Collisions/Reflection

General Collisions

The calculating whether or not two 2D objects collide is equivalent to calculating if the two shapes share a common area (intersect). For general polygons this is a difficult problem beyond the scope of what we can teach in a three week camp. However, for the types of games and movements we'll be doing in the camp, there are some collisions that are relatively easy to calculate, so this lecture will focus on those mechanics.

Reflection

For the purposes of this lecture, we'll only discuss how to calculate a perfectly elastic reflection off a vertical or horizontal wall. Further mathematics is needed (vectors and how to calculate their normal)) for collision of arbitrary walls. Luckily, we can make quite a few fun games just with reflections of horizontal and vertical walls.

In most simple games, an object will have some position (x, y) and a movement vector (dx, dy), where we would typically update x and y after each frame as follows:

```
x += dx
y += dy
```

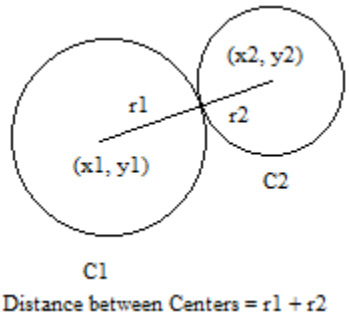
Unfortunately, this update might take an object off the screen. If we carefully think about what happens when an object bounces, we realize that we just need to change its direction. If the object's dx was positive and it went off the right end of the screen, we would want its dx to be negative. In fact, we also just want to preserve the magnitude of the dx. Similarly, if the object dx was negative and it was going off the left end of the screen, we would want to change the dx to be positive and also keep the same magnitude. In code, this change is quite easy:

```
dx = -dx
dy = -dy
```

Depending on how far off the screen the object has gone, some other adjustment might have to be made for x and y so that when the object turns around, it still isn't off the screen. (This might create a situation where the object is stuck, just to the left of the screen, bouncing back and forth.) It's best to do these adjustments specific to the game via testing.

Circle-Circle Collision

Given two circles, we usually know the position of the center (x, y) and the radius, r of each circle. Let the two circles for this section be C_1 and C_2 with centers (x_1, y_1) and (x_2, y_2) and radii r_1 and r_2, respectively. Consider the situation that the two circles are touching:



When two circles touch exactly, the distance between their centers is the sum of the two radii, as these line up in a straight line. But, since we know both centers, we have an alternate expression for the distance between the two centers via the distance formula:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

If the real distance between the centers (indicated by the formula above) is ***less than*** the sum of the two radii, then the two circles share positive area. If these two expressions are equal, the two circles touch at a point. If the distance is ***greater than*** the sum of the two radii, the circles do NOT touch.

Consider writing a circle class with instance variables x, y and r of type int, representing that the circle object has the center (x, y) with radius r. Here is the corresponding intersect method which returns true if and only if the circles share at least one point:

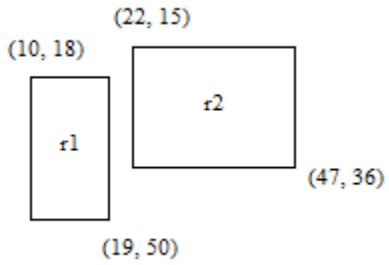
```
public boolean intersect(circle other) {
    return distsq(other) <=(r+other.r)*(r+other.r);
}

public int distsq(circle other) {
    return (x-other.x)*(x-other.x) + (y-other.y)*(y-other.y);
}
```

Axis-Aligned Rectangle Collision

Given two rectangles with sides parallel to the x and y axes, we can check four items, any of which guarantee no collision. If none of these are true, then there is an intersection between the two rectangles.

For each rectangle, assume that we have four instance variables: x, y, w and h, where (x,y) is the top left corner of the rectangle, w is the width of the rectangle and h is the height of the rectangle. The lower right corner has coordinates (x+w, y+h).

If the left side of one rectangle ***is to the right of*** the right side of the other rectangle, no collision can occur:



In this picture, the left end of r2 is x = 22, and this is greater than the right end of r1, which is x = 19. As can be seen, if this is true for a pair of rectangles in general, they can't intersect.

We can repeat this test on all four sides of r1. If any of these tests pass, they prove that the two rectangles do NOT intersect. If all four test fail, that is proof that there is some intersection between the rectangles.

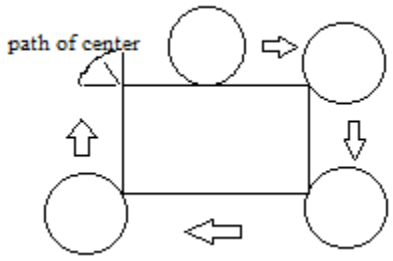
Here is code from the rectangle class that executes this, assuming the instance variables previously discussed:

```
public boolean intersect(rect other) {
    if (other.x > x+w) return false;
    if (x > other.x+other.w) return false;
    if (other.y > y+h) return false;
    if (y > other.y+other.h) return false;
    return true;
}
```

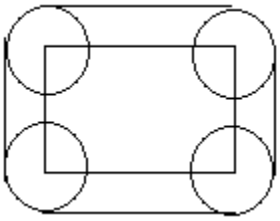
Axis-Aligned Rectangle and Circle Collision

It's a bit trickier to calculate if a circle and an axis aligned rectangle intersect, but we can use our current tools (basic geometry) to solve the problem as well.

The easiest way to view this problem is just how we viewed circle-circle intersection. We imagined the two objects perfectly touching. Now, consider a circle and rectangle touching at a single point. To consider all options, imagine rolling the circle on the outside of the rectangle:



As you can see, as the circle rolls on any of the four sides, the center just rolls parallel to that side, as would happen with the axle of a car. Only when we get to the corners is the behavior different. Here. The radius pivots around the corner, so the center of the circle traces the path of a quarter circle itself! This means that the range of locations where the center of the circle could be while intersecting the rectangle are as shown in this picture:



Thus, our goal will simply to be to check whether the center of the circle is within either of two rectangles in this picture, or one of the four quarter circles.

We can check to see whether or not the circle's center x coordinate is within the left and right bounds of the rectangle (we assume our method is in the rectangle class and takes in a circle named cir):

```
if (cir.x >= x && cir.x <= x+w)
```

If this is the case, then the y coordinate of the circle just has to be greater than or equal to y - cir.r and less then or equal to y + h + cir.r, since we have an extension of r on both sides of our rectangle, up and down, for this range of x.

The same logic applies symmetrically for y.

If all of these tests fail, then the only way an intersection can happen is if the center of the circle is within a distance of r with one of the rectangle corners. To implement this, we should use a method that takes the distance between two points. (Ideally our points would be objects, but for this example they are just stored separately as x and y coordinates.)

Here is the corresponding code from the rectangle class:

```
public boolean intersect(circle cir) {

    if (cir.x >= x && cir.x <= x+w)
        if (cir.y >= y-cir.r && cir.y <= y+h+cir.r)
            return true;

    if (cir.y >= y && cir.y <= y+h)
        if (cir.x >= x-cir.r && cir.x <= x+w+cir.r)
            return true;

    if (distsq(cir.x, cir.y, x, y) <= cir.r*cir.r) return true;
    if (distsq(cir.x, cir.y, x+w, y) <= cir.r*cir.r) return true;
    if (distsq(cir.x, cir.y, x, y+h) <= cir.r*cir.r) return true;
    if (distsq(cir.x, cir.y, x+w, y+h) <= cir.r*cir.r) return true;

    return false;
}

public static int distsq(int x1, int y1, int x2, int y2) {
    return (x2-x1)*(x2-x1) + (y2-y1)*(y2-y1);
}
```

Use of Java's Rectangle Class

But more simple way to do rectangle-rectangle intersection is use Java's Rectangle class's built-in .intersects() method:

```
Rectangle r1 = new Rectangle(c1.x, c1.y, c1.width, c1.height);
Rectangle r2 = new Rectangle(c2.x, c2.y, c2.width, c2.height);
if(r1.intersects(r2)) /* put action here */
```

What you put in the last part is totally up to you; if you simply want to make sure the objects aren't inside each other, you can move them both apart from each other a little bit. However, if you want to reflect an object's velocity, you can do that as well. After you do that, though, you'll need to make sure you place one of the objects outside of the other object, or else it will continue to register that the objects are colliding and infinitely try to reverse the object's velocity.