Checking on jumping a fence
Two ways to check if two line segments intersect if pixel coordinates

By Jeanine Meyer jeanine.meyer@purchase.edu http://faculty.purchase.edu/jeanine.meyer

In the implementation of a game or other applications, it may be necessary to determine if something has crossed over a line segment. The line segment could represent a fence or a door or paddle or something else in a graphics application involving simulation of motion. This article explains two ways to make the line segment crossing calculation.

A colleague, and very talented creator of games, Joe McKay, asked me for help in writing a function that determined if an animated ball was about to jump over a line segment serving as a paddle. The simulation of physical effects is performed at discrete intervals in contrast to how objects move continuously in space. The ball is displayed at one position and in the next instance, it would be displayed at another position. It jumps. Determining if the ball’s movement has jumped over an object represented by a line segment is a calculation typical for games and simulations. Recalling some high school algebra, I came up with a general solution. However, I soon realized vertical lines and parallel lines would be problematic. I developed two approaches for handling all cases.

I inserted my first solution into a program he had written using the Processing language. For this article for <jsmag>, I re-wrote both approaches using HTML5 and JavaScript. Figure 1 shows the opening screen of the program that accepts mouse input. The player will press down on the [left] mouse button, drag to indicate travel, and then release the button. Later I will show a program with a more elaborate interface since I could not depend on the tester acting as player, namely me, eyeballing the situation and using the mouse to test all the critical cases.

After pressing the mouse button down, dragging and then releasing to define two points, the program performs the calculation. The red square represents the first position; the blue square the second position; and the purple square, the intersection of the line containing the two points and the black line. Figure 2 shows what will be displayed when it is determined that the line segment has been crossed.
It is important in this situation to distinguish between lines crossing and line segments crossing. Figure 3 shows a case of the crossing being on the line of the paddle, but not on the paddle itself. The crossing is indicated by the purple square. My program does not issue an alert message in this situation, which is the proper course of action.

My first approach appears to solve Joe's problem both in terms of producing the correct answers and doing it fast enough to not interfere with the animation. It made what I term systematic use of fudge factors! However, I did feel the need to modify the initial approach to be something cleaner. At this point, I realized that to test the new program, I needed a way to generate various examples, specifically, the troublesome cases I now felt I could handle directly. This led to what is shown in Figure 4. Mouse down and dragging is still permitted as well as the option to randomly re-position the paddle, but the option also exists to enter in all the coordinate data to specify the two point/ball positions and the end points of the line segment.

Notice that there is a field labeled Result that the program uses for the results. Figure 5 shows an example with a line crossing.
You can view my programs as having a substantial amount of packaging around the crossing calculations, packaging that Joe does not use. Many times, you will need to construct such throw-away code to fully test parts of your programs. You, dear reader, can make use of the calculations in your applications or just read this as a refresher for algebra. I now will focus on an explanation of the mathematics of intersecting line segments.

**Algebra refresher**

The canonical form for the equation for a line that you may remember from algebra class is

\[ Y = M X + B \]

where \( M \) is the slope, the ratio of the change in the \( y \) (vertical) coordinates of points on the line to the change in the \( x \) (horizontal) coordinates. The \( B \) is the \( y \)-axis intercept: the point at which the line crosses the \( Y \)-axis, that is, the value of \( Y \) when \( X \) is zero.

The slope can be calculated from any two points on the line. Given the two \( x,y \) points

\[ (qx, qy) \text{ and } (px, py) \]

the slope is the ratio of the change in \( Y \) divided by the change in \( X \). This is

\[ (qy-py)/(qx-px) \]

Perhaps you already see one of the troublesome situations: what if \( qx \) is the same value as \( px \)? This would happen for vertical lines. Dividing by zero is a big no-no for programs. Let’s keep going and I will address this issue soon.

I am going to use a variation of the canonical format, based on slope and two points. I also will switch to all lower case letters and also use * for multiplication. The equation is

You can convince yourself that this is the right equation by inserting the value \( py \) for \( y \) and \( px \) for \( x \) and seeing that the equation is true (both sides equally zero) and then inserting \( qy \) for \( y \) and \( qx \) for \( x \) and, again, noting that both sides of the equation are equal.

If the task of the program is to compute the intersection of two lines, one line, the paddle, defined by the pair of points: \( (ax, ay) \) and \( (bx, by) \) and the other line, the moving ball, defined by the pair of points \( (qx, qy) \) and \( (px, py) \), I can use the canonical form of the two equations to solve first for \( x \) and then use this value in either equation to solve for \( y \). The canonical form of the other line is

\[ y - ay = ((by-ay)/(bx-ax))*x \]

To put it another way, I work off-line to manipulate the equations to determine the coding. My manipulation is to solve each equation for \( y \) and set the two equations equal to each other. THEN solve for \( x \). Keep in mind that the values \( px, py, qx, qy, ax, ay, bx, \) and \( by \), and are all, or, rather, will be all defined when the code is invoked. This \( x,y \) point is the intersection of the two lines. The next challenge is to determine if it is on both line segments. Think of this as being between the \( p \) and \( q \) points on the one line and between the \( a \) and \( b \) points on the other line. This is done by determining the ratio \( (x-ax)/(bx-ax) \) and \( (x-px)/(qx-px) \). These values all need to be between 0 and 1.

The code shown in Listing 1 computes the intersection and displays a purple square at the point of intersection. The code then calculates if this intersection is on both line segments. If you look back on Figure 3, you see a situation of the lines crossing, but the line segments not intersecting. The \( tline \) and \( tball \) variables are set with values indicating the position on the line relative to each pair of points defining the line segment. To be on the line segment, the values \( tline \) and \( tball \) must each be between 0 and 1.

```
mline = (by-ay)/(bx-ax);    // slope of paddle
mballs = (qy-py)/(qx-px);   // slope of ball travel
m = mballs/mline;
x = (ax-ay/mline-m*px+py/mline)/(1-m);
y = mline*(x-ax) + ay;
ctx.fillStyle = "purple";
ctx.fillRect(x,y,6,6); // marks intersection of lines
// is the intersection actually on both line segments
if (tline >=0 && (tline<=1) && (tball>=0) && (tball<=1))
{ return true;}
else { return false;}
```

Listing 1: Critical part of code

**Comment:** At this point, you may wonder if the upside down nature of computer screen coordinates is factor now. I certainly did. It turns out that it isn’t. The algebra still works.

When will this code NOT work? As I indicated earlier, the problems arise when the code attempts to divide by zero: when \( ax \) is the same as \( bx \), when \( px \) is the same as \( qx \) and when \( mline \) is the same as \( mballs \) (so \( m \) is equal to 1 and 1 is zero). These mathematical cases represent one or both of the line segments being parts of vertical lines or the line segments constituting a pair of parallel lines. These cases may never or rarely occur in an actual application, though the vertical line situation will occur if the application/game treats vertical borders as lines to check for crossing and if it makes sense to use the same code for the border crossing calculation as crossing other line segments. In any case, we really don’t want to depend on luck! I now will describe two approaches to handling the mathematically troublesome cases.

**Resolve problems by fudging values**

My first approach to the troublesome cases was to fudge the values. By this, I mean that if \( ax \) was the same as \( bx \), I added a very small number to \( bx \). I did the same thing for \( px \) and \( qx \) and also for the slopes. In addition, I adjust the checking for being at the line segments. The code is shown in Listing 2.

```
var fudge = .0001;
var mFudge = -fudge;
function crossOverLine (px, py, qx, qy, ax, ay, bx, by) {
    var mline, mballs, m, x, y, tline, tball;
    if (px==ax) { px+= fudge;}
    if (py==qy) {py+=fudge;}
    if (px==qx) { px+=fudge; }
    mline = (by-ay)/(bx-ax);
    mballs = (qy-py)/(qx-px);
    m = mballs/mline;
    if (m==1) { m = m+fudge;};
x = (ax-ay/mline-m*px+py/mline)/(1-m);
```

Listing 2: throw-away code

1 copy licensed to jeanine.meyer@purchase.edu via coupon code 6cihdd2 06/13/2013 03:53:01 PM
Resolve problems by cases

The other approach to addressing the troublesome cases is to detect and then do the calculations for each case. You can view all the code by using the View source option on the website cited in the Learn More section. Here I will sketch the cases.

If both lines are vertical lines, then my code first checks to see if the x values are the same. If this is so, then the code checks if the segments overlap. If this is not the case, then there is no crossing because these are two distinct, parallel vertical lines.

If the first (the a, b line) is vertical and the other line is not, then my code checks for crossing using a somewhat simpler algebra. The code needs to determine the minimum and maximum of the pairs px and qx and then checks if ax is in-between. The code does a similar check if the p,q line is vertical and the a,b line is not.

At this point in the coding, neither line is vertical so the slopes can be calculated. The next condition to check for is the two slopes being the same. There are two subcases: distinct, but parallel lines, in which case there is no intersection, or the same line. If it is the same line, then there is a check for overlap.

When all these cases have been rejected, then the problematic cases have been eliminated and the original solution, based on the algebra indicated earlier, can be performed.

Testing

Let me briefly describe the coding for testing the application. Again, it mostly is 'throw-away code' in terms of the given application. It is necessary because otherwise, I would not be able to test all the cases.

I set up a form for the tester/player/user to input values. The HTML for this is shown in Listing 3.

```html
<form onSubmit="return setvalues();" name="f">
  Point start px: <input name="pxv" value="" />
  Point stop qx: <input name="qxv" value="" />
  Line start ax: <input name="axv" value="" />
  Line stop bx: <input name="bxv" value="" />
  Line stop by: <input name="byv" value="" />
  Line start ax: <input name="axv" value="" />
  Line start ay: <input name="ayv" value="" />
  Line start bx: <input name="bxv" value="" />
  Line start by: <input name="byv" value="" />
  Result: <input type="text" name="results" />
</form>
```

Listing 3: Form for testing

Notice that the "return setvalues();" setting for the onSubmit attribute ensures that the browser does not refresh the screen.

I also provided a way to randomly position the paddle line segment, done using

```html
<button onClick="changeline();">Change the line randomly</button>
```

The changeline function is shown in Listing 4. The re-positioned line segment is drawn AND the values are displayed in the function.

```javascript
function changeline() {
  ax = 100 + Math.floor(Math.random()*1000);
  ay = 50 + Math.floor(Math.random()*500);
  bx = 100 + Math.floor(Math.random()*1000);
  by = 50 + Math.floor(Math.random()*500);
  drawline();
  document.f.axv.value = String(ax);
  document.f.ayv.value = String(ay);
  document.f.bxv.value = String(bx);
  document.f.byv.value = String(by);
}
```

Listing 4: The changeline function

The code for the determination of the line segment representing the ball movement may be similar to an actual game. I chose to use mouse down, drag and then mouse up. Since I do not attempt to track the movement, but just record the mouse down and then the mouse up positions, my code only needs to set up handling for the two events.

```javascript
function init() {
  canvas = document.getElementById("canvas");
  ctx = canvas.getContext("2d");
  canvas.addEventListener("mousedown",firstposition,false);
  canvas.addEventListener("mouseup",secondposition,false);
  ctx.strokeStyle = "black";
  ctx.lineWidth = 3;
  drawline();
}
```

Listing 5: The init function with the addEventListener calls

Both the firstposition and the secondposition functions determine the location of the mouse using standard coding. This standard coding makes allowances for differences among the browsers. The firstposition function, shown in Listing 6, draws the red square and also displays the value in the form fields.

```javascript
function firstposition(ev) {
  if ( ev.layerX || ev.layerX == 0) { // Firefox, ???
    px= ev.layerX;
    py = ev.layerY;
  }
  else if (ev.offsetX || ev.offsetX == 0) { // Opera, ???
    px = ev.offsetX;
    py = ev.offsetY;
  }
  ctx.fillStyle = "red";
  ctx.fillRect(px,py,10,10);
  document.f.pxv.value = String(px);
  document.f.pyv.value = String(py);
}
```

Listing 6: The firstposition function with the addEventListener calls
Listing 6: Function responding to mouse down

The `secondposition` function, shown in Listing 7, draws the blue square, displays the data in the form fields, and, if the positions aren’t identical, invokes the `crossOverLine` function. If the two positions are identical, it issues an alert box.

```javascript
function secondposition(ev) {
  if (ev.layerX || ev.layerX == 0) {
    qx = ev.layerX;
    qy = ev.layerY;
  } else if (ev.offsetX || ev.offsetX == 0) {
    qx = ev.offsetX;
    qy = ev.offsetY;
  }
  ctx.fillStyle = "blue";
  ctx.fillRect(qx,qy,10,10);
  document.f.qxv.value = String(qx);
  document.f.qyv.value = String(qy);
  if ((px==qx)&&(py==qy)) {
    alert("Start and end points are the same. Try again: mouse down, drag, then mouse up.");
  } else if (crossOverLine(px,py,qx,qy,ax,ay,bx,by)) {
    document.f.results.value = "Crossed the line";
  } else {
    document.f.results.value = "did NOT cross the line";
  }
}
```

Listing 7: Function handling mouse up event

I repeat: most of the code in my programs is for testing. If you need to make a determination of line segments crossing, you just need to copy the `crossOverLine` function.

Learn more

There are many sources, online and in-print and some sort of e-books, for learning HTML5 and JavaScript techniques. Here are links to my recent books and the websites for the two programs.

- Joe McKay’s personal web site is [http://www.joemckaystudio.com/](http://www.joemckaystudio.com/).

- **The Essential Guide to HTML5: Using Games to learn HTML5 and JavaScript**, [http://www.friendsofed.com/book.html?isbn=9781430233831](http://www.friendsofed.com/book.html?isbn=9781430233831), This is a text for beginners at programming as well as more experienced programmers who want to learn about HTML and JavaScript, including the new features of HTML5. A demonstration is given of collision detection between the playing piece and the walls in the chapter on mazes.

- **HTML5 and JavaScript Projects**, [http://www.apress.com/9781430240327](http://www.apress.com/9781430240327). This book is more advanced than the first one. The chapter on origami contains material on determining intersection of lines.

- To see the application in action and to view the source code, go to [http://faculty.purchase.edu/jeanine.meyer/html5/crossingtheline0.html](http://faculty.purchase.edu/jeanine.meyer/html5/crossingtheline0.html) for the first program "with fudging" and [http://faculty.purchase.edu/jeanine.meyer/html5/crossingthelineallcases.html](http://faculty.purchase.edu/jeanine.meyer/html5/crossingthelineallcases.html) for the second program, containing the more elaborate testing form and the program that handles all cases directly.

Jeanine Meyer lives just north of New York City and currently teaches at Purchase College/SUNY after many years at IBM, doing research on robotics and manufacturing and consulting on educational grants. She likes providing programming examples for her Mathematics/Computer Science and New Media students as well helping and learning from colleagues.