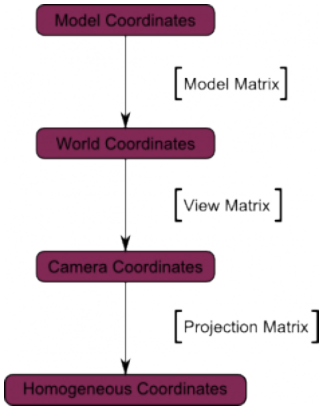
```
// Generates a really hard-to-read matrix, but a normal, standard 4x4 matrix nonethe
glm::mat4 projectionMatrix = glm::perspective(
    FoV,         // The horizontal Field of View, in degrees : the amount of "zoom".
    4.0f / 3.0f, // Aspect Ratio. Depends on the size of your window. Notice that 4/3
    0.1f,        // Near clipping plane. Keep as big as possible, or you'll get prec
```

```
6       100.0f       // Far clipping plane. Keep as little as possible.
7   );
```

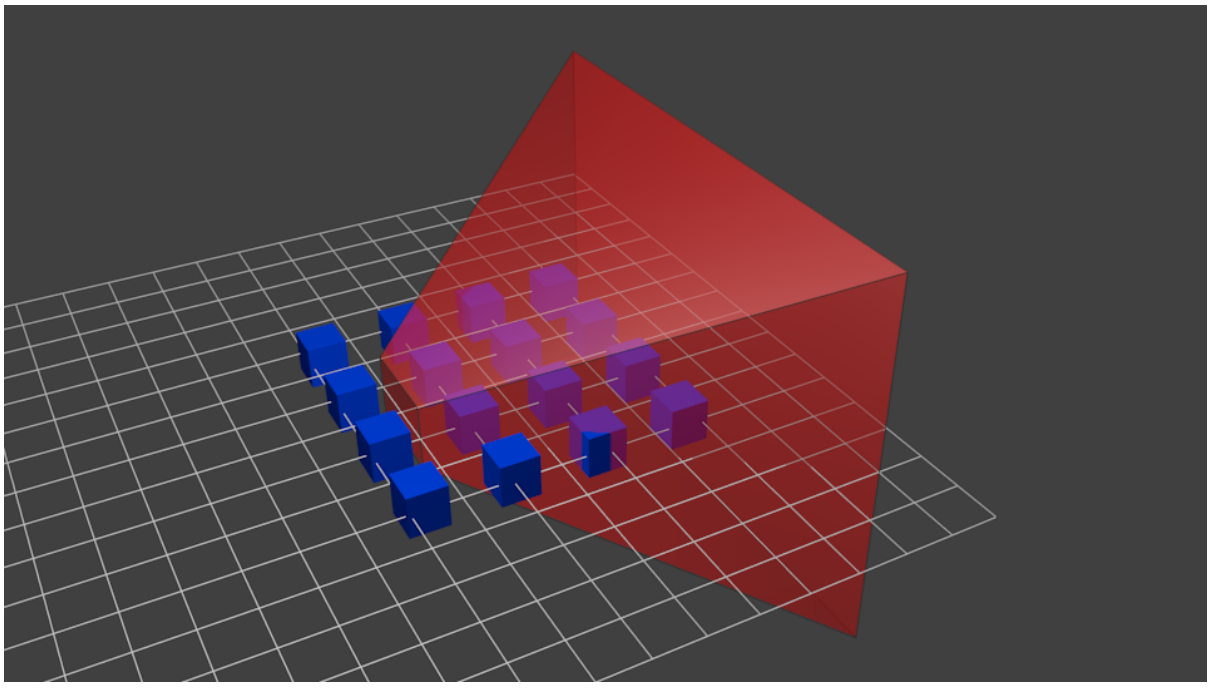
One last time :

We went from Camera Space (all vertices defined relatively to the camera) to Homogeneous Space (all vertices defined in a small cube. Everything inside the cube is onscreen).

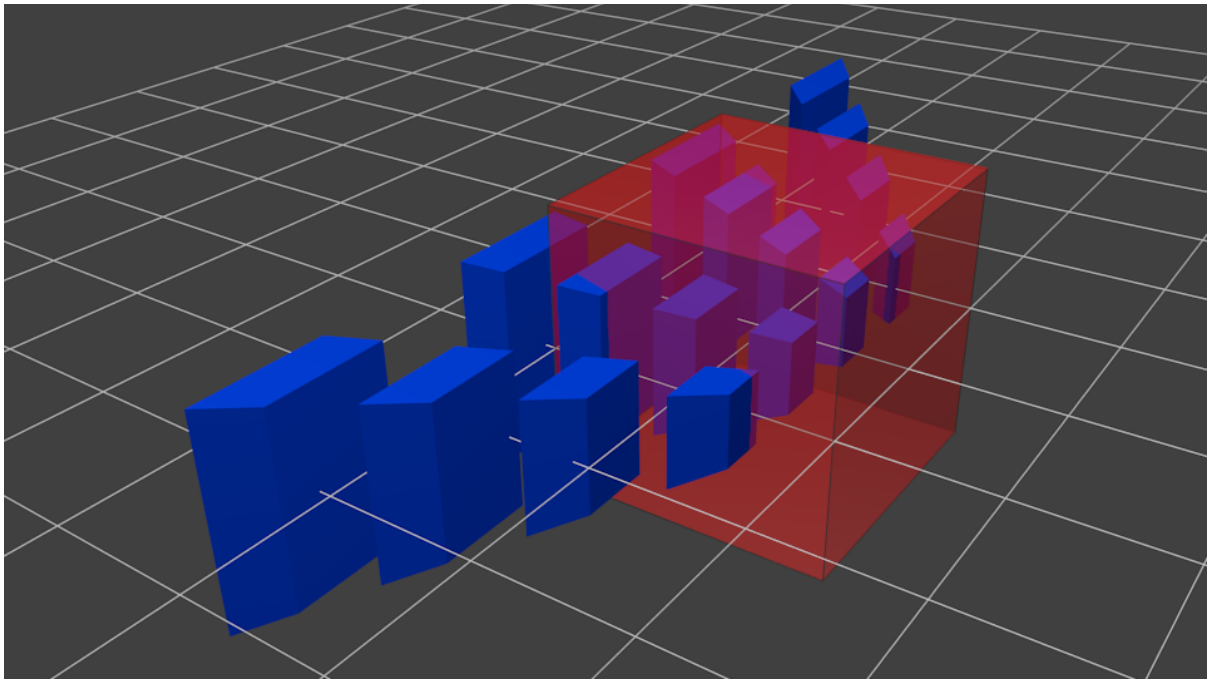
And the final diagram :



Here's another diagram so that you understand better what happens with this Projection stuff. Before projection, we've got our blue objects, in Camera Space, and the red shape represents the frustum of the camera : the part of the scene that the camera is actually able to see.

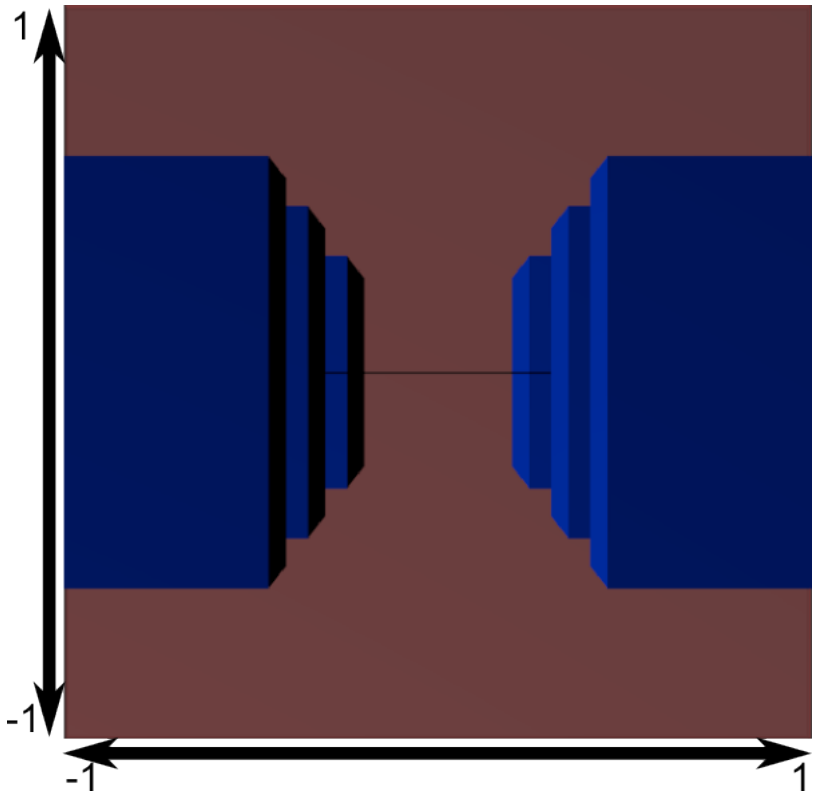


Multiplying everything by the Projection Matrix has the following effect :

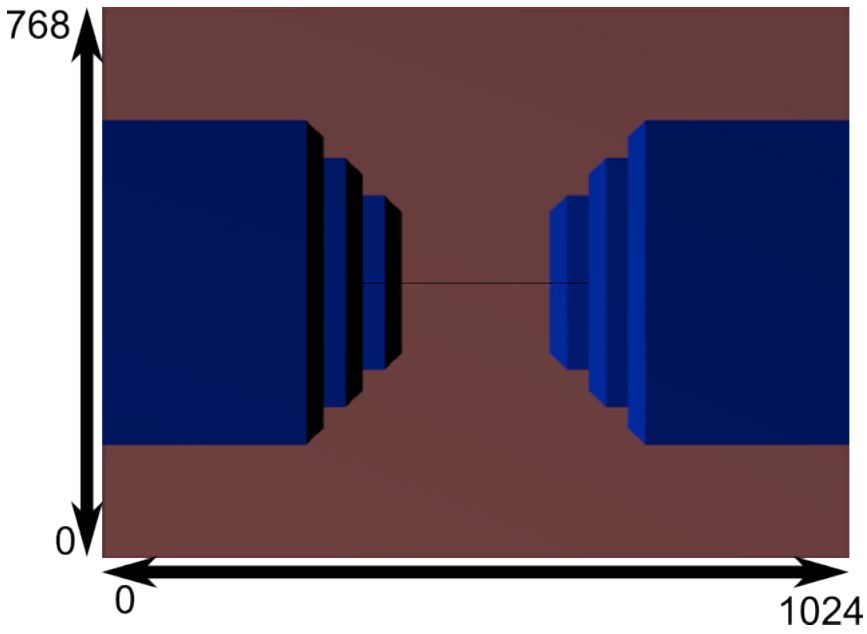


In this image, the frustum is now a perfect cube (between -1 and 1 on all axes, it's a little bit hard to see it), and all blue objects have been deformed in the same way. Thus, the objects that are near the camera (= near the face of the cube that we can't see) are big, the others are smaller. Seems like real life !

Let's see what it looks like from the "behind" the frustum :



Here you get your image ! It's just a little bit too square, so another mathematical transformation is applied to fit this to the actual window size :



And this is the image that is actually rendered !

Cumulating transformations : the ModelViewProjection matrix

... Just a standard matrix multiplication as you already love them !

```
// C++ : compute the matrix
glm::mat3 MVPmatrix = projection * view * model; // Remember : inverted !
```

```
// GLSL : apply it
transformed_vertex = MVP * in_vertex;
```

Putting it all together

First step : generating our MVP matrix. This must be done for each model you render.

```
// Projection matrix : 45° Field of View, 4:3 ratio, display range : 0.1 unit <-> 
glm::mat4 Projection = glm::perspective(45.0f, 4.0f / 3.0f, 0.1f, 100.0f);
// Camera matrix
glm::mat4 View       = glm::lookAt(
    glm::vec3(4,3,3), // Camera is at (4,3,3), in World Space
    glm::vec3(0,0,0), // and looks at the origin
    glm::vec3(0,1,0)  // Head is up (set to 0,-1,0 to look upside-down)
);
// Model matrix : an identity matrix (model will be at the origin)
glm::mat4 Model      = glm::mat4(1.0f);  // Changes for each model !
// Our ModelViewProjection : multiplication of our 3 matrices
glm::mat4 MVP        = Projection * View * Model; // Remember, matrix multiplicatio
```

Second step : give it to GLSL

```
// Get a handle for our "MVP" uniform.
// Only at initialisation time.
GLuint MatrixID = glGetUniformLocation(programID, "MVP");

// Send our transformation to the currently bound shader,
// in the "MVP" uniform
// For each model you render, since the MVP will be different (at least the M part)
glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP[0][0]);
```

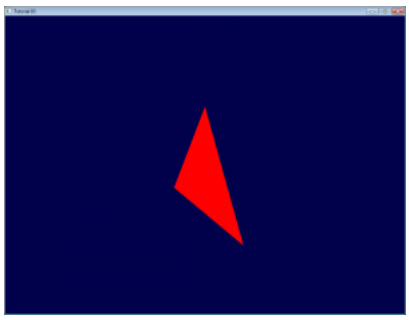
Third step : use it in GLSL to transform our vertices

```
in vec3 vertexPosition_modelspace;
uniform mat4 MVP;

void main(){

    // Output position of the vertex, in clip space : MVP * position
    vec4 v = vec4(vertexPosition_modelspace,1); // Transform an homogeneous 4D vecto
    gl_Position = MVP * v;
}
```

Done ! Here is the same triangle as in tutorial 2, still at the origin (0,0,0), but viewed in perspective from point (4,3,3), heads up (0,1,0), with a 45° field of view.



In tutorial 6 you'll learn how to modify these values dynamically using the keyboard and the mouse to create a game-like camera, but first, we'll learn how to give our 3D models some colour (tutorial 4) and textures (tutorial 5).

Exercises

Try changing the glm::perspective
Instead of using a perspective projection, use an orthographic projection (glm::ortho)
Modify ModelMatrix to translate, rotate, then scale the triangle
Do the same thing, but in different orders. What do you notice ? What is the "best" order that you would want to use for a character ?

Addendum

1 : [...]luckily for us, a 4×4 matrix can represent this projection[1] : Actually, this is not correct. A perspective transformation is not affine, and as such, can't be represented entirely by a matrix. After beeing multiplied by the ProjectionMatrix, homogeneous coordinates are divided by their own W component. This W component happens to be -Z (because the projection matrix has been crafted this way). This way, points that are far away from the origin are divided by a big Z; their X and Y coordinates become smaller; points become more close to each other, objects seem smaller; and this is what gives the perspective.
