

2 years ago · 3 Comments

Cameras on OpenGL ES 2.x - The ModelViewProjection Matrix (/cameras-on-opengl- es-2-x/)



Hello my friends!

In this article I'll talk about a very very important part of the 3D world. As you already know, this world behind of our devices' screen is just an attempt to recreate the beauty and complexity of the human eye. To do that we use cameras, which is in the real world the simulation of the human eye. To construct cameras we use mathematical equations.

In this article I'll treat about those cameras and equations behind it, the difference between convex and concave lenses, what are projections, matrices, quaternions and finally the famous Model View Projection Matrix. If you have some doubt, you know, just ask, I'll be glad to help.

Here is a little list of contents to help you to find something you want in this tutorial.

List of Contents to this Tutorial

- Cameras in the real world
- 3D history
- Projections
- Cameras in the 3D world
- The code behind the 3D world
 - Matrices
 - Matrices in Deep
 - Quaternions
- The code behind the 3D cameras
- Conclusion

At a glance

First let's see the basic about cameras, how it works in real world, the lens differences, how a zoom works, translations, rotations and some similar concepts. Right after consolidate these concepts let's enter deeply in the OpenGL and understand how all of that can fit in our application. So we finally go to the code, I'll give you the equations and explain how they work.

Do you remember from my OpenGL serie of tutorials when I said that Khronos has delegated many responsibilities and got focus on the most import part of the 3D world? (more precisely from the part 1, if not, you can [check it here \(http://blog.db-in.com/all-about-opengl-es-2-x-part-1\)](http://blog.db-in.com/all-about-opengl-es-2-x-part-1))

Well, from OpenGL ES 1.x to 2.x the cameras was one of those responsibilities that Khronos delegated. So now we must to create the cameras by ourselves. And wanna know? I Love it! With the shaders behavior we can have an amazing control on our applications, more than this, we are free to construct awesome 3D engines.

With the OpenGL controlling the cameras, we had only two or three kinds of cameras. But when we started programming the cameras by ourselves, we are able to create any kind of cameras. In this article I'll talk about the basic cameras: Orthogonal Camera and Perspective Camera.

OK, let's start!

Cameras in the real world

[top](#)

The human's eye is as convex lens, it converge the image to form the upside-down image on retina. Usually the camera's lens is formed by multiple lenses convexes and concaves ones, but the final image looks like more a convex lens, just like the human's eye.

The final image depend on many factors, not just the type of the lens, but in general words, the image bellow shows how a picture looks like behind each kind of lens.



Original Image



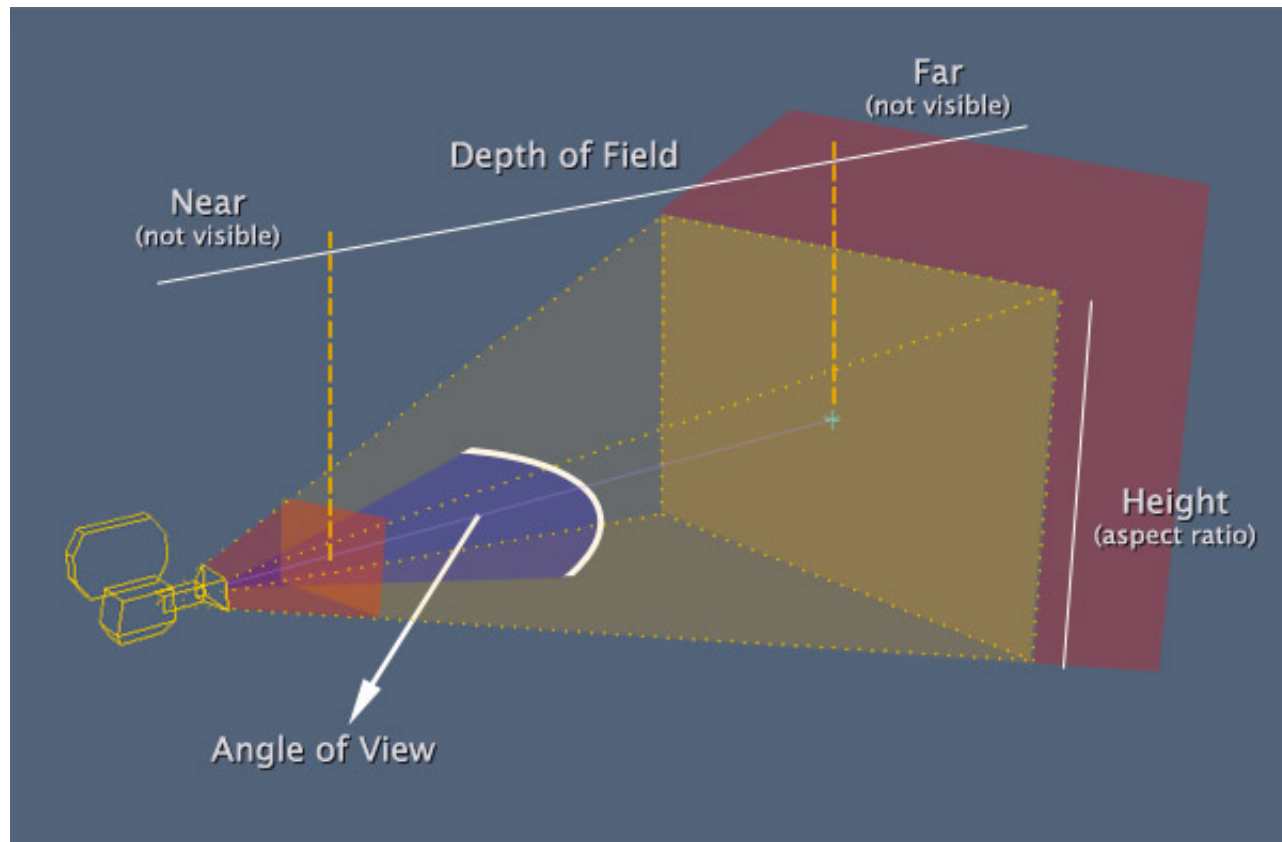
Behind a Convex Lens



Behind a Concave Lens

Both kinds could produce an image equal to the original one, I mean, with a tiny angle of distortion, depending on the distance of the object from the lens and the angle of view. The next image will show the most

important attributes of a camera.



The red areas in the image above are not visible to the camera, so any fragment in those areas will be clipped. The "Depth of Field" is the visible area, all fragments inside it will be visible. Commonly the word "Depth of Field" is also used to describe an special effect, the effect of Lens Blur. As the human's eyes have focus which make the object outside the focus seems blurred, the Lens Blur effect simulate that focus, making objects outside the focus seems blurred. So why I didn't put the attribute "Focus" on the image above? Because the focus is a special feature in just some cameras, the basic cameras in 3D doesn't implements focus behavior. The other important attribute is the "Angle of View", which represents the horizontal angle visible to the camera. Any fragment outside this angle will not be visible to the camera. Some times this "Angle of View" is also used to represent the vertical area, but usually we prefer to define the aspect ratio of the final image using the width and height.

The modern cameras are very accurate and can produce awesome effects using that attributes and combining the lenses types. Now let's back to our virtual world and see how we can transpose those attributes and behaviors mathematically. But before moving to the 3D cameras, we need to understand a little bit more about the maths in 3D world.

Short history about 3D world

[top](#)

Our grandpa of 3D world is Euclid, also known as Euclid of Alexandria. He lived in 323–283 BC (whoa, is a little bit old!) in the Greek city Alexandria. Euclid created what we use until today called Euclidean Space and Euclidean Geometry, I'm sure you heard these names before. Basically Euclidean Space is formed by 3 planes which give us the axis X, Y and Z. Each of those planes uses the traditional geometry, which has a lot of contribution from another greek, Pythagoras (570BC - 495 BC). Well it's not hard to figure out why Euclid developed their concepts, you know, the greeks love architecture and in order to construct perfect forms they needed to make all the calculus in a 3D imaginary world, without talk about their phylosophy and their passion by science.

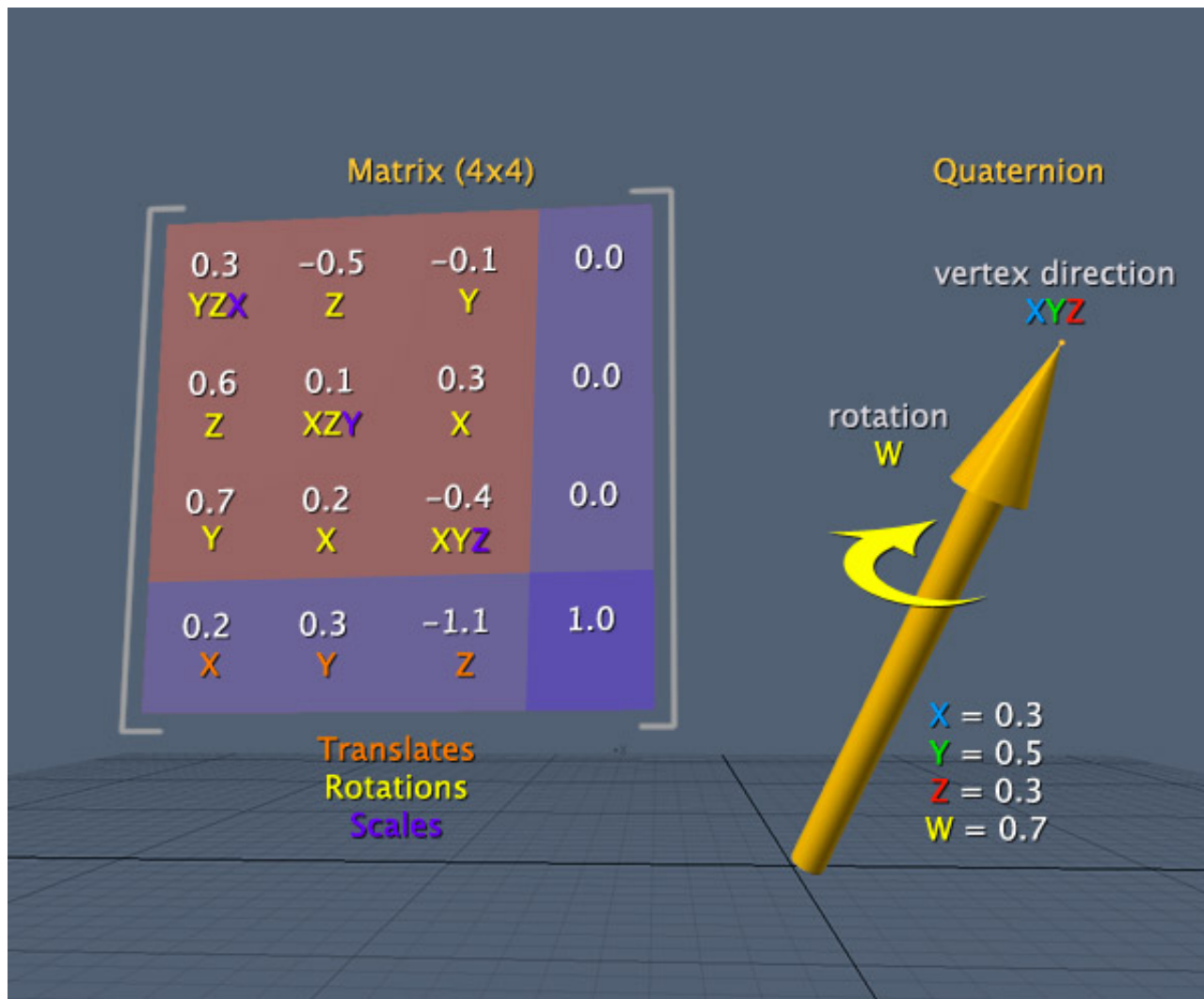
Advancing many years ahead in our Time Machine, we out in the beginning of 17th century, where a great man called René Descartes created something called cartesian coordinate system. That was amazing! It has created the bridge between the Euclid's theory and the linear algebra, introducing the matrices into Euclidean Transformations (translate, scale and rotation). Euclidean Transformations was made with traditional Pythagoras approaches, so you can imagine how many

calculations was there, but thanks to Descartes we are able to make Euclidean Transformations using matrices. Is simple, is fast, is pure beauty! Matrices in the 3D world are awesome!

But the matrices with Euclidean Transformations were not perfect. They produce some problems, the biggest one is related to rotations and is called Gimbal Lock. It happens when you try to rotate a plane and unintentionally the other two planes touche themselves, so the next rotation of one of those two planes will produce the Gimbal Lock, that means, it will involuntary rotate both locked axis. Many years later another great man called Sir William Rowan Hamilton, in 1843, created a method to deal with Euclidean Rotations and avoiding the Gimbal Lock, Hamilton created something called Quaternions! Quaternion is faster, better and the most elegant way to deal with 3D rotations. A Quaternion is composed by an imaginary part (complex number) and a real part. As in the 3D world we always use calculations with unity vectors (vectors with their magnitude/length equals to 1) we can discard the imaginary part of Quaternions and work only with the real numbers. Precisely, Quaternions was a Hamilton's thesis which include much more than just 3D rotations, but to us and to 3D world, the principal application is to deal with rotations.

OK, what does all of that have to do with cameras? It's simple, based on all this we start using 4×4 matrices to deal with Euclidean Transformations and use a vector with 4 elements to describe a point the space (X,Y,Z,W). The W is an Homogeneous Coordinate element. I'll not talk about it here, but just to let you know Homogeneous Coordinates was created by August Ferdinand Möbius in 1827 to deal with the concept of infinity in the Cartesian System. We'll talk about Möbius contribution later on, but shortly, the concept of infinity is very complex to fit into the cartesian system, we could use an complex imaginary

number for it, but this is not so good to real calculations. So to solve this problem, Möbius just added one variable W, which is a real number, and took us back to the world of real numbers. Anyway. The point is that a matrix 4x4 fits perfectly with vector 4 and as we use a single matrix to make the Euclidean Transformations into our 3D world, we think that it could be a good idea to use the same 4x4 matrix to deal with a camera in the 3D world.



The image above shows how a Matrix 4x4 and a Quaternions looks like visually. As you saw, the matrix has 3 independent slots for translation (position X,Y,Z), but the other instructions are mixed in its red area. Each rotation (X,Y and Z) affects 4 slots and each scale (X,Y and Z) affects 1 slot. A quaternion have 4 real numbers, 3 of then represent vertex (X,Y and Z) and this vertex forms a direction. The fourth value represents the rotation around its own pivot. We'll talk about quaternion later, but one

of its coolest features is that we can extract a rotation matrix from it. We can do so by constructing a Matrix 4x4 with only the yellow elements filled up.

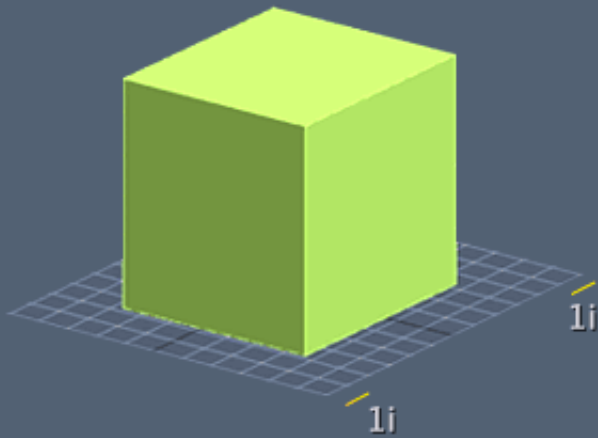
You could think now: "WTF!". Calm down, the practice is not so hard as the theory! Before put hands on code, we need to take just one more concept: the projections.

Projections

[top](#)

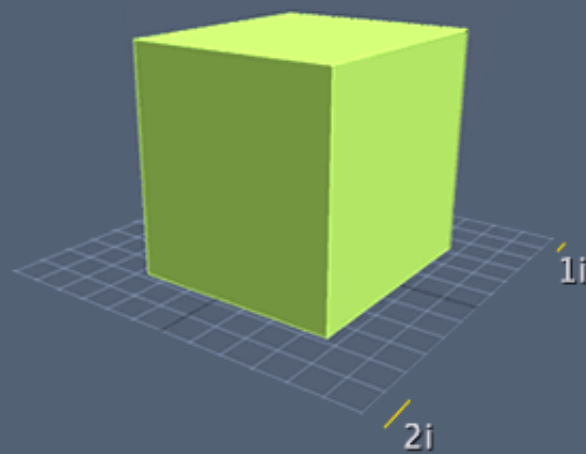
Instead of explaining this in a technically manner to you, I'll just show you! I'm sure you already know the difference of both projections types, maybe with another names, but I'm sure you know what that means:

Orthographic



- Everything seems equal
- No Vanish-Point
- Parallel lines never touch

Perspective



- Closest things seems bigger
- Has Vanish-Point
- Parallel lines touch at infinity



Did you see? It's very simple, the Orthographic projection is commonly used in 2D games, like Sim City or The Sims (old versions) or even the best seller Diablo (except the new Diablo III, which is really using the Perspective projection). The Orthographic projection doesn't exist in real world, it's impossible to the human's eyes receive the images at this way, because there is always has a vanish point on the images formed by our eyes. So the real cameras always capture the image with a Perspective projection.

Many people ask me about 2D graphics with OpenGL, well, here is my first tip about how to do it, you will need an Orthographic projection to create games like Sim City or The Sims. Games like Starcraft use a Perspective projection simulating an Orthographic projection. Is it possible? Yes! As everything related to a lens behavior, the final image lies on many factors, for example, a Perspective projection with a great Angle of View could seem more like an Orthographic projection, I mean, it's like to look to the ground from an airplane on air. From that distance, the cities look like a mockup and the vanish point seems no effect.



Before continuing we need to make a little digression to understand in deep the difference between those two projections. You remember from René Descartes and its cartesian system, right? In the linear algebra two parallel lines never touch, even at the infinity. How we could deal with the infinity idea in the linear algebra? Using calculations with ∞ (infinity denotation)? It's not useful. To create a Perspective projection we really need a vanish point and with it two parallel lines must touch at the infinity. So, how we can solve that? We can't! At least not using the linear algebra knowledge. We'll need something else.

Thanks to a man called August Ferdinand Möbius we can deal with this little problem. This man created something called "Homogeneous



Coordinates".

The idea is incredibly simple, is unbelievable (just as I like). Möbius just add 1 last coordinate number to any dimensional system the coordinate w . The 2D becomes 2D + 1 ($x,y \rightarrow x,y,w$), the 3D becomes 3D + 1 ($x,y,z \rightarrow x,y,z,w$). In the space calculations we just divide our original value by w , just it! Look at this pseudo code:

```
// This is the original cartesian coordinates. x = 3.0; y = 1.5; // This is  
new homogeneous coordinate. w = 1.0; // Space calculations. finalX =  
x/w; finalY = y/w;
```

The w will be 1.0 in the majority of cases. It will change just to represent the ∞ (infinity), in this case w will be 0.0! "WTF!!! Divisions by 0?" Not exactly. The w is often used to solve systems of two equations, so if we got a 0, the projection will be sent to the infinity. I know, seems confused in theory, but a simple practical example is the generation of shadows. When we have a light in a 3D scene, a light at the infinity or without attenuation, like a sun light in 3D world, we can simple create the shadows generated by that light using the w equals to 0. At this way the shadow will be projected on a wall or a floor exactly as the original

model is. Obviously the lights and shadows in the real world is stupidly more complex than this, but remember that in our virtual 3D world we are just kidding of copy the real behavior. With some few more steps we can simulate a shadow behavior more realistic, this is very good to a professional 3D software implements in its render, but to a game is not a good solution, more realistic shadow takes a lot of processing on the CPU and GPU. To a game, casts the shadows using the Möbius is pretty simple and looks very nice to the player!

OK, this is all about Projections, now let's move to the OpenGL and see how we can implement all these concepts with our code. Matrices and Quaternions will be our allies.

Cameras in the 3D world

[top](#)

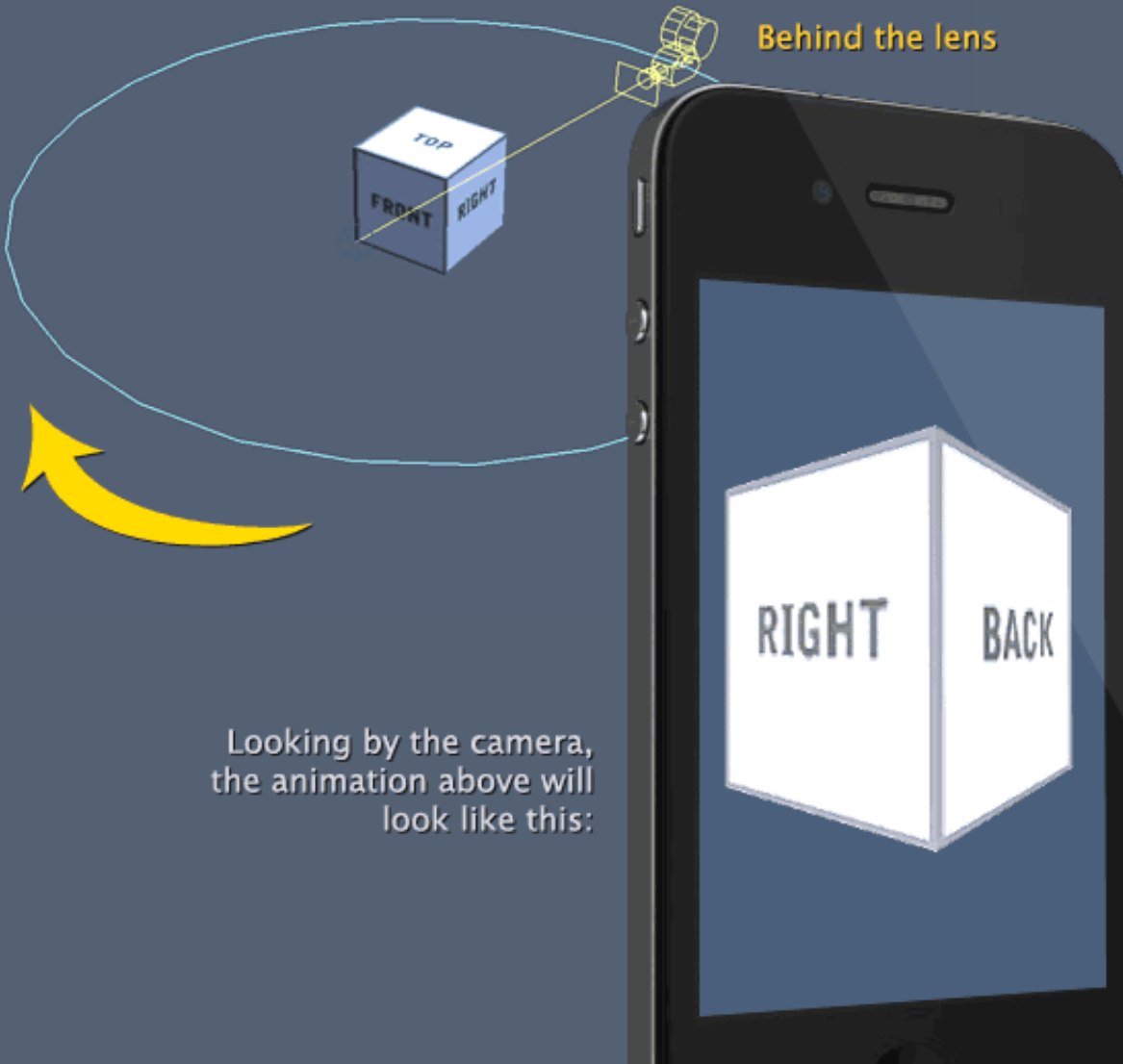
The first thing we need to understand is about how the transformations happen in the 3D world. Once we defined a 3D object structure, its structure will remain intact (structure is vertices, texture coordinates and normals). What will change at frame by frame will be just some matrices (often is just one matrix). Those matrices will produce temporary changes based on the original structures. So bear it in mind, "the original structure will never change"!

Thereby, when we rotate an object on the screen, in deeply, what we are doing is creating a matrix which contains information to make that rotation factor happen. Then in our shaders, when we multiply that matrix by the vertices of our object, on the screen, the object seems to be

rotating. The same is true for any other 3D elements, like lights or cameras. But the camera object has a special behavior, all the transformations on it **must be inverted**. The examples below can help to understand this issue:

Real transformation

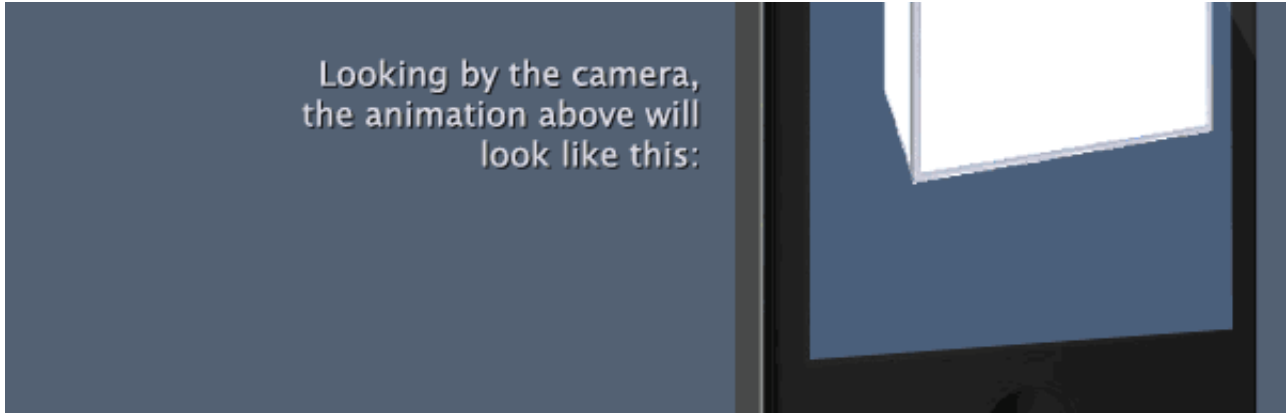
Behind the lens



Real transformation

Behind the lens





Looking by the camera,
the animation above will
look like this:

Notice in the pictures above that the resulting image on the device's screen is the same in both cases. This behavior drives us to an idea: every camera movement is inverted compared to the object space. For example, if the camera goes into +Z axis, this will produce the same effect as sending the 3D object to -Z axis. Rotate the camera in +Y will have the same effect as rotating the 3D object in its local -Y axis. So hold this idea, **every transformation at the camera will be inverted**, we'll use this soon.

Now the next concept about camera is how to interact the local space with the world space. In the examples above, if we think in rotating the object to -Y in local, this will produce the same results as rotating the camera at +Y and moving it in X and Y axis around the object, assuming the camera as the pivot. To deal with it, again the operations with matrices will save our day. To change from local space rotations to global space rotation all that we need is to change the order of the matrices in a multiplication ($A \times B = \text{local}$, $B \times A = \text{global}$, remember the matrices multiplication is not commutative). So we must **multiply the camera's matrix by the object's matrix**, in this order.

OK, I know, techniques seem very confusing, but trust me, the code is much simpler than you imagine. Let's review those concepts and dive into code.

- We never change the object structure, what changes is just some matrices, which will be multiplied by the original object's structure to achieve the desired result.
- On the camera, all transformations should be inverted before to construct the matrix.
- As the camera will be our eyes in the 3D world, we assume the camera is always the local space, so the final matrix will be the resulting of CameraMatrix X ObjectMatrix, in this exactly order.

The code behind the 3D world

top

I'll show you the formulas to do all those works and explain their usages, but I'll not show in deeply the mathematical logical behind the formulas, showing how the formula was created, is not my intention here. If you are interested to know in deeply how the formula was created I suggest you a book, a great book by the way, where all these formulas came from and also a great mathematical website:

- Recommended book: Mathematics for 3D Game Programming and Computer Graphics (http://www.amazon.com/Mathematics-Programming-Computer-Graphics-Second/dp/1584502770/ref=sr_1_2?ie=UTF8&qid=1300892618&sr=8-2).
- WebSite:<http://www.euclideanspace.com>
(<http://www.euclideanspace.com>)

The EuclideanSpace website has not a good layout, I know, it seems a little amateur, but trust me, all formulas in there are very reliable, ALL the formulas. Navigation is made by the top menu, it could seem confused at a glance, but it's very organized, thinking mathematically.

OK, so let's start with the matrices.

Matrices

top

Some guys see the matrix as black box with magic inside. Well, it really seems magical what it makes, but it's not exactly a black box. It's more like a very organized package and we can understand how that magic works, what is its "tricks", understanding its organization we could make great things with matrices. Remember that everything that is made by a matrix was also made by Euclid using only the Pythagoras and angle concepts. René Descartes just placed all that knowledge into a single package, called matrix. In 3D world we'll use a 4x4 matrix (4 lines with 4 columns), this kind of matrix is also known as square matrix. The fastest and simplest way to represent a matrix in programming language is through arrays. More specifically a linear unidimensional array with 16 elements.

Using a linear unidimensional array we could represent a matrix by two ways, row-major or column-major. This is just a convention, because in reality pre-multiply a row-major matrix or post-multiply a column-major matrix produces the same result. Well, as OpenGL prefers a column-major notation, let's follow this notation.

That is like the array indices are organized with a column-major notation:

Column-Major Notation

•

	0	4	8	12	
	1	5	9	13	
	2	6	10	14	
	3	7	11	15	

•

Now I'll show separately 5 kinds of matrices: Translation Matrix, Scale Matrix, Rotation X Matrix, Rotation Y Matrix and Rotation Z Matrix. Later we'll see how to join all these into one single matrix.

The most simple operation with 4x4 matrices is the translation, changing the X, Y and Z position. It's very very simple, you don't even need a formula. Here is what you need to do:

Translation Matrix

•

	1	0	0	X	
	0	1	0	Y	
	0	0	1	Z	
	0	0	0	1	

•

The second stupidly simple operation is the scale. As you may have seen in 3D professional softwares, you can change the scale individually to each axis. This operation doesn't need a formula. Here is what you need to do:

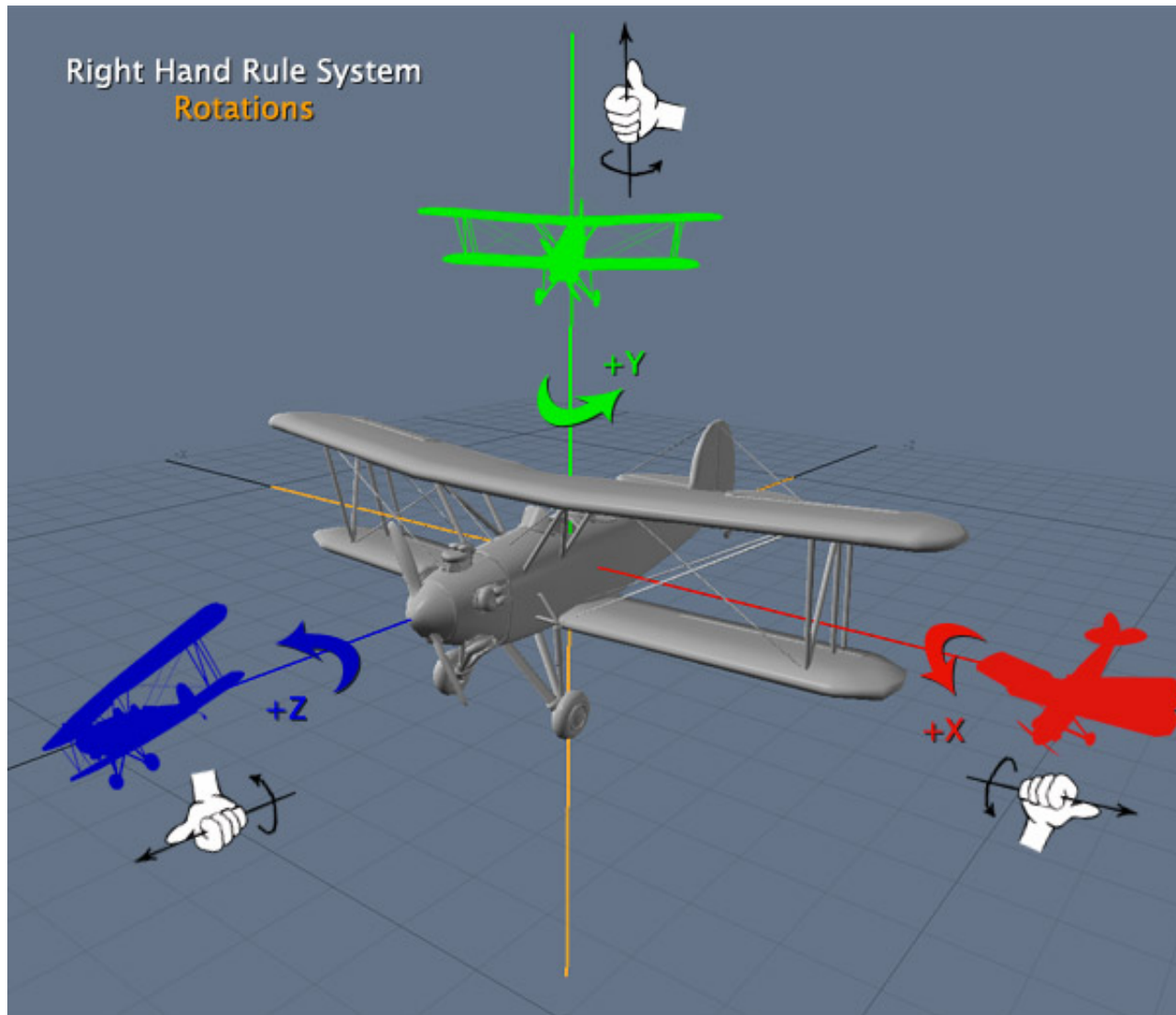
Scale Matrix

•

	SX	0	0	0	
	0	SY	0	0	
	0	0	SZ	0	
	0	0	0	1	

•

Now let's complicate it a little bit. It's time to make rotation with a matrix around one specific axis. We can think about rotations in the 3D world using the Right Hand Rule. The right hand rule defines the positive directions of all the 3 axes, besides it defines the rotations order as well. Align your thumb along a positive direction on an axis, now close the other fingers then the direction that your fingers are pointing to is the positive rotation of that axis:



We can create a rotation matrix using only one angle in one axis. To do that, we'll use sines and cosines. As you know, the angles should be in radians, not in degrees. To convert degrees to radians we use: **Angle * PI / 180** and to convert from radians to degrees we could use: **Angle * 180 / PI**. Well, my advice here, to gain performance, is: "let the PI / 180 and 180 / PI pre-calculated". Using C macros, I like to use something like this:

```
// Pre-calculated value of PI / 180.
#define kPI180      0.017453

// Pre-calculated value of 180 / PI.
#define k180PI     57.295780

// Converts degrees to radians.
#define degreesToRadians(x) (x * kPI180)

// Converts radians to degrees.
#define radiansToDegrees(x) (x * k180PI)
```

OK, so with the values of our rotations in radians, is time to use the following formulas:

Rotate X

$$\begin{matrix}
 & \cdot & & & & \\
 & | & & & & | \\
 & | & & & & | \\
 & | & & & & | \\
 & | & & & & | \\
 & | & & & & | \\
 & | & & & & | \\
 & \cdot & & & &
 \end{matrix}
 \begin{matrix}
 1 & 0 & 0 & 0 \\
 0 & \cos(\theta) & \sin(\theta) & 0 \\
 0 & -\sin(\theta) & \cos(\theta) & 0 \\
 0 & 0 & 0 & 1
 \end{matrix}
 \begin{matrix}
 | \\
 | \\
 | \\
 | \\
 | \\
 |
 \end{matrix}$$

Rotate Y

•

	$\cos(\theta)$	0	$-\sin(\theta)$	0	
	0	1	0	0	
	$\sin(\theta)$	0	$\cos(\theta)$	0	
	0	0	0	1	

•

Rotate Z

•

	$\cos(\theta)$	$-\sin(\theta)$	0	0	
	$\sin(\theta)$	$\cos(\theta)$	0	0	
	0	0	1	0	
	0	0	0	1	

•

Maybe you have seen the same formulas in other places with different minus signal in the elements, but remember, often they teach the traditional mathematical way, which uses the row-major notation, so remember here we are using the column-major notation, which fits right within OpenGL process.

Now it's time to join all of those matrices together. Just as a literal number, we need to multiply each one to get the final result. But the matrix multiplication has some special behaviors. You probably remember some of them from your high school or college.

- Matrix multiplication is not commutative. $A \times B$ is different than $B \times A$.
- Multiplying $A \times B$ is a calculation of the multiplication from each value of A rows by each value from B columns.
- To multiply $A \times B$, A matrix MUST have the number of columns EQUAL to the number of rows in B . Otherwise, multiplication can't be made.

Well, in the 3D world we always have square matrices, 4×4 or in some cases 3×3 , so we can multiply a 4×4 only for other 4×4 . Now, let's dive into the code using an array of 16 elements to compute all the above formulas:

Matrix Formulas with Array

```
typedef float mat4[16];
```

```
void matrixIdentity(mat4 m)
```

```
{
```

```
    m[0] = m[5] = m[10] = m[15] = 1.0;
```

```
    m[1] = m[2] = m[3] = m[4] = 0.0;
```

```
    m[6] = m[7] = m[8] = m[9] = 0.0;
```

```
    m[11] = m[12] = m[13] = m[14] = 0.0;
```

```
}
```

```
void matrixTranslate(float x, float y, float z, mat4  
matrix)
```

```
{
```

```
    matrixIdentity(matrix);
```

```
    // Translate slots.
```

```
    matrix[12] = x;
```

```
    matrix[13] = y;
```

```
    matrix[14] = z;
```

```
}
```

```
void matrixScale(float sx, float sy, float sz, mat4  
matrix)
```

```
{
```

```
    matrixIdentity(matrix);
```

```
    // Scale slots.
```

```
    matrix[0] = sx;
```

```
    matrix[5] = sy;
```

```
    matrix[10] = sz;
```

```
}
```

```
void matrixRotateX(float degrees, mat4 matrix)
{
    float radians = degreesToRadians(degrees);

    matrixIdentity(matrix);

    // Rotate X formula.
    matrix[5] = cosf(radians);
    matrix[6] = -sinf(radians);
    matrix[9] = -matrix[6];
    matrix[10] = matrix[5];
}
```

```
void matrixRotateY(float degrees, mat4 matrix)
{
    float radians = degreesToRadians(degrees);

    matrixIdentity(matrix);

    // Rotate Y formula.
    matrix[0] = cosf(radians);
    matrix[2] = sinf(radians);
    matrix[8] = -matrix[2];
    matrix[10] = matrix[0];
}
```

```
void matrixRotateZ(float degrees, mat4 matrix)
{
    float radians = degreesToRadians(degrees);
```

```
matrixIdentity(matrix);

// Rotate Z formula.
matrix[0] = cosf(radians);
matrix[1] = sinf(radians);
matrix[4] = -matrix[1];
matrix[5] = matrix[0];
}
```

And here is the code for a multiplication of two 16 arrays representing 4x4 matrices.

Matrix Multiplication

```
void matrixMultiply(mat4 m1, mat4 m2, mat4 result)
{
    // Fisrt Column
    result[0] = m1[0]*m2[0] + m1[4]*m2[1] + m1[8]*m2[2] +
m1[12]*m2[3];
    result[1] = m1[1]*m2[0] + m1[5]*m2[1] + m1[9]*m2[2] +
m1[13]*m2[3];
    result[2] = m1[2]*m2[0] + m1[6]*m2[1] + m1[10]*m2[2] +
m1[14]*m2[3];
    result[3] = m1[3]*m2[0] + m1[7]*m2[1] + m1[11]*m2[2] +
m1[15]*m2[3];

    // Second Column
    result[4] = m1[0]*m2[4] + m1[4]*m2[5] + m1[8]*m2[6] +
m1[12]*m2[7];
    result[5] = m1[1]*m2[4] + m1[5]*m2[5] + m1[9]*m2[6] +
m1[13]*m2[7];
    result[6] = m1[2]*m2[4] + m1[6]*m2[5] + m1[10]*m2[6] +
m1[14]*m2[7];
    result[7] = m1[3]*m2[4] + m1[7]*m2[5] + m1[11]*m2[6] +
m1[15]*m2[7];

    // Third Column
    result[8] = m1[0]*m2[8] + m1[4]*m2[9] + m1[8]*m2[10] +
m1[12]*m2[11];
    result[9] = m1[1]*m2[8] + m1[5]*m2[9] + m1[9]*m2[10] +
m1[13]*m2[11];
    result[10] = m1[2]*m2[8] + m1[6]*m2[9] + m1[10]*m2[10]
+ m1[14]*m2[11];
    result[11] = m1[3]*m2[8] + m1[7]*m2[9] + m1[11]*m2[10]
```

```

+ m1[15]*m2[11];

    // Fourth Column
    result[12] = m1[0]*m2[12] + m1[4]*m2[13] +
m1[8]*m2[14] + m1[12]*m2[15];
    result[13] = m1[1]*m2[12] + m1[5]*m2[13] +
m1[9]*m2[14] + m1[13]*m2[15];
    result[14] = m1[2]*m2[12] + m1[6]*m2[13] +
m1[10]*m2[14] + m1[14]*m2[15];
    result[15] = m1[3]*m2[12] + m1[7]*m2[13] +
m1[11]*m2[14] + m1[15]*m2[15];
}

```

As you know, standard C doesn't allow us to return arrays from a function, because we need to pass a pointer to our result array. If you are using a language which allows you to return an array, like JavaScript or ActionScript, you could prefer to return a literal array instead of working with a pointer.

Now one very important thing: "You CAN'T combine matrices directly, like using a rotationX formula above a matrix with another rotationZ, for example. YOU MUST TO CREATE EACH OF THEM SEPARATELY AND THEN MULTIPLY TWO BY TWO UNTIL YOU GET THE FINAL RESULT!".

For example, to translate, rotate and scale an object you must create each matrix separately and then perform the multiplication **((Scale * Rotation) * Translation)** to get the final transformation matrix.

Now let's talk about some tips and tricks with matrix.

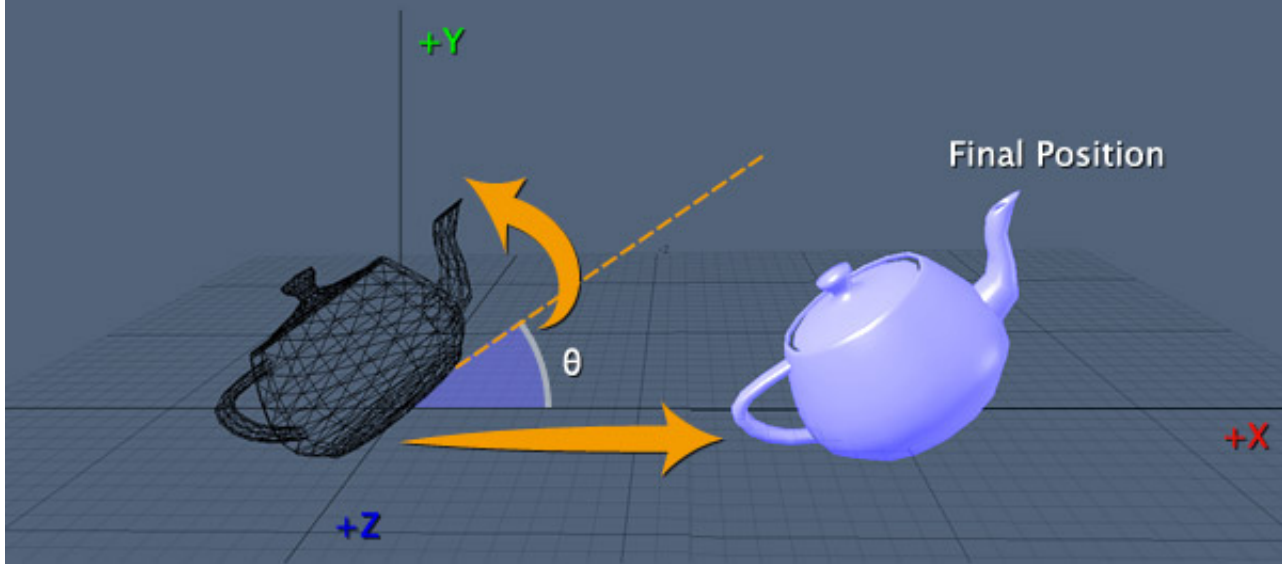
Matrices in Deep

top

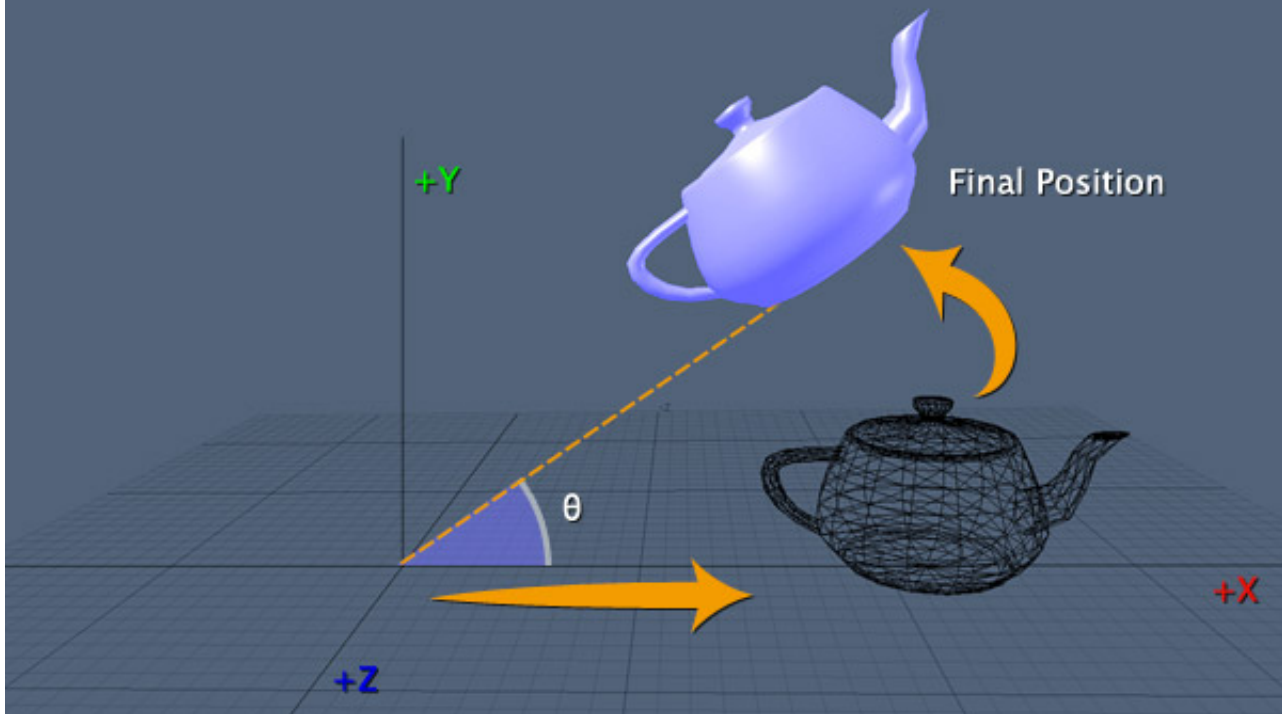
Is time to open that "black box" and understand what happens inside it. Sometime ago I worked with matrices without understand what exactly happens there, what exactly means pre-multiply or post-multiply a matrix by another, what's the purpose of transpose a matrix, if all matrix are column-majors, why uses the inverse, well... I need to say that everything in my world changed after watch some classes of Massachusetts Institute of Technology (MIT) about matrices. I want to share that knowledge with you:

1. What means pre or post-multiply matrices? This is the order of what the things happens. The second matrix in the multiplication WILL MAKE THE CHANGES FIRST! (lol). If we multiply $A \times B$ this means that B will happen first and then A. So if we multiply Rotation \times Translation this means that object first will translate and then will rotate. The same is true for scales too.

Translation x Rotation = **Rotation Happens First**

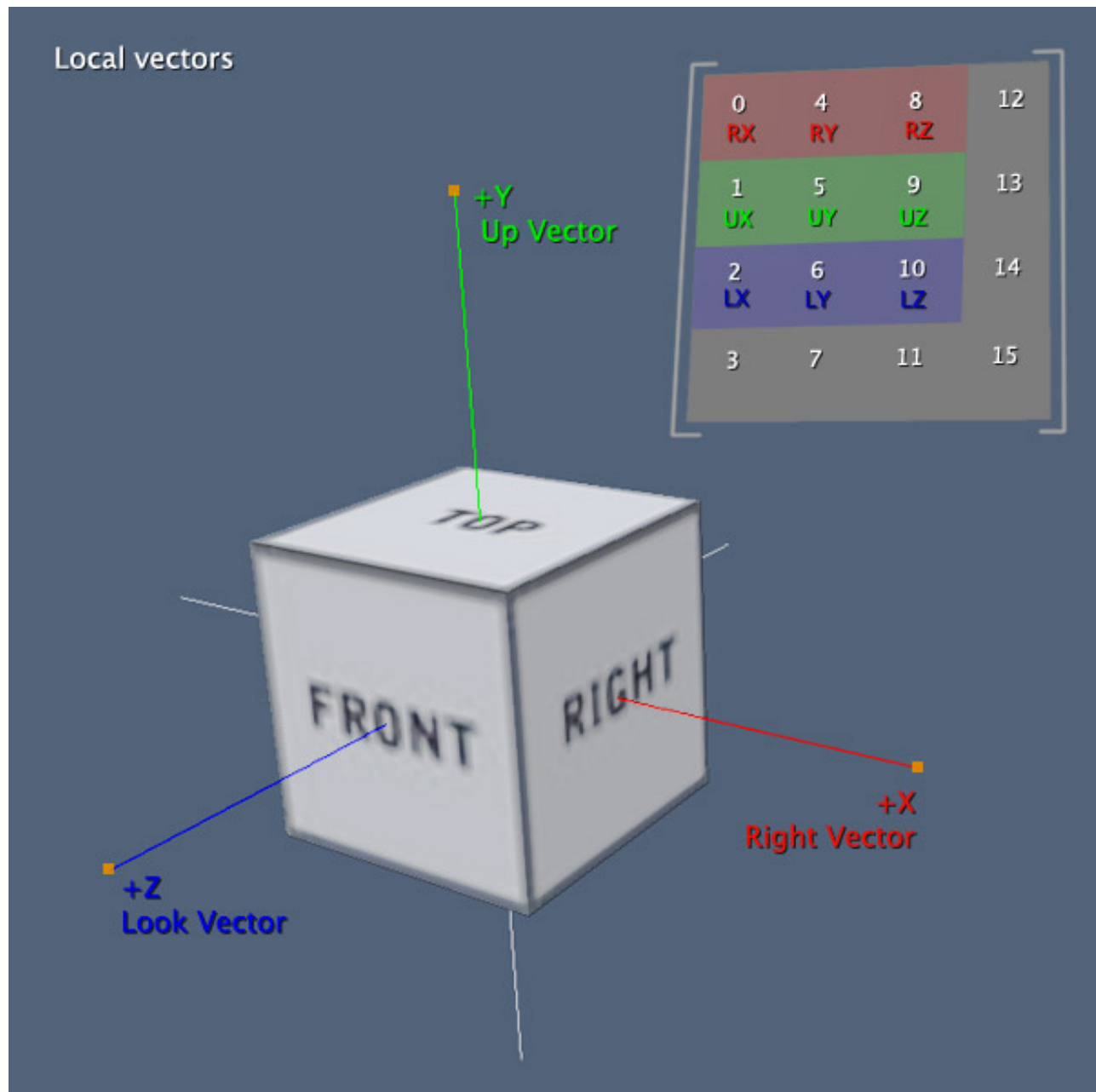


Rotation x Translation = **Translation Happens First**



2. Using the logic above, we can understand why the difference between local rotations and global rotations is just pre or post-multiply one rotation matrix by another. If you always post-multiply the new rotation matrices this means the object will first make the new rotation and then will rotate the old values, this is a local rotation. If you always pre-multiply the new rotations this means the object will first rotate to the old values and then rotate the new, this is a global rotation.
3. Any 3D object always has 3 local vectors: Right Vector, Up Vector and

Look Vector. These vector are very important to make the Euclidean Transformation on it (Scales, Rotations and Translations). Specially when you are making local transformations. The good news is: do you remember the rotations formulas? What that formulas make is transcribe the rotations angles to the vectors and place them in the matrix. So you can extract these vectors directly from a rotation matrix and the best thing is that these vectors are already normalized in the matrix.



4. The next cool thing is about orthogonal matrices (don't get confused with orthonormal, which is said about two orthogonal vectors). In theory the orthogonal matrix is that one with real entries whose columns and rows are orthogonal unit vectors, in very simple words,

orthogonal matrix is that one we call rotation matrix, without scales! I'll repeat this, it's very important: "Orthogonal is the ROTATION MATRIX, pure rotation matrix, without any scale!". Using the rotations formulas we get only unit vector and they are always orthogonal! What the hell are unit and orthogonal vectors? Is very simple to understand, unit vectors are that vector which the length/magnitude is equal 1, so the name "unit" vector. The orthogonal vector are said about two or more vector which has an angle of 90° between them. Look at the picture above again, notice that Right, Up and Look vectors are always orthogonal in 3D world.

5. Still in orthogonal matrices, if a matrix is orthogonal this means its inverse is equal its transpose. WOW! This is great! Because to calculate the inverse we need over than 100 multiplications and more than 40 sums, but to calculate the transpose we don't need any calculations, just change the order of some values. This is a big boost on our performance. But why we want the inverse of rotation matrix? To calculate the lights in the shaders! Remember that the real object never changes, we just change the computation of its vertices by multiplying they by a matrix. So to calculate a light, which is in the global space, we need the inverse of rotation matrix. Well, obviously we will need the inverse for other things, like cameras, so use the transpose instead of inverse could be great! And just to don't let any doubts, inverse matrix technacally represents a matrix which if we multiply it by the original (pre or post, doesn't matter here) will produce the matrix identity. In simple words, the inverse matrix is that which will revert all transformations from the original matrix.

You can extract the values of rotation, scale and transition from a matrix. Unfortunately don't exist a precise way to extract negative scales from a matrix. You could find the formulas to extract the values from a matrix in that book, 3D Game Programming, or in EuclideanSpace website. I'll

not talk about those formulas here because I have a better advice:
"Instead to try retrieve values from a matrix, is much much better you store user friendly values. Like store the global rotations (X,Y and Z), local scales (X,Y and Z) and global positions (X,Y and Z)."

Now is time to know about the great Hamilton contribution, the Quaternions!

Quaternions

top

Quaternions is to me the greatest package invention in 3D calculus. If the matrix is very organized and could seem "magical" to some people, the quaternions is what those people call "miracle". Quaternions is unbelievable simple. In stupidly simple words it is: "Take a vector as a direction and rotate it around own axis!".

If you make a little research by, you'll find many discussions about it. Quaternions is very polemic! Some guys love it, others hate it. Some say it is just a fad, others say it is amazing. Why all this around Quaternions? Well, is because using the rotations formulas we found a formula to produce rotations about an arbitrary axis, directly, avoiding the Gimbal Lock, or in other words, that formula produces the same effect as a quaternion (in fact is very similar). I'll not show this formula here, because I don't believe it is a good solution.

About the war of Quaternions X Rotation about an arbitrary axis you'll find people saying which this takes 27 multiplications, plus some sums, sine, cosine, vector magnitude against 21 or 24 multiplications in Quaternions and all this kind of annoying discussion! Whatever!!! With

the actual hardwares, you can cut 10.000.000 multiplications in your application and all the gain that you'll have is 0.04 secs (directly on the iPhone 4 the mark was 0.12 with 10.000.000 multiplications)! This is not remarkable. There are several thing much more important than multiplications to boost your application's performance. In reallity the difference between those number will be less than 1.000 per frame.

So what is the crucial point about the Quaternions X Rotations Formulas? My love about Quaternions comes from the fact that it is SIMPLE! Very organized, very clear and incredibly precisely when you make consecutive rotations. I'll show how to work with Quarternions, and you take your own decision.

Let's start with simple concept. Quaternions, as the name sugest, is a vector of order 4 (x,y,z,w). Just for convention we are used to take notation of Quaternions as w,x,y,z using the "w" first. But this really doesn't matter, because all operations with quaternions always will bring the letter x,y,z or w. An alert! Don't confuse the "w" of Quaternions with "w" from Homegeneous Coordinates, those are two things completely different.

As quaternion is a vector 4, many vector operations are applied to it. But just few formulas are really important: multiplication, identity and inverse. Before start with those three formulas, I want to introduce you the formula to extract a matrix from a quaternion. This is the most important one:

Quaternion To Matrix

.

```
// This is the arithmetical formula optimized to work  
with unit quaternions.
```

```
// | 1-2y2-2z2      2xy-2zw      2xz+2yw      0 |  
// | 2xy+2zw      1-2x2-2z2      2yz-2xw      0 |  
// | 2xz-2yw      2yz+2xw      1-2x2-2y2      0 |  
// |      0      0      0      1 |
```

```
// And this is the code.
```

```
// First Column
```

```
matrix[0] = 1 - 2 * (q.y * q.y + q.z * q.z);  
matrix[1] = 2 * (q.x * q.y + q.z * q.w);  
matrix[2] = 2 * (q.x * q.z - q.y * q.w);  
matrix[3] = 0;
```

```
// Second Column
```

```
matrix[4] = 2 * (q.x * q.y - q.z * q.w);  
matrix[5] = 1 - 2 * (q.x * q.x + q.z * q.z);  
matrix[6] = 2 * (q.z * q.y + q.x * q.w);  
matrix[7] = 0;
```

```
// Third Column
```

```
matrix[8] = 2 * (q.x * q.z + q.y * q.w);  
matrix[9] = 2 * (q.y * q.z - q.x * q.w);  
matrix[10] = 1 - 2 * (q.x * q.x + q.y * q.y);  
matrix[11] = 0;
```

```
// Fourth Column
```

```
matrix[12] = 0;  
matrix[13] = 0;
```

```
matrix[14] = 0;
matrix[15] = 1;
```

.

Just like the matrices formulas, this conversion always produces an orthogonal matrix with unit vectors. In some places you may find an arithmetical formula which uses " $w^2+x^2-y^2-z^2$ " instead " $1-2y^2-2z^2$ ". Don't worry, this is because the original quaternions from Hamilton was much more complex. They have an imaginary part (i, j and k) and could be more than just unit quaternions. But as in 3D world we always work with unit vectors, we can discard the imaginary part of quaternions and assume which they will always be unit quaternions. Because this optimization, we can use the formula with " $1-2y^2-2z^2$ ".

Now, let's talk about other formulas. First, the multiplication formula:

Quaternion Multiplication

.

```
// Assume that this multiplies q1 x q2, in this
order, resulting in "newQ".
newQ.w = q1.w * q2.w - q1.x * q2.x - q1.y * q2.y -
q1.z * q2.z;
newQ.x = q1.w * q2.x + q1.x * q2.w + q1.y * q2.z -
q1.z * q2.y;
newQ.y = q1.w * q2.y - q1.x * q2.z + q1.y * q2.w +
q1.z * q2.x;
newQ.z = q1.w * q2.z + q1.x * q2.y - q1.y * q2.x +
q1.z * q2.w;
```

.

That multiplication formula has exact the same effect as multiply a rotation matrix by another, for example. Just as matrix multiplication, multiplying quaternions is not a commutative operation. so $\mathbf{q1} \times \mathbf{q2}$ is not equals to $\mathbf{q2} \times \mathbf{q1}$. Note that I'll not show here the arithmetical formula to multiplication. This is because the original multiplication with two vector 4 are much more complex than a cross multiplication by a vector 3 and it needs a matrix multiplication, this could confuse our minds (in fact the arithmetical formula will result in the code above). Let's focus on what is important. But if you have intrest in know more about multiplications with vector 4, try this:

<http://www.mathpages.com/home/kmath069.htm>

[\(http://www.mathpages.com/home/kmath069.htm\)](http://www.mathpages.com/home/kmath069.htm)

The identity. The identity quaternion produces the identity matrix, here is the formula:

Quaternion Identity

.

$$\mathbf{q} \cdot \mathbf{x} = 0;$$

$$\mathbf{q} \cdot \mathbf{y} = 0;$$

$$\mathbf{q} \cdot \mathbf{z} = 0;$$

$$\mathbf{q} \cdot \mathbf{w} = 1;$$

.

OK, now this is the formula to invert a quaternion, aka "conjugate quaternion":

Quaternion Inverse

```
.  
  
    q.x  *= -1;  
    q.y  *= -1;  
    q.z  *= -1;  
  
    // At this point is a good idea normalize the  
    quaternion again.  
.
```

I love this inverse formula. Because its simplicity! It's so simple! And these three lines of code has the exactly the same effect as take the inverse of a matrix! (Ô.o)

Yes dude, if you are working with quaternions to rotations, instead to invert the matrix, making more than 100 multiplications and sums, you can simple make these three lines above. As I said before, is not because the reduction of processing, but is more in reason of the simplicity! Quaternions are stupidly simple!

Is everything OK until here? Remember to ask, if you have doubts. Let's proceed now to those two formula which are reason of the "Quaternions WAR". Let's put some rotations angles into it. To do that, we have two ways: using the quaternion's concept and informing a vector as direction and an angle to rotate around this direction or we can use the Euler angles (X, Y and Z) informing the three angles as they are. This last one takes more multiplications, but is more user friendly because is just like set the angles to the matrices rotations formulas.

First setting the quaternion by a direction vector and an angle:

Axis to Quaternion

```

•
// The new quaternion variable.
vec4 q;

// Converts the angle in degrees to radians.
float radians = degreesToRadians(degrees);

// Finds the Sin and Cosin for the half angle.
float sin = sinf(radians * 0.5);
float cos = cosf(radians * 0.5);

// Formula to construct a new Quaternion based on
direction and angle.
q.w = cos;
q.x = vec.x * sin;
q.y = vec.y * sin;
q.z = vec.z * sin;
•

```

To produce consecutive rotations you can make multiply quaternions. Just as the matrix approach, to produce a local or a global rotation just change the order of the multiplication ($q_1 \times q_2$ or $q_2 \times q_1$), and remember, just as the matrix, when you multiply $q_1 \times q_2$ this means: "do q_2 first and then q_1 ".

Now, here is the formula to convert Euler Angles to Quaternions:

Euler Angles to Quaternion

•

```
// The new quaternion variable.
vec4 q;

// Converts all degrees angles to radians.
float radiansY = degreesToRadians(degreesY);
float radiansZ = degreesToRadians(degreesZ);
float radiansX = degreesToRadians(degreesX);

// Finds the Sin and Cosin for each half angles.
float sY = sinf(radiansY * 0.5);
float cY = cosf(radiansY * 0.5);
float sZ = sinf(radiansZ * 0.5);
float cZ = cosf(radiansZ * 0.5);
float sX = sinf(radiansX * 0.5);
float cX = cosf(radiansX * 0.5);

// Formula to construct a new Quaternion based on
Euler Angles.
q.w = cY * cZ * cX - sY * sZ * sX;
q.x = sY * sZ * cX + cY * cZ * sX;
q.y = sY * cZ * cX + cY * sZ * sX;
q.z = cY * sZ * cX - sY * cZ * sX;
```

•

As you saw, I organized the order of code to make the angles in order Y, Z and X. Why? Because this is the order in which the rotation will be produced by the quaternion. Using this formula, can we change this order? NO, we can't. This formula is to produce this kind of rotation (Y,Z,X). By the way, this is what we call "Euler Rotation Order". If you

want to know more about rotation order, or what that means, watch this video, is really great <http://www.youtube.com/watch?v=zc8b2Jo7mno>
(<http://www.youtube.com/watch?v=zc8b2Jo7mno>)

Great! This is the basic about quaternions. Obviously we have formulas to retrieve the values from a Quaternion, extract the Euler angles, extract the vector direction, etc. This kind of thing is good just to you check what is happening inside the quaternions. My advice here is the same as the matrix: "Always store an user friendly variable to control your rotations".

Now let's back to the matrices and finally understand how we can create camera lenses.

The code behind the 3D cameras

[top](#)

Wow! Finally we are ready to understand how to create a camera lens. Now is easy to figure out what we need to do. We need to create a matrix which affect the vertices positions according to their depth. By using some concepts we saw at the beginning (Depth of Field, Near, Far, Angle of View, etc) we can calculate a matrix to make elegants and smooth transformations to simulate the real lenses, which is already a simulation of the human's eyes.

As explained early, we can create two kind of projections: Perspective and Orthographic. About the both kinds, I'll not explain here the mathematical formula in deep, if you are intrest in the concepts behind the projection matrices, you can find a real good explanation here:

http://www.songho.ca/opengl/gl_projectionmatrix.html

(http://www.songho.ca/opengl/gl_projectionmatrix.html). Oh right, so now let's focus on the code. Starting with the most basic kind, the Orthographic projection:

Orthographic Projection

.

```
// These paramaters are lens properties.  
// The "near" and "far" create the Depth of Field.  
// The "left", "right", "bottom" and "top" represent  
the rectangle formed  
// by the near area, this rectangle will also be the  
size of the visible area.  
float near = 0.001, far = 100.0;  
float left = 0.0, right = 320.0, bottom = 480.0, top =  
0.0;  
  
// First Column  
matrix[0] = 2.0 / (right - left);  
matrix[1] = 0.0;  
matrix[2] = 0.0;  
matrix[3] = 0.0;  
  
// Second Column  
matrix[4] = 0.0;  
matrix[5] = 2.0 / (top - bottom);  
matrix[6] = 0.0;  
matrix[7] = 0.0;  
  
// Third Column  
matrix[8] = 0.0;  
matrix[9] = 0.0;  
matrix[10] = -2.0 / (far - near);  
matrix[11] = 0.0;  
  
// Fourth Column
```

```
matrix[12] = -(right + left) / (right - left);  
matrix[13] = -(top + bottom) / (top - bottom);  
matrix[14] = -(far + near) / (far - near);  
matrix[15] = 1;
```

.

As you noticed, the Orthographic projection doesn't have any "Angle of View", this is because it doesn't need. As you remember from the orthographic project, it make everything seems equal, the units are always squared, in other words, orthographic projection is a linear projection.

The code above show to us what we imagine before. The projection matrix will affect slightly the rotations (X, Y and Z), affect directly the scales (the major diagonal) and more incisive on the vertex positions.

Now, let's see the perspective projection, a little more elaborated case:

Perspective Projection

.

```
// These paramaters are about lens properties.  
// The "near" and "far" create the Depth of Field.  
// The "angleOfView", as the name suggests, is the  
angle of view.  
// The "aspectRatio" is the cool thing about this  
matrix. OpenGL doesn't  
// has any information about the screen you are  
rendering for. So the  
// results could seem stretched. But this variable  
puts the thing into the  
// right path. The aspect ratio is your device screen  
(or desired area) width divided  
// by its height. This will give you a number < 1.0  
the the area has more vertical  
// space and a number > 1.0 is the area has more  
horizontal space.  
// Aspect Ratio of 1.0 represents a square area.  
float near = 0.001, far = 100.0;  
float angleOfView = 45.0;  
float aspectRatio = 0.75;  
  
// Some calculus before the formula.  
float size = near * tanf(degreesToRadians(angleOfView)  
/ 2.0);  
float left = -size, right = size, bottom = -size /  
aspectRatio, top = size / aspectRatio;  
  
// First Column  
matrix[0] = 2 * near / (right - left);
```

```

matrix[1] = 0.0;
matrix[2] = 0.0;
matrix[3] = 0.0;

// Second Column
matrix[4] = 0.0;
matrix[5] = 2 * near / (top - bottom);
matrix[6] = 0.0;
matrix[7] = 0.0;

// Third Column
matrix[8] = (right + left) / (right - left);
matrix[9] = (top + bottom) / (top - bottom);
matrix[10] = -(far + near) / (far - near);
matrix[11] = -1;

// Fourth Column
matrix[12] = 0.0;
matrix[13] = 0.0;
matrix[14] = -(2 * far * near) / (far - near);
matrix[15] = 0.0;

```

Understanding: Wow, the formula changes slightly, but now there are no effect on X and Y position, it just changes the Z position (depth). It continues affecting the rotation X, Y and Z, but there are a great interference on the Third Column, what is that? That is exactly the calculus about the modifications to produce the perspective and the factors to adjust the aspect ratio. Note that the last element of the third column is negative. This will inverse the stretches of the aspect ratio when the final matrix was generated (multiplied).

Now talking about the final matrix. This is a very important step. Unlike the other matrices multiplication, at this time you can't change the order, otherwise you'll get unexpected results. This is what you need to do:

Take the camera View Matrix (an inverted matrix containing the rotations and translations of the camera) and POST-Multiply it by the Projection Matrix:

PROJECTION MATRIX x VIEW MATRIX.

Remember, this will produce the effect as: "Do the VIEW MATRIX first and then do PROJECTION MATRIX".

Now you have what we are used to call *VIEWPROJECTION MATRIX*. Using this new matrix you will POST-Multiply the *MODEL MATRIX* (matrix containing all the rotations, scales and translations of an object) by the *VIEWPROJECTION MATRIX*:

VIEWPROJECTION MATRIX x MODEL MATRIX.

Again, just to reinforce, that means: "Do the *MODEL MATRIX* first and then do *VIEWPROJECTION MATRIX*". Finally! Now you have what is called *MODELVIEWPROJECTION MATRIX*!

CONGRATULATIONS!

Gasp! Oof!

OK, I know what you are thinking... "WTF! All this just to produces a simple stupid matrix!" Yeh, I thought the same. Doesn't has a more simple or fast way to do that?

Well, I think the answer to that questions is on the conclusion of this article. Let's go!

Conclusion

[top](#)

Henceforward, the things will become much more complex than this. Matrices and Quaternions is just the beginning of the journey to construct a 3D engine or personal framework. So if you didn't take a decision yet, maybe is time to take one. I think you have two choices:

1. You could construct a Framework/Engine by yourself.
2. You could take an existing Framework/Engine and learn how to use it.

Just as any "choice", both have positives and negatives points. You need to think and decide what is better for your purposes. I think a third choice, construct a little "template" to use in your projects could not be a good one, so personally I discard this option as a choice. There are so many things to be done in the 3D world that probably we would be crazy or very lost trying to fit everything into one or few templates. So my last advice is: "Take a choice".

Anyway, the next tutorial I'll make will be more advanced. For now, let's revise everything of this tutorial:

- Cameras could has a Convex or Concave Lens. Cameras also has some properties like Depth of Field, Angle of View, Near and Far.
- In 3D world we can work with a real projection called Perspective and an unreal projection called Orthographic.
- The Camera should work as the inverse of a normal 3D object.
- We never change the original structure of a 3D object, we just take the result of a temporary change.
- Those changes can be made using matrices. We have formulas to rotate, scale and translate a matrix. We also has the Quaternions to

work with the rotations.

- We also use a matrix to create the camera's lens using a Perspective or Orthographic projection.

This is all for now.

Thanks for reading, see you in the next tutorial!



Steve Jobs

1955 – 2011

You changed the way as I live and the world around me.
You taught me to always keep thinking in "One More Thing".

Thank you Steve,
dear mentor.

WRITTEN BY

Diney Bomfim ()

Published on

February 04, 2014

SPREAD THE WORD





Join the discussion...



sree · 2 years ago

really informative. Thanks a lot

3 ^ | v · Reply · Share ›



chinese · 2 years ago

great tutorial dude ... keep going this way :D

3 ^ | v · Reply · Share ›



zego · a year ago

greate article, thanks

^ | v · Reply · Share ›

ALSO ON BLOG.DB-IN

WHAT'S THIS?

All about OpenGL ES 2.x - (part 2/3)

9 comments • 2 years ago

MD Aminuzzaman — This is the best OpenGL ES tutorial i have seen/found so far around the web. Many many thanks for very informative ...

NinevehGL is HERE!

3 comments • 2 years ago

dineybomfim — Thanks, Yeah, shame on me about this long delay. The project is not dead, but it's in a sleep phase. Maybe it comes live ...

Swift 2.0 is now better than Objective-C

2 comments • 8 months ago

dineybomfim — I totally agree with you, More pure Swift libs for all! ;)

All about Shaders - (part 1/3)

5 comments • 2 years ago

dineybomfim — Hi, thank you. Shame on me buddy, I must get back to this series and finish it. I'll, for sure! ;)

✉ Subscribe

🗨 Add Disqus to your site Add Disqus Add

🔒 Privacy