7

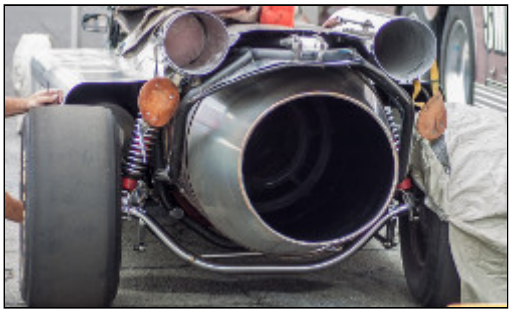
Making a Game Engine: Transformations

6

By **James Lambert** | Published Feb 15 2014 01:17 PM in Math and Physics
Peer Reviewed by (joew , dejaime, Dave Hunt)

game engine | transforms | matrices

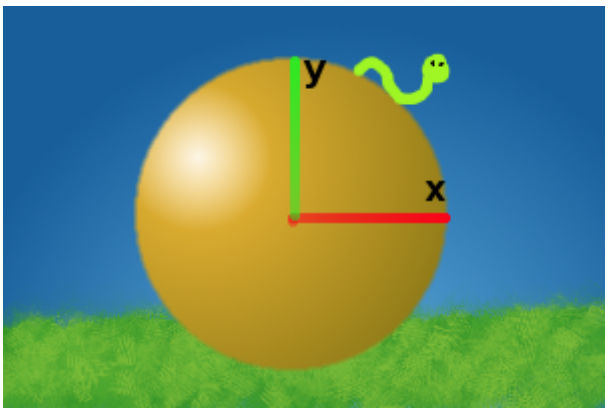
See Also:
Making a Game Engine: Core Design Principles



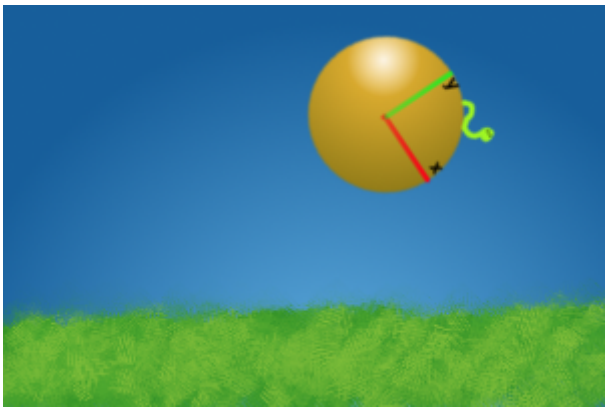
Moving objects around in a scene is not hard to do. You simply draw objects at different locations each frame. Adding rotation, and resizing on top of that makes the code a little trickier but still not hard to manage. But then start trying to move objects relative to another such as putting a sword in a character's hand and things can get really tricky. The sword should stay in the character's hand regardless of what direction they rotate or where they move. This is where using matrices to represent these transformations is extremely valuable.

Coordinate Systems

In order for a coordinate to have any meaning there needs to be a point of reference to base that coordinate off of. Imagine you have a dot on a piece of paper. The dot does not have a coordinate unless you come up with a way to measure its location. One way would be to determine the distance the point is from the left side of the page and its distance from the bottom. These two measurements give the coordinate meaning since it has a point of reference, the bottom left corner of the page. The point of reference is known as a basis and any point measured relative to the same basis are said to be in the same coordinate system. If a worm were on a ball you could devise a way to measure the location of the worm on the ball. This coordinate would be the worm's location in the ball's coordinate space.



The ball itself also has a location but its location is measured in world coordinates. Let's assume that the location of the ball is measured from the bottom left corner of the image to the center of the ball. If you were to throw that ball the worm would move with the ball. The position of the worm in world coordinates would move as the ball moves, even though its position in ball coordinates stay the same. Transformations give us a way to convert between these coordinate systems relatively easy using matrices allowing us to easily keep the worm on the ball as it travels through the air.



Matrices have a lot of useful properties that lend themselves to solving many complex problems. We will be using them to transform objects on the screen. This article will focus on three simple transformations: translation, rotation, and scaling. I will be using 2D transformations for the sake of simplicity. The principles in the article also apply to 3D transformations. If you want to do 3D transformations, search around on the web for 3D transformation matrices. The jump from 2D to 3D is trivial with the exception of rotations. I am assuming you already know what matrices are and how to multiply them, what the inverse and transpose of a matrix are, and other basic matrix operations. (For help with matrices, try this video series)

So far this talk of coordinate spaces and matrices may seem to be complicated things when all you need to do is move or rotate objects. How exactly do these things help move objects on the screen anyway? Well, imagine you

define a character sprite with a width of 50 and a height of 100. You could define four points that represent the corners of the sprite image (-25, 0), (-25, 0), (25, 100), (25, 0). These points represent the corners of the sprite relative to the character. So what whappens when the character moves? The transform of the player converts the points of the sprite to world space. Then the camera transform converts from world space to screen space to be drawn. If the player clicks on the screen you can use the inverse of these transforms to determine where on the character they clicked.

Basic Transformations

Translation

A translation simply changes an object's location. The matrix used to represent this is pretty simple too.

$$\begin{pmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{pmatrix}$$

T_x and T_y is the number of units to move in the x and y directions respectively.

Rotation

Rotates points around the origin.

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The above matrix will rotate counter clockwise by θ radians.

Scaling

Scaling an object changes its size. Scalar values in the range (0, 1) make objects smaller. A scale of 1 keeps it the same size, a scale greater than 1 makes it bigger. Really all the scale is doing is taking the current size and multiplying by the scale value.

$$\begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Scales along the x and y directions by S_x and S_y respectively. When $S_x = S_y$ the size will change uniformly. If you only modify a single value then the object will stretch meaning you can make an object wider by setting S_x to a value greater than 1 and keeping y the same. You can even make a value negative which will flip the object.

Using Matrices as Transforms

Transforming Points

Now that we have a few different transformation matrices we need a way to transform points using them. To do this

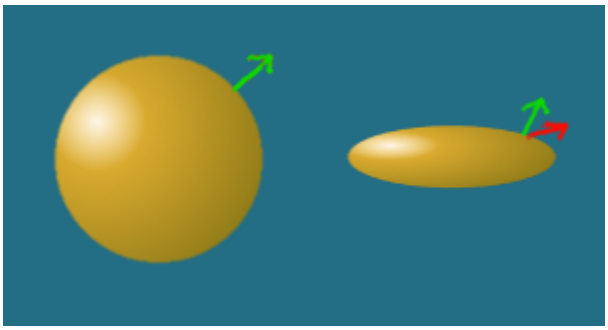
you simply treat the point as a 1x3 matrix padding the extra slots with a 1 and multiplying it with the transform matrix.

$$\begin{pmatrix} x' \\ y' \\ w \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

The result of that multiplication is a another 1x3 matrix. Just taking the top two elements of the resulting matrix gives a new x, y pair of the point in the new coordinate system.

Transforming Directions

Transforming a direction is slightly different than transforming a point. Take a look at the example below.



The ball has been squashed. This is achieved by scaling the y direction down while keeping the x the same. If directions were transformed using the same matrix that were used to transform points, then you would get the red arrow as a result, but notice how the red arrow is no longer perpendicular to the ball. We want the green arrow that still is. To get that result, use the following equation.

$$(M^T)^{-1} \cdot \begin{pmatrix} x \\ y \\ 0 \end{pmatrix} = ((\begin{pmatrix} x & y & 0 \end{pmatrix} \cdot M^{-1})^T$$

As shown above, to get the proper matrix for transforming direction you need the inverse transpose of the transform matrix. You can avoid transposing a 3x3 matrix if you simply do a row vector multiplication in front of the matrix. Also take note that the third element is a 0 instead of a 1. This makes transforming a direction uneffected by any translations. This is proper behavior for transforming a direction since directions still point the same way regaurdless of where they are moved to.

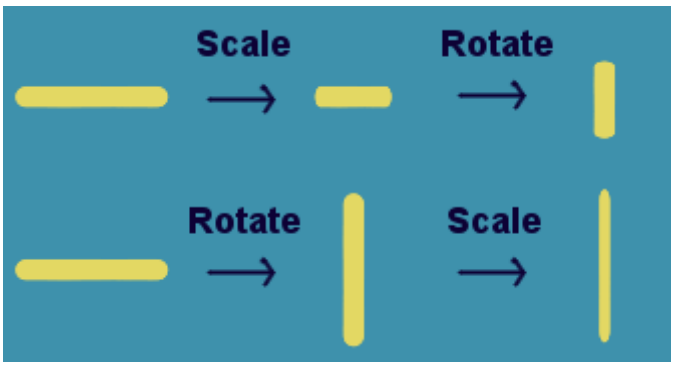
Concatenating transforms

The three basic tranformations listed above aren't that useful on their own. To make things interesting we want to be able to join them together allowing objects to be translated, rotated, and scaled simultaneously. To do this we simply multiply the matrices together.

$$M = T \cdot R \cdot S$$

The resulting matrix, *M*, is a combination of scaling an object, rotating it, then translating it. Keep in mind that the order of *T*, *R* and *S* does matter. Since *S* is the right-most matrix its transformation will be applied first when transforming a point followed by *R*, followed by *T*. To see this grab some object nearby, such as a pencil, and hold it so it is pointing to the right. Now imagine that the pencil is first scaled by half in the x direction making the pencil

shorter but not changing its thickness. Then the pencil is rotated 90 upward. The result is a short pencil with the same width pointing upward. Now switch the transformation order. First rotate it upward then scale. The resulting trasnform is a pencil of the same length but different width. So pay attention to the transform order.



One thing to note is that roatations pivot around the origin. This means if you move an object up ten units then rotate it clockwise 90 degrees it will end up 10 units to the right. If you want the object to pivot around its own center, rotate it first then move it.

Since matrices can be multiplied to concatenate transforms you can easily keep an object at the same position relative to another, like the example of the worm on a ball described above. You first create a transform using the position and orientation of the worm on the ball. This transform represents the conversion of the worm's local coordinates to the ball coordinates. You then create the transform of the ball relative to world coordinates. Since the worm is on the ball, to find the transform from worm coordinates to world coordinates you simply multiply them together.

```
Matrix insectToWorldTransform = ballToWorldTransform * insectToBallTransform;
```

Column Vector vs Row Vector

Before moving into the next section I should mention the difference between column vectors and row vectors. I have been using column vectors in this article. This means vectors are represented as a 3x1 matrix that is multiplied on the right of the transform.

$$\begin{pmatrix} x' \\ y' \\ w \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

The alternative are row vectors. A row vector is a 1x3 matrix and is multiplied on the left of the matrix.

$$\begin{pmatrix} x' & y' & w \end{pmatrix} = \begin{pmatrix} x & y & 1 \end{pmatrix} \cdot \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix}$$

Mathematically, these two methods are the same. Either one works. Some prefer how compact column vectors are to write out on paper. Others find row vectors to be more intiuative. The important thing is to pick one and be consistent with it. However, the graphics package you are using may require matrices for row vectors or column vectors. If you are using the other method you will need to convert your matrices before passing them in. To do so, simply transpose the matrix. As an example, the translate matrix using column vectors is

$$\begin{pmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{pmatrix}$$

The same matrix for row vectors is

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_x & T_y & 1 \end{pmatrix}$$

Also, keep in mind that when using row vectors the order to multiply matrices in is reversed.

$$(T \cdot R \cdot S \cdot v)^T = v^T \cdot (T \cdot R \cdot S)^T = v^T \cdot S^T \cdot R^T \cdot T^T$$

So using row vectors, the left most matrix is applied first to the point and the matrix on the far right is applied last. An easy way to remember the difference is the matrix closest to the vector is applied first. This means the order is left to right using row matrices and right to left using column matrices.

Transform Class

Now that we have covered some basics of how transforms work let's build a transform class. Since we don't want to deal with matrices at a high level, we will allow the transform to be manipulated using a position, rotation, and scale. The transform class will also have children to allow for a heirarchy. When the parent transform moves all of the children move with it and the children's position, rotation, and scale will be defined relative to its parent. Finally the transform will extend the component class that is attached to a game object. Game objects and components are touched upon in this article.

```
class Transform extends Component
{
    // the parent transform of this transform
    // if it is null then the parent transform
    // is the world coordinate system
        private Transform parent;
    // all of the transforms that have this
    // transform set as their parent
    private Transform[] children;

    // the position relative to the parent transform
    private Vector2 localPosition = new Vector2(0.0f, 0.0f);
    // rotation relative to the parent
    private float localRotation = 0.0f;
    // scale relative to the parent
    private Vector2 localScale = new Vector2(1.0f, 1.0f);

    // specifies if the localToWorldTransform
    // needs to be recalulated
    private bool isDirty = false;
    // the transform that converts local coordinates
    // to world coordinates
    private Matrix localToWorldMatrix = Matrix.identity;

    // specifies if the worldToLocalMatrix
    // needs to be recalculated
    private bool isInverseDirty = false;
    // the transform that converts world cooridnates
    // to local coordinates
    private Matrix worldToLocalMatrix = Matrix.identity;
```

There is a lot of code there so let me explain the key points. First of all, I am using a dirty flag to indicate when the transform matrices are out of date. That allows the position, rotation, and scale to be changed multiple times and the matrix is only actually recalculated once it is requested, reducing the amount of uneeded recalculations. Also take note that the dirty flag is only set if the transform is not already dirty. This is to keep the `setDirty` call from propagating to all of the children every time the transform is modified and only setting the dirty flag when necessary.

The inverse matrix is also stored in the transform to keep from having to calculate it everytime it is needed. It has its own dirty flag so it isn't calculated if it isn't needed.

Using the Transform Class

Now that we have a transform class, let's see how it can be used. First of all, the `localToWorldMatrix` can be used in draw calls. Most drawing libraries will allow you to specify a matrix to position objects on the screen.

```
class Renderer extends Component
{
    void render(graphics)
    {
        graphics.setTransformMatrix(this.gameObject.transform.getLocalToWorldMatrix(
        graphics.drawSprite(this.frame);
    }

    // whatever else the renderer does
    // would go here
    ...
}
```

Keep in mind the above code doesn't account for the view transform. Think of the view transform as the camera for the scene. If you wanted to be able to scroll the view of the scene you should specify a camera object and multiply all transforms by the `worldToLocalMatrix` from the transform of the camera.

```
        graphics.setTransformMatrix(
            camera.transform.getWorldToLocalMatrix() *
            gameObject.transform.getLocalToWorldMatrix());
```

The transform could be used in game logic code

```
class Mine extends Behavoir
{
    void update(float deltaTime, InputState input)
    {
        // transforming the zero vector gets that
        // transform's origin in world coordinates
        Vector2 playerWorldPosition = player.transform.transformPoint(new Vector2(0.

        // take the players world position and convert it the mine's
        // local coordinates
        Vector2 playerLocalPosition = this.transform.inverseTransformPoint(playerWor

        // since playerLocalPosition is the players position relative to the mine
        // the magnitude of the position is the distance from the mine to the player
        if (playerLocalPosition.magnitudeSquared() < sensorRange * sensorRange)
        {
            this.detonate();
        }
    }
}
```

Conclusion

Transforms may take a while to fully grasp but once understood help simplify problems that would otherwise be a huge challenge to tackle. Hopefully this article helped you understand how to use them in your games better.

Article Update Log

5 Feb 2014: Initial release

License

Comments

Strewya

Feb 11 2014 03:26 AM

My suggestion is to mention the other way matrices can be represented and multiplied. You use a colum-vector based representation, and thus multiply your vector and matrices in reverse order (at least as i see it): TRS**v**. I personally find this awkward, and use a row-vector representation, and multiply vector and matrices in **v**SRT order, as it feels more natural to me. You don't have to change your article (which is really useful), but just mention the other approach as well.

Burnt_Fyr

Feb 11 2014 07:17 PM

+1 to what Strewya said above... It took me a long time....a really long time to really peg down column vs row vectors, and it's because there were so many articles back then that never mentioned which convention they were using.

I would also add, that your title is a little misleading, as you don't cover hierarchical transforms at all.

NC1988

Feb 11 2014 09:53 PM

that's really helpful for me. thank you!

Rohithzhere

Feb 12 2014 12:16 AM

Great article!!Thanks a lot!

slayemin

Feb 12 2014 05:53 PM

I agree with Burnt_Fyr: The title is misleading. There is no mention of a transform heirarchy in this article. It's just the basics of transformation matricies. You really should add in a section which talks about scene graphs and what you think are the best practices for implementing them.

Imagine an arm for a moment. It's attached to a torso at the shoulder joint. From the shoulder, you've got a bicep, which leads to an elbow joint, which then leads to a wrist joint, and finger joints. The hand is holding a flail, which is a stick with a chain, with a ball attached to the chain. If the shoulder rotates, the arm moves and all joints attached to it should move. If the elbow moves, all objects past the elbow should move. If the flail swings, the ball should rotate around the anchor point. If the flail is dropped, subsequent rotations of a shoulder should not change the flail position. This can all be done with a scene graph and matricies. When you title your aricle with "hierarchical transforms", this is what I'm expecting to read about. So... Keep writing, you're not done yet.

DemonRad

Feb 13 2014 07:36 AM

Totally agree, scene graph is a very interesting topic, I'd like to see an article both about a standard scene graph and a lazy scene graph. => bottlenecks of scenegraphs, and ways to reduce them.

dejaime

Feb 13 2014 11:37 AM

As others said, it looks like you neglected the hierarchy side of the transformations, limiting the article to simpler ones. Maybe you could add that specific content or change the article title to limit its scope to reflect what is really being covered (what, by itself, is not that little content).

#edit: changed my vote due to the title change.

Bacterius

Feb 16 2014 02:59 AM

You could probably fit a paragraph about why we need a 3x3 matrix (i.e. homogeneous coordinates) for 2D transforms, especially since a lot of stuff you find online about 2D transformations show 2x2 matrices for rotation and scaling but completely fail to mention translation which is where the extra dimension comes in, so it would perhaps be useful to provide *some sort* of explanation as to where it is coming from.

Technically you don't need the last row (or last column in row-major order) since 2x3 (resp. 3x2) transformation matrices form a group over affine transformations. You only need it for nonlinear operations such as perspective projection which have no place in geometric transforms, however getting rid of it does make the notion of transpose/inverse a bit strange (though still perfectly well-defined). Anyway, one certainly doesn't need to know that to use them, but I stand by my first paragraph.

Note: Please offer only **positive**, **constructive** comments - we are looking to promote a positive atmosphere where collaboration is valued above all else.