Lazy Foo' Productions

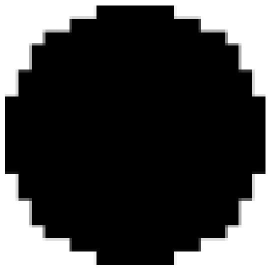
Per Pixel Collision

Last Updated 12/09/07

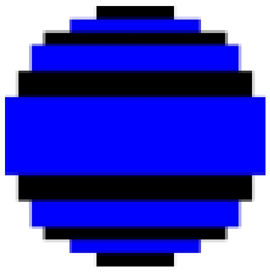
You've already learned how to check collision with rectangles. Of course not everything in video games is a rectangle and there's often a loss of accuracy when dealing with non rectangular shapes. Here you'll learn to get collision accuracy down to the pixel.

Everything is made out of rectangles, even this circle:

Don't see it? Let's magnify it:



Still don't see it? How about now:



Every image on a computer is made out of pixels, and pixels are squares which happen to be rectangles. So when you're checking collision between any shapes, you check if the two groups of rectangles have collided.

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

In this program we include the vector library along with our other standard ones. Vectors are kind of like arrays that are easier to manage.

```
//The dot
class Dot
{
    private:
    //The offsets of the dot
    int x, y;

    //The collision boxes of the dot
    std::vector<SDL_Rect> box;

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

    //Moves the collision boxes relative to the dot's offset
    void shift_boxes();

    public:
    //Initializes the variables
    Dot( int X, int Y );

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

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

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

    //Gets the collision boxes
    std::vector<SDL_Rect> &get_rects();
};
```

Here we have a revised version of the dot class.

We have the offsets and velocities from before, and now we have a vector of SDL_Rects to hold the dot's collision boxes.

In terms of functions, we now have shift_boxes() which moves the boxes in relation to the offset. I'll explain what that means later.

There's also the contructor which sets the dot at the offsets in the arguments and we have our event handler from before. This time I chose to seperate the move() and show() functions as opposed to having them both in the show function like before. We also have get_rects() which gets the dot's collision boxes.

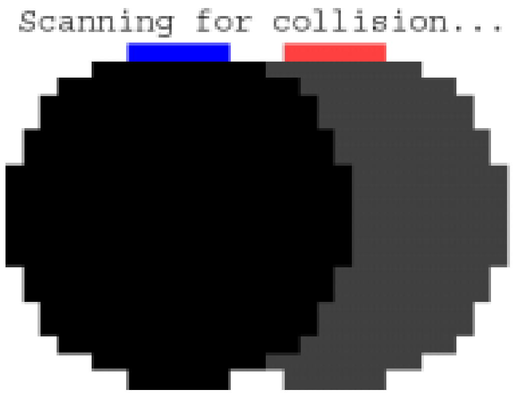
```
bool check_collision( std::vector<SDL_Rect> &A, std::vector<SDL_Rect> &B )
{
    //The sides of the rectangles
    int leftA, leftB;
    int rightA, rightB;
    int topA, topB;
    int bottomA, bottomB;

    //Go through the A boxes
    for( int Abox = 0; Abox < A.size(); Abox++ )
    {
        //Calculate the sides of rect A
        leftA = A[ Abox ].x;
        rightA = A[ Abox ].x + A[ Abox ].w;
        topA = A[ Abox ].y;
        bottomA = A[ Abox ].y + A[ Abox ].h;

        //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;

            //If no sides from A are outside of B
            if( ( ( bottomA <= topB ) || ( topA >= bottomB ) || ( rightA <= leftB ) || ( leftA >= rightB ) ) == false
            {
                //A collision is detected
                return true;
            }
        }
    }

    //If neither set of collision boxes touched
    return false;
}
```

Here we have our collision detection function.

In takes in two vectors of SDL_rects, then checks collision between the two sets of rectangles.

This function gets a rectangle from vector A, then checks if it collides with any rectangles from vector B, then gets another rectangle from vector A, then checks if it collides with any rectangles from vector B and so on until either a collision is found or all the rectangles have been checked.

So when the function is checking for collision it would operate like this:



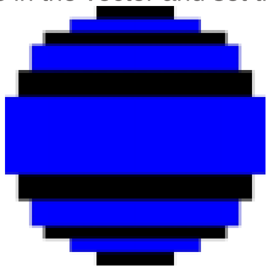
Like from last time, the function returns true if there's a collision and false if there is no collision.

```
Dot::Dot( int X, int Y )
{
    //Initialize the offsets
    x = X;
    y = Y;

    //Initialize the velocity
    xVel = 0;
    yVel = 0;

    //Create the necessary SDL_Rects
    box.resize( 11 );

    //Initialize the collision boxes' width and height
    box[ 0 ].w = 6;
    box[ 0 ].h = 1;

    box[ 1 ].w = 10;
    box[ 1 ].h = 1;

    box[ 2 ].w = 14;
    box[ 2 ].h = 1;

    box[ 3 ].w = 16;
    box[ 3 ].h = 2;

    box[ 4 ].w = 18;
    box[ 4 ].h = 2;

    box[ 5 ].w = 20;
    box[ 5 ].h = 6;

    box[ 6 ].w = 18;
    box[ 6 ].h = 2;

    box[ 7 ].w = 16;
    box[ 7 ].h = 2;

    box[ 8 ].w = 14;
    box[ 8 ].h = 1;

    box[ 9 ].w = 10;
    box[ 9 ].h = 1;

    box[ 10 ].w = 6;
    box[ 10 ].h = 1;

    //Move the collision boxes to their proper spot
    shift_boxes();
}
```

Now here's the dot's constructor.

It sets the dot's offsets to the arguments given, and initializes the velocity of the dot.

Then we create 11 collision boxes in the vector and set them like this:



At the end we set the boxes relative to the dot's offset.

```
void Dot::shift_boxes()
{
    //The row offset
    int r = 0;

    //Go through the dot's collision boxes
    for( int set = 0; set < box.size(); set++ )
    {
        //Center the collison box
        box[ set ].x = x + ( DOT_WIDTH - box[ set ].w ) / 2;

        //Set the collision box at its row offset
        box[ set ].y = y + r;

        //Move the row offset down the height of the collision box
        r += box[ set ].h;
    }
}
```

You may be asking yourself what I mean by "Setting the boxes relative to the dot's offset".

Say if you move the dot 100 pixels over, but when it goes over the other dot it doesn't detect the collision.

The reason for this is that when you move the dot, you have to move collision boxes along with it and that's what this function does.

Don't worry how I did it, it was just a fancy way of doing:

```
box[ 0 ].x = x + 7;
box[ 0 ].y = y;

box[ 1 ].x = x + 5;
box[ 1 ].y = y + 1;
```

and so on.

```
void Dot::handle_input()
{
    //If a key was pressed
    if( event.type == SDL_KEYDOWN )
    {
        //Adjust the velocity
        switch( event.key.keysym.sym )
        {
            case SDLK_UP: yVel -= 1; break;
            case SDLK_DOWN: yVel += 1; break;
            case SDLK_LEFT: xVel -= 1; break;
            case SDLK_RIGHT: xVel += 1; break;
        }
    }
    //If a key was released
    else if( event.type == SDL_KEYUP )
    {
        //Adjust the velocity
        switch( event.key.keysym.sym )
        {
            case SDLK_UP: yVel += 1; break;
            case SDLK_DOWN: yVel -= 1; break;
            case SDLK_LEFT: xVel += 1; break;
            case SDLK_RIGHT: xVel -= 1; break;
        }
    }
}
```

Here's the dot's event handler. As you can see the dot's velocity is only one pixel per frame so if you notice it's going slow just know its intentional. If you can only move one pixel at a time you can better see the per pixel collision.

```
void Dot::move( std::vector<SDL_Rect> &rects )
{
    //Move the dot left or right
    x += xVel;

    //Move the collision boxes
    shift_boxes();

    //If the dot went too far to the left or right or has collided with the other dot
    if( ( x < 0 ) || ( x + DOT_WIDTH > SCREEN_WIDTH ) || ( check_collision( box, rects ) ) )
    {
        //Move back
        x -= xVel;
        shift_boxes();
    }

    //Move the dot up or down
    y += yVel;

    //Move the collision boxes
    shift_boxes();

    //If the dot went too far up or down or has collided with the other dot
    if( ( y < 0 ) || ( y + DOT_HEIGHT > SCREEN_HEIGHT ) || ( check_collision( box, rects ) ) )
    {
        //Move back
        y -= yVel;
        shift_boxes();
    }
}
```

Here's the dot's move function that we seperated from the show function.

Its pretty much the same story as before. We move the dot, and if the dot went off the screen or over the vector of rectangles, move back. There is one key difference however.

Whenever we move the dot, we call shift_boxes() to move the collision boxes along with the dot. The collision boxes will do no good if they do not go along with the dot.

```
void Dot::show()
{
    //Show the dot
    apply_surface( x, y, dot, screen );
}
```

Here's our show function that applies the dot to the screen.

```
std::vector<SDL_Rect> &Dot::get_rects()
{
    //Retrieve the collision boxes
    return box;
}
```

Here's the function that gets the dot's collision boxes.

```
    //Make the dots
    Dot myDot( 0, 0 ), otherDot( 20, 20 );
```

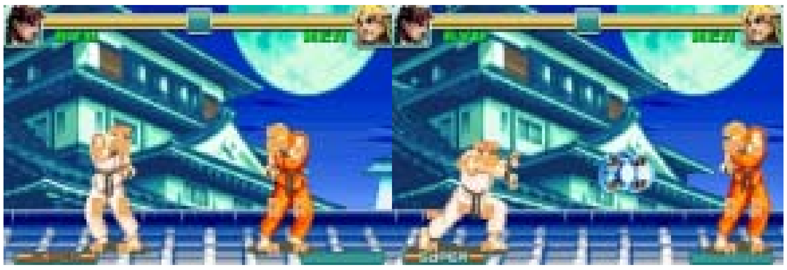
In our main function we generate two Dot objects, "myDot" which is the dot we're moving and "otherDot" which is the dot that's sitting still.

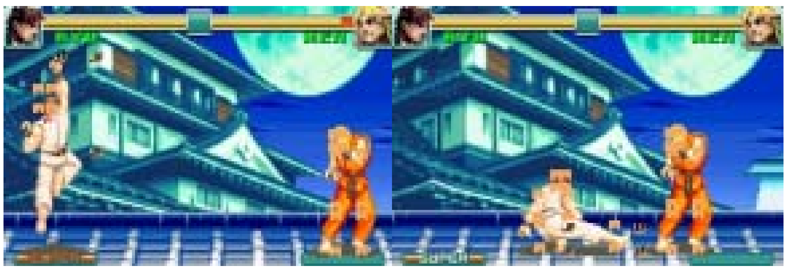
```
//While the user hasn't quit
while( quit == false )
{
    //Start the frame timer
    fps.start();

    //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( otherDot.get_rects() );

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

    //Show the dots on the screen
    otherDot.show();
    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. We handle events, move the dot, fill the screen white, show the dots, update the screen, and cap the frame rate. Now you can check collision with whatever you want.

There is one note I want to make about per pixel collision. Even though you can check for collision down the the pixel, 99% of the time you don't have to.

The perfect example of this is super street fighter 2 turbo.
If you have the GBA version, when you activate the akuma glitch you can see the corners of the collision boxes:





As you can see the collision detection is not down to the pixel.

When it comes to collision detection, down to the pixel accuracy isn't always needed. In many cases it's a waste of CPU power to check collision down to the pixel. There is such thing as accurate enough. It's up to you to decide how much accuracy you need.

Download the media and source code for this tutorial here.

Visual Studio users might encounter a problem compiling the source. There's 2 ways to fix it:
1) Change "Runtime Library" from "Multithreaded DLL" to "Multithreaded Debug DLL".
2) Compile in release mode. When you switch from Debug to Release make sure to put "SDL.lib SDLmain.lib" and all the other libraries you used in the additional dependencies field again.