Pixel-Perfect Collision

Written by: Phaelax (5Apr2017)

First of all, a shout out to blink0k for his theory which gave me this whole idea.

Audience

Though the language syntax I will use to demonstrate the concept is AppGameKit (AGK), the logic can be used by anyone who is looking to further understand the concept of pixel-based collision systems.

The Problem

Detect if two sprites overlap by using pixel-perfect accuracy instead of bounding boxes.

The Theory

The idea is to convert each sprite into a bit mask using it's alpha channel. Each bit represents 1 pixel in the image where 0 means that pixel is transparent and 1 means it contains a color. It doesn't matter what color or even its level of opacity is, we only need to be concern about which pixels are visible and which aren't. If a pixel is represented by a 4-byte color value (RGBA), then a sprite of 256x256 would require 262,144 bytes. If we store only the alpha value then it's only 65,536 bytes. But we don't need a variable alpha value, a simple 1 or 0 will do just fine. So instead of storing the alpha as series of bytes, we can cut that size down by 8 times and store each as a single bit. Our alpha mask is now a mere 8,192 bytes! If you have over 100 sprites on the screen, you can see how quickly this starts to save memory.

You would then determine which part of these bit masks overlap when comparing two sprites. If two overlapping pixels both contain a 1, then the sprites have intersected and you can exit the loop early.

The Bit Mask

How do we extract the alpha values and store them into an array of bits? First you'll need a way to access the image data related to the sprite. To do this in AGK we'll use memblocks.

```
image = getSpriteImageID(sprite)
mem = createMemblockFromImage(image)
```

Next, determine how large our bitmask array should be. This is calculated easily by multiplying the width of the image by its height then dividing it by 32 (because 32 bits in an integer). Keep in mind you can't have a fractional integer, so always round up. At most you will have is 7 unused bits, which will remain 0.

```
maskSize = ceil(getImageWidth(image) * getImageHeight(image) / 32)
mask as integer[maskSize]  // initialize bit mask array
```

The size of an image memblock is equal to the width of the image times its height times 4 plus a 12 byte header. It's multiplied by 4 because each pixel is represented by a 4-byte color value. The first 12 bytes of the memblock are three integers containing the width, height, and depth of the image, respectively. After that follows the pixel data. Every 4 bytes is a new pixel in the image, but we will be concerned with the one containing the alpha channel, which is the 4th byte.

Loop through all pixels in the image and store it's alpha value as a single bit into the array. Remember, each integer in the array will contain 32 bits; 8 pixels.

```
bit = 0
for i = 12 to getMemblockSize(mem) - 4 step 4
      if getMemblockByte(mem, i+3) > 0
            k = floor(bit / 32)
            b = bit - k * 32
            mask[k] = mask[k] || (1 << b)
      endif

      inc bit
next i
```

If you're new to bitwise operations, don't worry I'll explain what that weird line means. Ok, first set up the FOR loop. We start at 12 because there is a 12 byte header at the beginning (bytes 0 through 11) and we can skip those. Every 4 bytes equals a single pixel, thus we step by 4 each iterations to get to the next pixel in the memblock data. The value of '**i**' will always point to the beginning of the 4-byte pixel. When you're grabbing a single color component from a memblock, the alpha channel is stored last.

Alternatively, you could get an integer representing the entire color in a single number, at which the alpha channel is technically listed first. Confused? Don't worry, it's called endianness and determines the byte order. It's outside the scope of this tutorial but feel free to google it, it's handy information to understand. However, an understanding of endianness is not needed for this. All you need to know is that the alpha channel is the 4th byte in each iteration of the loop.

If the alpha value is anything other than 0, then it contains a visible pixel and we should flip the corresponding bit in the mask. The value of **bit** is incremented each iteration to match a pixel in the image. Technically, **bit** is the same as **i-12.** Because there are 32 bits to each integer, every 32 pixels goes into a new integer. And as our mask is defined as an array of integers we must determine which integer this bit will fit into, which is why we divide the bit variable (think of bit as an index) by 32. The resulting value of the division can be a decimal value, thus either floor it (round down) or truncate the fractional portion. The whole number from the division represents the integer we'll be manipulating. The fractional part will be used to determine which bit in that integer to set. Let's look at an example.
Below is the binary representation of 8 bytes, or two integers.

Let's say we have pixel number **74** and its alpha value is greater than 0.

74 / 32 = 2.3125 *(value of 'k' in the above code example)*

Remember we only want the whole number here as this is the index to the integer in the bitmask array, which is **2**.

74 – 2*32 = 10 *(value of 'bit' in the above code example)*

mask[2] = mask[2] || (1 << 10)

Ok, so what this last line means is take the existing integer in the array and bitwise OR it with another number which has had its bits shifted left 10 places over. Still confused? It helps to look at this in binary.

[10001000001100001000000000100010] **mask[2]** (represents the integer value of 2,284,879,906)
OR
[00000000000000000000100000000000] **(1 << 10)** (represents the integer value of 1,024)
=
[10001000001100001000010000100010] (represents the integer value of 2,284,880,930)

What you see for the value of mask[2] is it's binary representation. The other is what happens when you shift the bits of number whose value is 1 over by 10 places. This bit shifting is key to how we get and set the alpha values into handful of integers. When you bitwise OR the two numbers together, the resulting number will compare the two binary numbers and flip the bit in any place where either number is a 1. In other words, if this is true OR that is true then it's true. We do this to keep the existing bits already stored in the mask intact.

And that covers the setup to building a bitmask of a sprite. Next we'll see how to use this data for collision checks.

The Collision

First, you check for intersection using the bounding boxes of the sprites. We do this first because it's a simple check that barely costs anything in performance. And if the two sprites' boxes don't overlap then we can skip over the real collision check. If you're uncertain what I mean by bounding box, it's just a simple rectangle which defines the basic shape of the sprite's image. Regardless of how images appear on screen, they're all squares or rectangles.

The red rectangle is this sprite's bounding box. In AGK, the sprite is positioned by the upper left corner. Given a sprite at **[x,y]** the box would be defined as **[x, y, x+spriteWidth, y+spriteHeight]**.

I'll use pseudo code to make this easier to read, but this is how you can check if two boxes overlap at all. Assume two sprites designated **A** and **B** with properties **X** and **Y** to denote their position and **width** and **height** to represent their dimensions.

IF (B.x < A.x+A.width) AND (B.x+B.width > A.x) AND (B.y < A.y+A.height) AND (B.y+B.height > A.y)

If that condition is true, then the two bounding boxes intersect and so we should continue with the bitmask check.

Before we can compare our two bit masks we need to determine which part of the two sprites overlap. To do this we'll need to write a new function which takes 8 parameters, box A and box B. These are the bounding boxes of the sprites and we'll define them for this function by giving the top left coordinate of the box. So the first 4 parameters of the function will be for box A and the other 4 parameters box B.

Function getIntersection(A.x, A.y, A.x+A.width, A.y+A.height, B.x, B.y, B.x+B.width, B.y+B.height)

Now define the sides of the intersecting box.

left = max(A.x, B.x)
top = max(A.y, B.y)
right = min(A.x+A.width, B.x+B.width)
bottom = min(A.y+A.height, B.y+B.height)

Create a User-Defined Type (UDT) and call it Rectangle. This will contain 4 variables: X, Y, WIDTH, HEIGHT. This will define the intersecting rectangle and allow us to return the relevant data easily from the function.

```
Type Rectangle
     x as integer
     y as integer
     width as integer
     height as integer
EndType
```

Now define the rectangle's properties and return it from the function. Here's what the entire intersection function would look like.

```
function getIntersection(ax1, ay1, ax2, ay2, bx1, by1, bx2, by2)
     left   = max(ax1, bx1)
     top    = max(ay1, by1)
     right  = min(ax2, bx2)
     bottom = min(ay2, by2)

     r as Rectangle

     r.x      = left
     r.y      = top
     r.width  = 0
     r.height = 0

     if right  > left then r.width  = right - left
     if bottom > top  then r.height = bottom - top
endfunction r
```

The bounding boxes are all in world space, and hence so is the intersection rectangle. We must translate this rectangle's origin to that of each of the sprites. To do that, simply subtract the sprite's origin from the rectangle's origin.

```
x1 = r.x - A.x
y1 = r.y - A.y

x2 = r.x - B.x
y2 = r.y - B.y
```

Next, we will need the sprites widths so we can translate a pixel with an [X,Y] coordinate to a single dimension of data. Now loop through the pixels contained within that intersection rectangle

```
for y = 0 to r.height-1
      for x = 0 to r.width-1
            // translate pixel space into single dimension
            aPos = (y1+y)*A.width + x1+x
            bPos = (y2+y)*B.width + x2+x

            ai = floor(aPos / 32)
            ab = aPos - ai*32
            aValue = (A.mask[ai] >> ab) && 1

            bi = floor(bPos / 32)
            bb = bPos - bi*32
            bValue = (B.mask[bi] >> bb) && 1

            if aValue = 1 and bValue = 1 then exitfunction 1
      next x
next y
```

Eww, more bitwise operations! Have no fear, it's very similar to what was done above. Instead of using OR **||** we use AND **&&**. This time shift the bit we want all the way to the right. This will put the bit in the first position of the integer. The number 1 in binary will appear as a line of zeros except for a single bit flipped at the very beginning. We'll use this as a sort of mask when we AND it with the real value. What this means is instead of returning true for each bit where one of the two numbers being compared is a 1, this only returns true if both are 1. And if only our first bit in the integer is set (as in 1) then only that bit from our masked value will be returned. Compare the overlaying bits (or pixels) from the two sprites. If both equal 1 then the sprites have collided.

Conclusion

That's it! It's really quite simple once you grasp the concept. If you change the mask array to two dimensions instead of one, you could easily adapt this code to suppose animated sprites. The first dimension of the array would match up to a sprite's frame, the second dimension containing the bit data for that frame.