2d Transformations

Intro

You will probably find, as I did, that 2 dimensional transformations, while being not as cool as 3-d transformations, are very versatile and really easy to use. What would your favorite game be like without an overhead map to guide your way! This tutorial covers the hard-core transformations for scaling, rotating, and translating 2d points. As I mentioned, all are very easy to use and understand so let's get to it!

Data Representation

If we just wanted to do 2 dimensional transformations on one point, this subject wouldn't be that big of a deal, but let's get serious, it's much more likely that we will be transforming hundreds or even thousands of points at a time. With that many points, we must devise a way to contain point data without making it any harder to read. Since we are just starting out, let's just put the data into a structure like this:

```
typedef struct point
{ float x,y;
  point(){x=y=0.0;
} point2d;
```

Since we are declaring this structure, we could easily dynamically allocate as many instances as we wanted! What data type to use also depends on the application. If we really wanted to get serious about speed and precision, we could use 32 bit longs and fixed point math. For now, I won't confuse you and I'll stick to slower floating point representation that is easier to read! Let's get to those transformations!

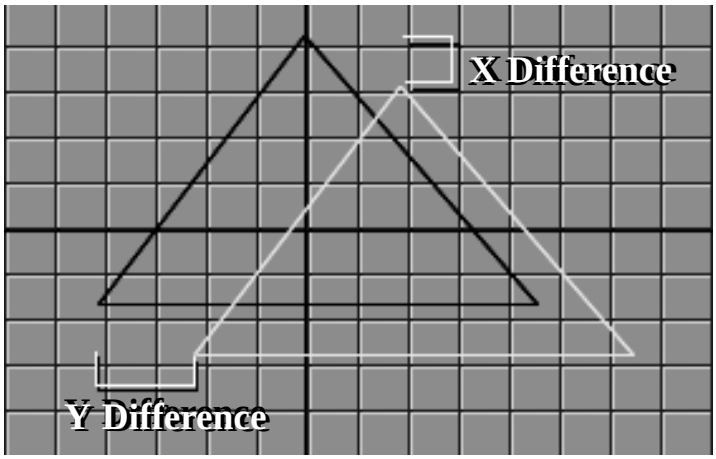
Translation

Translations with 2 dimensional points are extremely easy! All we need to do is determine how much we want to move the object in the x and y direction, and add those variables to the point's x and y respectively.

We will always be doing translations since the upper left of our screen is (0,0) and we want (0,0) to be the middle of the screen. If we didn't do the alignment, we would just be able to see anything with a positive x and a negative y which leaves out 3/4 of the viewing field! Let's encapsulate all of the appropriate code within its own class so we can reuse the code whenever and wherever we want, and make it really readable. Remember to design your class or group of functions for a particular application. I like to build classes and make them flexible enough so I can reuse them in just about anything.

You may want to narrow down the scope of the functionality a bit yourself. Here's the function that will allow us to do translations.

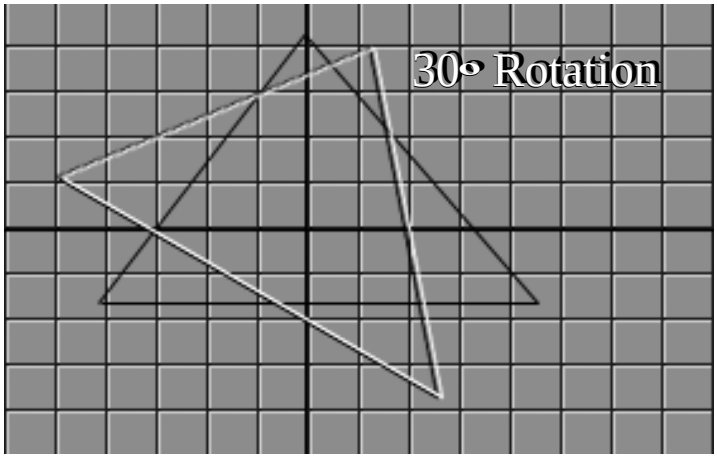
```
point2d Transform2d::TranslatePoint(float xdiff, float ydiff, point2d &point)
{ point2d newpoint;
  newpoint.x=point.x+xdiff;
  newpoint.y=point.y+ydiff;
  return newpoint;
}
```



This function allows us to add a certain amount to the x and y coordinates of a point, but notice that we don't do the alignment here, we'll get to that later. Also notice that we are preserving the value of the point passed to us, instead we create a newpoint and return that instead.

Rotation

Rotation is the only 2d transformation that actually requires any amount of thought, so beware! Each point is modified using an ancient equation that you probably learned in high school. The only tricky aspect of this function is that each object rotates around (0,0), so if we want to position an object's center at (1000,1000) and have it rotate around its center, we must first rotate it then translate it so it will appear correct!



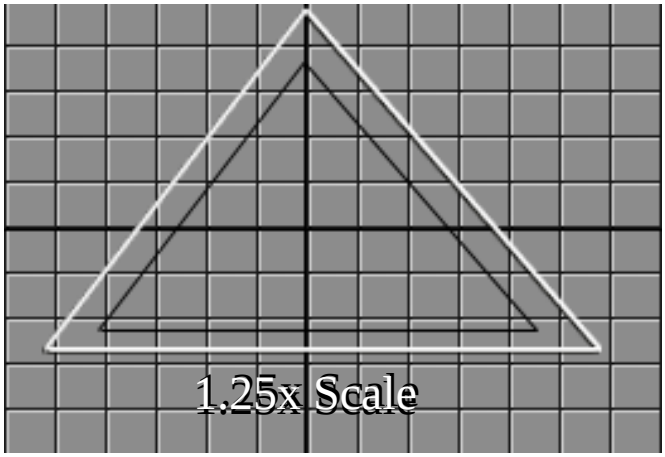

```
point2d Transform2d::Rotate(float angle,point2d &point)
{ float xold,yold;
  point2d newpoint;
  xold=point.x;
  yold=point.y;
  newpoint.x= xold*cos(angle)-yold*sin(angle);
  newpoint.y= xold*sin(angle)+yold*cos(angle);
  return newpoint;
}
```

This equation should kinda remind you of transforming Cartesian coordinates into polar coordinates. If it doesn't, it probably means you weren't paying attention in class!! We first save the old x and y coordinates and then use them in our equations. They are both very similar to each other, but notice the - and + if you get those switched or matched these equations will produce distorted results which are kinda cool, but not what we want!! Remember that with these equations, the object will always be rotated around (0,0) so if you want something rotating about its own center and not be located at (0,0), you must first rotate it and then translate it! Also notice that we are returning a point instead of modifying the one passed to it. We are doing this because we always want to retain the positions of the original points. Modifying and saving the data to the same point will produce distorted, unreliable results.

Scaling

This is a really neat feature to include if you are going to use 2d points for an overhead map because you can zoom in an out to get the amount of detail you like! This function is also dependent on the fact that all points must be centered around (0,0). If we don't follow this rule we will get object flying all over the place! Take a look at the equation and see!

```
point2d Transform2d::Scale(float scale,point2d &point)
{ point2d newpoint;
  newpoint.x= point.x*scale;
  newpoint.y= point.y*scale;
  return newpoint;
}
```



As you can see with this simple equation, if an object's points are not centered around (0,0) it will be hurled in a direction and look like its being translated and if it does manage to look scaled, it won't look correct. So one more time, make sure that you objects are centered around (0,0)!!

Thre's a nifty little program that demonstrates some of the capabilities of 2d transformations located at www.invsersereality.org! Thanks for reading and I hope you come back soon to take a gander at the rest I have to offer! If you have any question, comments, complaints, whatever! send me some Feedback.

Contact Information

I just wanted to mention that everything here is copyrighted, feel free to distribute this document to anyone you want, just don't modify it! You can get a hold of me through my website or direct email. Please feel free to email me about anything. I can't guarantee that I'll be of ANY help, but I'll sure give it a try :-)

Email : deltener@mindtremors.com
Webpage : http://www.invsersereality.org