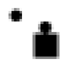
Lazy Foo' Productions

Circular Collision Detection



Last Updated 11/15/07

Besides rectangles, the most common shape you'll have to deal with is a circle. In the last tutorial we had to use 11 collision boxes for a circle. This tutorial will teach you a more efficient way to handle circles and collision detection.

```
#include "SDL/SDL.h"
#include "SDL/SDL_image.h"
#include <string>
#include <vector>
#include <cmath>
```

This tutorial is going to require the distance formula so we include a header for math.

```
//A circle stucture
struct Circle
{
    int x, y;
    int r;
};
```

We have to create our own circle structure for this program. "x" and "y" are the offsets of the center of the circle. "r" is the radius.

```
//The dot
class Dot
{
    private:
    //The area of the dot
    Circle c;

    //The velocity of the dot
    int xVel, yVel;

    public:
    //Initializes the variables
    Dot();

    //Takes key presses and adjusts the dot's velocity
    void handle_input();

    //Moves the dot
    void move( std::vector<SDL_Rect> &rects, Circle &circle );

    //Shows the dot on the screen
    void show();
};
```

Here's yet another revision of the Dot class.

Everything is pretty much the same except for two differences. This time we have a Circle structure instead of a vector of SDL_Rects. Also, in the move() function we check collision with a vector of SDL_Rects and a Circle.

```
double distance( int x1, int y1, int x2, int y2 )
{
    //Return the distance between the two points
    return sqrt( pow( x2 - x1, 2 ) + pow( y2 - y1, 2 ) );
}
```

This function we made gives us the distance between given 2 points. This is pretty much the only real math used in this program.

For those of you using visual studio you may need to type cast those integers to doubles.

```
bool check_collision( Circle &A, Circle &B )
{
    //If the distance between the centers of the circles is less than the sum of their radii
    if( distance( A.x, A.y, B.x, B.y ) < ( A.r + B.r ) )
    {
        //The circles have collided
        return true;
    }

    //If not
    return false;
}
```

Checking collision between two circles is pretty easy. All you have to do is check whether or not the distance between the centers of the circles is less than the sum of their radii.

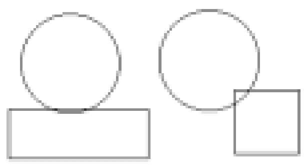
If it is less, a collision has happened, otherwise there's no collision.

```
bool check_collision( Circle &A, std::vector<SDL_Rect> &B )
{
    //The sides of the shapes
    int leftAv, leftAh, leftB;
    int rightAv, rightAh, rightB;
    int topAv, topAh, topB;
    int bottomAv, bottomAh, bottomB;

    //The corners of box B
    int Bx1, By1;
    int Bx2, By2;
    int Bx3, By3;
    int Bx4, By4;
```

This function check collision between a circle and a vector of rectangles. Checking for collision between a circle and a rectangle gets a bit tricky.

There's essentially two types of possible collisions:



The circle can either touch a side or a corner of the rectangle.

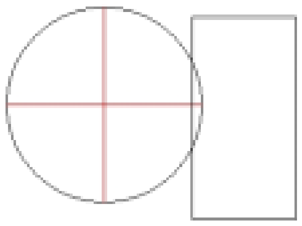
So we have to take into account the sides of the circle, the sides of the rectangle and corners of the rectangle.

```
    //Calculate the sides of A
    leftAv = A.x;
    rightAv = A.x;
    topAv = A.y - A.r;
    bottomAv = A.y + A.r;

    leftAh = A.x - A.r;
    rightAh = A.x + A.r;
    topAh = A.y;
    bottomAh = A.y;
```

You may be wondering why we have two sets of sides for the circle.

When we check for collision along the sides of the circle, we form a cross:



So when we check for circle side collisions, we check if the box collides with either of the lines.

How do you check collision with a line? A line is just a rectangle with a thickness of one, so all of the same principles apply.

```
    //Go through the B boxes
    for( int Bbox = 0; Bbox < B.size(); Bbox++ )
    {
        //Calculate the sides of rect B
        leftB = B[ Bbox ].x;
        rightB = B[ Bbox ].x + B[ Bbox ].w;
        topB = B[ Bbox ].y;
        bottomB = B[ Bbox ].y + B[ Bbox ].h;

        //Calculate the corners of B
        Bx1 = B[ Bbox ].x, By1 = B[ Bbox ].y;
        Bx2 = B[ Bbox ].x + B[ Bbox ].w, By2 = B[ Bbox ].y;
        Bx3 = B[ Bbox ].x, By3 = B[ Bbox ].y + B[ Bbox ].h;
        Bx4 = B[ Bbox ].x + B[ Bbox ].w, By4 = B[ Bbox ].y + B[ Bbox ].h;
```

Now that we have calculated the sides of the circle, it's time to go through the rectangles.

First we calculate the sides and corners of the rectangle.

```
        //If no sides from vertical A are outside of B
        if( ( ( bottomAv <= topB ) || ( topAv >= bottomB ) || ( rightAv <= leftB ) || ( leftAv >= rightB ) ) == false )
        {
            //A collision is detected
            return true;
        }

        //If no sides from horizontal A are outside of B
        if( ( ( bottomAh <= topB ) || ( topAh >= bottomB ) || ( rightAh <= leftB ) || ( leftAh >= rightB ) ) == false )
        {
            //A collision is detected
            return true;
        }
```

Then we check if the rectangle collides with the vertical or horizontal lines of the circle.

```
        //If any of the corners from B are inside A
        if( ( distance( A.x, A.y, Bx1, By1 ) < A.r ) || ( distance( A.x, A.y, Bx2, By2 ) < A.r ) || ( distance( A.x, A.y, Bx3, By3 ) < A.r ) || ( distance(
        {
            //A collision is detected
            return true;
        }
    }

    //If the shapes have not collided
    return false;
}
```

After that we check if any of the corners from the rectangle are inside the circle.

If the function goes through and no rectangles from the vector have collided with circle, the function return false.

```
    //Make the shapes
    std::vector<SDL_Rect> box( 1 );
    Circle otherDot;

    //Set the shapes' attributes
    box[ 0 ].x = 60;
    box[ 0 ].y = 60;
    box[ 0 ].w = 40;
    box[ 0 ].h = 40;

    otherDot.x = 30;
    otherDot.y = 30;
    otherDot.r = DOT_WIDTH / 2;
```

In the main() function we create the 2 shapes the Dot is going to interact with.

```
//While the user hasn't quit
while( quit == false )
{
    //Start the frame timer
    fps.start();

    //While there's events to handle
    while( SDL_PollEvent( &event ) )
    {
        //Handle events for the dot
        myDot.handle_input();

        //If the user has Xed out the window
        if( event.type == SDL_QUIT )
        {
            //Quit the program
            quit = true;
        }
    }

    //Move the dot
    myDot.move( box, otherDot );

    //Fill the screen white
    SDL_FillRect( screen, &screen->clip_rect, SDL_MapRGB( screen->format, 0xFF, 0xFF, 0xFF ) );

    //Show the box
    SDL_FillRect( screen, &box[ 0 ], SDL_MapRGB( screen->format, 0x00, 0x00, 0x00 ) );

    //Show the other dot
    apply_surface( otherDot.x - otherDot.r, otherDot.y - otherDot.r, dot, screen );

    //Show our dot
    myDot.show();

    //Update the screen
    if( SDL_Flip( screen ) == -1 )
    {
        return 1;
    }

    //Cap the frame rate
    if( fps.get_ticks() < 1000 / FRAMES_PER_SECOND )
    {
        SDL_Delay( ( 1000 / FRAMES_PER_SECOND ) - fps.get_ticks() );
    }
}
```

Here's the main loop, pretty much everything is the same story as before. I would like to point one thing out.

When we show the other dot, we don't blit the image at its offset. The offset is the center of the circle, so we have to blit the dot image at the upper-left corner. We do this by subtracting the radius from the offset.

Download the media and source code for this tutorial here.

News FAQs Games Tutorials Articles Contact Donations