Understanding Affine Transformations With Matrix Mathematics

by Kah Shiu Chong 17 Feb 2012

Difficulty: Intermediate Length: Long Languages: English ▼

Effects Flash

This post is part of a series called You Do The Math.

◀◀ Circular Motion in AS3: Make One Moving Object Orbit Another

▶▶ The Math and ActionScript of Curves: Drawing Quadratic and Cubic Curves

Inspired by Prof. Wildberger in his lecture series on linear algebra , I intend to implement his mathematical ideas with Flash. We shall not delve into the mathematical manipulation of matrices through linear algebra: just through vectors. This understanding, although diluting the elegance of linear algebra, is enough to launch us into some interesting possibilities of 2x2 matrix manipulation. In particular, we'll use it to apply various shearing, skewing, flipping, and scaling effects to images at runtime.

Final Result Preview

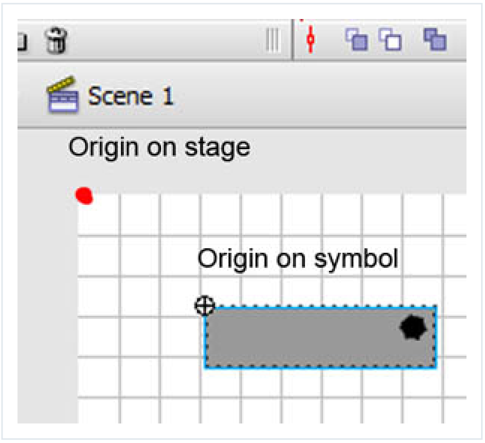
Let's take a look at the final result we will be working towards. Press the four directional keys - up, down, left, right - to see some effects we can achieve with affine transformations.

If you only use the left and right arrow keys, the fish appears to swim around in a pseudo-3D isometric space.

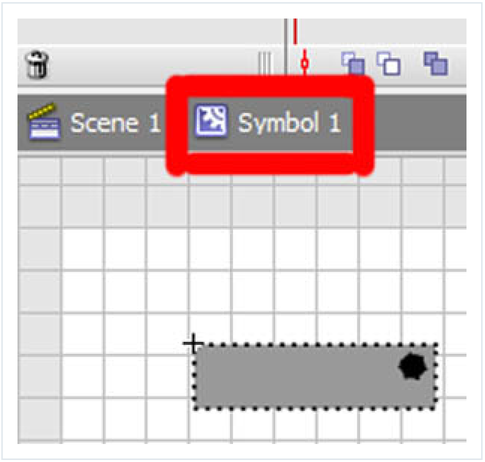
Step 1: **Different Coordinate Spaces**

Graphics are drawn onto coordinate spaces. So in order to manipulate them, especially to translate, rotate, scale, reflect and skew graphics, it is vital that we understand coordinate spaces. We generally make use of not just one, but multiple coordinate spaces in a single project - this is true not only for designers using the Flash IDE, but also for programmers writing ActionScript.

In Flash IDE this happens whenever you convert your drawings into MovieClip symbols: each symbol has its own origin.



The image above shows the origin of the stage' coordinate space (red dot), and that of the symbol's coordinate space (registration point marked by crosshair). To know which space you are in currently, observe the bar beneath the timeline of Flash IDE as shown by the image below.



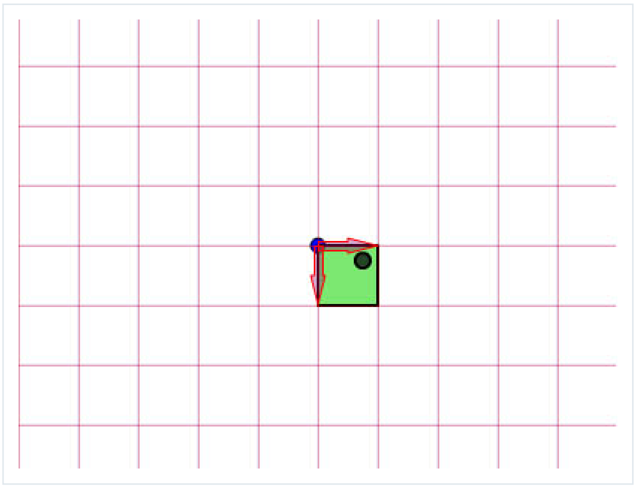
(I'm using Flash CS3, so its location may differ for CS4 and CS5.) What I want to emphasize is the existence of different coordinate spaces, and the fact that you're already familiar with using them.

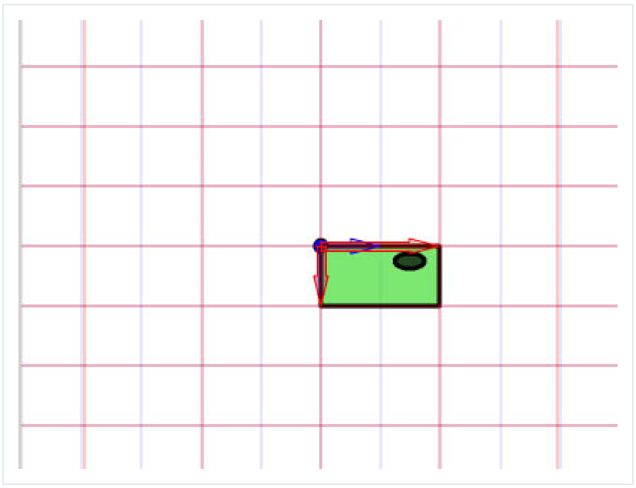
Step 2: **The Rationale**

Now there's a good reason for this. We can use one coordinate space as a reference to change the other coordinate space. This may sound alien, so I've included the Flash presentation below to facilitate my explanation. Click and drag the red arrows. Play around with it.

In the background is a blue grid, and in the foreground is a red grid. The blue and red arrows are initially aligned along the x- and y-axis of Flash coordinate space, whose center I have shifted to the middle of the stage. The blue grid is a reference grid; the gridlines will not change as you interact with the red arrows. The red grid, on the other hand, can be reoriented and scaled by dragging the red arrows.

Note that the arrows also indicate an important property of these grids. They indicate the notion of a unit of x and a unit of y on their respective grid. There are two red arrows on the red grid. Each of them indicate the length of one unit on the x-axis and the y-axis. They also dictate the orientation of the coordinate space. Let's take the red arrow pointing along the x-axis and extend it to be twice as long as the original arrow (shown in blue). Observe the following images.

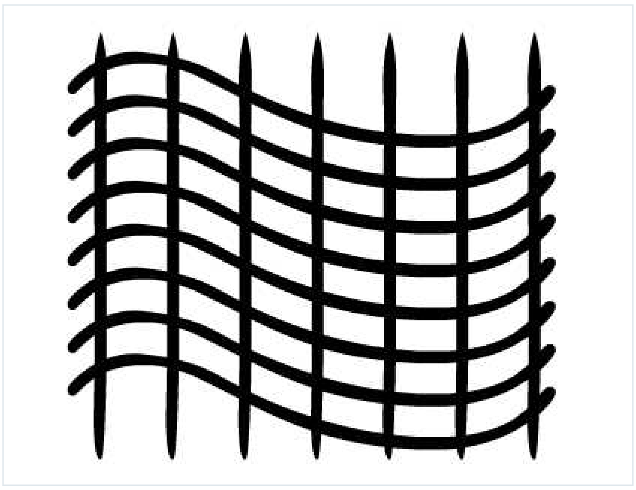




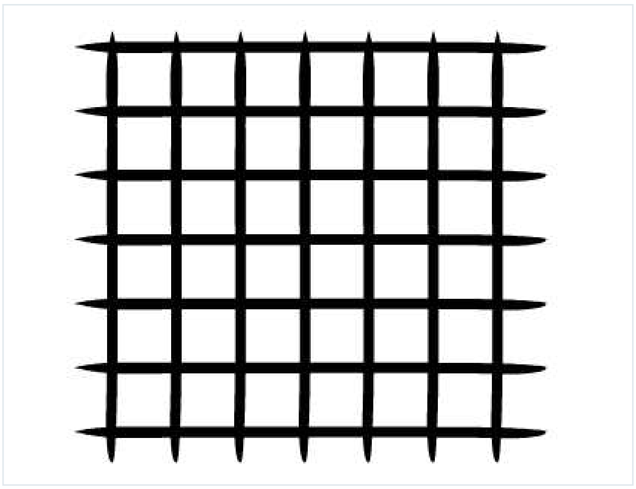
We see that the image (the green box) drawn on the red grid is now stretched horizontally, due to the fact that this red grid it is drawn onto is now twice as wide. The point I'm trying to make is rather simple: you can use one coordinate space as a basis to change another coordinate space.

Step 3: Affine Coordinate Space

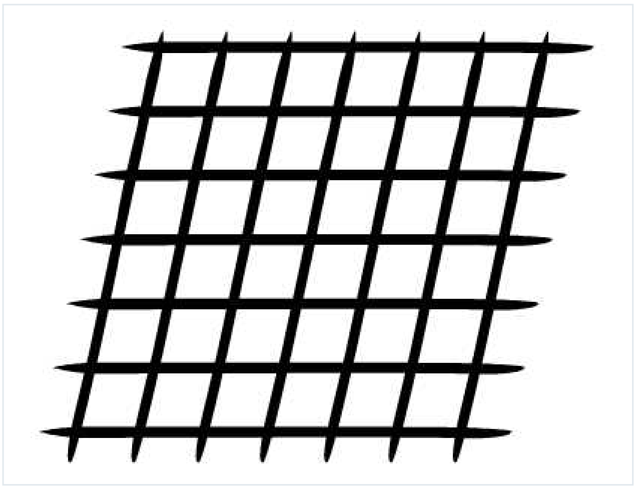
So what's an "affine coordinate space"? Well, I'm sure you are careful enough to observe that these coordinate spaces are drawn using parallel grids. Let's take the red affine space for example: there's no guarantee that both the x-axis and the y-axis are always perpendicular to each other, but rest assured that however you try to tweak the arrows, you will never arrive at such a case as below.



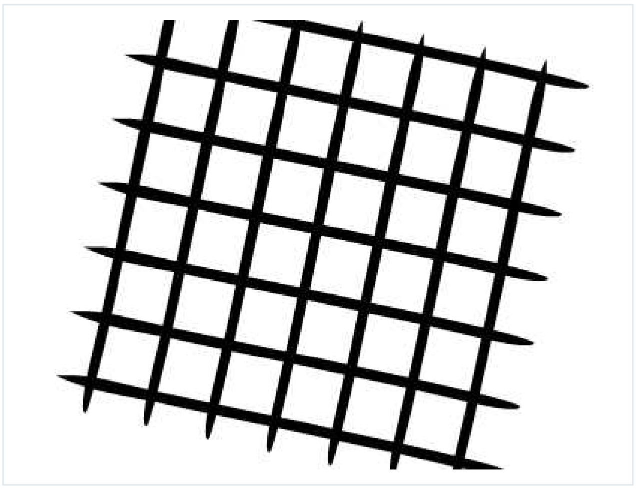
This coordinate space is *not* an affine coordinate space.
In fact, x- and y-axes usually refer to the Cartesian coordinate space, as shown below.



Note that the horizontal and vertical grids are perpendicular to each other. Cartesian is a *type* of affine coordinate space, but we can transform it to other affine spaces as we prefer. The horizontal and vertical grids do not necessarily have to be perpendicular to each other.



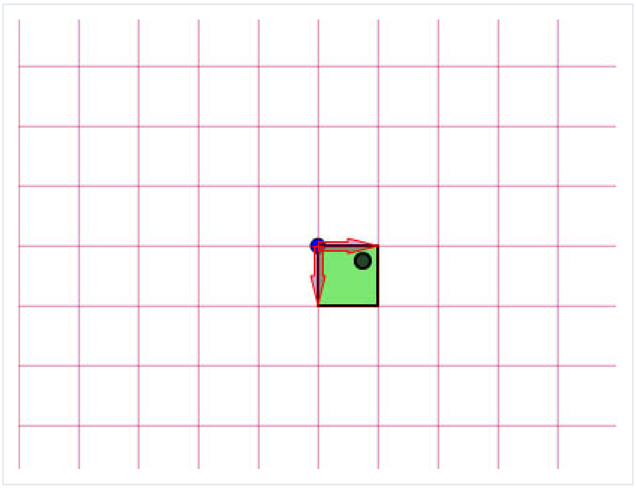
Example of an affine coordinate space



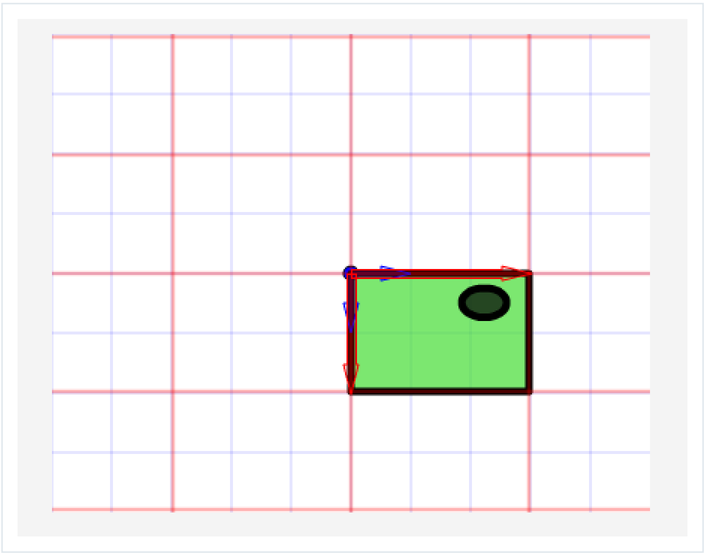
Another example of an affine coordinate space

Step 4: Affine Transformations

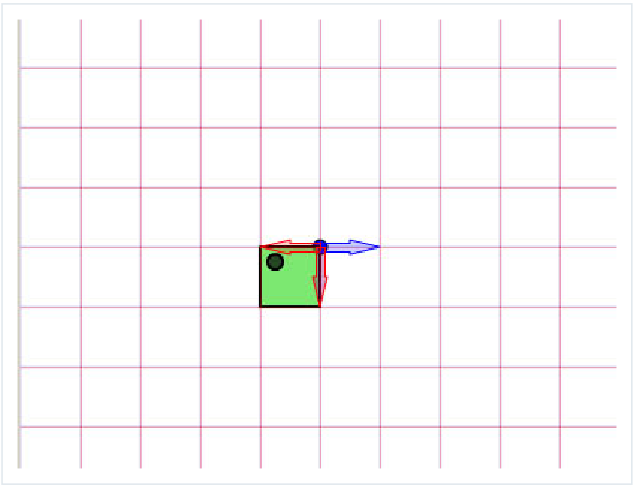
As you might have guessed, the affine transformations are translation, scaling, reflection, skewing and rotation.



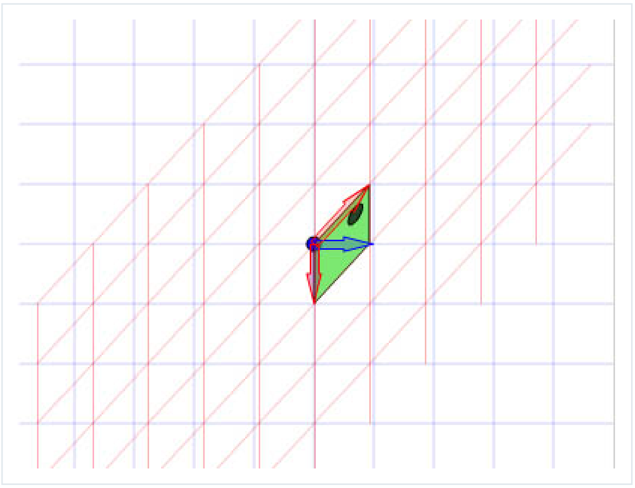
Original affine space



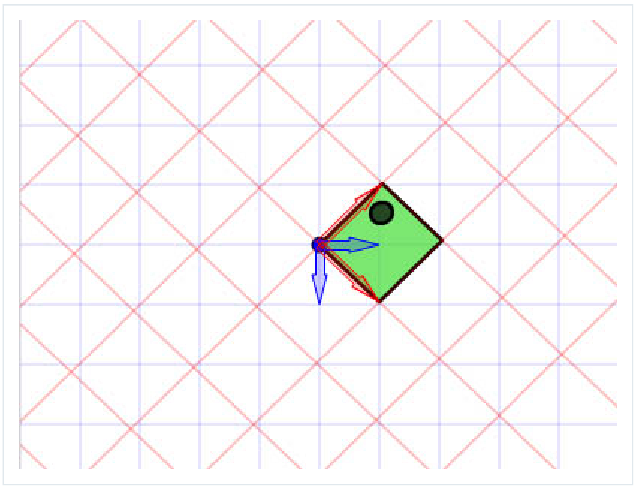
Scaled affine space



Reflected affine space



Skewed affine space



Rotated and scaled affine space

Needless to say, physical properties such as `x, y, scaleX, scaleY` and `rotation` depend on the space. When we make calls to those properties, we are actually transforming affine coordinates.

Step 5: Understanding Matrix

I hope the images shown above are explicit enough to drive home the idea. This is because for a programmer working with FlashDevelop, we will not see those grids that the Flash IDE conveniently displays for designers. All of these have to live in your head.

Apart from imagining these grids, we also need to enlist the help of `Matrix` class. Thus, having a mathematical understanding of matrices is important, so we shall revise the operations of matrix here: addition and multiplication.

Matrix multiplication:

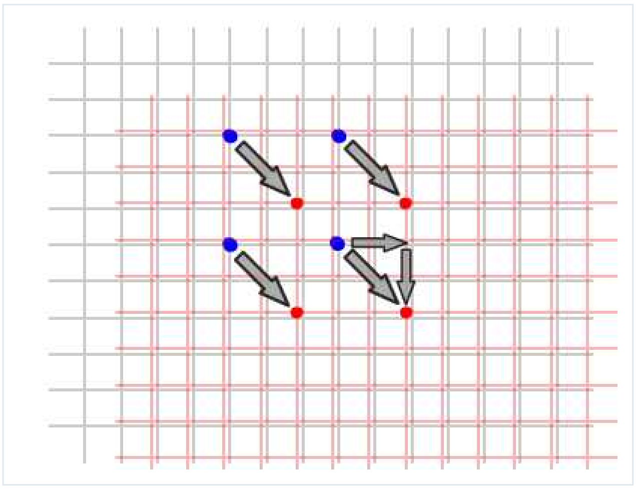
$$\begin{bmatrix} a & c \\ b & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax+cy \\ bx+dy \end{bmatrix}$$

Matrix addition:

$$\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} tx \\ ty \end{bmatrix} = \begin{bmatrix} x + tx \\ y + ty \end{bmatrix}$$

Step 6: Geometrical Meaning of Matrix Addition

Matrix operations convery meanings geometrically. In other words, you can picture what they mean on a graph. Let's assume that we have four points in our coordinate space and would like to shift them to a set of new locations. This can be done using matrix addition. Check out the image below.



As you can see, we are actually shifting the whole local coordinate space (red grids) where these four points are drawn. The notation for performing these operations is as shown below:

Matrix addition:

$$\begin{bmatrix} x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \end{bmatrix} + \begin{bmatrix} tx \\ ty \end{bmatrix}$$

$$= \begin{bmatrix} x_1 + tx & x_2 + tx & x_3 + tx & x_4 + tx \\ y_1 + ty & y_2 + ty & y_3 + ty & y_4 + ty \end{bmatrix}$$

We can also see that this shift can actually be represented using a vector of (tx, ty). Let us differentiate vectors and static points in coordinate spaces by our use of parentheses and square brackets. I've rewritten them in the image below.

Matrix addition:

$$\begin{bmatrix} x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \end{bmatrix} + \begin{pmatrix} tx \\ ty \end{pmatrix}$$

$$= \begin{bmatrix} x_1 + tx & x_2 + tx & x_3 + tx & x_4 + tx \\ y_1 + ty & y_2 + ty & y_3 + ty & y_4 + ty \end{bmatrix}$$

Step 7: ActionScript Implementation

Here's a simple implementation of matrix addition. Check out the comments:

```
public class Addition extends Sprite
{
    public function Addition()
    {
        var m:Matrix = new Matrix();        //instantiate matrix
        m.tx = stage.stageWidth * 0.5;      //shift in x
        m.ty = stage.stageHeight * 0.5;     //shift in y
        var d:DottedBox = new DottedBox();  //create the custom graphic (dotted box is a Sprite)
        addChild(d);
        d.transform.matrix = m;         //apply the matrix to our graphic
    }
}
```

Step 8: Geometrical Meaning of Matrix Multiplication

Matrix multiplication is somewhat more sophisticated than matrix addition but Prof Wildberger has elegantly broken it down to this simple interpretation. I shall humbly attempt to reiterate his explanation. For those who'd like to dive deeper into the understanding of linear algebra that leads to this, check out the professor's lecture series.

Let's start off by tackling the case of the identity matrix, I.

Matrix multiplication:

$$\begin{bmatrix} a & c \\ b & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax+cy \\ bx+cy \end{bmatrix}$$

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

$$= \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \text{ aka identity matrix, I}$$

From the image above we know that multiplying an arbitrary matrix, A, by the identity matrix, I, will always produce A. Here's an analogy: 6 x 1 = 6; the identity matrix is likened to the number 1 in that multiplication.

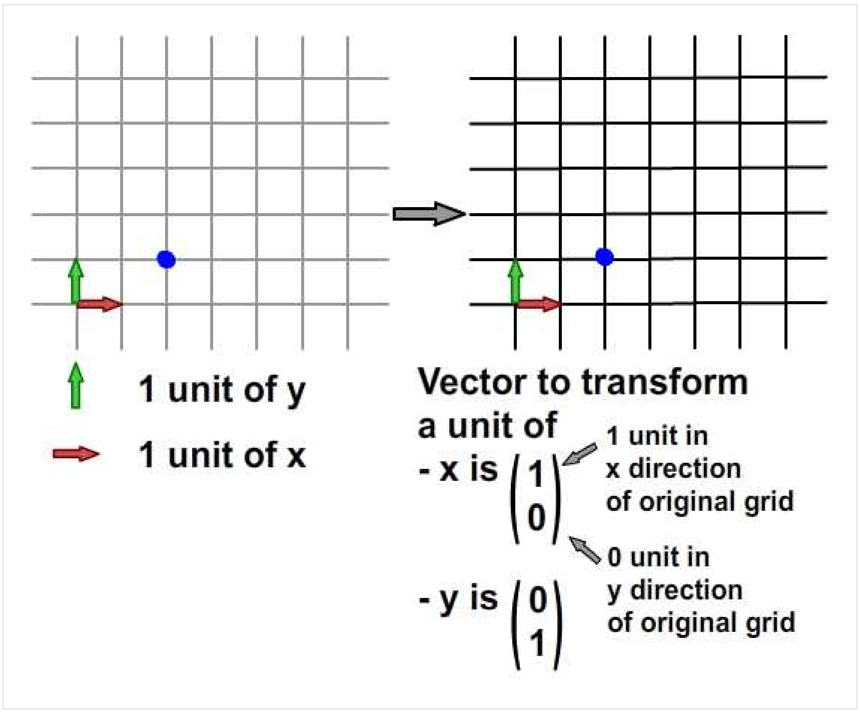
Alternatively, we can write the result in the following vector format which will greatly simplify our interpretation:

Matrix multiplication:

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

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} x + \begin{pmatrix} 0 \\ 1 \end{pmatrix} y = \begin{bmatrix} 1x+0y \\ 0x+1y \end{bmatrix}$$

The geometric interpretation of this formula is shown in the image below.


1 unit of y
1 unit of x
Vector to transform
a unit of
- x is $\begin{pmatrix}1\\0\end{pmatrix}$
1 unit in x direction of original grid
0 unit in y direction of original grid
- y is $\begin{pmatrix}0\\1\end{pmatrix}$

From the Cartesian grid (left grid), we can see the blue point is located at (2, 1). Now if we were to transform this original grid of x and y to a new grid (right grid) according to a set of vectors (below the right grid), the blue point will be relocated to (2, 1) on the new grid - but when we map this back to the original grid, it's the same point as before.

Because we are transforming the original grid to another grid that shares the same vectors for x and y, we see no difference. In fact, the changes of x and y in this transformation is nil. This is what it meant by *identity matrix*, from a geometrical point of view.

However, if we try to perform a mapping using other transformations, we shall see some difference. I know this was not the most revealing example to start with, so let's move on to another example.

Step 9: **Scaling Along X**

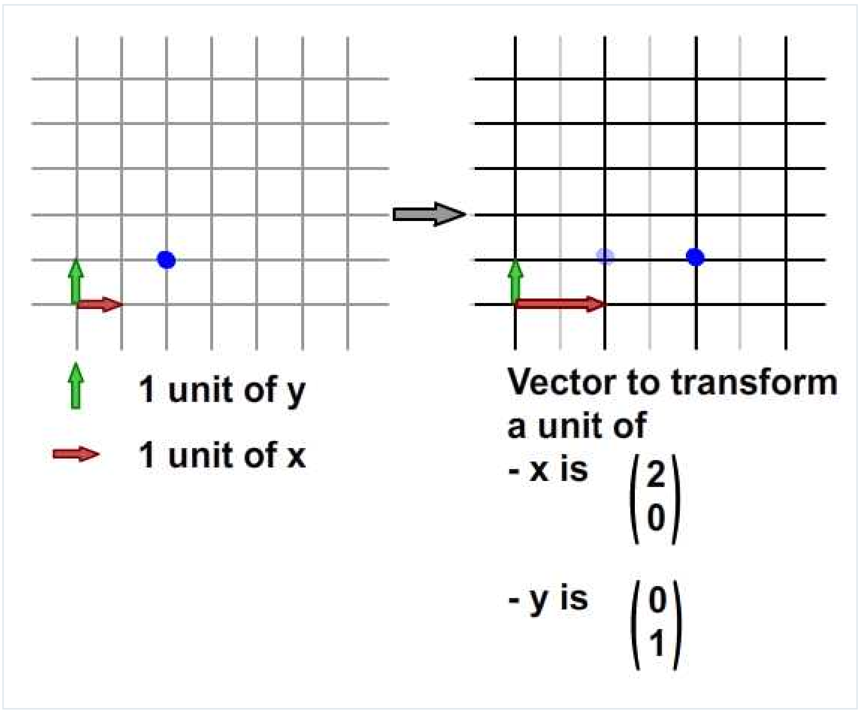

1 unit of y
1 unit of x
Vector to transform
a unit of
- x is $\begin{pmatrix}2\\0\end{pmatrix}$
- y is $\begin{pmatrix}0\\1\end{pmatrix}$

Image above demonstrates a scaling of the coordinate space. Check out the vector of x in transformed coordinate space: one unit of the transformed x accounts for two units of the original x. On the transformed coordinate space, the coordinate of the blue point is still (2, 1). However, if you try to map this coordinate from the transformed grid onto the original grid, it is (*4*, 1).

This whole idea is captured by the image above. How about the formula? The result should be consistent; let's check it out.

I'm sure you recall these formulae. Now, I've added their respective meanings.

Matrix multiplication:

Transformation matrix

Transformed coordinate

$$\begin{bmatrix}a & c\\ b & d\end{bmatrix}\begin{bmatrix}x\\ y\end{bmatrix}=\begin{bmatrix}x'\\ y'\end{bmatrix}$$

Original coordinate

$$=\begin{bmatrix}ax+cy\\ bx+dy\end{bmatrix}$$

Transformation vector on original y

$$\begin{pmatrix}a\\ b\end{pmatrix}x+\begin{pmatrix}c\\ d\end{pmatrix}y=\begin{bmatrix}x'\\ y'\end{bmatrix}$$

Transformation vector on original x

Now to check out the numerical result of our scaling example.

- Original coordinate: (2, 1)
- Vector on transformed x-axis: (2, 0)
- Vector on transformed y-axis: (0, 1)
- Expected result: (2*2 + 0*1, 0*2 + 1*1) = (4, 1)

$$\begin{pmatrix}2\\ 0\end{pmatrix}2+\begin{pmatrix}0\\ 1\end{pmatrix}1=\begin{bmatrix}4\\ 1\end{bmatrix}$$

They do agree with each other! Now we can happily apply this idea to other transformations. But before that, an ActionScript implementation.

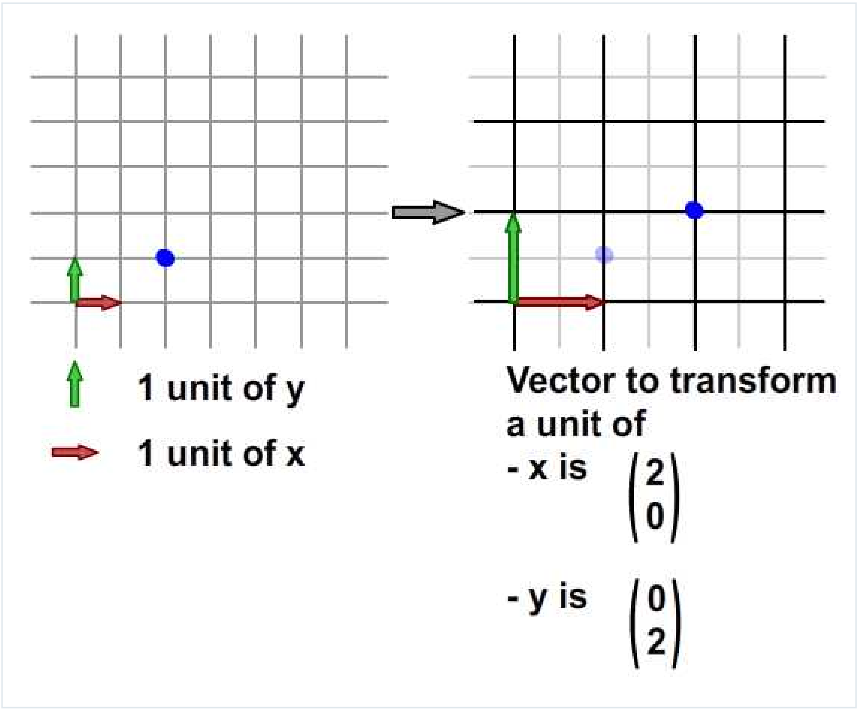
Step 10: ActionScript Implementation

Check out the ActionScript implementation (and the resulting SWF) below. Note that one of the overlapping boxes is being stretched along x by a scale of 2. I have highlighted the important values. These values will be tweaked in the later steps to represent different transformations.

```
public class Multiplication extends Sprite
{
    public function Multiplication()
    {
        var ref:DottedBox = new DottedBox();    //create reference graphic
        addChild(ref); ref.x = stage.stageWidth * 0.5; ref.y = stage.stageHeight * 0.5;

        var m:Matrix = new Matrix();        //instantiate matrix
        m.tx = stage.stageWidth * 0.5;      //shift in x
        m.ty = stage.stageHeight * 0.5;     //shift in y
        m.a = 2; m.c = 0;
        m.b = 0; m.d = 1;
        var d:DottedBox = new DottedBox();  //create the custom graphic
        addChild(d);
        d.transform.matrix = m          //apply the matrix onto our graphic
    }
}
```

Step 11: Scaling X and Y


1 unit of y
1 unit of x
Vector to transform a unit of
- x is $\begin{pmatrix}2\\0\end{pmatrix}$
- y is $\begin{pmatrix}0\\2\end{pmatrix}$

Here we've scaled the grid by a factor of two along both the x- and y-axes. The blue point is at (2, 1) in the original grid before the transformation, and (4, 2) in the original grid after the transformation. (Of course, it's still at (2, 1) in the *new* grid after the transformation.)

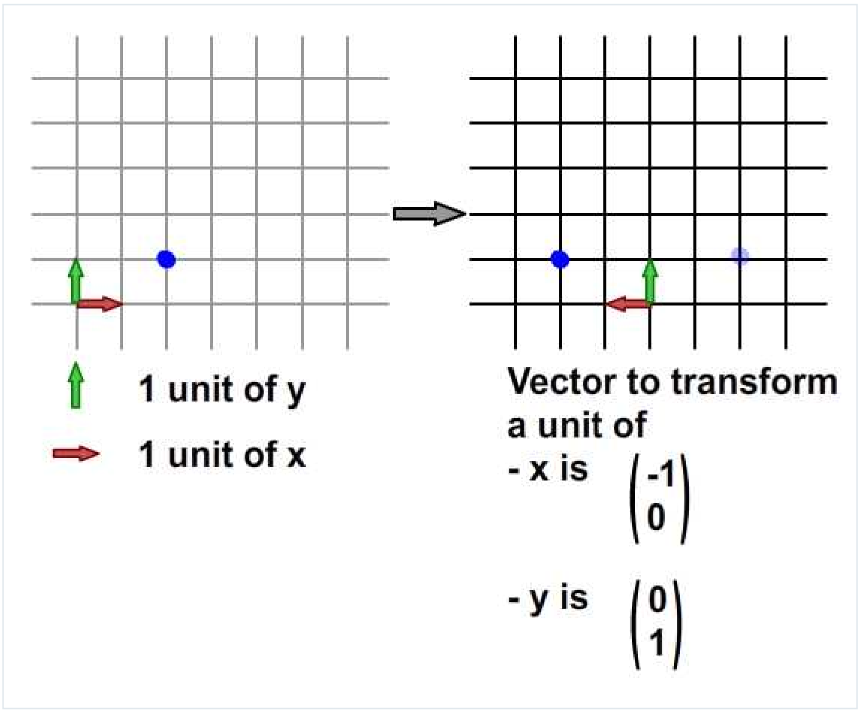
And to confirm the result numerically...

$$\begin{pmatrix}2\\0\end{pmatrix}2+\begin{pmatrix}0\\2\end{pmatrix}1=\begin{bmatrix}4\\2\end{bmatrix}$$

... they match again! To see this in the ActionScript implementation, just change the value of `m.d` from 1 to 2.

(Note that the direction of stretch from y is downwards, not upwards, because y increments downwards in Flash but upwards in the normal Cartesian coordinate space that I used in the diagram.)

Step 12: **Reflection**

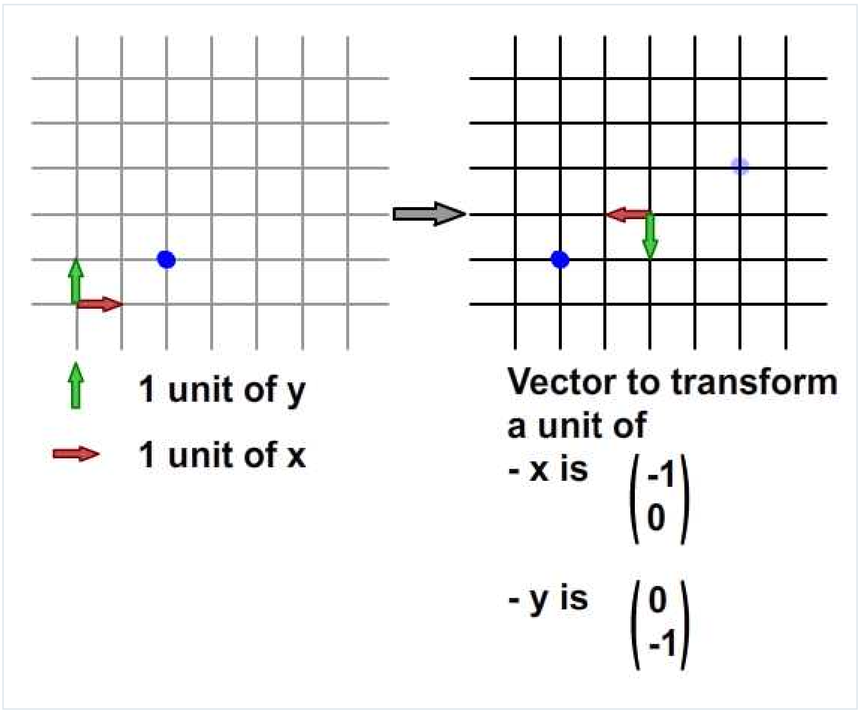

1 unit of y
1 unit of x
Vector to transform a unit of
- x is $\begin{pmatrix}-1\\0\end{pmatrix}$
- y is $\begin{pmatrix}0\\1\end{pmatrix}$

Here we've reflected the grid along the x-axis using these two vectors, so the position of the blue point in the original grid changes from (2, 1) to (-2, 1). The numeric calculation is as follows:

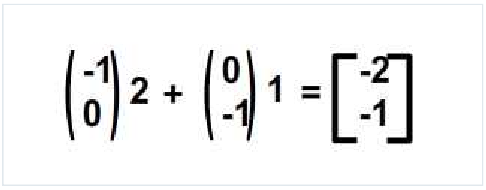
$$\begin{pmatrix}-1\\0\end{pmatrix}2+\begin{pmatrix}0\\1\end{pmatrix}1=\begin{bmatrix}-2\\1\end{bmatrix}$$

The ActionScript implementation is the same as before, but using these values instead: `m.a = -1, m.b = 0` to represent the vector for the x transformation, and: `m.c = 0 and m. d = 1` to represent the vector for the y transformation.

Next, what about reflecting simultaneously on x and y? Check out the image below.


1 unit of y
1 unit of x
Vector to transform a unit of
- x is $\begin{pmatrix}-1\\0\end{pmatrix}$
- y is $\begin{pmatrix}0\\-1\end{pmatrix}$

Also, numerically computed in image below.

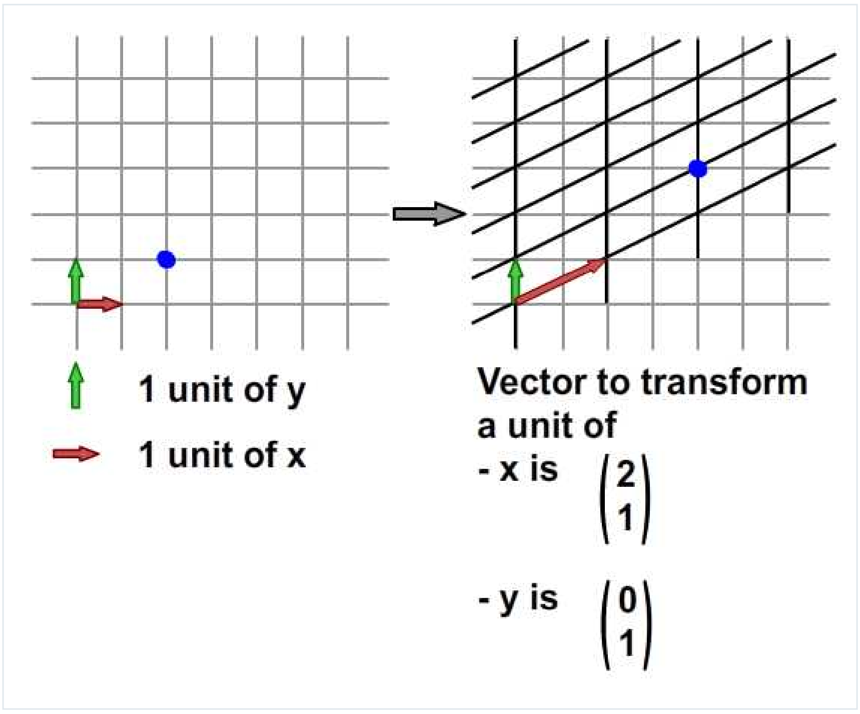


$$\begin{pmatrix}-1\\0\end{pmatrix}2+\begin{pmatrix}0\\-1\end{pmatrix}1=\begin{bmatrix}-2\\-1\end{bmatrix}$$

For the ActionScript implementation... well, I'm sure you know the values to put into the matrix. `m.a = -1, m.b = 0` to represent the vector for the x transformation; `m.c = 0 and m. d = -1` to represent the vector for the y transformation. I've included the final SWF below.

Step 13: **Skewing and Shearing**

Skewing comes with a little fun. For the case of the image below, the transformed grid has had its x-axis reoriented and scaled. Compare the red arrows in both grids below: they are different, but the y-axis remains unchanged.


1 unit of y
1 unit of x
Vector to transform a unit of
- x is $\begin{pmatrix}2\\1\end{pmatrix}$
- y is $\begin{pmatrix}0\\1\end{pmatrix}$

Skewing

Visually, it seems that distortion happens along the y-direction. This is true because our transformed x-axis now has a y-component in its vector.

Numerically, this is what happens...

$$\begin{pmatrix}2\\1\end{pmatrix}2 + \begin{pmatrix}0\\1\end{pmatrix}1 = \begin{bmatrix}4\\3\end{bmatrix}$$

In terms of implementation, I've listed the tweaks below.

- `m.a = 2`
- `m.b = 1`
- `m.c = 0`
- `m.d = 1`

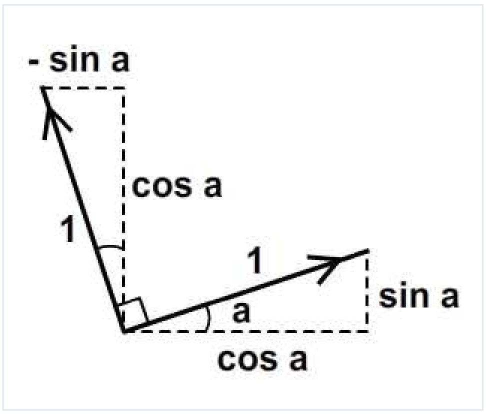
I'm sure at this point you'd like to try out things yourself, so go ahead and tweak

- the orientation of transformed y-axis while maintaining the x-axis
- the orientation of both axes altogether

I have included the Flash output for both cases as below. For readers who'd like some help with these values, check out `Multiplication_final.as` in the source download.

Step 14: Rotation

I consider rotation a subset of skewing. The only difference is that in rotation, the magnitude of a unit of both x and y-axis is maintained, as is the perpendicularity between the two axes.



ActionScript actually provides a method in the `Matrix` class, `rotate()`, to do this. But let's go through this anyway.

Now we do not want to alter the magnitude of a unit length in x and y from the original grid; just to change the orientation of each. We can make use of trigonometry to arrive at the result shown in the image above. Given an angle of roation, a, we'll get the desired result by using vectors of (cos a, sin a) for x-axis and (-sin a, cos a) for y-axis. The magnitude for each new axis will still be one unit, but each axis will be at an angle of a, compared to the originals.

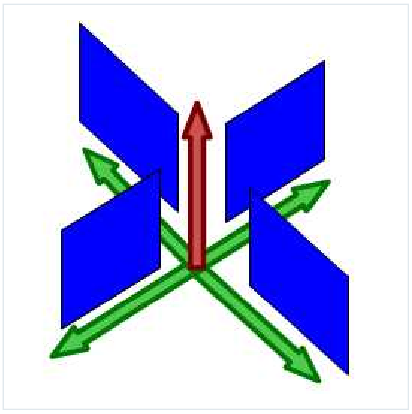
For Actionscript implementation, assuming that the angle, a, is 45 degrees (that is, 0.25*Pi radians), just tweak the matrix values to the following:

```
var a:Number = 0.25*Math.PI
m.a = Math.cos(a); m.c = -1*Math.sin(a);
m.b = Math.sin(a); m.d = Math.cos(a);
```

The full source can be referred to in `Multiplication_final.as`.

Step 15: Application

Having a vector interpretation of a 2x2 matrix opens up space for us to explore. Its application in manipulating bitmaps (`BitmapData, LineBitmapStyle, LineGradientStyle`, etc.) is widespread - but I think I'll save that for another tutorial. For the case of this article, we shall attempt to skew our sprite at run-time so that it looks like it's actually flipping in 3D.



View of a pseudo-3D isometric world

From the image above we can see that, in a world with an isometric view, any graphic that's "standing" keeps its y-axis vector unchanged while the x-axis vector is rotates. Note that a unit of length for the x- and y-axis does not change - in other words, no scaling should happen in either axis, just rotation around the x-axis.

Here's an example of this idea in Flash. Click anywhere on the stage and begin dragging around to see the fish skew. Release to stop your interaction.

Here's the important bit of Actionscript. I've highlighted the crucial lines that handle the x-axis rotation. You can also refer to `FakeIso.as`.

```
private var f1:Fish, m:Matrix;
private var disp:Point;
private var axisX:Point, axisY:Point;

public function FakeIso() {
    disp = new Point(stage.stageWidth * 0.5, stage.stageHeight * 0.5);
    m = new Matrix();
    m.tx = disp.x; m.ty = disp.y;       //displace to the center of stage
    f1 = new Fish(); addChild(f1);
    f1.transform.matrix = m;        //apply transformation to onto fish

    axisX = new Point(1, 0);        //vector for x - axis
    axisY = new Point(0, 1);        //vector for y - axis
    stage.addEventListener(MouseEvent.MOUSE_DOWN, start);   //start interaction
    stage.addEventListener(MouseEvent.MOUSE_UP, end);   //end interaction
}

private function start(e:MouseEvent):void {
    f1.addEventListener(Event.ENTER_FRAME, update);
}

private function end(e:MouseEvent):void {
    f1.removeEventListener(Event.ENTER_FRAME, update);
}

private function update(e:Event):void {
    axisX.setTo(mouseX - f1.x, mouseY - f1.y);  //determine orientation (but magnitude changed as well)
    axisX.normalize(1);         //fix magnitude of vector with new orientation to 1 unit
    apply2Matrix();             //apply matrix onto fish
}
```

```
private function apply2Matrix ():void {
    m.setTo(axisX.x, axisX.y, axisY.x, axisY.y, disp.x, disp.y);
    f1.transform.matrix = m;
}
```

Here, I have used the Point class for storing vectors.

Step 16: Add Keyboard Control

In this step, we shall attempt to add keyboard controls. The fish's location will update according to its velocity, `velo`. We'll define incremental steps for positive (clockwise) rotation and negative (anti-clockwise) rotation as well.

```
velo = new Point(1, 0); //velo will be used to define x-axis
axisY = new Point(0, 1);
delta_positive = new Matrix(); delta_positive.rotate(Math.PI * 0.01); //positive rotation
delta_negative = new Matrix(); delta_negative.rotate(Math.PI * -0.01); //negative rotation
```

Upon a key press, `velo` will rotate:

```
private function keyUp(e:KeyboardEvent):void {
if (e.keyCode == Keyboard.LEFT) {
    velo = delta_negative.transformPoint(velo) //rotate velo counter-clockwise
}
else if (e.keyCode == Keyboard.RIGHT) {
    velo = delta_positive.transformPoint(velo) //rotate velo clockwise
}
}
```

Now for each frame, we shall attempt to color the front side of the fish, and skew the fish as well. If the velocity, `velo`, has a magnitude of more than 1 and we apply it to the fish's matrix, `m`, we will get a scaling effect as well - so in order to eliminate this possibility, we shall normalise the velocity and then only apply that to the fish's matrix.

```
private function update(e:Event):void {
    var front_side:Boolean = velo.x > 0      //checking for the front side of fish
    if (front_side) { f1.colorBody(0x002233,0.5) }  //color the front side of fish
    else f1.colorBody(0xFFFFFF,0.5)         //white applied to back side of fish

    disp = disp.add(velo);          //update current displacement with velocity
    var velo_norm:Point = velo.clone();     //in case velo > 0, we need to recalculate 1 unit of length fo
    velo_norm.normalize(1);         //note that x-axis more than 1 will perform scaling. We dont want tha
    m.setTo(velo_norm.x, velo_norm.y, axisY.x, axisY.y, disp.x, disp.y);
    f1.transform.matrix = m;
}
```

Step 17: Your Fish

Click the stage, then press the left and right arrow keys to see make the fish change direction.

Step 18: Another Keyboard Control

To spice things up, let us allow the control of the y-axis vector as well.

```
private function keyUp(e:KeyboardEvent):void {
if (e.keyCode == Keyboard.LEFT) {
    velo = delta_negative.transformPoint(velo)
}
else if (e.keyCode == Keyboard.RIGHT) {
    velo = delta_positive.transformPoint(velo)
}
if (e.keyCode == Keyboard.UP) {
    axisY = delta_negative.transformPoint(axisY)
}
else if (e.keyCode == Keyboard.DOWN) {
    axisY = delta_positive.transformPoint(axisY)
}
}
```

Also to determine the front side of the fish, we now need to incorporate the y-axis in. Here's the code for that:

```
var front_side:Boolean = velo.x * axisY.y > 0
if (front_side) { f1.colorBody(0x002233,0.5) }
else f1.colorBody(0xFFFFFF,0.5)
```

Step 19: Your No-So-Regular Fish

Well, for some the result of controlling both axes may prove to be a little confusing, but the point is that you now can skew your fish, translate it, reflect it, and even rotate it! Try out the combos of up + left, up + right, down + left, down + right.

Also, see if you can maintain the "front" side of fish (fish will be grayed). Hint: Tap up continuously, then left, then down, then right. You are making a rotation!

Conclusion

I hope you find matrix math a valuable asset to your projects after reading this article. I'm hoping to write a little more on applications of 2x2 matrix in little Quick Tips branching out of this article, and on `Matrix3d` which is essential for 3D manipulations. Thanks for the read, terima kasih.

Kah Shiu Chong

A graduate in Physics, acquainted with Actionscript 2.0 as a introduction to programming and got hooked ever since. Currently serving as instructor in University College of Technology and Innovation, Malaysia.

10 Comments **Activetuts+** 1 **Login**

Recommend 2 Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

blatha • 4 years ago
Hi All your tutorials are great but this one really blew me away. Very creative!
1 • Reply • Share ›

Devon • 4 years ago
Thanks for the tutorial! Can you also tell us about affine warping of triangle?
• Reply • Share ›

Adrian H • 5 years ago
I used to be scared of matrices, they seemed too complex for me. Now I might try using them :D Thank you for the tutorial!
• Reply • Share ›

Ace • 6 years ago
Great tutorial! Too bad I didn't come across it earlier because I could have really used it a week ago. :-/ Oh well... got it bookmarked now.
• Reply • Share ›

tibbi • 6 years ago
And the image at step 5 has a typo, cy instead of dy :)
• Reply • Share ›

shiu → tibbi • 6 years ago
Hi and good day. Thanks for flagging the error. I've corrected it. As for the swf file, its compiled for flash player 11.0. Perhaps an update on your browser player would help?
• Reply • Share ›

tibbi → shiu • 6 years ago
cool thanks, btw same typo is at the beginning of 8th step :) cy instead of dy. Sometimes i have some flash issues at using chrome, using chrome 12.0.742.91 . will check some updates.
• Reply • Share ›

tibbi • 6 years ago
Fish doesnt seem to move in Chrome for me, however it works fine in Firefox. Problem is on my side, or its just not optimised for chrome ?
• Reply • Share ›

tibbi → tibbi • 6 years ago
Its changing colors, just not moving
• Reply • Share ›

Michael James Williams → tibbi • 6 years ago
That's weird. It's working for me in Chrome. We have been having some odd issues with our SWF hosting recently, however. Which versions of Flash and Chrome are you running?
• Reply • Share ›

Subscribe Add Disqus to your siteAdd DisqusAdd Privacy