Introduction to Intermediate Algorithms

Previously, I gave you more “run of the mill” algorithms. However, I feel that one of the great things about University (as opposed to a Community College) is that they encourage professors to bring their unique experiences and research into the courses they teach.

In keeping with that, I would like to spend this week on discussing some straight-forward algorithms in my field (computer game design). These two algorithms would be considered relatively basic in the game programming world but still may require more (or different) thinking than the basic algorithms we discussed last module.

In this module, I am going to go over:

- 2D Collision Detection using “bounding box” collision
- Mouse Click Processor in a game with various screen resolutions

I hope you enjoy this module. I enjoy discussing all things game design and am happy to inject the topic where relevant to the course at hand.

2D Collision Detection

In many genres of computer games, we use the concept of basic physics to simulate a more realistic experience. Recall the old-fashioned Super Mario Brothers game: how strange would it be if Mario could fall right through the floor or run through a brick on the ground?

When we draw computer graphics to the screen, these are reduced to streams of bits containing pixel color information (red, green, blue, alpha) and screen coordinate data (x, y). There is nothing within the actual graphical data that we can use to handle physics. The computer doesn't "just know" that two blocks of graphical data (often referred to as "sprites" or "raster images" in 2D game development) are not supposed to overlap. Therefore, we have to create the construct of physics for the game engine.

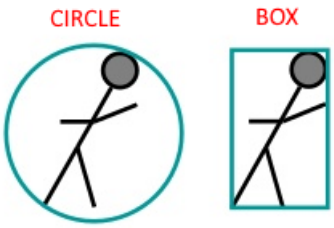
Since we are talking about 2D game physics here, there are generally two types of "rigid bodies" we can create. (A "rigid body" is a type of structure that we use as a "fence" of sorts around a sprite to denote the points at which we would "touch" another object).

The two most common types of rigid bodies in 2D game development are:

- Bounding Boxes
- Circles

You might wonder why we don't use "pixel perfect" collision detection for the most accurate physics. The reason is performance. Games are a real-time medium with graphics going at least 60 frames per second optimally. Physics tend to run at twice the graphics render rate which is a minimum of 120 frames per second (more on why physics run twice the rate later!) Consider that you would have to compare every pixel in a sprite against every pixel of every other sprite a minimum of 120 times a second. Even with modern processing, this is a huge burden. Therefore, we approximate collision buffers to "close enough" and increase performance by orders of magnitude.

Bounding Box vs. Circle Collision Buffer

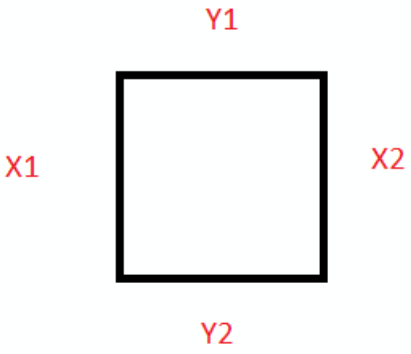


The image above shows how both types of collision buffers operate. Which one you use should be determined by the object you are trying to “contain”.

In this module, however, I will be explaining how we use the bounding box.

FYI: The circle works by taking the x, y coordinates at the center of the buffer and having a radius for how far the circle extends from the center. Collisions are determined by finding the distance between two circle buffers and checking to see if the sum of both circle’s radius is greater than the distance. If the distance is less than the sum, you have a collision. You could easily derive a Java solution for this if you desire.

For the bounding box, we have a set of 4 coordinates for each collision buffer.



- X1 = The most LEFT coordinate on the x-axis
- Y1 = The most TOP coordinate on the y-axis (screen height coordinates are inverted from Cartesian!)
- X2 = The most RIGHT coordinate on the x-axis
- Y2 = The most BOTTOM coordinate on the y-axis (bottom is greater y coordinate than top!)

Next we will go over how to determine if there is a collision between two boxes…

Bounding Box Collision Algorithm

When we are trying to determine whether or not two boxes collide, we might be tempted to use conditionals to check for a collision state. Although this is possible, it is less straight-forward than reversing the logic.

What if we, instead, tested for two boxes that we know WILL NOT collide? Then, if the boxes pass our “tests” then we know the boxes collide by default!

There are four conditions where we know we did not collide (because it would be impossible):

1. box1.X1 > box2.X2
2. box1.X2 < box2.X1
3. box1.Y1 > box2.Y2
4. box1.Y2 < box2.Y1

Therefore, if we test for these conditions and none of them are true, we know we have a collision!

Why do Physics engines tend to run twice the frame rate?

Testing for if a collision is going to happen is tedious. What is much simpler (and computationally cheaper) is to advance the moving parts of the collision buffers and let the collision happen. Then, if you detect a collision, you can reset the collision buffer back to its previous state before the renderer changes the graphical position of the sprite. Therefore, the overlap is happening behind the scenes, but the player never sees it because it never reaches the renderer.

Mouse Click Processor

When you start working with GUI applications, you will use the Java “Swing” package. It provides a (relatively) easy group of classes that can handle basic Windows style applications. It provides support for buttons, radio buttons, drop boxes, etc. When using this, you do not have to worry about screen resolution much as the structure of the Swing package takes care of much of that for you and you work within a “layout.”

With serious game design, however, you are going to be designing full screen graphical applications. These applications require you to manually handle all of the coordinates and input “action listeners” as Java calls them. Games are real-time and optimizing speed is of utmost importance. As great as the Swing class can be for basic Windows apps, it is not suitable to the performance required by games.

We still will use a few things from the basic Windows toolbox such as “JFrame” to get started but then we manually handle all graphics pipeline stuff including the renderer.

There are many things to discuss regarding getting into full screen GUI mode and drawing sprites, etc. However, this particular algorithm is going to focus on dealing with mouse input, processing it, and normalizing input for multiple resolutions.

For a crash course, when we wish to receive real-time mouse input, our input class must implement the MouseListener interface. This interface automatically requires you to override the following:

- mouseClicked – This method reports when a user presses down the mouse button and releases it
- mousePressed – This method reports when a user presses down the mouse button
- mouseReleased – This method reports when a user releases a previously held mouse button

You do not actually have to write anything in the body of all of these. You choose how you wish to process your mouse input. I find that for the quickest and most accurate response, I use “mouseReleased” since it doesn’t record the coordinates until someone releases the button. Using “mousePressed” will become problematic if someone drags the mouse a bit before releasing it because it will record the coordinates at the point the user pressed; not released. This makes the “feel” seem off. The mouseClicked seems like it would be the perfect middle ground but does not feel very responsive in practice. For the purposes of this module, I will use mouseReleased (although they all work very similar, in practice.)

The mouseReleased method

When you implement the MouseListener, the method prototypes for the overridden methods will automatically appear if you are using Eclipse and click “Add unimplemented methods”.

Since we are using the mouseReleased to handle our input, the prototype looks as follows:

```
public void mouseReleased(MouseEvent arg0){…}
```

The MouseEvent object is passed to our method by the JVM (Java Virtual Machine) as an interrupt as soon as a mouse event is detected. We read from this object to know “what is going on” and, therefore, how we can respond to it.

In general, there are two pieces of information we are primarily concerned with here. These are:

1. The mouse button that triggered the event
2. The x, y coordinates of the mouse at the point of the event happening

To acquire the button pressed, we can create an int and read it as follows:

```
int button = arg0.getButton();
```

But how do we know what this means in terms of actual buttons? Java has constants for the common buttons:

MouseEvent.BUTTON1 – This is the mouse’s left button

MouseEvent.BUTTON2 – This is the mouse’s middle button (if applicable)

MouseEvent.BUTTON3 – This is the mouse’s right button

Other buttons of note are:

4 – This is the integer for the mouse’s back button

5 – This is the integer for the mouse’s forward button

Acquire the position of the event

When we want to acquire the position of the mouse at a given time (such as when we process a mouse event), we can use the Java built in API method as follows:

```
Point p = MouseInfo.getPointerInfo().getLocation();
```

This will return a “Point” object into “p” that you can extract the coordinates from. Point is a lightweight object that packs a set of float coordinates into a single entity. We can turn these into integers as follows:

```
int x = (int) p.getX();

int y = (int) p.getY();
```

Now we know where the event occurred (assuming we act swiftly upon the triggered event!) This will help us to compare the “spot of the event” to the coordinates of where things are placed on the screen.

Note: A mouse cursor can be a small or a large item. It is important to note that only the “hotspot” of the mouse is saved. On a standard arrow, this is the pixel at the very tip of the mouse cursor.

Clickable "RECT" of screen items

When we have a point and click style game, we generally set up a rectangle of coordinates (similar to the bounding box of 2D physics engines) to denote the “clickable region” of the graphical item. You can create a custom class (or use a Java class; I prefer to build my own to customize but your mileage may vary). I will build a class as follows:

```
public class RECT{

        // Fields

        private int x1;  // This is the most left coordinate on x-axis

        private int y1;  // This is the most top coordinate on y-axis

        private int x2;  // This is the most right coordinate on x-axis

        private int y2;  // This is the most bottom coordinate on y-axis

        …Constructor and get/set methods below…

}
```

Note: If you already built a class for your collision buffer, you can likely reuse this for your clickable rectangle.

Finding when an item has been "clicked"

Using the x, y variables for the mouse coordinates upon an event and also the RECT, we can figure out when an item is clicked as follows:

```
// Assume RECT called “item” that contains an item’s clickable rectangle region
if (x >= item.getX1() && x <= item.getX2()){
if(y >= item.getY1() && y<= item.getY2()){
        // This is the code to respond to the item being clicked
}}
```

What about different screen resolutions?

Unfortunately, gone are the days when you could support just a single screen resolution. Chances are, you need to support at least 3 of the most common screen resolutions; even for 2D games. If you are using a commercial engine like Unity, this isn't a big deal as they handle most of that for you. If you are writing from scratch in Java, the burden is on you.

My recommendation: Do not EVER support screen resolutions with different ASPECT RATIOS. (An "aspect ratio" is the ratio of max pixels high to max pixels wide. So called "Full HD" is 1920 x 1080. This creates an aspect ratio of 16:9 (16 / 9 = ~1.78). The reason why I say this is because trying to change the aspect ratio of your high resolution source images will result in a "skewed" image that looks unaesthetic.

Since we are not going to use differing aspect ratios, we need to assign a "native" resolution that acts as a reference point for the other possible resolutions. My opinion is that the highest resolution your game supports should be the native resolution. For my example, I will use 1080p as my native resolution.

The purpose of the native resolution is as a reference point. Based on this reference point, we will create a global variable called "scaleFactor" that is the percentage of the native resolution that our current resolution is. If the player has a native resolution monitor (or support for that graphics mode), the scaleFactor will be 1.0f which is ideal.

Note: Your scaleFactor variable should be a float or a double.

Once you figure out what resolution the user can support (in your graphics initialization code, always try for the "best" resolution down to the "worst"), you can calculate scale factor as:

```
scaleFactor = currentPixelsWide / referencePixelsWide;
```

Since we preserve only the native aspect ratio, we could also calculate it as:

```
scaleFactor = currentPixelsHigh / referencePixelsHigh;
```

Therefore, assuming our native resolution is 1920 x 1080, but the player has a laptop that only supports 1280 x 720, our scale factor would be:

```
scaleFactor = 1280 / 1920 = ~0.67
```

Finding when an item is "clicked" with scale factor

Since we want to keep all of our item coordinates based on native coordinates for our own ease of development, we can use this to adjust for the differences “on the fly”.

Using the code we used before, but with a slight modification, we can still determine a click even with differing resolutions:

```
// Assume RECT called “item” that contains an item’s clickable rectangle region (native coords)

if (x >= (int)(item.getX1() * scaleFactor) && x <= (int)(item.getX2() * scaleFactor)){

if (y >= (int)(item.getY1() * scaleFactor) && y <= (int)(item.getY2() * scaleFactor)){

        // Process the item clicked here…

}}
```

Note: Your program will also have to resize the native graphics in the beginning loading phases of your game for non-native resolutions for this all to work correctly visually. This is beyond the scope of this algorithm, but I can give you some code or ideas for it if this is something you are interested in for your personal coding.