

# Bandersnatch for C64 v0.1: the story so far

*Warning: this article contains Black Mirror: Bandersnatch spoilers, proceed at your own risk.*

In January I got sucked into the gripping interactive film that is [Black Mirror: Bandersnatch](#) on Netflix. While the average critic finds it to be over-flawed it's fair to say that a great many people got a lot out of the experience of guiding our avatar Stefan as he gradually loses his marbles over and over again, trying to finish and perfect his game also called Bandersnatch.

After my third or fourth go through [I decided to remake the game Stefan is working for on C64](#) instead of as he did on ZX Spectrum. I [researched the clues of his work in the film](#), and [considered it's philosophy](#) and links to the [larger Black Mirror project](#). If you don't quite know what I'm talking about, click on of those links and read up on the story so far.

Now finally I have something to show for it, a 6194 line file of highly annotated assembler code with data, compiling into a playable demo of Bandersnatch for Commodore 64 (AKA C64), which I call...

## **BNDRSNTCH**

### **Overview of game (so far)**

1. Movement through iconic maze corridors in first person rendering
2. Choice screen when confronted by Pax (if you find him!)
3. New feature: map view
4. And a few other technical tid-bits

In this article I will share a few notes on each of these feature areas, prefaced with instructions for play and a general note on the code, and wrapped up with a description of my intentions for v0.2 and beyond.

## How to play

Playing the game is a two step process:

1. Download the game file using [this link](#).
2. Run it on this [online C64 emulator](#) or download the great [VICE C64 emulator](#) and run it on that.

The playable game file is `main.prg`, this is a file format supported by all C64 emulators worth their salt so you should also be able to run it on your favourite emulator also.

All files are hosted on the GitHub project at <https://github.com/simon-butler/bndrsntch>

**Note that from this point on there will be “spoilers” of the game itself.** If you want to enjoy and explore this first version as we did in the old days, without much of an idea of what you’re getting yourself into, load it up now and give it a bash before reading ahead!

## **Overview pt 1: first person in the maze**

The maze appears to be where the bulk of Bandersnatch gameplay occurs. There are several features of note to the maze.

- Fixed perspective
- Two-colour walls (blue and black), with block colour splitting at the vertical half-way point
- Red floor
- Pipes on ceiling (not recreated in my clone yet)
- Sprite overlay when encounter a character (e.g. Pax)
- Sounds effects and melody fragment

### **Fixed perspective**

- The fixed perspective is of course a common constraint of game of the setting era of 1984. True 3D was simply beyond the reach of most (all?) of the games machines at the time, and even where it was attempted it would greatly impact of the resources left for a complete game. Packing as much visual

punch into the limited computers available meant using perspective restrictions and tricks.

## Two-colour walls

- A note on the colours. The colours used on the ZX Spectrum are far more vivid and “pure” than those on the C64. On the C64 we see more a pastel colour palette. Compare the above screenshot with my recreation on the C64. It does seem to be washed out, but that’s only in comparison. Within context of the C64 it looks just fine.
- The two colour wall is an interesting choice. In my implementation it allowed me to make some savings while drawing, I set the background colour to light blue (looks more purple but it’s “blue”) meaning I did not have to explicitly draw the solid parts of light blue colour. When the screen is rendered, I first paint the top half black and the bottom half light blue, then draw in the detail.
- In the film we see the inspiration for Stefan’s game in one of his flashbacks or video cassette tape watching sessions, when we manage to get on the *Program And Control Study* (P.A.C.S.) paths. Here we see Stefan as a child, blindfolded and led by his father (himself an agent of the government and apparently the main conductor of the study) through an industrial-looking hallway, painted in two colours and with exposed pipes on the ceiling, as in the game.

- In symbolic terms it is clear that the common thread between these three situations is loss of autonomy, control and imprisonment. As many have noted one of the main themes of *Bandersnatch* is the illusion of choice and free will. The wall styling creates a visual link for this theme.

## **Red floor**

- The colours red, yellow and blue are highly symbolic and meaningful (see my dissection of symbols in [this article](#)). The branching glyph (AKA the White Bear symbol) gives us the idea of paths, and the floor itself is a pathway. Thus we have “follow red path” in [Stefan’s game design notes](#), as well as “follow yellow stones”.
- The yellow path is the “good path” in *Bandersnatch* and the red path a bad path. So we can see that the child Stefan is on the same bad red path as the agent character is in his game.

## **Ceiling pipes**

- I don’t yet have this in the game as it they are quire detailed and I thought that nice-to-have details like this could wait until the core functionality is implemented. I will be adding them in a later version, probably by using custom character based graphics, as opposed to sprite based graphics.
- What they mean is less clear. They seem to imply an industrial settings, perhaps even underground.

## **Sprite overlay**

- There are two characters you can encounter that we see in Stefan's demo, either Pax or the Agent. Stefan's demo shows the character far away, closer and then in full screen (which is the choice screen). This fits well with the sprite graphics modes available on C64 which offer a normal mode and a expanded size mode, for both horizontal and vertical dimensions. This allows us to double the size of a sprite while using the same sprite data.

## **Sounds effects and melody fragment**

I created a basic rendition of the sound effects for Bandersnatch that we see in the film, but it's not very exact. This was one of the last steps I did in the current batch of work and what you hear is just a first draft.

You will hear a short clipped tone when the player character moves to a new map position successfully. You will also get a basic creation of the glitchy low rumblings when you encounter Pax, as well as the melody when you are face to face with him. I copied this by ear and translated the notes into Hz which are then played through the C64's SID chip (as Colin says in the film "great sound chip, the Commodore"). This process was fairly straightforward, but again I am not completely happy with the sounds quality, it's too soft and not harsh enough to compliment the style of the film. This will be improved.

## **Notes on coding for first person view**

I had two concerns when implementing drawing the graphics for this view on the screen:

1. Render speed
2. Image data storage space

The C64 has a very small amount of memory which is available to the programmer. Technically it is 64 kilobytes of addressable memory but there are major and minor caveats to this. I won't go into too many of the details, but they are well known and fairly easy to research if you are motivated.

Since the corridor views are static images (they do not move or animate in any complex way) we *could* store these as images and simply draw to screen. This however would use almost 2K per image if we do not use any compression techniques; 1K for the character to use, and 1K for colour data.

I determined that there are 7 corridor section types (see left above), but Stefan's original graphical rendering *also shows the next corridor section*, if not blocked by a wall. This gives 30 distinct corridor and wall configurations, which would mean 60K used, if it was done in the most straightforward way possible. Of course, this is not necessary, we can compress the data in a few different ways if we are clever. The screen is mirrored, and there are only 10 unique "screen halves", so we can create the required screen on the fly from two screen halves, left and right.

Since what is drawn is made up of lots of similar blocks (using a custom C64 character map), I decided to store the screen as plotting drawing instructions in a custom data format instead of directly as screen data. This is like the difference between using a raster image format such as BMP and using a vector image format such as SVG.

I codified these as you see the comments in the following screenshot from the code.

The difficulty here was that interpreting this data as drawing instructions is not as fast as just drawing some data directly to the screen. To overcome this I used the standard technique of a drawing buffer, where the drawing data is drawn first to an off-screen area of memory and then copied to the screen all at once. To further speed this up I implemented some so called “speed code” in the memory copy, using unrolled loops. In programming a loop is generally achieved using a counter and then looping over the same instructions. When highly efficient code is required however this wastes a few CPU cycles every loop iteration in both the counter number check and jumping back to the start of the loop. Unrolled loops take up more memory but save on a surprising amount of CPU cycles. [There's a good article on it here.](#)



Even with all this, there is a noticeable flicker when the screen is updated, though it's significantly less than without the speed improvements. I intend to improve this further in a future release. It should be noted however that in Stefan's demo we do see a less than perfect screen drawing rendering also.



## Overview pt 2: choice screen

The choice screen is where the main mechanism of the game is expressed, the choice between one of two alternatives. This is the fork in the road shown in the branching glyph, where we can take only one path, rejecting the other. This is also how *Bandersnatch* is “played” on Netflix.

I copied the graphics as faithfully as possible while using 8x8 square blocks to create the face of Pax (the blue “lion” demon). I actually did not copy the above graphic, I copied the one available on the fake Tuckersoft website (which I can no longer find, it might have been removed). I later investigated a way to create it using 4x4 blocks but it was more complex and I decided to stick with the larger version for this v0.1 demo. As you can see it’s a little too large for the screen.

I added an additional feature here that is from the *Bandersnatch* film, not from Stefan’s ZX Spectrum game *in* the film, and that is the white bar timer at the bottom that recedes from both sides to the middle. When the white bar shrinks to the middle, your choice is made. You can press left and right (  or  on the keyboard) to change your choice. It also makes a sound when you change selection.

In future I may add the ability to choose the option with the joystick fire button, as we see Mohan Thakur (pictured below) doing, but I do like the suspense of having to wait also, as it’s done in the film.

The choice time expires after ten seconds (as Stefan has designed), and then you'll be rewarded with a resolution screen and a short original melody I composed. I won't show you here so you can discover that on your own! Then the game will restart from the very start, and this will loop endlessly for now. Though it's in the service of the demo, I actually like this and it will be similar to what you get in the fully developed version. Back in the day this was what happened when you failed (or even completed) games. Back to the title screen.

### **Notes on coding for choice screen**

I needed to develop a few things for this screen:

1. Centered text
2. Text choice change
3. Timer with on screen update and expiration

The difficult part for me was the timer. In actual fact the process is fairly simple, but it wasn't easy to come by the correct knowledge, and it was hard to debug. It's the only part of the code where concurrency (or rather program flow interruption) is a necessary consideration.

As you can see there are some particular steps here to prepare the timer. The timer takes the technical form of an "interrupt", which is a very common feature of microprocessor architecture. An

“interrupt” allows certain hardware events to interrupt the current program flow and then return to it afterwards. This makes all kinds of important features possible, such as hardware I/O and screen drawing. We can also set up our own interrupts in software, which is very handy indeed and a must for interactive games.


I’m using a technique here which ties the interrupt to the screen refresh system, on what’s called a “raster line”, a particular horizontal line of pixels. Remember that the C64 was build for CRT televisions, so the clock is synced for 50 Hz (times a second) in PAL regions (e.g. Europe) and 60 Hz in the USA. The target region is configured at the start of the code.

The handler routine then is called 50 or 60 times a second, depending on region, and allows us to accurately gain an awareness of the passage of time, thus allowing the creation of a timer. See the routines `choice_timer_prepare`, `choice_timer_handler` and `choice_timer_draw` in the code for more details.

## Overview 3: the map, a new feature

This feature has the following main components:

1. Shows current location, look ahead and previously visited locations
2. Shows facing direction
3. Can move in this mode in the same way as you can in first person view

4. Can switch between first person view and map view with  key

One of the first challenges I had to solve was how to store the map data, complete with exits, and allow the user to navigate this. I narrowed down the corridors to 7 types, as I previously stated, but 5 of these are different when rotated. They need to be rotated when the user is not facing up, so we actually have a few steps to do while you navigate. This is because from a first person perspective forward is always ahead, but actually if you turn left say you will go from facing north (up) to facing west (left), for example.

I decided to use screen memory as a map, and use the graphics characters from the standard C64 character ROM as map information. This would mean I could double it as a map view, as well as storing it in a standard way that would be easy to edit using the [PETSCII](#) drawing tool I'd been using, [Petmate](#).

When in map mode this is actually what is displayed, but all colours are set to black initially, so you can't see anything on the black background (however there is a bug, you can just see them flash up briefly!) So we just colour in your current map position, and what I call the "look ahead" position is a less vivid grey, if you can indeed look ahead.

In considering this I came to realise that the corridor / first person view has very little distinguishing features to help players navigate it. Having the ceiling piping would help, but even still, it could be a

little confusing. So I decided to make it so that you will continue to see the map area of the locations you have already been as you traverse the maze. You'll see these in the map view mode.

Technically I make sure to save this colouring to an off screen buffer when the user switches back to first person mode, and restore when they switch again to map mode. In future I would like to improve this as it's currently not using memory optimally, but it's okay for now.

Another small component to this feature is displaying the facing direction in text in the middle of the screen. This is not perhaps the best way to do it but for now I'm okay with it. I'd like to redesign this and perhaps use some sprites (i.e. small high detail images) to indicate the direction in future.

## **Overview 4: other interesting bits**

1. Loading screen
2. Code commenting
3. General notes on code

### **Loading screen**

I decided to add a little loading screen, which is kind of hack on the default loader which is added by default by [ACME](#), the C64 assembly code compiler I used. You can't actually customise it out of the box to do the classic star field or random coloured line segments when loading, which is so iconic on the C64, but I did realise that all it's

really doing is copying data to memory, and since the screen is also in memory, I can just get it to copy some character data there! So what we get on start up is this (though it does not seem to show in the online emulator for some reason)

This memory is overwritten after first time start up so you'll only see it once.

## **Code commenting**

I spent a lot of “extra” time on the project so far just making sure there were excellent comments. This was not only for you, dear code reader, but also for myself, as writing in assembly is damn hard! And I didn't want to forget why I'd done something. I usually code in verbose programming languages with advanced high level features, and in this dive into low level programming I brought some of those habits of verbosity with me.

For example, consider the follow code screenshot, an entire assembly routine to write the colour value of a traversed map location to the off screen map so that when you move through there in first person mode you'll see your path when you switch to map mode.

As you can see I have added a signature for the routine in comments before it, with instructions of how to use it, what to expect and also noting side effects and “return” values (the areas of memory or CPU registers that are set as a result).

I found this to be extremely important and I hope that those of you who can read it will make good use of it.

## General notes on code

Some other notes on the code, in brief. Obviously most people will not be interested in this. This is vaguely in order as they appear in the source file `main.asm`

- The memory map, i.e. what areas of memory are used, appears at the top of the file for reference.
- Some handy VIC, CIA and SID chip addresses are stored as constants, for better readability.
- Any one byte address which is prefaced with a “Z\_”, for example `Z_SCR_X`, is a [zero page address](#). These addresses are used for [indirect addressing](#), player data and other temporary data.
- The main code is the first code, starting at address `$1000` and is the high level structure of the game. I used routines where possible to keep this structure compact and readable.
- There are several sections of assembly routines then, organised by feature: game logic, map drawing, first person drawing, choice, sound, special effects and info, and general.
- This is followed by game data, which includes tables, string data, screen copy (loading and map screens), character set

copy (for first person mode drawing), draw instructions( also for first person mode), sprite data, screen buffer definition (largely for completeness), and finally image data.

Right now it's a huge file of over six thousand lines, but this should also be broken up and linked at a later stage.

## What's coming in v0.2

*BDNRSNTCH v0.1* is almost complete, in terms of what we see from Stefan's demo. There are a few things missing though, which I would like to complete before moving on to fill out the game [based on his pen and paper designs](#):

1. The yellow "agent" as an encounterable character, with sprite and choice screen
2. Title screen high resolution image, with square by square loading
3. Glitches! There has to be some beautiful glitches
4. Joystick / joypad control
5. Improved sound effects

In fact I have created these as [feature tickets](#) on the [GitHub project](#), so you can keep an eye on things there.

**And after that?**



As I've been saying, I want to not only clone Stefan's work so far (as we see it on screen) but also finish the actual game, based on his designs and my interpretation of them. First I'll do the above, and then it's time to plumb the depths for new material!