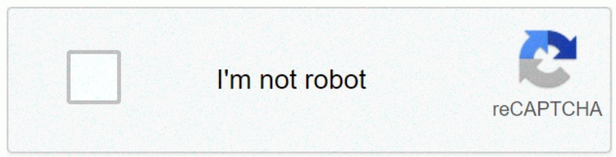

Continue

Affine transformation calculator

Inspired by Prof. Wildberger's lecture series linear algebra, I intend to apply my mathematical ideas to Flash. We don't let tasteless mathematical manipulation through linear algebra: just through vectors. This understanding, while diluting the elegance of linear algebra, is enough to trigger us with some interesting possibilities of 2x2 matrix manipulation. In particular, we use it to apply different cutting, cropping, reversing, and tag effects to images at runtime. The end result is we look at the final result we are working for. Press the four direction keys - up, down, left, right - to see some of the effects that we can achieve with the changes we are securing. If you only use left and right arrow keys, the fish floats around the pseudo-3D isometric space. Step 1: Different coordinate spaces in the graphics are drawn onto the coordinate spaces. So to manipulate them, especially translate, rotate, scale, reflect and skewed graphics, it is important that we understand the coordinates of the spaces. We generally use not only one, but multiple coordinate spaces in one project – this applies not only to designers using Flash IDE, but also to programmers writing ActionScript. In Flash IDE this happens when you convert your drawings to MovieClip symbols: each symbol has its own origin. The above image shows the origin of the stage coordinate space (red dot) and the symbol coordinate space (cross-shaped registration point). To find out which room you're in right now, follow the bar below the Flash IDE timeline, as shown in the image below. (I use Flash CS3, so its location can vary from CS4 and CS5.) I want to stress the existence of different coordinate rooms and the fact that you are already familiar with their use. Step 2: Recital Now is a good reason. We can use one coordinate room to change the other coordinate space. This may sound alien, so I've added a Flash presentation below to facilitate my explanation. Click and drag the red arrows. Play with it. In the background there is a blue network and a red network in the foreground. The blue and red arrows are originally aligned along the X and y axes of the Flash coordinate space, in the middle of which I have shifted to the center of the stage. The blue grid is the reference network; gridlines do not change when interacting with red arrows. The red grid, on the other hand, can be redirected and scaled by dragging red arrows. Note that the arrows also show an important feature of these grids. They indicate the concept of x-unit and unit y in their respective grid. The red grid has two red arrows. Each shows the length of one unit on the x-axis and on the y axis. They also dictate the orientation of the coordinate space. We take the red arrow on the x axis and extend it twice as long as the original arrow blue). Follow the following images: We see that the image (green box) drawn from the red net is now stretched horizontally because this red grid being pulled onto is now twice as wide. The point I'm trying to make is quite simple: you can use one coordinate space as a base to change the other coordinate space. Step 3: Affine Coordinates Space So what are the affine coordinates of space? I'm sure you're careful enough to make sure these coordinate rooms are drawn using parallel nets. Let's take the red affine space for example: there is no guarantee that both the X-axis and the y-axis will always perpendicular to each other, but you can be sure that but you try to tweak the arrows, you can never reach such a case as below. This coordinate room is not the coordinate room. In fact, x and y axes usually refer to the cartes coordinate space as shown below. Note that the horizontal and vertical squares are perpendicular to each other. Descartes is a type of affine coordinate room, but we can turn it into other insurers, as we prefer. Horizontal and vertical networks need not necessarily be perpendicular to each other. Example of affine coordinate space Another example of affine coordinate space Step 4: Affine Conversions As you might have guessed, conversions are related to translation, scaling, reflection, rotation, and rotation. Original fixing room Scaled fixing space Reflected fixing space Skewed fixing space Rotated and scaled space Without saying clear, physical features such as x, y, scaleX, scaleY and rotation depend on the space. When we call these qualities, we actually change the coordinates of the affine. Step 5: Understanding the Matrix I hope that the images shown above are clear enough to draw home the idea. This is because the programmer works with FlashDevelop, we do not see these networks that Flash IDE conveniently displays by designers. All of these have to live in your head. In addition to portraying these networks, we must also bring with us the help of the Matrix class. Therefore, having a mathematical understanding of the matrices is important, so we look at the activity matrix here: adding and multiplying. Step 6: Adding matrix to matrix operations has geometric meaning. In other words, you can imagine what they mean by the graph. Let's say we have four points in our coordinate room and would like to take them to new locations. This can be done using the matrix addition. Check out the picture below. As you can see, we actually shift the entire local coordinate space (red grids) where these four dots are drawn. The indication to perform these actions is as follows: We can also see that this exchange can actually be represented by a vector (tx, ty). We distinguish between vectors and static dots in coordinate rooms when using brackets and square brackets. I re-wrote them on the picture. Step 7: Apply ActionScript Here you have to apply the matrix in addition. Check out the comments: Public Class In addition, Sprite { public feature Upgrade() { 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 dotted box(); //create custom graphic(dotted box in Sprite) addChild(s); d.transform.matrix = m; //apply matrix to our graphics } Step 8: Geometric meaning Matrix multiplication matrix multiplication is somewhat more complex than the matrix addition, but Prof Wildberger has elegantly distributed it down to this simple interpretation. I humbly try to repeat his explanation. For those who'd like to dive deeper into the understanding of the linear algebra that leads to it, check out the professor's lecture series. Let's start with the identity matrix case, I. From the above image, we know that multiplying arbitrary matrix A by the identity matrix, I, always produces A. Here's an analogy: 6 x 1 = 6; the identification matrix has been compared to its multiplied number 1. Alternatively, we can write the result in the following vector format, which greatly simplifies our interpretation: The geometric interpretation of this formula is shown in the screenshot below. Cartesian grid (left grid), we see blue dot located (2, 1). Now that we would have to convert it to the original grid x and y the new grid (the correct grid) according to the set of vectors (below the correct grid), the blue dot will be relocated (2, 1) to the new grid - but if we map it back to the original grid, it will be the same point as before. Because we're turning the original network into another network that shares the same vectors for x and y, we don't see the difference. In fact, the changes x and y in this transformation are zero. This is what it meant by the identity matrix, from a geometric point of view. However, when we try to do mapping using other changes, we see some difference. I know it wasn't the most revealing example to start with, so let's move on to another example. Step 9: Scale the x image above shows the scaling of the coordinate space. See x vector in converted coordinate space: one unit of converted x is two units of the original x. The redesigned coordinate room still has a blue point coordinate (2, 1). However, if you try to map this coordinate from the converted grid to the original coordinate, it is (4, 1). This whole idea is captured in the image above. How about a formula? The result should be consistent; Let's check it out. I'm sure you remember those formulas. I added their meanings. Now check out the numerical result of our tag example. Initial coordinate: (2) Vector on transformed x-axis: (2,0) Vector on redesigned y-axis: (0,1) Expected result: (2*2 + 0*1, 0*2 + 1*1) (4, 1) They agree with each other! Now we can happily apply this idea to other changes. But before that, actionscript implementation. Step 10: Apply ActionScript to Check out ActionScript Implementation (and result in SWF) below. Note that one of the overlapping boxes is stretched along the x scale of 2. I have brought forth important values. These values are changed in later stages to represent different changes. 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 dotted box(); //create custom graphic addChild(s); d.transform.matrix = m //apply matrix onto our graphics } } The blue dot is in the original grid (2, 1) before the conversion and in the original grid after the conversion (4)(2). (Of course, it's still (2, 1) in the new grid after the transformation.) And to confirm the result numerically... ... they fit again! To see this ActionScript implementation, simply change the m.d value from 1 to 2. (Note that the y is stretching downwards, not upwards, because y by a step by a point by a point, but upwards in the normal descartes coordinate room I used in the chart.) Here we have reflected the grid along the x-axis using these two vectors, so that the position of the blue point in the original grid changes (2, 1) to (-2, 1). The numeric calculation is as follows: The application of ActionScript is the same as before, but using these values: m.a = -1, m.b = 0 to represent the vector for the x-conversion, and: m.c = 0 and m. d = 1 represents the y transformation vector. Next, how about we mirror the X and y at the same time? Check out the picture below. Also calculated numerically in the picture below. To apply ActionScript... I'm sure you know the values to put in the matrix. m.a = -1, m.b = 0 to represent the vector of the x conversion; m.c = 0 and m. d = -1 represents the y-transformation vector. I have added the final SWF below. Skewing comes with a bit of fun. For the image below, the redesigned grid is redirected and scaled. Compare the red arrows in both grids below: they are different, but the y axis remains the same. Visually skewing, it seems to be a distortion of i-direction. This is true because the x-axis around us is now the y-component of its vector. Numerically, that's what happens... Regarding implementation, I have listed the tweaks below. m.a = 2 m.b = 1 m.c = 0 m.d = 1 I'm sure at this point like try things yourself, so go ahead and tweak the orientation around the y-axis while maintaining the x-axis orientation so on the axis together I've added flash output in both cases as below. For readers who want help with these values, see Multiplication_final.as a source download. I consider the rotation to be a subset of skewing. The only difference is that both the x and y-axis units are maintained during rotation, as is the perpestal ness between the two axes. ActionScript actually provides a method in the matrix class, rotating() to do so. But let's still go through it. Now we do not want to change the unit length size in x and y from the original grid; just change the orientation of each. We can use trigonometry to get to the result shown in the above screenshot. Taking into account the angle of inclination, a, we get the desired result using the x-axis and (-sin a, cos a) vectors for the x-axis and (-sin a, cos a) for the y axis. Each new axis has a single unit of magnitude, but each axis is at original angles compared to the originals. For the implementation of actionscript, assuming that the angle a is 45 degrees (i.e. 0.25*Pi radians), simply tweak the matrix values as follows: 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 entire source may be referred to Multiplication_final. Step 15: The application for taking vector interpretation of the 2x2 matrix opens up space for us to explore. Its application by manipulating bitmap images (BitmapData, LineBitmapStyle, LineGradientStyle, etc.) is widespread - but I think I saved that in another tutorial. For the case of this article, we try to distort our sprite run-time so that it looks like it's actually flipping 3D. The view of the pseudo-3D isometric world above image shows that in the world with an isometric view, each graphic image that is standing keeps its y-axis vector unchanged when the vector of the X axis rotates. Note that the x and y-axis length units do not change - in other words, you should not mastata in either axis, just rotation around the x axis. Here's an example of this idea for Flash. Click anywhere on stage and start seeing the fish whistle. Release to stop your communication. Here is an important bit of Actionscript. I highlighted important features for the rotation of the X-axis. You can also refer FakeIso.as. era var f1: Fish, m:Matrix; eravar disp:Punkt; private var axisX:Point, axisY:Point; public feature FakeIso() { disp = new Point (stage.stageWidth * 0.5, stage.stageHeight * 0.5); m = new Matrix(); m.tx = disp.x; m.ty = disp.y; //place located in the middle of step f1 = new Fish(); addChild(f1); f1.transform.matrix = m; //apply transformation onto fish axisX = new point(1,0); /vector/axisY = new Point(0,1); //vector for y - stage.addEventListener(MouseEvent.MOUSE_DOWN, start); interaction stage.addEventListener (MouseEvent.MOUSE_UP, end); end communication } 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); //specify orientation (but magnitude changed as well) axisX.normalize(1); //fix magnitude vector new orientation 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've used point class storage vectors. Step 16: Add keyboard control at this step, we will try to add keyboard controls. The location of the fish will be updated according to its speed, speed. We define incremental steps for positive (clockwise) rotation and negative (counterclockwise) rotation. velo = new point 1.0; the velo is used to determine the x-axis Y = 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 At keystroke, the velo: private keyUp(e:KeyboardEvent):void { if (e.keyCode == Keyboard.LEFT) { velo = delta_negative.transformPoint(velo) //rotate velo counter-clockwise } If (e.keyCode == Keyboard.RIGHT) { velo = delta_positive.transformPoint(velo) //rotate velo 24/7}} Now for each frame, we will try to paint the front of the fish and also distort the fish. If speed, speed, is of a magnitude of more than 1 and we apply it to the fish matrix, m, we can tag the effect as well – so in order to eliminate this option, we normalize the speed and then only apply that to the fish matrix. private function update(s):event):void { var front_side:Boolean = velo.x > 0 //checking for the front of the fish if (front_side) {f1.Body color(0x002233,0.5) } //color fish front face f1.colorBody(0 xFFFFFF,0.5) //white, applied to the back of the fish disp = disp.add(velo); //update flow shift at var velo_norm:Point = velo.clone(); //, if speed > 0, we need to recalculation of 1 length unit for x. velo_norm.normalize(1); //note that the x axis is greater than 1, performs scaling. We do not want to now 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 on the stage, then press left and right arrow keys to see how the fish change direction. To spice things up, let us check the y-axis vector as well. private keyUp(e:KeyboardEvent):void { if (e.keyCode == Keyboard.LEFT) { velo = delta_negative.transformPoint(velo) } other than (e.keyCode == Keyboard.RIGHT) { velo = delta_positive.transformPoint(velo) } than (e.keyCode == Keyboard.up) { axisY= } Otherwise, if (e.keyCode == Keyboard.DOWN) { axisY = delta_positive.transformPoint(axisY) } Also set the front of the fish, we must now add the y-axis into Here is the code that: var front_side: Boolean = velo.x * axisY.y > 0 if (front_side) { f1.colorBody(0x002233,0.5) } otherwise f1.colorBody(0xFFFFFF,0.5) Step 19: Your No-So Regular Fish Well, some result control on both axes may prove a little confusing, but the thing is that you can now distort your fish , translate, mirror it, and even rotate it! Try combos up + left, up + right, down + left, down + right. Also check if you can store the front side of the fish (the fish is gray). Hint: continuously tap, then left, then down, then right. You're rotating! Conclusion I hope you will find a matrix of mathematics valuable assets in your projects after reading this article. I hope to write a little more applications in the 2x2 matrix with little Quick Tips branching out of this article and Matrix3d which is important for 3D manipulations. Thanks for reading, terima kasih. Kasih, kasih, kasih.