08214 Lab 4_1: Animation, Simple Collision Detection and Response

In this lab we will learn about animation, simple collision detection and physical simulation. There are many different approaches to collision detection, with advantages and disadvantages. In this lab most of the work will be done in 2d, but many aspects translate directly into 3d. Other techniques (such as calculating the shortest distance from a point to a line) have different meaning and uses in three dimensions (cylinder intersections, for example).

There is a lot of maths involved. Try to stay focused and don't get disheartened. If you get stuck, get help! Help is available from demonstrators in labs, from the forum or from module staff via email. It is expected that in order to complete this lab fully it will take at least three hours, so expect to spend a significant amount of your own time outside of labs working through this.

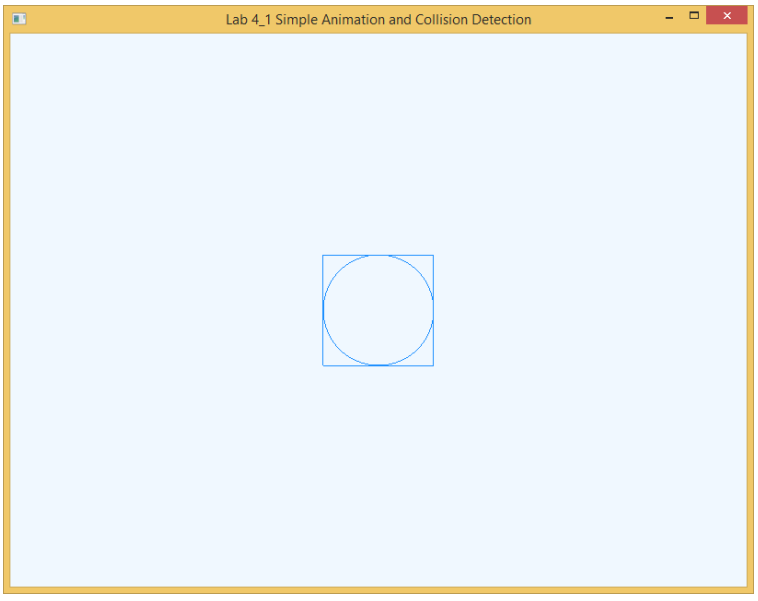
If you have any problems (or spot a mistake) one way you can get help is on the forum thread for lab 4.1 here at

http://forums.net.dcs.hull.ac.uk/p/1953/6565.aspx

In the Program.cs file change the m_CurrentLab value to Lab.L4_1

```
private static Lab m_CurrentLab = Lab.L4_1;
```

This will run the correct window for this lab. You should see something like this:



Let's take a look at the code. A lot of this should look fairly familiar by now. Here are the important things to note for this lab. In the OnLoad method where we set up the vertex buffer objects we create a

circle and a square. Both are centred around 0,0. The circle has a radius of one and the sides of the square are two units in length.

In the OnRenderFrame method we render the buffer arrays using LineLoop.

```
GL.UniformMatrix4(uModelMatrixLocation, true, ref mSquareMatrix);
GL.BindVertexArray(mVertexArrayObjectIDArray[0]);
GL.DrawArrays(PrimitiveType.LineLoop, 0, 4);

Matrix4 circleMatrix = Matrix4.Identity;

GL.UniformMatrix4(uModelMatrixLocation, true, ref circleMatrix);
GL.BindVertexArray(mVertexArrayObjectIDArray[1]);
GL.DrawArrays(PrimitiveType.LineLoop, 0, 100);
```

The square is given a model matrix mSquareMatrix, and the circle is given the identiy matrix.

This lab is all about animation and physics. We're going to be animating the circle by changing its position each frame. First we need to create a Vector3 data member called mCirclePosition to store the circle's position. In the OnLoad method instantiate the circle to 0,0,0.

In OnRenderFrame change the line that sets the circleMatrix to be the Identity matrix to the following:

```
Matrix4 circleMatrix = Matrix4.CreateTranslation(mCirclePosition);
```

This translates the circle to the circle position. Finally we're at the really important bit! We're going to override another of OpenTK's methods; the OnUpdate method. This is where we update our simulation. In this case, we're going to change the position of the circle by a small amount. Add the following code to the Lab4_1Window class:

```
protected override void OnUpdateFrame(FrameEventArgs e)
{
    base.OnUpdateFrame(e);
    mCirclePostion.X = mCirclePostion.X + 0.2f;
}
```

Run the code. You should see the circle disappear off to the right, but it moves extremely quickly. Commit your code to SVN and then we'll set about adding some more control.

L41T1 Have a fast moving circle

At the moment our circle is moving too fast. What's more, this will move at different speeds on different computers. We need to make our simulation processor independent by using a timer to work out how much time has passed since the last update, and then use that to calculate the distance we should move the circle. A simple Timer class has been provided for you. Create a new member variable in the

Lab4_1Window class of type Timer called mTimer. In the OnLoad method after the class is constructed call the Start method. This effectively starts the timer.

```
mTimer = new Timer();
mTimer.Start();
```

Alter the OnUpdateFrame method to use the timestep.

```
protected override void OnUpdateFrame(FrameEventArgs e)
{
    base.OnUpdateFrame(e);
    float timestep = mTimer.GetElapsedSeconds();
    mCirclePosition.X = mCirclePosition.X + 0.2f * timestep;
}
```

What this does is it changes the amount the sphere moves according to the time. The circle should now move at a rate of 0.2 units per second. If you run the code, it should take the sphere ten seconds to leave the right hand side of the box, which is two units across.

L41T2 Have a slower, processor independent circle by calculating the new position by multiplying the x offset by the timestep

Using the actual time isn't always a good thing. There are times when you will be debugging simulation code and you will want to stop the program running whilst you work out what is going on. When you start the code running again this can result in a very big timestep. For this reason it is suggested that you add a mockGetElapsedSeconds which always returns an appropriate value.

Really, instead of hard coding the circle's movement using magic numbers we should store how the circle moves somewhere. Create a Vector3 member variable called mCircleVelocity. In the OnLoad method instantiate mCircleVelocity to be (0.2f,0,0), then in the OnUpdateFrame method use that vector, multiplied by the timestep, to adjust the sphere's position for all three components, instead of just the X component.

```
mCirclePosition = mCirclePosition + mCircleVelocity * timestep;
```

Check your code is working, then commit your code to SVN with the message:

L41T3 Created a velocity vector member variable for the circle

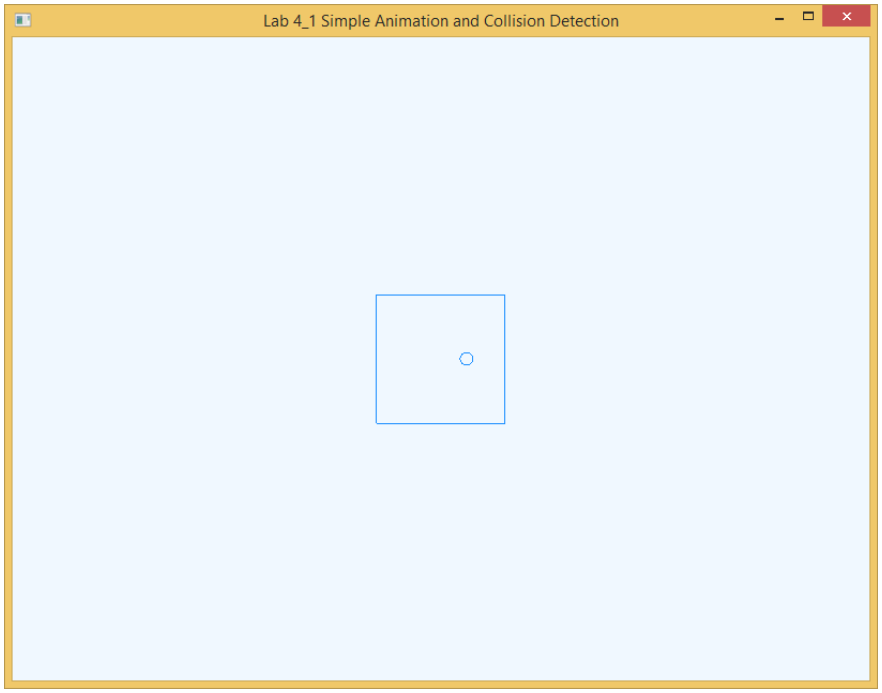
We want to start bouncing the circle around the square, but at the moment there isn't much room inside. Let's make the circle smaller by creating a member variable of type float called mCircleRadius and assign it a value of 0.1f. In the OnRenderFrame method, before we render the circle change the line that says

```
Matrix4 circleMatrix = Matrix4.CreateTranslation(mCirclePosition);
```

To be

```
Matrix4 circleMatrix = Matrix4.CreateScale(mCircleRadius) *
Matrix4.CreateTranslation(mCirclePosition);
```

Run your code and you should see something like this:



Commit your code to SVN

L41T4 Can resize circle using a radius member variable

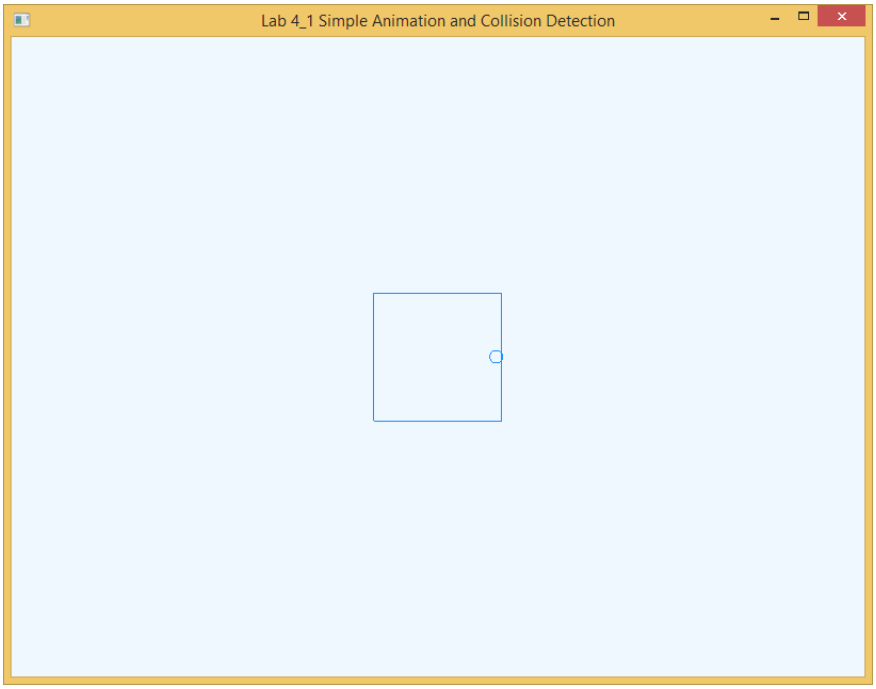
Next we want to stop the circle when it reaches the edge of the box. In this case the box's right hand edge has an x value of 1. After you calculate the sphere's new position add this code:

```
if (mCirclePosition.X > 1)
{
        mCircleVelocity = Vector3.Zero;
}
```

If the circle position is greater than one the circle velocity is set to zero, and the circle should stop moving.

L41T5 Can stop the circle when its centre is at the edge of the square

This stops the circle in the middle of the circle. We need to take into account that the circle has a radius of one unit. We need to stop the circle when it is one circle radius away from the edge of the box. Change the conditional statement to take this into account.



When it stops in the correct place submit your code to SVN

L41T6 Can stop the circle before is escapes the square

Do this for the other four sides of the square and test your solution with circle velocities going up, down and left as well as right. When you are finished change your velocity to right again and commit to SVN.

L41T7 Circle is stopped by all four sides of the axis aligned square

Now instead of just setting the velocity to zero let's add some collision response. For an axis aligned square, it turns out collision response is pretty easy. If we've collided with a side aligned with the y plane we simply reverse the sign of the x component of the velocity. If we collided with a side aligned with the x plane, we reverse the sign of the y component of the velocity. See the hint below.

```
mCircleVelocity.X = -mCircleVelocity.X;
```

Make the necessary changes and then test with a velocity that has both an x and a y component. If it works commit your code to SVN.

L41T8 Can bounce a circle off of an axis aligned line

There is one thing that you might have to look out for. If you look at the algorithm you'll notice that we actually allow the circle to penetrate the line a little before stopping it. This can sometimes result is some nasty juddering effects as the circle gets stuck on the wrong side of the line. One way to prevent this is to store the previous position of the circle before calculating the new position. If the circle collides with a wall then move the circle back to its previous spot which we know to be wholly inside the square.

This adds a small inaccuracy, but it's usually not noticeable. There are other ways to reduce or remove this error (for example testing a position value with half the timestep to see if it's better), but this method is fine for now. Check your code still works and commit to SVN:

L41T9 Made sure no part of the circle can ever leave the square

So far so good, but what if the box is rotated? In the onLoad method change the line that says:

```
mSquareMatrix = Matrix4.CreateScale(1f) * Matrix4.CreateRotationZ(0.0f) *
Matrix4.CreateTranslation(0, 0, 0);
```

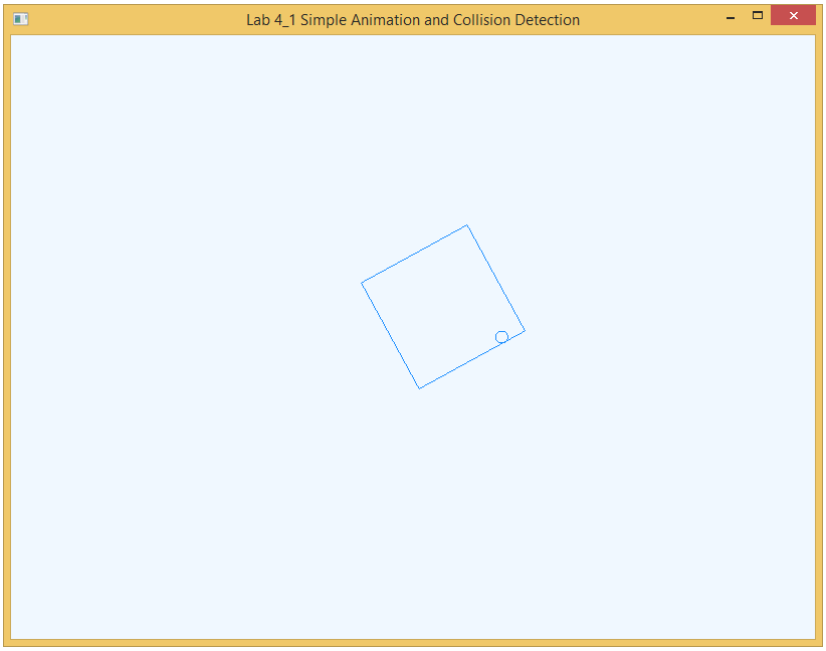
To be

```
mSquareMatrix = Matrix4.CreateScale(1f) * Matrix4.CreateRotationZ(0.5f) *
Matrix4.CreateTranslation(0.5f, 0.5f, 0);
```

This will rotate and translate the square. If you run your code you'll see that the collision code is no longer working properly. To fix this, we're going to move the circle into square space but applying the inverse matrix. After you update the circle positon in the OnUpdate method add the lines:

```
Vector3 circleInSquareSpace = Vector3.Transform(mCirclePosition, mSquareMatrix.Inverted());
```

And then perform your tests on circleInSquareSpace instead. If you run your code you should see that the circle now stops in the appropriate place, but bouncing has stopped working.



For now, let's commit and then worry about the bouncing later.

L41T10 Can collide circle in square space with a translated, rotated square

Take a minute to think about what you just did. You moved the circle into square space.This can be a confusing concept, so to try to make it easier to grasp we're going to draw both spaces at the same time. Add the following code to the OnRenderFrame Method:

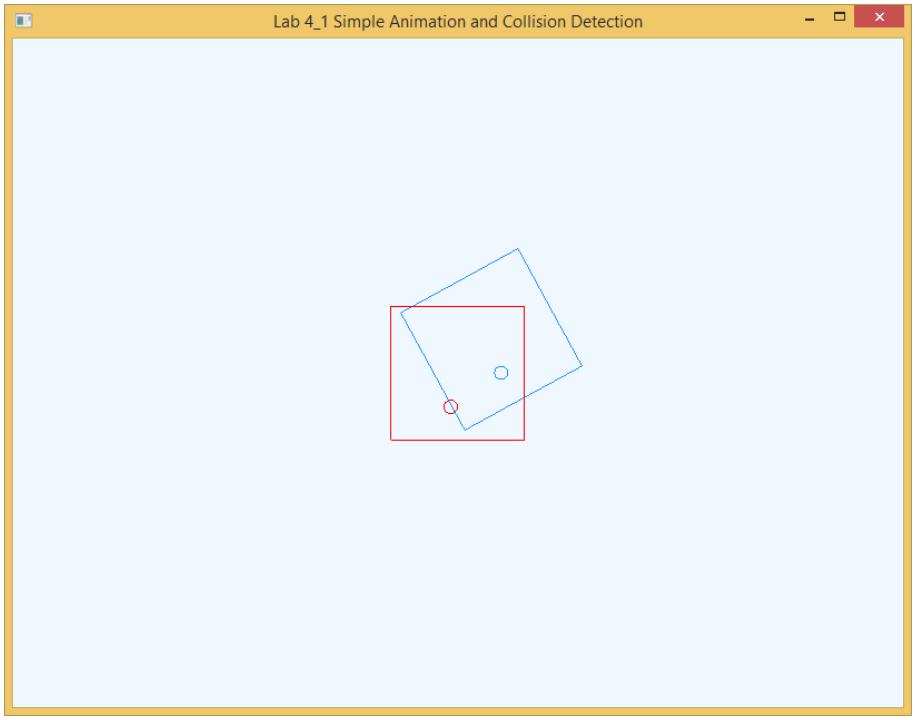
```
GL.Uniform4(uColourLocation, Color4.Red);

Matrix4 m = mSquareMatrix * mSquareMatrix.Inverted();
GL.UniformMatrix4(uModelMatrixLocation, true, ref m);
GL.BindVertexArray(mVertexArrayObjectIDArray[0]);
GL.DrawArrays(PrimitiveType.LineLoop, 0, 4);

m = (Matrix4.CreateScale(mCircleRadius) * Matrix4.CreateTranslation(mCirclePosition)) *
mSquareMatrix.Inverted();

GL.UniformMatrix4(uModelMatrixLocation, true, ref m);
GL.BindVertexArray(mVertexArrayObjectIDArray[1]);
GL.DrawArrays(PrimitiveType.LineLoop, 0, 100);
```

Take a minute to look at the code. What the code does is it redraws the scene in square space. Matrix m is set to the square matrix multiplied by its inverse matrix. This is really just a test and should always result in the identity matrix. Similarly the inverse of the square matrix is also applied to the circle. This moves the circle into square space, where the square is always an axis aligned square centred at the origin. In this space our collision calculation is far easier (because the square is axis aligned).



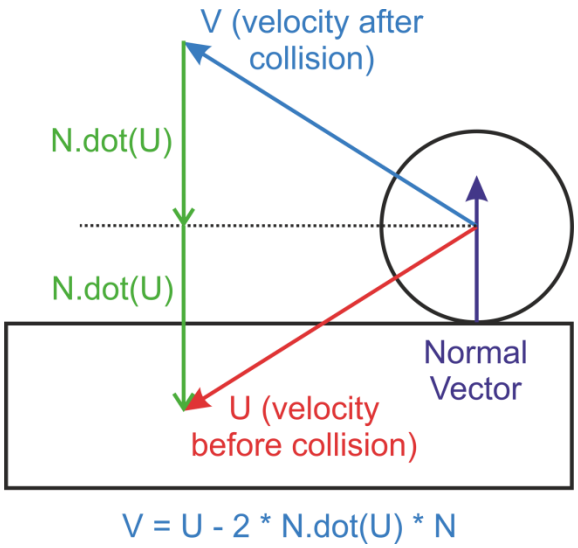
Run the code to see the effect. The red circle and square are in square space. The blue circle and square are in world space. Making a complicated problem simpler is half the battle. Commit your code to SVN:

L41T11 Visualised the problem in world space and square space

Right, so let's fix the collision response. It's actually the translation that's causing us the most bother with the collisions, so set the translation of the squareMatrix to zero.

```
mSquareMatrix = Matrix4.CreateScale(1f) * Matrix4.CreateRotationZ(0.5f) *
Matrix4.CreateTranslation(0.0f, 0.0f, 0);
```

The collisions still aren't right, but we're not stuck in a wall anymore. In the previous solution we took advantage of the special case where we were bouncing off of an axis aligned surface. Now we should take a more general approach. Fortunately, it turns out that bouncing a ball off of a surface and bouncing light of a surface are pretty similar so we can reuse the same skills we used when we calculated lighting. Using the dot product we can calculate a reflected vector with respect to the normal vector. Remember the dot product equals **|a||b|Cos(angle)**. Combine that with some trigonometry and we find the reflected vector as shown below. This is quite an important concept to grasp.



Alter the way you calculate the new velocity after a collision. Here's a hint for colliding with one of the edges. For the rest you're on your own (but if you need help, ask ☺)

```
Vector3 normal = new Vector3(-1, 0, 0);
mCircleVelocity = mCircleVelocity - 2 * Vector3.Dot(normal, mCircleVelocity) * normal;
```

Run your code and you should see that not a lot has changed. The circle is bouncing in just the same way. The good thing though is that because of our code now uses the normal vector were framing our problem in a more general, useful way. Commit your code to SVN.

L41T12 Bounces are still incorrect, but the calculation uses normal vectors now

To solve our problem we need to transform the normal vectors in the similar way to how we transformed the square. Before using the normal vectors apply the square transform to them apply the rotation matrix portion of the square. Here is the calculation for the right side. You can do the rest on your own.

```
Vector3 normal = Vector3.Transform(new Vector3(-1, 0, 0), mSquareMatrix.ExtractRotation());
```

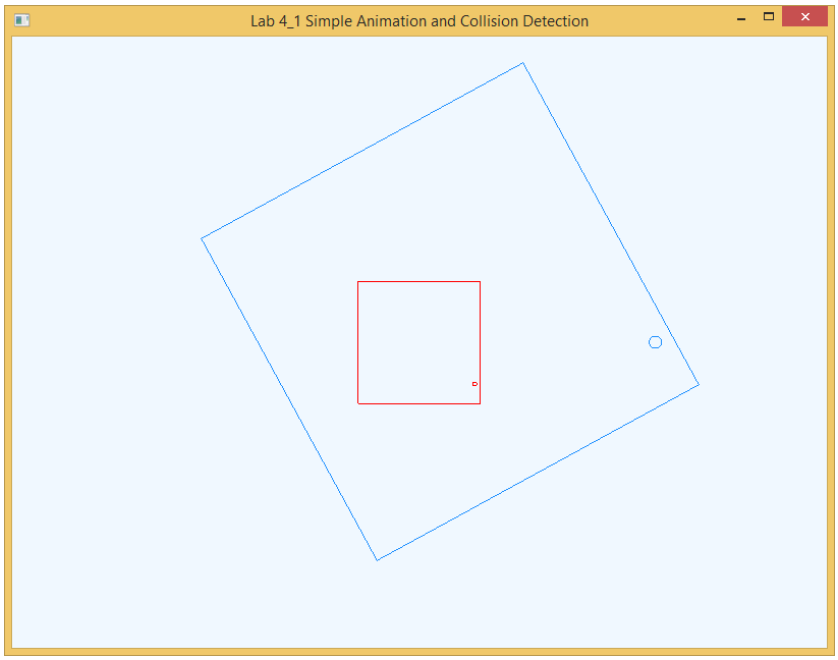
Add the translation back in and see if your code works.

```
mSquareMatrix = Matrix4.CreateScale(1f) * Matrix4.CreateRotationZ(0.5f) *
Matrix4.CreateTranslation(0.0f, 0.0f, 0);
```

If it's working commit to SVN.

L41T13 Bounces are working with a rotated, translated square

The battle isn't over yet though. Go back to the line of code in the onLoad method that sets up the square matrix and set the scale to be 3. Run your code and see the result.



Notice that by making the square three times bigger in blue world space has made the circle three times smaller in square space (where the square is kept the same size). Also notice that our collisions are broken again. That's because we only transformed the point at the centre of the circle into square space. In our collision calculation we're using the radius of the circle, but we're using that in world space. In square space it has been shrunk.

Make the appropriate alterations to your collision code. Here is a hint to help you get started.

```
if (circleInSquareSpace.X + (mCircleRadius / mSquareMatrix.ExtractScale().X) > 1)
```

This Test that your code and then commit your code to SVN:

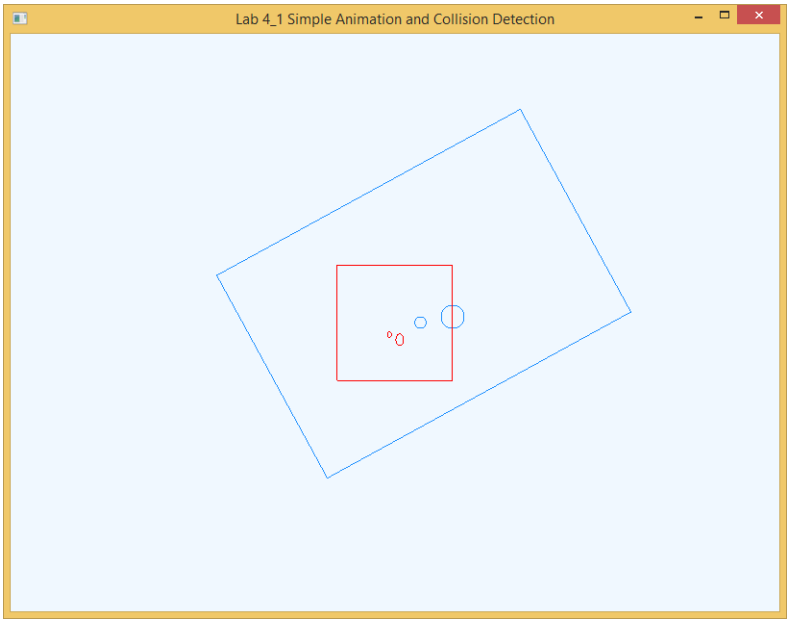
L41T14 Fixed collisions under scaling

Before we move on, go back to the onLoad method and change the square matrix to be scaled different amounts in different directions.

```
mSquareMatrix = Matrix4.CreateScale(3f, 2f, 1f) * Matrix4.CreateRotationZ(0.5f) *
Matrix4.CreateTranslation(0.5f, 0.5f, 0);
```

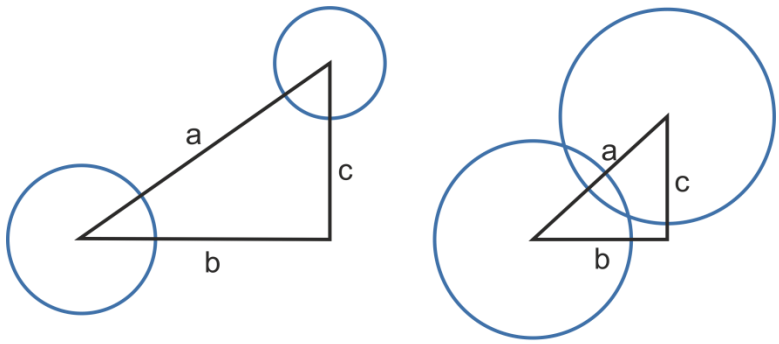
This appears to work, but it's worth noting that usually this wouldn't work because scaling differently in different directions can squash the normal vectors out of shape. In this case, all of the normal vectors are axis aligned, so they are never affected by the scale factor in more than one direction at a time. If this weren't true we would have to apply the same fix we applied in our lighting calculations, which was to rotate our normal using the inverse transpose matrix.

So far, so good. The next thing we've going to do is collide our circle with another, fixed circle. Create member variables for the position and radius for another circle. Assign them values in OnLoad and in OnRenderFrame draw the new circle.



L41T15 Can draw a second circle

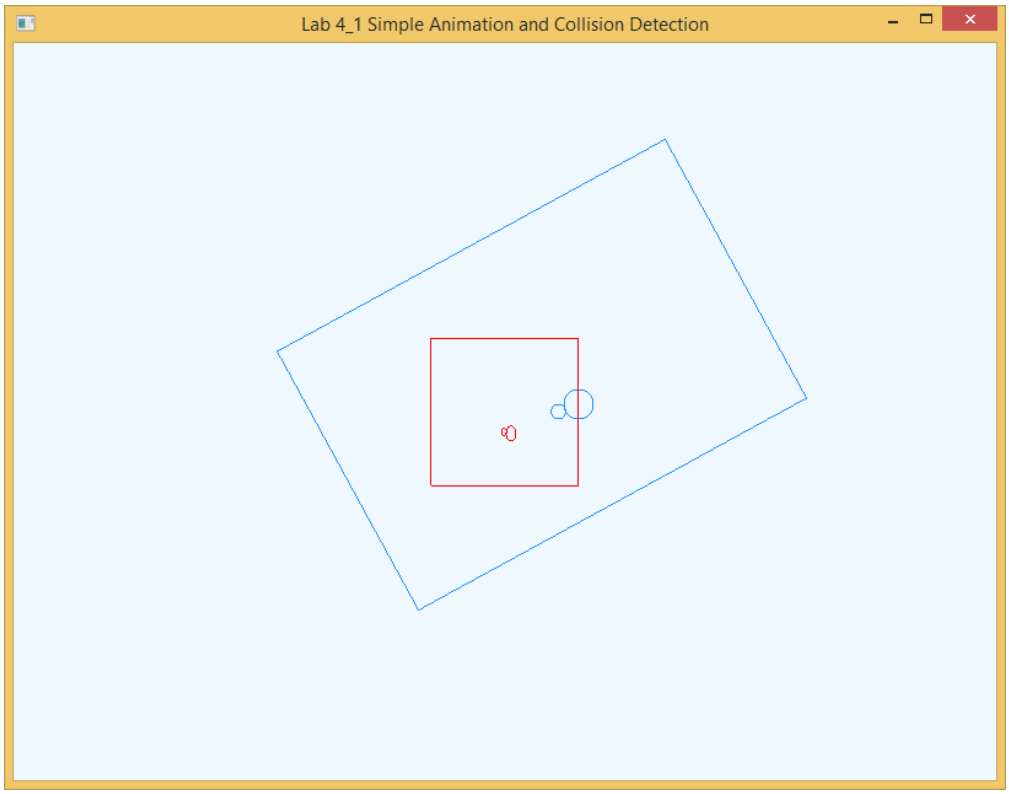
The next thing we need is code to detect a collision with our circle. This is pretty easy if you remember Pythagoras's theorem.



Pythagoras's Theorem
$a^2 = b^2 + c^2$

In the diagram above we can see how to calculate the distance between the centre points of two circles using Pythagoras's theorm. If the distance is less than the sum of the two circle radii then they have

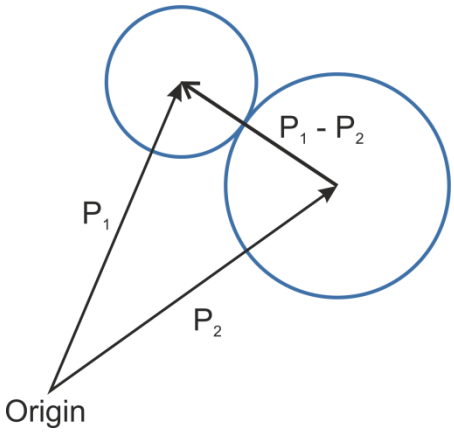
collided. In your update method add a test that sets the moving sphere's velocity to zero when it intersects with the stationary sphere. You actually don't need Pythagoras's theorem as OpenTK has a built in length property for vectors, but it's good to understand the theory. To get the vector between two points subtract one from the other.



Test your moving sphere stops when it collides with the stationary sphere. If it does then commit your code to SVN.

L41T16 Can stop the first circle when it touches the second

We can calculate a collision response in a pretty similar way to how we did the square. We don't have to worry about changing from one space to another because the circles are both in the same space. The only potential problem is calculating the normal vector, but that's pretty easy too, and it turns out we've already done some of the work. We want to calculate the normal vector at the point of collision. We know that the vector between the two circle centres (that you may have calculated to check for collisions) passes through that point. To calculate the normal all we need to do is make sure that the vector is pointing in the right direction (i.e. out of the stationary sphere) and make it a unit length. The diagram below might help you to understand what's happening.



Calculate the normal vector, and then use the same dot product principle that we used earlier to calculate the new velocity.

```
Vector3 normal = (mCirclePosition – mCirclePosition2).Normalized();
mCircleVelocity = mCircleVelocity - 2 * Vector3.Dot(normal, mCircleVelocity) * normal;
```

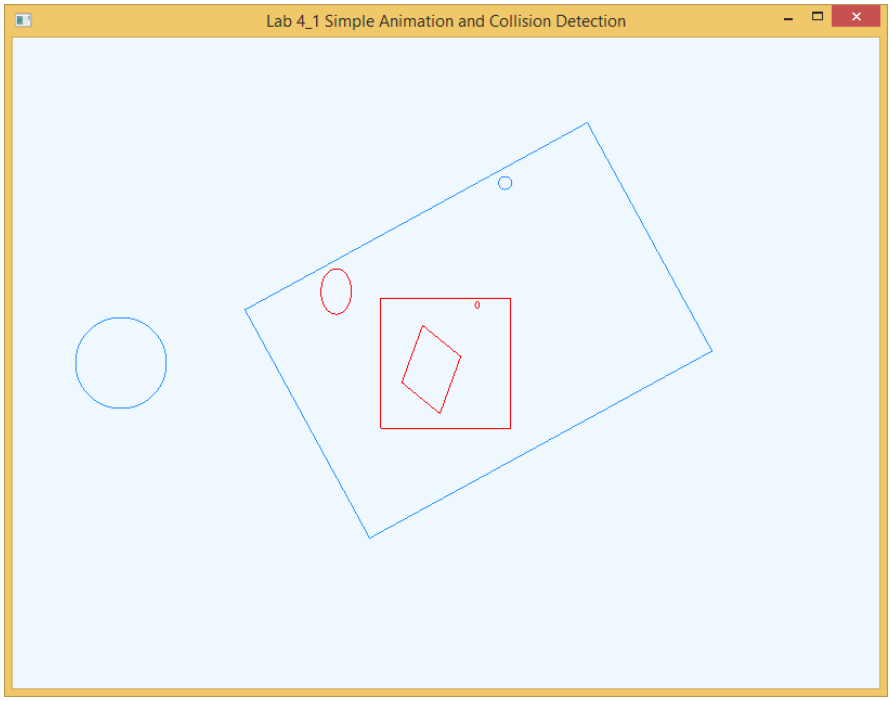
Don't forget, we are still allowing the circles to intersect each other. We should be moving the circle back to an old position to avoid buggy behaviour. Once your collision response is working commit your code to SVN.

L41T17 Calculated normal vector for circle-circle collision and calculated response

Get rid of your sphere and replace it with another square. For speed, just move the stationary sphere somewhere out of the way. Create another square matrix member variable and set it up like this.

```
mSquareMatrix2 = Matrix4.CreateScale(1f) * Matrix4.CreateRotationZ(0.0f) *
Matrix4.CreateTranslation(0.0f, 0.0f, 0);
```

Render it in blue world space and red square space.

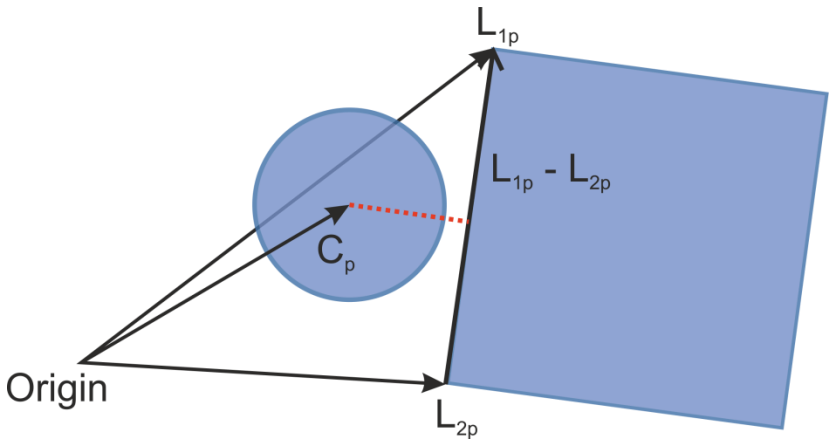


We have a slight issue here, because the original square in red square space is being rendered exactly on top of the new square in world space. The skewed square in red square space is the new square in world space. We could spend some time separating the spaces out, or setting them to appear with different key presses, but for the time being it's quicker and easier to recognize that the axis aligned red square is the large square in red square space, and is exactly on top of the smaller square in blue world space. It's not worth worrying too much about because as soon as we move that square our problems go away.

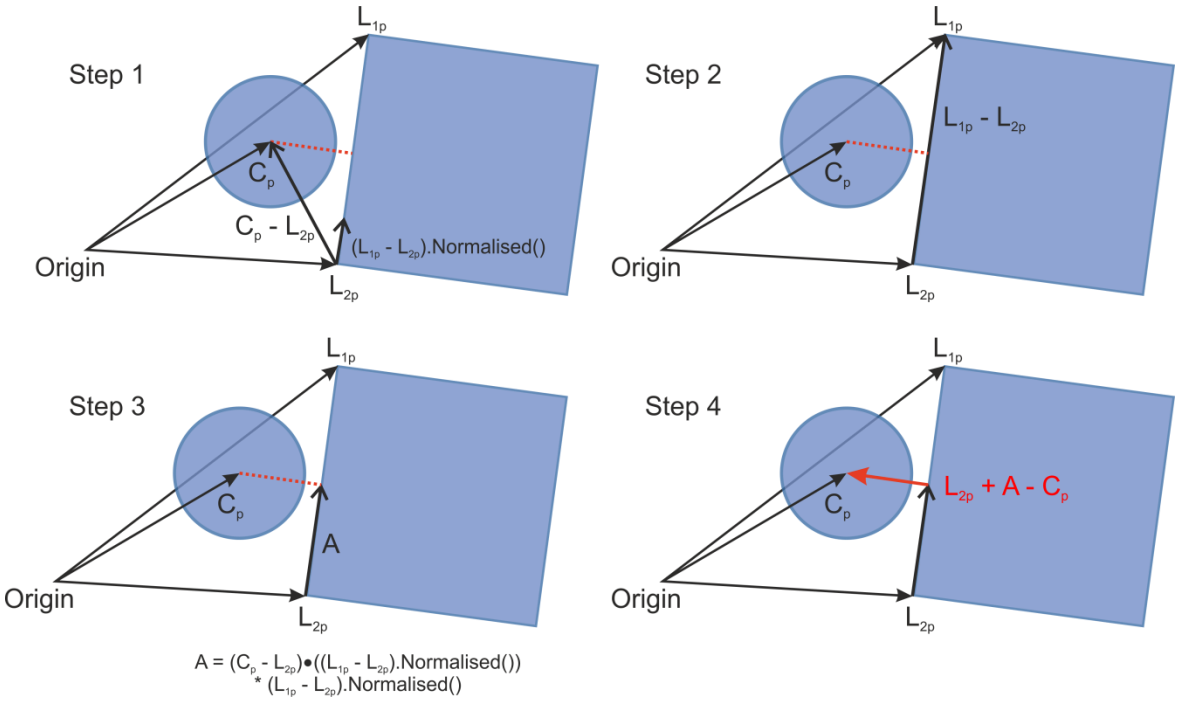
Commit your code to SVN:

L41T18 Added a new square in world space (but I can't see it because it's under a different square in red square space

This time, instead of colliding with the inside of the square we're going to collide with the outside of the square. In fact, this method (although not the most efficient) can be used to detect collisions with any shape. We're going to find the shortest distance between each line segment and the circle. If that distance is less than the radius of the circle then it has collided.



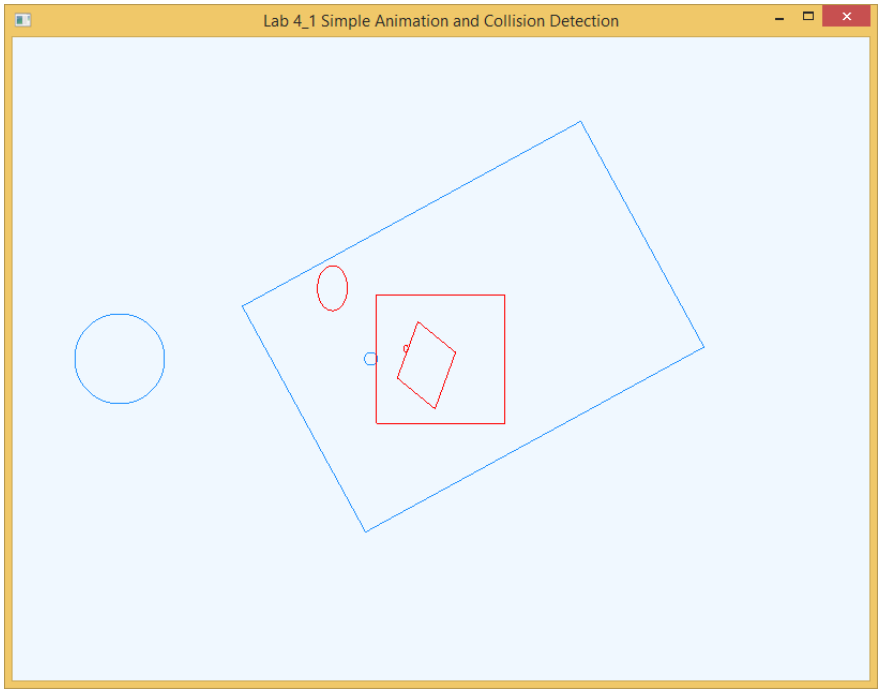
What we're looking for is the length of the red dotted line. If that is less than the radius of the sphere then the sphere has collided with the line. Once again, we can calculate the length of this line using the dot product and a little trigonometry.



$$A = (C_p - L_{2p}) \bullet ((L_{1p} - L_{2p}).\text{Normalised}()) * (L_{1p} - L_{2p}).\text{Normalised}()$$

The process is illustrated in the diagrams above. Let's try adding this to our code. For the left hand edge of the box the vertices are (-1, -1, 0) and (-1,1,0).

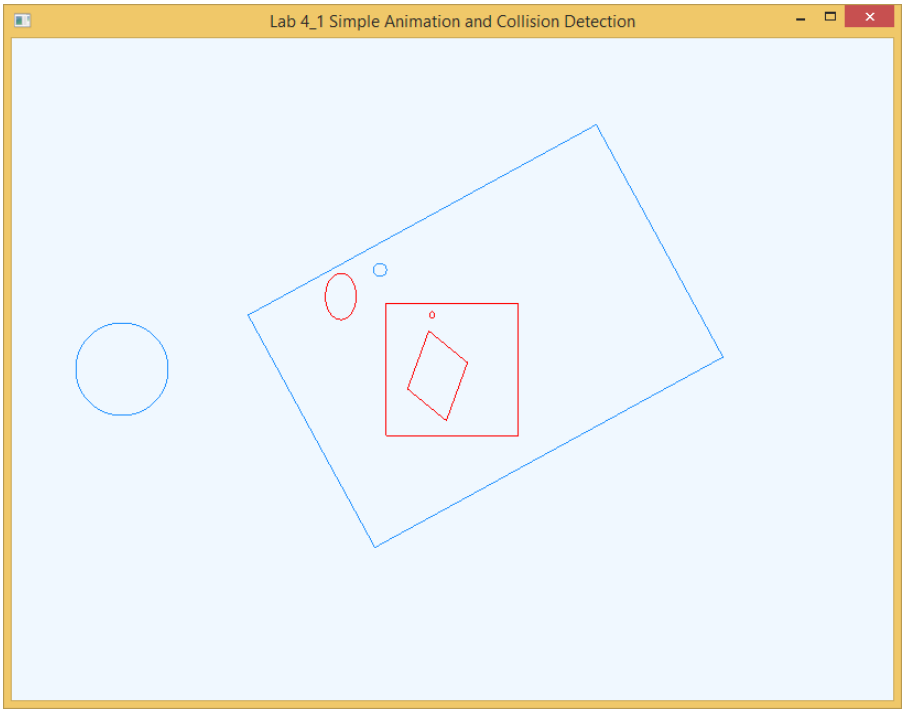
This is some complicated maths, so take it slow, and when you're debugging your code test the values to see if they are reasonable and what you expect.



Once you get the circle to stop at the outside edge of the square commit your code to SVN.

L41T19 Can collide circle with a line

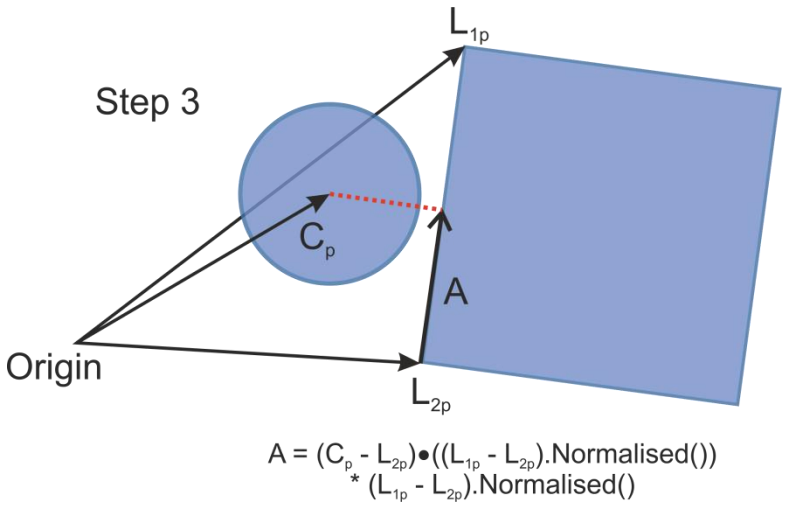
One thing that we haven't addressed is the end points of the line. What you currently have is an infinite line. In the screenshot below the circle has been stopped because the line for the left hand side of the new square extends out past its boundaries.



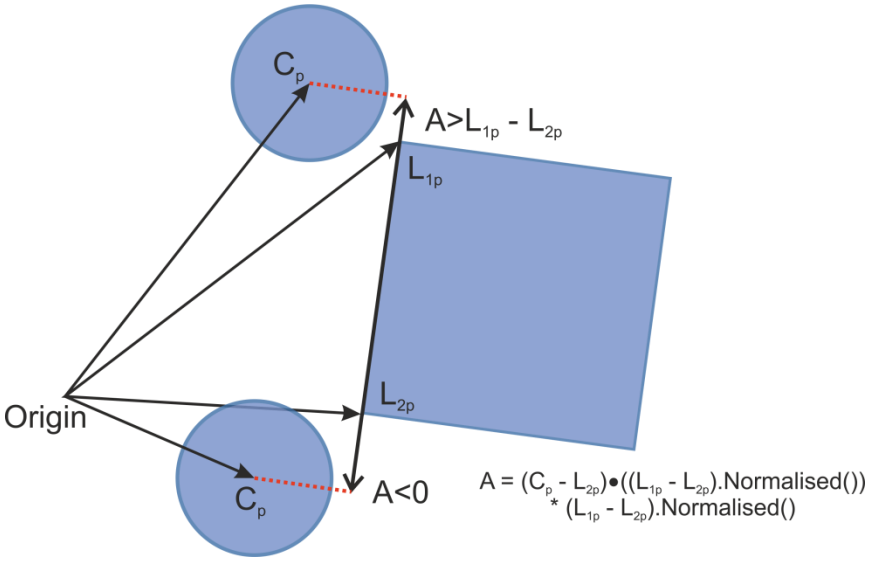
Set up your scene to replicate this issue, then commit you code to SVN before we set about solving it.

L41T20 Identified problem – treating a line segment as an infinite line

To fix this issue lets have another look at how we calculated the perpendicular point again.



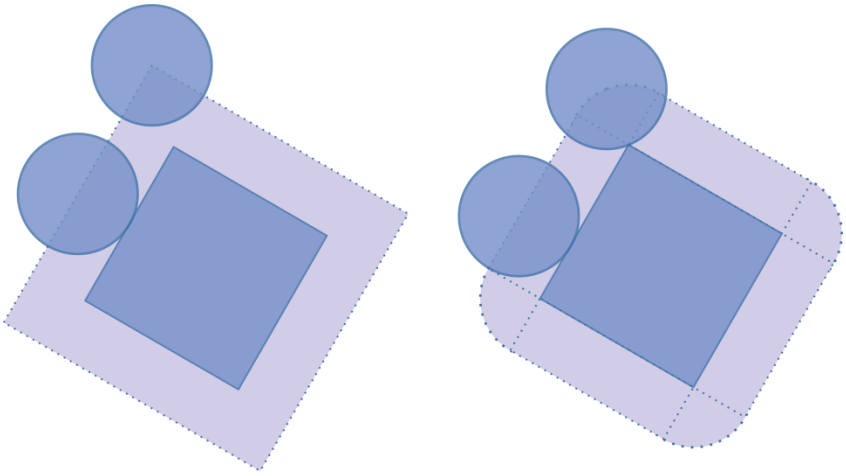
At one point we calculate the length of A (by getting the dot product of the circle position minus the start of the line and the line itself). We can use that to figure out if the centre of our circle is passing beyond the ends of the line.



In the diagram above if A is negative the circle is passing below the start of the line segment. If A is greater than the length of the line segment it is passing above. Note that in the diagram A is calculated to be a vector – and its length is always positive. To test the lower bounds you will have to calculate A as a scalar and skip the final step (multiplying by the unit vector). It's worth noting that this method measures from the centre of the circle, so other parts of the circle are still able to intersect with the line.

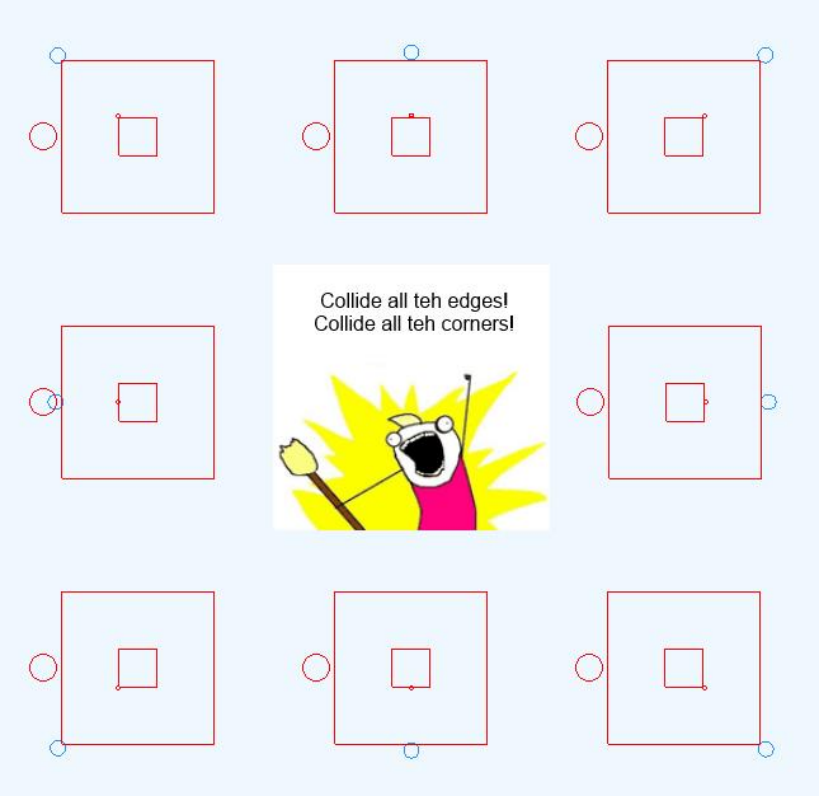
You could add a circle's radius to the ends of A, but that would give us some false collisions in the very corners.

Alternatively you could treat the vertices as if you are colliding a circle with another circle (with no radius). These concepts are demonstrated in the diagram below.



The diagram on the left shows what would happen if we extended the line segment by the radius of the circle. The diagram on the right shows a collision with a circle with zero radius at the corners.

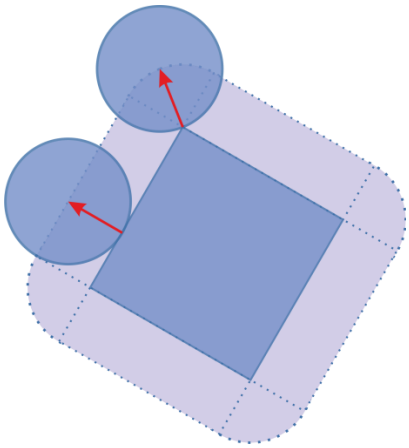
Add the line segment collision for all four lines, and add a circle collision at each vertex. You may want to think carefully about how you could use methods to make this easier for you, and test all the edges and corners. To make life easier for me set the large square matrix to the identity. It should have any affect because all these collision are happening in world space.



Once you've done that and tested it thoroughly commit your code to SVN.

L41T21 Can collide accurately with all the line segments and vertices in a box

The next step is to calculate a collision response. In order to do that we need to calculate the normal vector at the point of collision. In both cases we can get the normal by taking the circle position vector and subtracting the point of collision from it. In the case of the line segment we had to calculate that as part of our collision detection algorithm. For the corner collisions the point of collision is the corner itself.



Once you've got all the collision responses right commit your code.

L41T22 Calculating the correct collision responses for a square made up of four line segments

The next question is what happens if this square is scaled, rotated or translated? Well, we have two choices. We can either move the square into world space or we can move the circle into square space. Last time we moved the circle into square space, so let's try that.

To do this, first you need to calculated the circle position in square space by transforming it by the inverse of the square matrix (make sure you pick the correct square matrix – remember we have two!). Use that position in your collision calculations. When it comes to setting the velocity you also need to covert the velocity into square space by transforming it by the inverted rotation component of the square matrix, then you need to perform the collision response in square space and then covert the velocity back into world space by transforming by the original (non-inverted) rotation. If you're worried about the normal getting squashed you needn't be because the normal vectors are being calculated in square space, so they should be correct. One thing that you do have to correct for, however, is the distance that you allow the circle to be from the square. As the circle isn't circular in square space, and the line segments are no longer axis aligned this presents a problem. One solution is to extract the scale from the square matrix and proportionally scale the individual x, y and z components of the calculated vector from the closest point to the circle.

Once you're done that, and tested commit your code to SVN.

L41T23 Calculated the correct responses for line segments that have been rotated and translated by moving everything into square space.

Now let's try colliding with our square (made up of lines) in world space. To do this you should simply transform the end points of each line by the square matrix, and then use the algorithm we developed before. That's it. Commit your code to SVN.

L41T24 Calculated collision and response in world space. It was so much easier that it makes you wonder why we bothered with the other method

So why do we need two methods? Well, maybe whoever set this work is particularly sadistic, or maybe it's good to understand that there is more than one way to do things. Often the best way is dependent on the specific problem, and sometimes, because of the nature of the problem you can take short cuts, or you have to be aware of other issues that might not be immediately apparent. For example, what would the best way be if you we colliding with a hundred circles, or had a shape made up of a hundred lines?

If you are feeling brave then before moving on, as one last challenge try colliding with a set of lines translated and rotated by more than one matrix. Pick whatever method you like, but be aware how the space you are working in changes from one space to another. This is the beginning of something called a scene graph. The idea of being able to define spaces within other spaces, and then change the rotation, translation or scale in one space and have that change automatically propagate to all the other nested spaces is really useful, and is similar to something called a scene graph.

L41T25 Can calculate collision and response in nested space under multiple scales, translations and rotations

Summary

This lab was really hard work but you've made it through. Although this lab only really dealt with two dimensional problems moving from 2D to 3D is not that hard, and many techniques are the same. For example, you learnt how to calculate collide between a circle and axis aligned and non-axis aligned boxes, which translates well into 3D. You learnt how to collide a circle with another circle, which also translates very easily into 3D. Finally you learnt how to calculate the distance from a point to a line. In 3 dimensions you can use that for a ball cylinder collision.

Congratulations on making it this far!