nesdoug

10. Background Collisions

A bit more challenging than Sprite vs Sprite collisions. Since most games use square blocks of 16×16 metatiles and 16×16 sprites, we will use this as the example. Currently, I have it set to move only 1 pixel R or L or U or D per frame. Every time we move, we need to check for collision. I just check every frame.

The standard, apparently, is to allow the move, check collisions, then eject if collision.

First we're going to allow the X movement, check collision, eject in X direction. Then we're going to allow the Y movement and eject in Y if collision.

Another thing… It's a pain to try to read from the PPU. You'd have to calculate the correct nametable address, AND fetch it from the PPU (during V-blank) in a timely manner in which you can then have game logic around movements and still have enough V-blank time to update Sprites…so let's drop that idea.

What we are going to do is have a map of the background metatiles that we can nicely fit in 1 page of RAM. This can index to a collision array to indicate which tiles are solid. In my example, metatile 0 = not solid and metatile 1 = solid, so I don't need to index to a solidity array. If we had more metatiles, we would need a solidity array.

[On a side note, I'm going to load this map into the RAM, but I suppose this was an unnecessary step. I could have just assigned a pointer to the ROM location of the collision map, and indexed using that pointer.]

How do we make a collision map? Sounds time consuming. Well, I used Tiled to make the collision map. First I constructed metatiles in NES Screen Tool (not too hard, we only have 2 here). And, I took a screenshot and cropped it in Photoshop to a 256×256 image. Then I brought that into Tiled, with 16×16 map of 16×16 tiles. Then I redrew the background using metatiles, and then Exported .csv format files (1 file per background). I had to edit the .csv file slightly so it looks like this…

```
const unsigned char c2[]={
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,
0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,
0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,
0,0,0,0,1,0,0,0,0,1,0,0,0,0,0,0,
0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
```

```
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
};
```

Then I included it in the C code, and referenced its address in an array of addresses. (If any of this is confusing, I included all the files with the source code).

Whenever we press 'start' to load a new background, it also loads the collision map into $300-3ff of the RAM. Note, I had to edit the .cfg file to get this to work like I wanted. Specifically, I added a "MAP" segment and defined it to be at $300 and $100 in size. In the C code, I just defined an empty array to be there…

```
#pragma bss-name(push, "MAP")
unsigned char C_MAP[256];
```

[Again, probably not necessary, it COULD have gone anywhere, but I like knowing that it's exactly at $300. When I open the hex editor debugging tool of FCEUX, and scroll to $300, it's nice to know that I'm looking at the collision map.]

and did this when Start is pressed, to transfer the collision map from the ROM to the RAM at $300.

```
p_C_MAP = All_Collision_Maps[which_BGD]; // pointer to a collision map
 for (index = 0;index < 240; ++index){
 C_MAP[index] = p_C_MAP[index]; // transfer to the RAM
 }
```

When I looked back at this later, I thought, "Why didn't I use memcpy? So I replaced it with memcpy, and it turns out this innocent looking "for loop" takes 42688 cycles to complete… 9 times longer than memcpy. Here's how it should look…

```
void __fastcall__ memcpy (void* dest, const void* src, int count);
```

```
p_C_MAP = All_Collision_Maps[which_BGD]; // pointer to a collision map
memcpy (C_MAP, p_C_MAP, 240);
```

Then I wondered, if memcpy is really the fastest? So, I rewrote this transfer in ASM, just to see if it could be done more efficiently. (That source code is bellow, in the lesson8c.zip)

The ASM version was only 4% faster than memcpy, so I guess it's safe to use memcpy to transfer bytes. They all appear to work exactly the same, so it's a bit of a moot point. Later on, when our code is much longer, and we're worried about fitting all the logic in 1 frame, then we can worry about which code is faster.

And we do a little math to see if we are colliding with the background. First we do moves R or L, calculate the right and left side of the sprite (ours is narrow), and then check (if right) the top right and bottom right corners, to see if they are inside a solid tile….

```
if ((joypad1 & RIGHT) != 0){



//top right
  corner = ((X1_Right_Side & 0xf0) >> 4) + (Y1_Top & 0xf0);
  if (C_MAP[corner] > 0)
   X1 = (X1 & 0xf0) + 3; //if collision, realign



//bottom right
  corner = ((X1_Right_Side & 0xf0) >> 4) + (Y1_Bottom & 0xf0);
  if (C_MAP[corner] > 0)
   X1 = (X1 & 0xf0) + 3; //if collision, realign
 }
```

Why +3? If you remember from before, our sprite is blank on the 3 pixels on the left (and right).

If collision left, we adjust right…

X1 = (X1 & 0xf0) + 12;

Then I move Up or Down, and then check the top or bottom corners for collision. Our sprite is a full 16 pixels tall…

If collision down, adjustment upward…

Y1 = (Y1 & 0xf0);

If collision above, adjust downward…

Y1 = (Y1 & 0xf0) + 7;

[I just made all these up quickly. They seem to work. I suppose, if our sprites are moving faster than 1 pixel per frame, this logic might not work, and would have to be rewritten.]

Again, sprites always show up on the screen 1 pixel lower than you would expect. You could make an adjustment before updating the Y coordinates to the sprite RAM (OAM). On platform games, I think it looks ok. Top-down games might look odd (as though the character is standing on the wall below him.)

Here's the source code, the 3 versions only differ in that one 'for loop' / 'memcpy' / asm transfer. I thought it would be less confusing if they were separated.

http://dl.dropboxusercontent.com/s/usbt4evqf4bn41y/lesson8.zip (http://dl.dropboxusercontent.com/s/usbt4evqf4bn41y/lesson8.zip)

http://dl.dropboxusercontent.com/s/w3fvsw93e4wwb20/lesson8B.zip (http://dl.dropboxusercontent.com/s/w3fvsw93e4wwb20/lesson8B.zip)

http://dl.dropboxusercontent.com/s/dxiohi67uheaazk/lesson8C.zip (http://dl.dropboxusercontent.com/s/dxiohi67uheaazk/lesson8C.zip)

November 29, 2015April 15, 2017 dougfraker

4 thoughts on "10. Background Collisions"

1. **Boring Danger (@fritzvd)** says:
   May 2, 2016 at 2:02 pm Edit
   Hi Doug,

   I'm really enjoying these tutorials.

   At the moment I'm trying out some stuff with the backgrounds and I was wondering if you can say anything about the RLE-encoding. I was trying to make my own version of the header files, but failed. I couldn't really tell from using the NES screen tool or the UNRLE asm function which RLE algorithm was being used.

   Any way you could elaborate on that?

   Fritz

   Reply

   dougfraker says:
   May 3, 2016 at 2:19 am Edit
   Shiru wrote the RLE code. But it goes like this. First byte {1} is a unique byte, called "tag". Next byte {2} is a tile to go to the nametable. If the next byte {3} is the "tag" then the following byte {4} is how many more repeats. If the next byte {3} is not the "tag" then the that byte {3} becomes the

next new value (tile). If the byte after “tag” {4} is a zero (zero repeats), then that is an End Of File marker, and it exits.
…the “tag” value appears to be the first unused tile value, generated by the NES Screen Tool.

2. Reply

Boring Danger (@fritzvd) says:
May 12, 2016 at 7:31 am Edit
Ok. Thanks. I thought it should be something like this. But there are quite a few implementations, and my reading of asm isn’t quite up to savvy 😉

Thanks a bunch

3. Reply

jomo0825 says:
January 30, 2017 at 1:49 pm Edit
This is definitely the best CC65 NES tutorial I’ve ever seen. Nice and clean!

Reply