Lockdown Dizzy: developing Fast Food on the Nintendo Switch

The Oliver Twins explain how they made their first game in 25 years – and how you can use FUZE to make your own

AUTHOR
PHILIP AND ANDREW OLIVER

The Oliver Twins have been making games since the early eighties, and can now be found at their new consultancy firm, Game Dragons. **gamedragons.com**

Back in November 1988, Andrew and I had the crazy idea of writing a game in one weekend and getting it released in time for Christmas. By Monday morning, after very little sleep, we had 30 playable mazes. Within a few more days we had a game ready to ship on Amstrad and Spectrum: *Fast Food*, starring our egg hero, Dizzy. Sadly, there were some delays in publishing and it was finally released in April 1989 – but still, it went on to be a huge success, with ports to the Commodore 64, Atari ST, Amiga, and PC. It sold over 150,000 copies – not bad for a simple maze game created in less than a week.

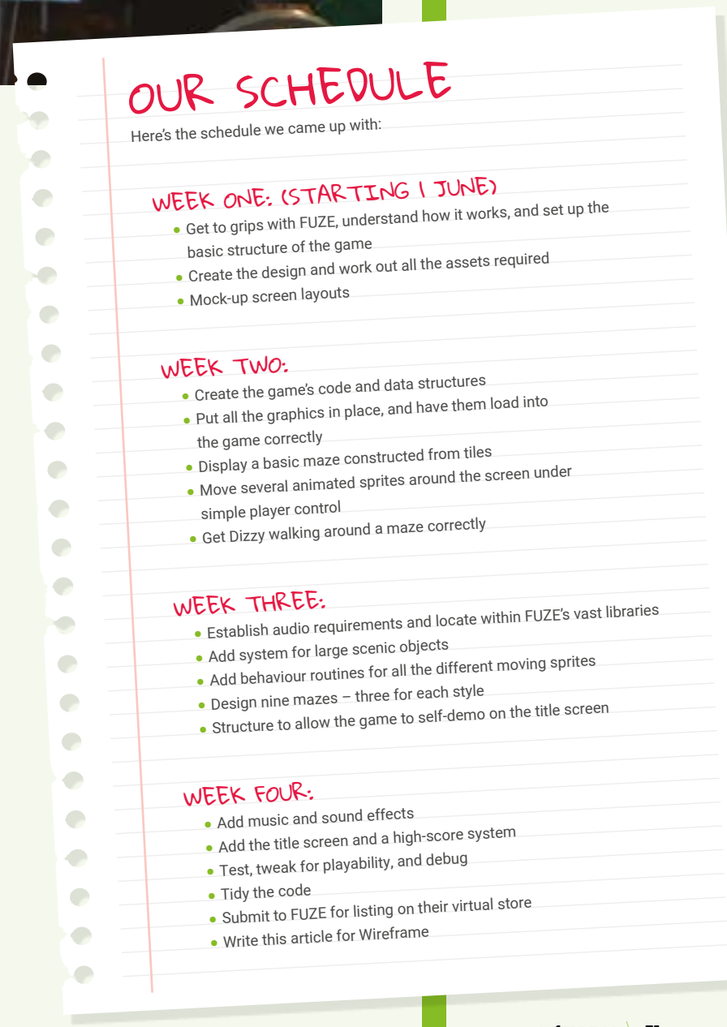
Dizzy's back in a shinier, HD form on FUZE.

Over the years, we've been asked to write new Dizzy games, but we're not up to speed with current development languages and toolsets. We just haven't had the time. When the Nintendo Switch came along there were, yet again, lots of requests for Dizzy games on Nintendo's neat new hybrid console. Andrew and I love the Switch and the idea, but making new, state-of-the-art games is expensive and high-risk. But recently, as consultants at our company, Game Dragons, we met the team at FUZE Technologies who have a passion to teach people how to code. They always do this through the challenge of making games – something we fully endorse.

The programming taught in schools tends to be task-based and uninspiring. Young people love games. If you want to teach them programming, let them make games – it's how our generation learned to program. FUZE Technologies have developed the FUZE programming language which is based on BASIC, which is designed to be easy to learn. This makes it the perfect stepping stone between Scratch and more complex languages like Lua, Python, Java, C++, and C#.

Last year, FUZE released a version of the platform on the Nintendo Switch, allowing the console's users to become amateur developers. We were really impressed by this, and made the throwaway comment that we could probably write Dizzy games in FUZE and how it would be cool to have them run on a Switch. The team at FUZE jumped at this and challenged us to do it. We said that *if* we were to do this, we'd want to remake *Fast Food*, as it was an easy game to make and also fun to play. The next thing we knew, they'd produced a mock-up screenshot – always the best starting point for any game. We gave some feedback and a list of the other graphics required, and over the weeks that followed, they produced these too.

FUZE programming being taught in a school club.



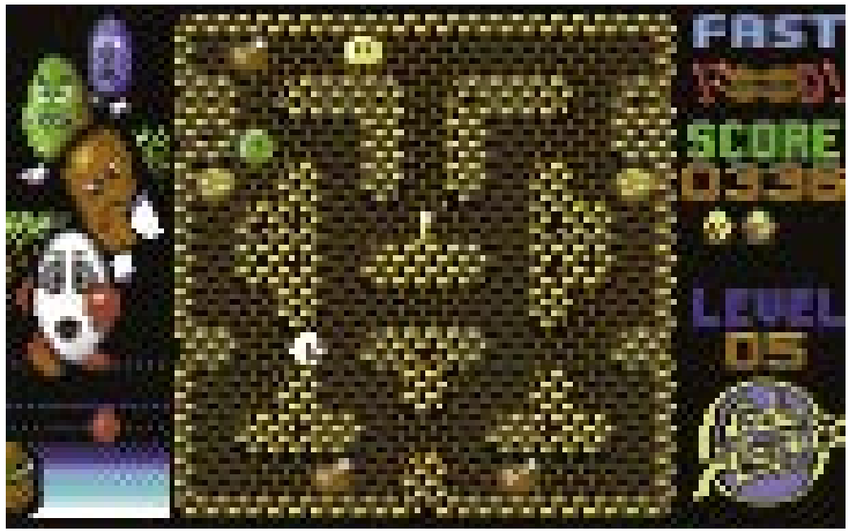
> **"If you want to teach young people programming, let them make games"**

We hoped that someone else would write the game, but then came coronavirus and lockdown, and Andrew and I suddenly found we had more time on our hands. We therefore agreed to write *Fast Food Dizzy* on Switch within FUZE, to be given away free with their FUZE Player.

START WITH A PLAN

With any project, you need to have a plan. The plan may change along the way, but it's always better to have a plan than nothing at all. We wanted to create a fun game that was easy for players to open the code, see how it works, then modify it to make their own maze games. This meant the code had to be elegant, flexible, and easy to read and modify. Even if players only modify a few mazes and tinker with a bit of the code, they'll have a greater depth of understanding of how games are made. ➔

Fast Food appeared on a number of systems, including ZX Spectrum, Amstrad CPC, and Amiga, pictured here.



The original game was written in less than a week, but that's because we were 'in the zone' both in terms of knowing the programming language and also having all the tools in place. Added to that, we were working insane hours without any distractions. This time, while we were in lockdown, there were still many distractions, plus we were working remotely, and working in a language we hadn't used before. We figured we should be able to do this in a month, between other work. (You can see our full schedule in the box on the previous page.)

Due to feature creep, things didn't exactly follow the plan – but then again, neither did the government's plan with lockdown...

CONCEPT

The original *Fast Food* was initially based on 8-bit technology, occupying 32kB of RAM and with a screen resolution of 160 × 200 pixels on the Amstrad CPC 464. The Atari ST and Amiga versions took up 512kB RAM with a resolution of 320 × 200 pixels.

Technology and player expectations have moved on a lot over the last 30 years, and while we could give the game a retro feel by using pixel graphics, we felt players expect more from a Switch. The Switch screen resolution is 1280 × 720 in handheld mode, increasing to 1920 × 1080 when docked. Artist Jonathan Temples created the original concept screenshot that helped us buy into the project, and later produced a bunch of other assets at our request.

You can read more about FUZE on the Nintendo Switch in our feature in Wireframe #24.

The Oliver Twins researching for the original *Fast Food* game in 1988.

Fast Food is a maze game inspired by the original *Pac-Man*. What's different is that the food doesn't want to be eaten, and players, controlling Dizzy, have to chase it around the mazes. The mazes are also occupied by guards that must be avoided, and there are power-ups to keep things varied. It all sounds pretty easy, and a good example game for FUZE.

"The project started as a simple maze game – but how impressive could we make it?"

While developing the game, we inevitably thought of more interesting features. If there had been a pressing deadline, they may have been shelved, but the lockdown seemed to keep getting extended, so we had more time. The project started as a simple maze game – but how impressive could we make it? We both agreed that we were keeping it restricted to 2D, even though FUZE has extensive 3D capabilities.

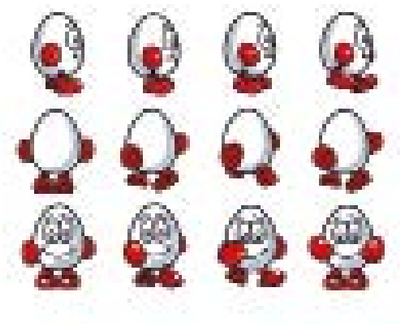
Our first big decision was, what size should the mazes be? The original versions were all single screens, but that's quite limiting and FUZE could easily do scrolling, so within the first week, we decided to change from fixed-screen mazes to scrolling, giving far larger maps and more depth. Then we wondered about the resolution and size of the characters on the screen. The graphics had all been drawn as 64 × 64 pixel textures, yet we were displaying them at 16 × 16 pixels, to give a good playable area of visibility.

This felt a bit of a waste since they looked great close up. So Andrew started experimenting with zooming in and out. It looked fantastic to have a dynamic camera and showed the best of both worlds – it just created a fair bit more work.

As well as improving the graphics and sounds, we felt the design could be enhanced, too. Dizzy collects stars in his adventure games, so we thought it would be a good idea to include these in *Fast Food*. While it's possible to complete a maze without collecting all the stars, players are encouraged to collect them all and receive a special bonus if they succeed.

Meanwhile, Jonathan created a bunch of hats for Dizzy, which the original *Fast Food* didn't have. While they looked great, we were wondering what to do with them. In the original game, Dizzy could collect a shield which he could use to turn the tables on the guards, much like the power-pills in *Pac-Man*. But while the guards fled from Dizzy and started to flash, nothing happened to Dizzy to show he possessed this special power, so we thought Dizzy wearing a hat would show he was now in predator mode.

This led to quite a lot more coding and work figuring out offset tables to ensure that, as Dizzy ran along in any direction, the hat moved correctly with his animation. This meant we could put hats in the mazes to be collected and we would need to find another use for the shield or lose it. Never wanting to waste good graphics, we turned it into an invisibility shield. Now when Dizzy collected the shield he'd become translucent, and the guards wouldn't be able to see or catch him.



Dizzy's walk cycle comprises four frames of animation for each of the four cardinal directions.

DESIGNING THE TITLE SCREEN

A classic trick of eighties games was to use the game in auto-play to create an 'attract mode', so this became the title screen. It was the cheapest approach both in terms of memory and development time and worked well. In our original games, we had a routine we used that recorded us playing and we would play that through the input routine and the game would then just appear to play itself. We were tempted to write this but instead allowed it to just use the logic routines that were there already for the guards, just changing the target for the closest food. The title screen was further improved on by overlaying the title of the game and other important title screen information like inviting the player to press start and further instructions. We felt this approach would work well for us since we wanted to keep the code focused to the game, rather than the presentation.

DIZZY AGAIN

Says artist Jonathan Temples: "When I was working on the Commodore 64 in the late eighties and early nineties, creating games for Codemasters, I was offered my first Dizzy creative experience with the C64's *Panic! Dizzy* loading screen. Soon after I was asked to create all the graphics for the new different-styled *Bubble Dizzy*. Just like this Switch game, *Fast Food Dizzy*, *Bubble Dizzy* was a departure from the original adventure exploring Dizzy games and was an underwater puzzle jumping platform game. I never thought I'd get the chance to do another retro 8-bit-type game again, never mind Dizzy. But once I was contacted by Jon Silvera, managing director at FUZE, I knew it was a brilliant opportunity. Plus if you look behind me (below) – that's the original Commodore 64C that I created the graphics for *Panic! Dizzy* and *Bubble Dizzy* on, way back in the early nineties."

Jonathan Temples working on *Fast Food*'s graphics in July 2020.

DEVELOPMENT TOOLS & ENVIRONMENT

Normally when developing games, you need a PC or Mac, the target hardware, a compiler, an art package, and various other software tools. Since we were developing this entirely on the Switch within FUZE, however, we had everything we needed in the package. To make things easier to use, we purchased two external USB 3.0 keyboards from Amazon at £20 each and hooked our Switches to monitors.

Andrew took care of the coding but had never used FUZE before. As mentioned, the FUZE language was inspired by BASIC, and anyone can get things running on the FUZE with a few lines of code. The typical BASIC cliché we grew up on was:

```
10 Print "Hello World"
20 Goto 10
```

New fast food and power-up assets, courtesy of the great Jonathan Temples.

This wouldn't work in FUZE because it's considered poor coding practice. **Goto** isn't a feature of any other programming language other than BASIC, and line numbers aren't referenced in code; instead, FUZE uses function names. So FUZE looks more like the programming language, C:

```
Loop
Print("Hello World")
Repeat
```

You can quickly progress to add more functionality, and it's really easy. Especially given that the context-sensitive help is built-in and you can find out more about each function.

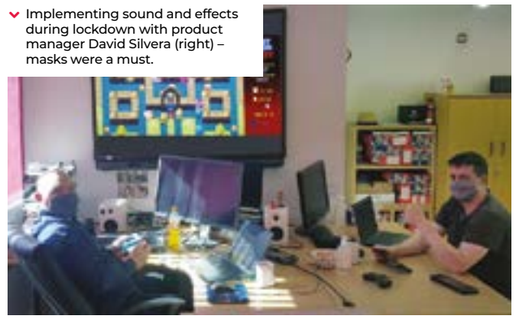
The **PrintAt** command allows you to define the position and colour of the text to print. It's based on 0,0 being the top left corner of the screen and works on character spacing which is based on the current font and size, which is also definable. So this code prints 'Hello World' in red 10 times, from the top left corner down:

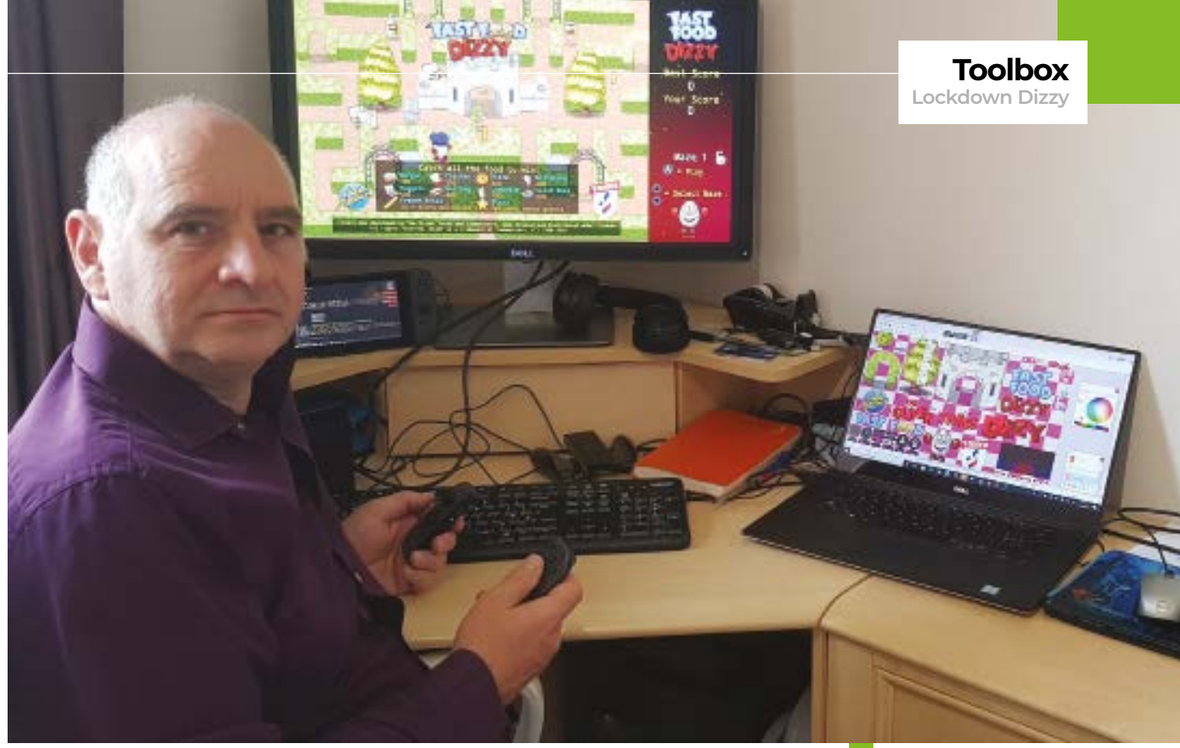
```
For a=0 to 10 Loop
PrintAt(10,10+a,Red,"Hello World")
Repeat
```

Again, it feels like a C-based language, but without much of the fuzzy syntax, and doesn't require defining all the variables upfront. While FUZE is able to handle all this background automation, it also allows you to define it, so it becomes very simple to code as you start to explore further. You can also define your own structures and functions.

Andrew was able to set up the moving entities, called Actors, into structures quickly and easily:

Implementing sound and effects during lockdown with product manager David Silvera (right) – masks were a must.





Andrew Oliver coding in his lockdown study.

```
Struct _Actors[
                .x
                .y
                .speed
                .anim
                .state
Sprite   .image
String   .name
]
_Actors  Actor[16]    //gives me 16 actor
structures
Actor[0].x+=Actor[0].speed  //add the speed to
Actor[0]  x coord.
```

Note that you don't need to define types of variables in structures. They're all assumed to be floating-point numbers, but if you want to override this to an integer, array, or string, you can easily instruct it.

```
        Actor[0].name="Dizzy"
```

So now all the variables are packaged up nicely for each 'Actor' they could be passed to created functions:

```
        For a=0 to 16 loop
                Move(Actor[a])
Draw(Actor[a])
        Repeat
```

Within the **Move** and **Draw** functions, they can call on further functions for collision with the maze walls and the various other Actors. It was so simple, and part of the reason for the feature creep was because adding functionality was so easy and intuitive.

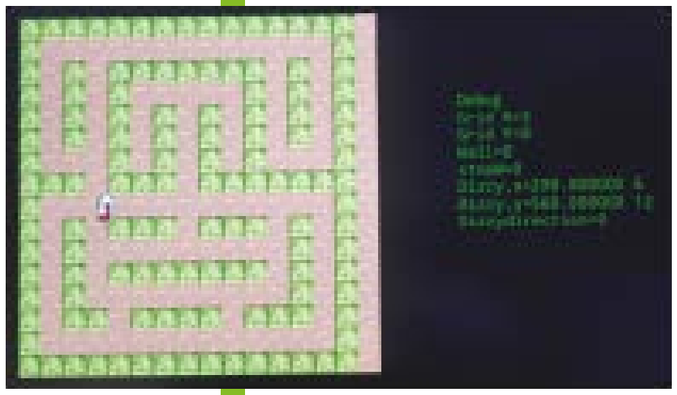
MAZES

When creating any 2D game, it's common to use tiles to construct the background, so the first thing to do was to work out all the different tiles required. Jonathan created three base tile sets: Garden, Temple, and Ice. The mazes were to be stored in a table of data – one entry per tile. A basic maze can be created with just two states per tile: solid (impassable) or not solid (passable). We obviously wanted much more tile variety, so we gave each space one byte-length integer. We ordered the tiles so that any tile under 128 would be a solid wall of some description, so it was an easy test for the code in terms of collision, while anything over 128 would be open space or used to set up the maze with the starting locations of moving entities (Actors).

To make the walls look more natural and fit together nicely, we needed around 24 tiles in each style to cater for corners, shadows, and so forth. Tiles 128 through 159 were a variety of floor tiles which would need to be printed before the walls and sprites. This would ensure they were not just the deepest depth but that walls and foreground sprites with transparencies would have floor showing through.

Beyond this, we assigned groups to all the other entities which needed locating in the mazes. ➔

A hat for every occasion – and all four directions.



Fast Food early in development. We have Dizzy in a maze!

160 - 175: Overlay sprites: Gates, trees, etc
175: Dizzy (his starting point).
176 - 185: Hats
186 - 189: Guards
190: Fries
191: Spinning Star
192 - 199: Fast Food
200 - 207: Power-Ups

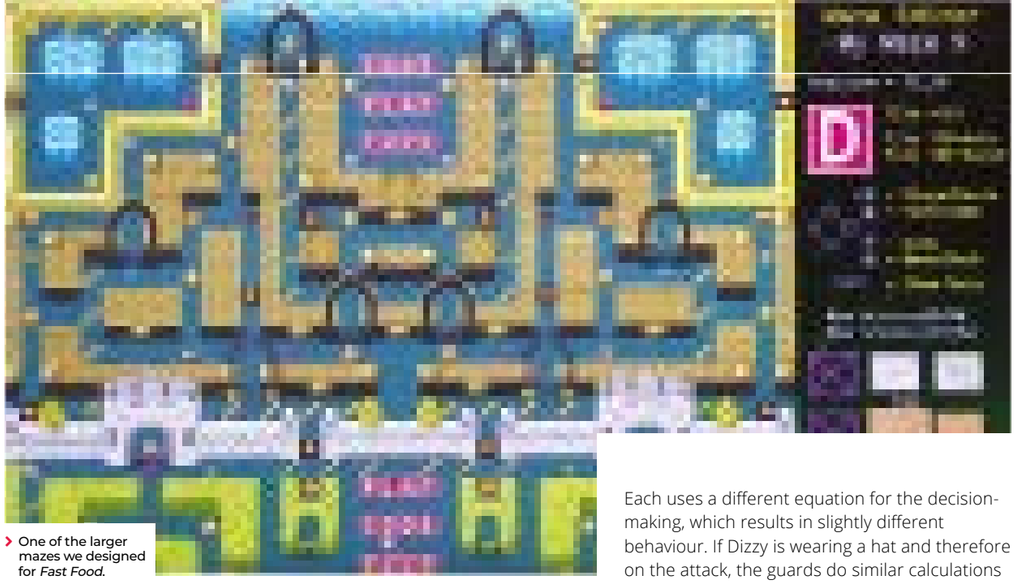
Tiles were drawn on 64 × 64 pixels textures with four-byte colour depth: three bytes for RGB plus one byte for a transparency value. These were drawn on PC, stored as PNGs, and imported into FUZE.

As a new maze starts, the code works through the maze data, copying it to a temporary RAM buffer and when it finds a Guard (Tile 186) it adds it to the Actors list. This has 64 slots to hold all the Actor details. Once the code has put the guard into the Actors list, it clears the maze data slot back to the floor (Tile 128). Stars and Fries are separate lists as they don't need to move or have as much functionality as the Actors. We also had space for 64 'Scenery' objects. There were around 15 large sprites: trees, arches, and even a castle. These are simply put down in the editor in a single 'floor' space as a representative icon and added to the scenery list, which is a list of sprites with coordinates that are correctly sorted by depth, so that Actors appear and move correctly in front and behind them.

The Switch is a powerful enough console, but FUZE is an interpreted language, so you still need to use smart techniques to ensure the game runs quickly. We put a limit on the maze size of 60 × 60 tiles. We limited both Actors and Scenery sprites to 64 slots, but allowed as many Stars or Fries as empty floor space would allow, and this allowed the game to still maintain a respectable 30 frames per second (fps) during play. Most mazes don't use anything like this, and the game mostly runs at 60 fps, which looks very smooth.

FRIES WITH THAT?

In the original *Fast Food*, there were only four items: burgers, chicken, pizzas, and milkshakes, so we decided to increase the menu selection and even added salad bowls to encourage healthy eating. Real fast food also usually comes with fries; these weren't in the original game, but we loved the 'chain mechanic' in classic games like *Bomb Jack*, where players are encouraged to collect chains of items for exponentially increasing scores. Because of this, fries were added and made to spin faster the more they were worth. Tasty.



One of the larger mazes we designed for *Fast Food*.

CREATING THE MAZE EDITOR

As with the original, there are a number of different maze graphic styles, including hedge mazes, brick walls, ice, and pipes. All these tiles and features had to be turned into fun, aesthetically pleasing mazes with the right amount of challenge. The original idea was to have arrays (data tables) in the code that describe the mazes. There would be a number for each tile type, then players could go into the code and change maps by overtyping the data. But we then decided to create ten large mazes – mostly because we had ten hat types and thought it would be good to have a different hat per maze.

"What started out as a passing comment became a reality due to the coronavirus"

So we needed a Map editor. The easier it was to design the mazes, the better they would be. It would be tough, then, to type these back into the code, and so we added an extra feature to save and load mazes, so that players could feel free to do so without ruining the built-in levels.

THE ACTORS

In *Fast Food*, the enemies – or guards – are programmed to behave in a similar way to the ghosts in *Pac-Man*. When Dizzy is vulnerable, they chase him by looking at his coordinate and direction and then heading towards him. Each uses a different equation for the decision-making, which results in slightly different behaviour. If Dizzy is wearing a hat and therefore on the attack, the guards do similar calculations but for the opposite direction. All movement uses floating-point values, and '1' was chosen as the default speed for Dizzy. All other actors were then given speed values based on this and would all move at different speeds.

SOUND AND MUSIC

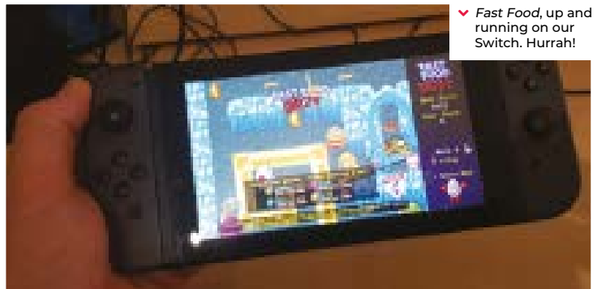
FUZE readily plays MP4s, and already contained a set of original *Dizzy* soundtracks, so it was easy to assign these to the title screen and individual mazes. FUZE also has a library of stock sound effects to use, and Andrew populated the game with these. It became apparent pretty quickly, however, that these sounds were all from different sources in different styles and didn't really do the game justice. Many of the sound effects were created by FUZE's David Silvera, and he agreed to create a set of more bespoke sounds for Dizzy. When the game was complete, we met at the FUZE offices and implemented the new sound effects together.

THE FINAL STAGES

Most games go through Alpha and Beta mastering phases. Our console games always had around a month for each, maybe more, but with a game of this scope, we fitted all this into around ten days. During development, we made mock mazes to test new features. Within the last two weeks, it became virtually feature complete, so then the final mazes had to be created in around a space of a week. I created these with the help of my daughter to type in large amounts of data. I reported bugs to Andrew while he was also finding his own bugs to fix. Andrew enjoyed receiving and playing the new mazes as the layouts were a complete surprise to him.

Once the sound effects were implemented, Andrew and Dave played the game from start to finish by way of a final QA test and signed off the master on 20 August. The game ended up at around 3000 lines of code, taking around 80 days. Whilst that was far more time than we had originally envisaged, the game was far more than we had planned and we were proud of what we had made. This is the first time we've made a game since the mid-nineties!

What started out nearly a year ago as a passing comment became a reality due to the coronavirus. What started as a quick four-week port of an old *Dizzy* maze game became almost two and a half months of development. But if you're going to do something, it's worth doing it well, and all this hopefully serves as a good example of what can be created in FUZE with a bit of time and effort. We hope *Fast Food Dizzy* will attract more people to learn to code – and perhaps some will go on to become professional game developers in the future.



Fast Food, up and running on our Switch. Hurrah!

Philip Oliver busily making mazes in FUZE.