A 2d collision detection tutorial, including a C implementation. first draft, please email comments!

Ulf Ekström
ulfek@ifm.liu.se

July 12, 2002

1 Introduction

This tutorial tries to explain a commonly used approach to 2d collision detection for use in games. A special 'mask' is created for each sprite, and is used for the overlap detection. This method is suitable for pre-rendered or hand-drawn graphics as it gives pixel-perfect collision detection. The method is also pretty fast and does not slow down the game noticably when compared to other existing methods.

The method has been commonly used in games ever since the days of the Commodore 64, and is well understood even though the actual implementation can be a bit tricky to get right. A GPL'd implementation of the ideas in this tutorial can be found at on the internet [1].

2 How to know when things collide

The heart of the problem we are trying to solve goes something like: There are some sprites in the game, each one with a mask and a position. We would like to determine if these sprites have collided, and possibly some details of the collision such as the point of intersection. We begin by checking each pair of sprites to see if they overlap. This method has it's problems with a large number of sprites, and we will improve it later on, but for now it will suffice.

2.1 The bitmask

In the following discussion we assume a 32-bit machine, but the same points are valid for 16 or 64 bits as well. A bitmask is essential a 1 bit per pixel image, and to store the bitmask we use a struct like

```
struct bitmask{
int width, height;
unsigned long *bits;
}; .
```

The `bits` pointer is used to access the actual mask. Each line of bits in the mask requires a whole number of int's, to speed up the intersection tests we are doing later. The remaining bits are set to 0. Assuming we have allocated some memory for the bits it is now easy to set and clear bits in the mask using the bitwise AND and OR operators. The idea is to set all 'solid' bits to 1 and all the unoccupied bits to 0. In practice this means that we use an alpha channel or a special magic color to mark the transparent areas of the images. It is also possible to OR a mask on top of another mask in a fast way, which can be used to construct a large mask out of smaller 'brushes'.

Masks created in this way are pretty memory-efficient. They require only a little more than a bit per pixel of the original image, which is typically 16 or 32 times less than the memory used for the actual image.

[1] `www.ifm.liu.se/~ulfek/project/bitmask-1.0.tar.gz`

2.2 Detecting a collision

We have now created bitmask for our sprites and want to know if they actually collide. Again we shall use the bitwise AND (&) operator, together with shifts ($\ll$ and $\gg$ in C). The idea is to identify the common area of the two masks, and then perform a bitwise AND to check if they share a common '1'-bits. This way we can check 32 bits at a time which speeds up the test enormously. Once we find an overlap we can return from the function since it is now clear that the sprites have collided. The first thing to check is however if the bounding rectangles of the masks intersect at all. Since most pairs of sprites doesn't overlap at a given time we will perform many such bounding-box tests and thus need them to be very fast.

It is clear that the most computationally expensive case is the one where the sprites are so close that their bounding boxes overlap, but not so close that there is an actual collision. We then need to examine each overlapping line of the masks, only to find that there is no collision. Depending on the type of games it may therefore be a bad idea to use a very large 'background' mask, if it's mostly empty. For most cases there will however be no performance problems.

2.3 Finding a point of intersection

By modifying the above code slightly we can find the coordinates of the first point where the masks overlap, counting top-down, left to right. In the bitmask implementation of this tutorial this is done by examining the bits of the single unsigned int that was found by the collision detection algorithm above.

A possible problem is that the point found by this algorithm isn't really the 'best' point from the game point of view. It would perhaps be better if one could find the center of the overlapping area. This has however not been implemented, and would probably require substantially more CPU time.

2.4 Computing the area of intersection

To know how 'severe' the collision is it is nice to know the number of overlapping bits, or pixels in the sprite images. This is easily added to our original collision detection function by counting the bits in all nonzero overlap tests. A fast bitcount function is used for this purpose, and results in a function that is perhaps 50% as fast as the original collision detection routine. We obviously only need to count the bits if a collision has been detected and is interesting for game purposes, so the overall impact of using this approach is very small.

2.5 Determining an angle of collision

A very robust way to determine the angle at which the sprites collided can be had from the gradient of the overlap area. Calculate the gradient of the overlap area $f(x, y)$, where x and y are the differences in position of the two sprites.

$$\nabla f(x,y) \approx (f(x+1,y)-f(x-1,y), f(x,y+1)-f(x,y-1)) \quad (1)$$

This gradient vector will point in the direction where the overlap increases the most, and this can be used as the normal vector of the collision. Overall this method is very good at finding good looking normals, and has very few weaknesses. It does however require four calls to the overlap area function, and can therefor be a bit expensive. It should only be used when actually needed.

2.6 Some tricks and tips

Sometimes you don't actually need pixel-perfect detection. If your games runs at very high resolution you may not need more precision than, say, 4 pixels. In this case you can create masks at this resolution and use them for the collision detection. Just rememeber to scale your sprite coordinates by the same amount..

Your game might be more fun to play if the masks are a little smaller than the sprite image, or if you use

the overlap area to apply only a little damage to the player if the overlap is small. An automatic way to get this effect it to set all bits that borders on a 0 bit to 0. This will also remove unwanted noise from the masks. On the other hand you may want to detect collision before they are visible on screen, in which case you can grow the masks a little bit.

As a general guideline it is wise not to store each overlap in a list, but instead use function pointers or object oriented techniques to handle each collision as it is detected. This leads to cleaner and faster code, at least in my experience.

Note that the mask does not *have* to be the shape of an actual sprite image. In a pseudo-3d game like Diablo or Baldurs Gate it is best to use a special mask based on how the sprites look from above.

3 How to know when they stay apart

We now know how to find collisions, and while that's the primary point of this tutorial there is another side of the story which is just as important when there is a large number of objects in the game. The 'problem' is that most of these objects does not overlap. Using the method above we would still have to test each and every pair of sprites. With 200 sprites in the game this means 20000 pairs, and that is a bit much even if we can discard most of the pairs at the bounding-box stage. But there are better ways!

One obvious (depending on the game) improvement is that we only need to consider moving sprites. We keep two groups, one moving and one static, and only test moving vs moving and moving vs static. The static sprites can even be placed in a 2d grid to easily find possible candidates for collision. Another solution may be to not check every groups of sprites vs itself. If it is decided that bullets cannot hit each other then a lot of tests can be avoided.

A general and effective solution is to separate the sprites by their position. I have used a quicksort-inspired algorithm which gave good performance for a few hundred sprites. It uses an array of pointers to the sprites which can be largely recycled between different frames of the game. The method alternatively sorts by x and y coordinate, and stops when each group is small enough to check with the all-pairs approach, or when it seems impossible to partition the sprites further. It works by selecting a divisor, a coordinate by which to partition the sprites. Sprites entirely to the left (or above) of the divisor are placed first in the array, while sprites entirely to the right (or below) are places at the end. In between are sprites which intersect the divisor.

$$[unsortedsprites] \rightarrow [L, I, R] \tag{2}$$

We know that the sprites in groups L and R cannot intersect since they are on different sides of the divisor. Hopefully both these groups contains about the same number of sprites, and the undecided I-group should be as small as possible. It is now possible to partition the groups further if is seems profitable. An advantage of this method is that it handles arbitary sprite positions and sizes, and that it does not require any dynamic memory allocation, and does not require a complete sorting of the sprites. The method is much faster than a static grid, at least for a moderate number of moving sprites.

4 Other methods

If you have a lot of dynamic geometry I suggest you use a vector-based collision detection scheme. This may or may not lead to a faster game — on one hand it may be more elegant and you might be able to do more advanced things like having freely rotating objects; on the other hand it is highly non-trivial to get working correctly, and it may be harder to produce contours from the sprite images in an automatic fashion. For more information on this subject I suggest you look for 3d collision detection schemes and try to convert them down to 2d. See the net for a lot of info on this subject.

It may also be nice to use RLE encoded masks. This require a special compilation stage after the mask have been created, and makes the masks effectively read-only. It may however speed up testing of very large masks, and is probably worth a closer look.