Squeezing the NES

How the Oliver Twins managed to fit Super Robin Hood onto a 64kB NES cartridge

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**

Download the code from GitHub: **wfmag.cc/wfmag34**

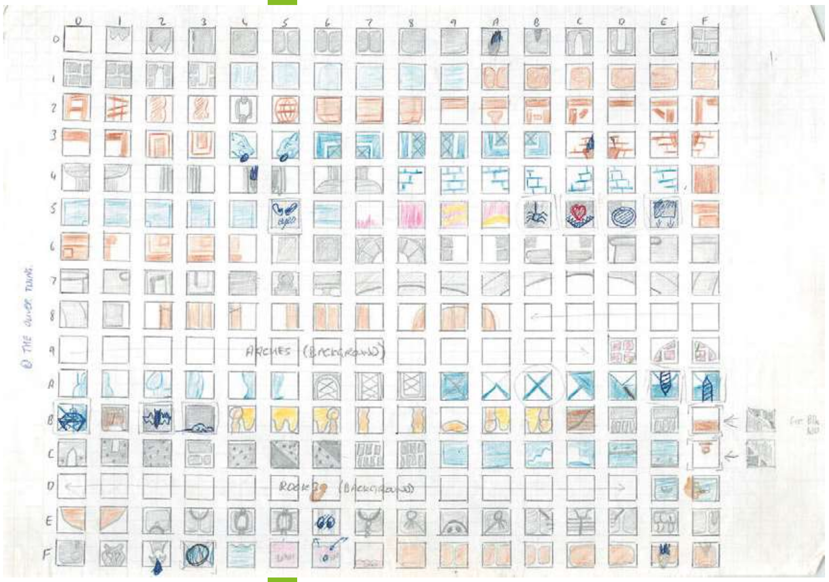
We came late to the development of NES games, having decided to write *Fantastic Dizzy* in early 1990. We stayed longer on 8-bit computers than other developers, due to the success we were having with the *Simulator* and *Dizzy* series. We had a slick pipeline and tools, and could design and produce games quickly across the Spectrum and Amstrad, which then got ported to C64, Atari ST, and Amiga.

We'd resisted change because it meant we'd have to learn new computer architectures, 68000 assembler, then recode all our engine, tools, and get hit with large bills for all the artwork and music. It would have been a massive investment of time and money, and we could see that many developers were struggling to make money on the ST and Amiga due to the high costs of development and the ease with which players were able to pirate the finished games.

When we visited the Consumer Electronics Show in January 1990, we saw the success Nintendo were having, that they had eliminated piracy and were selling games at high prices and in vast numbers. We looked at these games and were confident we could do something of similar quality – and in the same amount of development time – to our current titles: that is, about four to six weeks each. The NES was based on the 8-bit 6502 chip, which we already knew from the Commodore 64, and we knew we'd still be able to do a lot of the graphics ourselves if we needed to.

Our first game for the NES ended up being *Fantastic Dizzy*, released in 1991, which was a mash-up of ideas from our first three *Dizzy* titles in one large game, with a few other subgames added for maximum value. It took about nine months to write, and used an expensive 128kB cartridge. Codemasters had signed a distribution deal with Camerica in Canada to sell NES titles in the US and Canada; they needed a catalogue of games, so we needed to start producing them faster and cheaper. A lot of our time was spent on learning how the console worked and producing the toolchain, standard library routines, and the overall development environment.

The Oliver Twins' sketches for *Super Robin Hood's* background graphics.



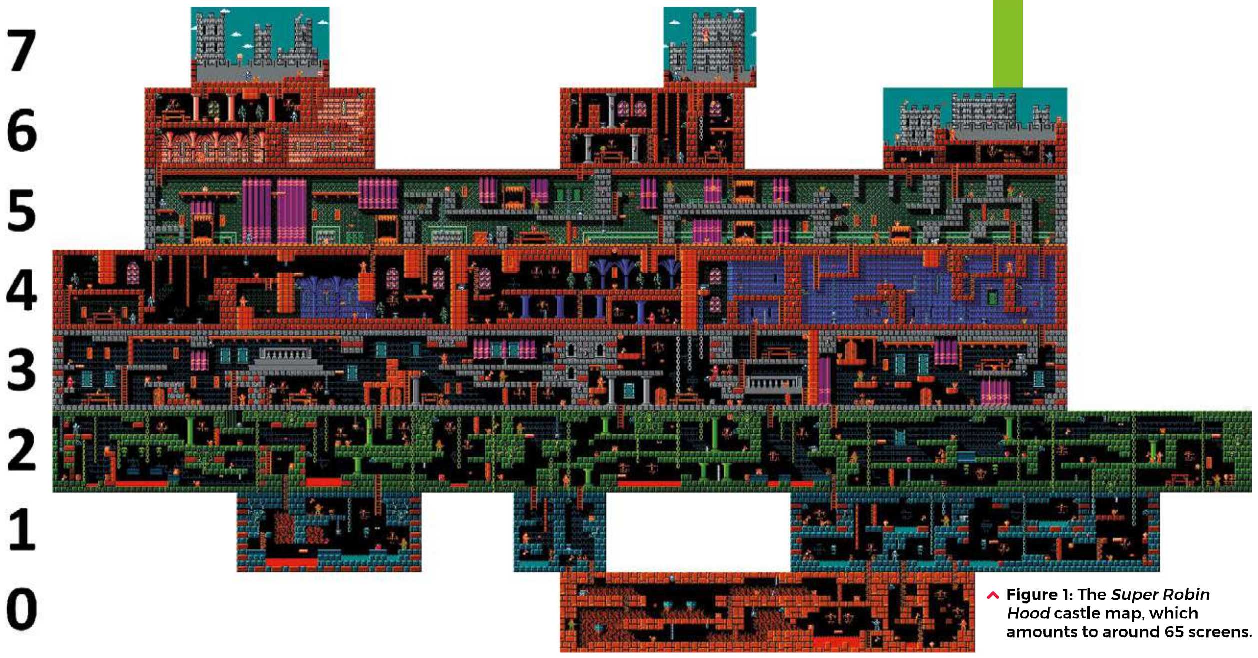


Figure 1: The *Super Robin Hood* castle map, which amounts to around 65 screens.

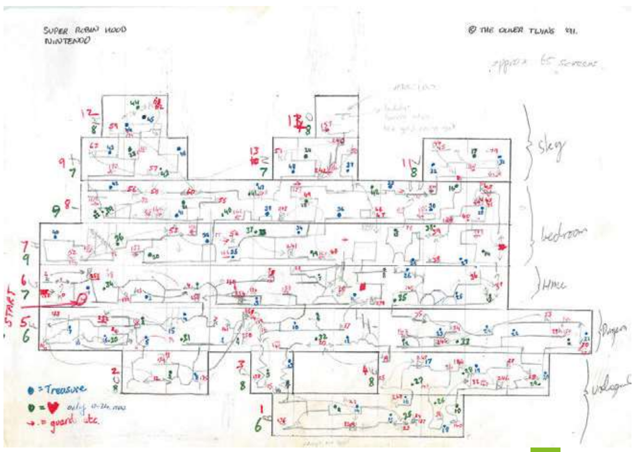
Our development environment consisted of two 8086 PCs, both fitted with floppy drives and 20MB hard drives. These had PDS (Programmer Development System) cards installed, cabled to Codemasters NES Development Boards, that in turn were cabled to retail NES consoles. Each PC had a text editor and 6502 compiler. We'd back up and transfer data between the PCs via 5¼ inch floppy disks. The PCs had

"Our development environment consisted of two 8086 PCs"

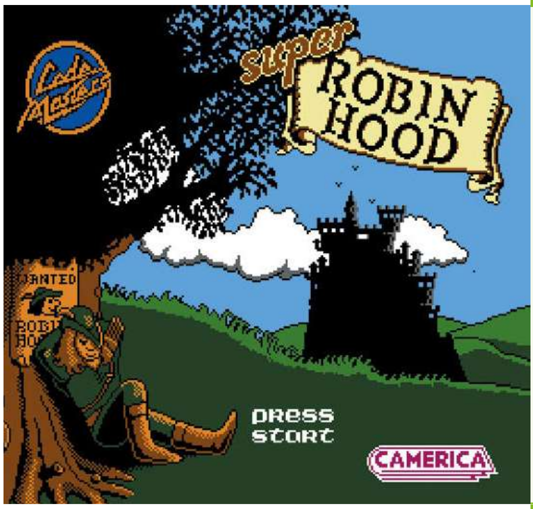
monochrome monitors, while the NES consoles were connected to colour TVs – one was PAL and the other NTSC, to ensure compatibility with televisions in other parts of the world. In addition, we had an Amiga 500 with a colour monitor running Deluxe Paint III, a package we used for creating all the graphics. The whole set up cost almost £10,000. In today's money, that's around £20,000 – and as self-employed developers, we had to pay this up front and recoup the cost via royalties, so it was a huge investment.

For our second NES game, we decided to revisit the first title we made for Codemasters, *Super Robin Hood*, first released for the Amstrad CPC in 1985. The game had a fundamentally good concept and didn't need a huge story with lots of scripts and puzzles; it could make

➡



An early sketch of the map, with the player's path and items carefully plotted out.



THE TITLE SCREEN

Once the overall game was complete, we saw how much memory was left and allocated this to the title screen. This picture only used about 400 unique characters, which took around 8kB memory, with the character map being 32 wide by 30 high, plus the colour palette colour information. We also used a trick so that halfway down the screen an interrupt swapped the Sprite Character set with the Background Character set.

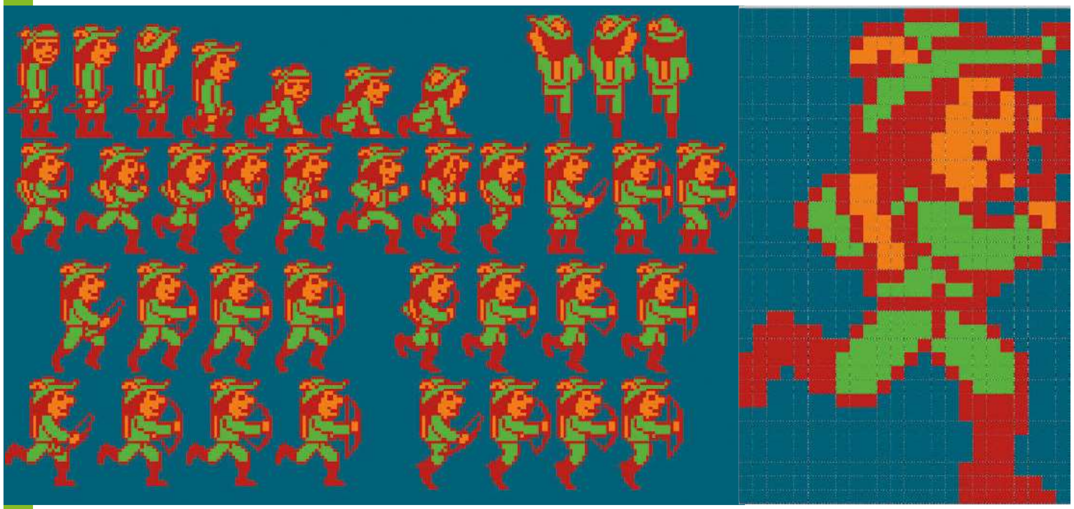


Figure 2: A few of Robin Hood's sprite animations. These were flipped when Robin Hood moved left.

INSIDE THE NES

First hitting Japan in 1983, the Nintendo Entertainment System was Nintendo's debut games console. Inside the grey box, there was a MOS 6502 CPU, similar to the Commodore 64. It contained just 2kB of onboard RAM; game cartridges could increase this, but doing so made them more expensive to produce. Cartridge sizes ranged from 8kB to 1MB, but 128kB to 384kB were the most common. There was an additional 2kB video RAM, 256 bytes of 'object attribute memory' to store the positions, colours, and tile indices of up to 64 8×8 pixel sprites on the screen, and 28 bytes to allow selection of background and sprite colours. The standard display resolution was 256 horizontal pixels by 240 vertical pixels, and could display up to 24 colours at once from a palette of 52. The NES could display 32 × 30 background characters with up to 64 overlaid sprites. The graphics for the characters were held in the game cartridge, either fixed in ROM or reprogrammable in more expensive RAM.

good use of a set of solid routines and game mechanics developed for *Dizzy*, and we'd be able to produce it much faster and for a cheaper 64kB cartridge. We also relished the idea of making scrolling levels, as they'd feel so much nicer than the original, flick-screen game.

THE GAME

Super Robin Hood was a typical platformer, but its English folklore theme meant we had a built-in plot (Robin rescuing Maid Marion) and a great projectile weapon: a bow and arrow was much better for an 8-bit game than a gun, since the arrows can move slowly and are more visible as a result. When the enemies use these, it also makes sense in terms of gameplay, since there's time for the player to dodge the arrows. (Bows and arrows don't tend to attract the negative political baggage that guns tend to come with, either.)

The game was set in a medieval castle filled with guards and other obstacles to navigate. The adventure element was made more interesting by the inclusion of keys to open new routes, ensuring it wasn't a linear game, and took some memory and navigation to work out the best routes. The castle itself was a side-on maze, with each floor distinguished by its own theme: the lowest floor consisted of chain-lined dungeons and rocky walls, and as you rose up through the floors, you passed through feast halls, living rooms, bedchambers, and ultimately the ramparted roofline and towers, where Maid Marion awaited rescue.

TECHNICAL DESIGN

The game was written to fit on a 64kB ROM cartridge, which also had 8kB RAM for redefining the background character and sprite graphics. To put that in perspective, *Super Mario Bros.* was a 40kB cartridge: 32kB for the game and data, with 8kB ROM for the background and sprite character sets.

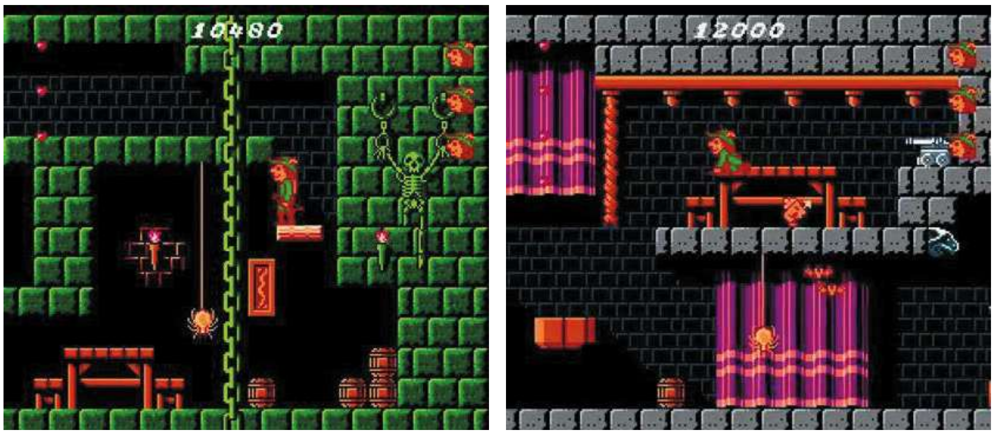
The NES's architecture forces some great memory-saving restrictions. First, it allows up to 256 four colour, 8×8 pixel characters for the backgrounds, and 256 characters for sprites. Each character set adds up to just 4kB in total, which is really efficient on memory – by contrast, a single iPhone App Icon amounts to about 43kB (120×120 pixels in 24-bit colour).

When printing a background character, the NES also uses different predefined palettes – 48 in total. We typically chose black for the background, and then three shades of a colour for each of the palettes. The background was character mapped (32 × 30 characters) and also had the ability to scroll – which kind of made us wish we'd had a C64 back in the day, as this is such a powerful feature.

CASTLE BACKGROUNDS

When the player started the game, the background level graphics needed to be transferred into the background character set. We could load in different graphic sets from the main game ROM, which was great for changing background level graphics between floors. We decided on eight floors in the castle, but only three unique background character sets, as we had a couple of floors that used similar environment styles and used colour palette changes to make them look more varied:

Floors 0, 1, and 2 – Dungeons – all used the same character set.



Super Robin Hood saved memory by reusing some background sprites with different colour palettes, like the stone blocks you can see here.

Floors 3, 4, and 5 – Halls and Bedrooms – share the same character set.
Floors 6, 7, and 8 – Ramparts, End Screen, Font – all in the third character set.

With each background character set taking 4kB, this amounted to 12kB for all background graphics. The rippling lava in the deepest parts of the dungeons was achieved by redefining the lava character between a set of different characters, each with ripples in different positions, rather than changing the characters on the screen.

The background maps for all the levels took a large chunk of memory by themselves. The full castle map took up around 65 screens (see **Figure 1**). With each screen being 32 characters wide and 30 characters high, this translated into 62kB of memory if it wasn't optimised or compressed in some way. We started by mapping everything with 256 unique 2×2 character blocks, reducing this to almost a quarter. In fact, the map only had to be 14 blocks high, as TVs used to lose a little around the edges and you could set edge characters to black, creating a slight black border top and bottom. This meant the map data took 14.5kB of the ROM (65 screens × 16 vertical strips × 14 blocks per strip).

FOREGROUND GRAPHICS

Hardware sprites were similar to background characters, except colour 0 in each palette was always transparent, and you could display up to 64 sprites anywhere on the screen simultaneously. These would display on top of the background without taking up any processing power – this was a big step up from pixel-mapped computers like the BBC, Spectrum, and Amstrad, where displaying sprites took most of the available processor time. The sprite graphics were largely taken up by Robin Hood's animations – all the guards and other enemies were static, so they only took up 4kB of memory.

The NES's hardware did come with one minor restriction: it couldn't display more than eight sprites on a single horizontal raster line. We did our best to design around this restriction by alternating the order we displayed the sprites – this meant that, when more than eight sprites needed to be displayed on a line, you'd see some flickering of the first and last sprites. In *Super Robin Hood*, this led to some arrows flickering as they approached Robin. We did our best to position guards and other enemies on different levels to reduce this. Vertical adversaries like spiders and chains with spiked balls used lots of sprites; these were in vertical lines and so didn't cause such issues, but it was challenging coming up with lots of vertically based threats that made for good gameplay.

"The background maps took up a large chunk of memory"

The Robin Hood animation should have used around 12 sprites (typically 3 characters wide by 4 deep) per frame, and there were 74 frames of animation, which without optimisation would have required a whopping total of 888 characters. Obviously, we didn't use this amount; to reduce it, we developed a special tool that helped locate duplicate characters, allowing us to move the sprites to create blanks, and look for opportunities to mirror existing characters – for when Robin Hood was moving left and

TECH NOIR

Fun fact: The Apple II, which used the Motorola 6502 processor, was featured in James Cameron's 1984 film, *The Terminator*. It clearly shows that Arnold Schwarzenegger's T-101 ran on 6502 code, taken from the Apple II manual.

Due to licensing disputes with Nintendo, *Super Robin Hood* didn't emerge until 1993, on the *Quattro Adventure* four-in-one cartridge.

Super Robin Hood's sprite set, which largely comprised Robin Hood's animations. The rest was given over to assorted enemies, barrels, and other objects.

right, say. This was another of the NES's great hardware features. By doing this, we were able to create all 74 frames of animation from just 120 unique sprite characters (see **Figure 2**).

AUDIO

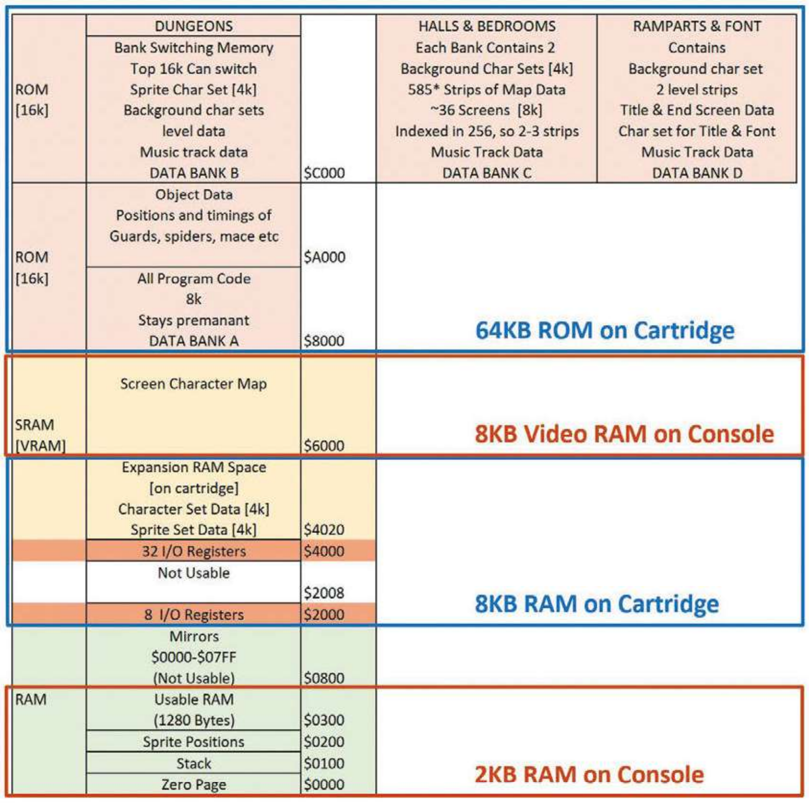
The music and sound effects were developed by Allister Brimble to meet the brief we delivered. We weren't musicians, and always outsourced all our audio. The music driver was written by Gavin Raeburn, and was the driver we used on all our Codemasters NES games. There were ten soundtracks and around 20 sound effects. Altogether, the data and code took only 4kB.

BANK SWITCHING

The ROM size of *Super Robin Hood* was 64kB, which seems like the obvious convenient 'addressable' size of memory given that two 8-bit registers (16-bit addressing) can index memory up to 64kB.

The console reserves the first 32kB for its local RAM for variables, character data, and screen memory, but, this is *not* 32kB of usable memory (see the memory map in **Figure 3**). The 64kB cartridge ROM is split into four banks of 16kB each; the first remains permanently mapped at $8000. Banks B, C, and D can switch, but are only accessible when resident at $C000.

Figure 3: The *Super Robin Hood* cartridge's memory map.



ROM [16k]	DUNGEONS Bank Switching Memory Top 16k Can switch Sprite Char Set [4k] Background char sets level data Music track data DATA BANK B	$C000	HALLS & BEDROOMS Each Bank Contains 2 Background Char Sets [4k] 585* Strips of Map Data ~36 Screens [8k] Indexed in 256, so 2-3 strips Music Track Data DATA BANK C	RAMPARTS & FONT Contains Background char set 2 level strips Title & End Screen Data Char set for Title & Font Music Track Data DATA BANK D
ROM [16k]	Object Data Positions and timings of Guards, spiders, mace etc	$A000		
	All Program Code 8k Stays premanant DATA BANK A	$8000	64KB ROM on Cartridge	
SRAM [VRAM]	Screen Character Map	$6000	8KB Video RAM on Console	
	Expansion RAM Space [on cartridge] Character Set Data [4k] Sprite Set Data [4k]	$4020		
	32 I/O Registers	$4000		
	Not Usable	$2008		
	8 I/O Registers	$2000	8KB RAM on Cartridge	
	Mirrors $0000-$07FF (Not Usable)	$0800		
RAM	Usable RAM (1280 Bytes)	$0300		
	Sprite Positions	$0200		
	Stack	$0100		
	Zero Page	$0000	2KB RAM on Console	

PROGRAMMING

The NES used the 8-bit 6502 chipset, which predates the Z80 chipset from the ZX Spectrum and Amstrad. It only has 56 instructions, and when you remove the useless ones like BCD (Binary Coded Decimal), it leaves less than 50 instructions and feels far more limited than the Z80. Fundamentally, you have three registers:

A (main Accumulator)
X & Y (for indexing and arithmetic).
S (stack pointer – note it's only 8-bit, which means you can't nest routines too deeply)
P (Processor status – a set of flags).
8 bits = 1 Byte = 0–255 number possible. In HEX, this is $00–$FF.

The processor makes good use of 'paging' its memory. That is, you have a total addressable memory up to 64kB using 16-bit addressing (two bytes combined – high and low). But if you use a high byte to address each 'page' (256 bytes) then you can index into the next 255 bytes, only using the 8-bit X or Y registers.

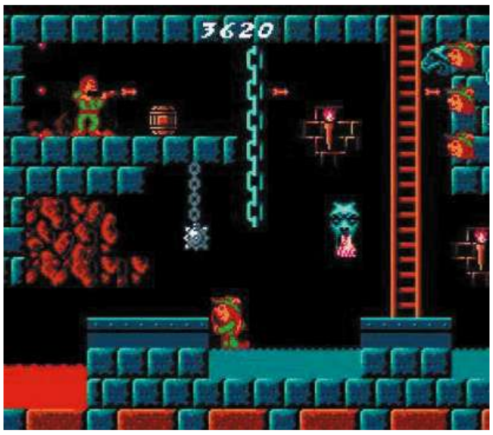
RESERVED PAGES

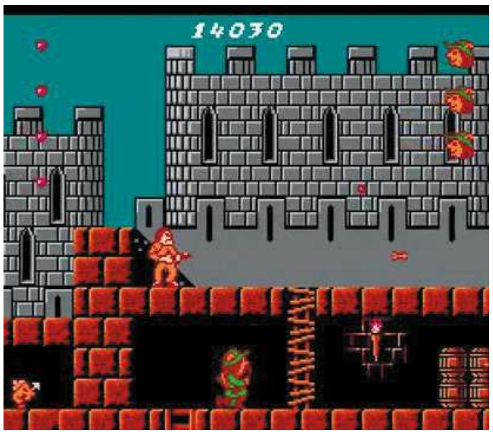
#00 'Zero Page': General variable workspace. Anything here only uses 1-byte indexing, which is shorter and faster. We put *all* variables for the entire game in these 256 bytes. Remember, since the game is stored on a cartridge, it's entirely ROM (Read Only Memory).

#01 'Stack': The 'S' register stores 2-byte addressed 'return locations' here, when it enters subroutines.

A simple piece of code will look like this:

```
Add7to16bitVariable      ; Routine Label
    CLC           ; CLear the Carry flag.
    LDA $23    ; LoaD Accumulator,   getting
the low byte of a variable in Zero Page
[usually given a name]
    ADC #$07  ; ADd a Constant 7, carry will
be set if result > 255  # Means Number, $
means HEX
    STA $23     ; STore Accumulator - saving
```





The score was placed high on the screen so as to avoid a clash with other sprites on the same horizontal line.

```
the low byte
    LDA $24     ; LoaD Accumulator - getting
the high byte
    ADC #$00   ; ADd with Carry  - adding zero
to add any carry that might have been set
above
    STA $24      ; STore Accumulator - saving
the high byte
```

You can also see a longer code snippet in **Figure 4**. With such basic instructions, each piece of code might look long-winded at first glance. Bear in mind though, that most lines of code only take up one or two bytes, so while it looks long, it takes up very little memory. If you're interested in learning more about 6502

"The NES used the 8-bit 6502 chipset, which predates the ZX Spectrum and Amstrad"

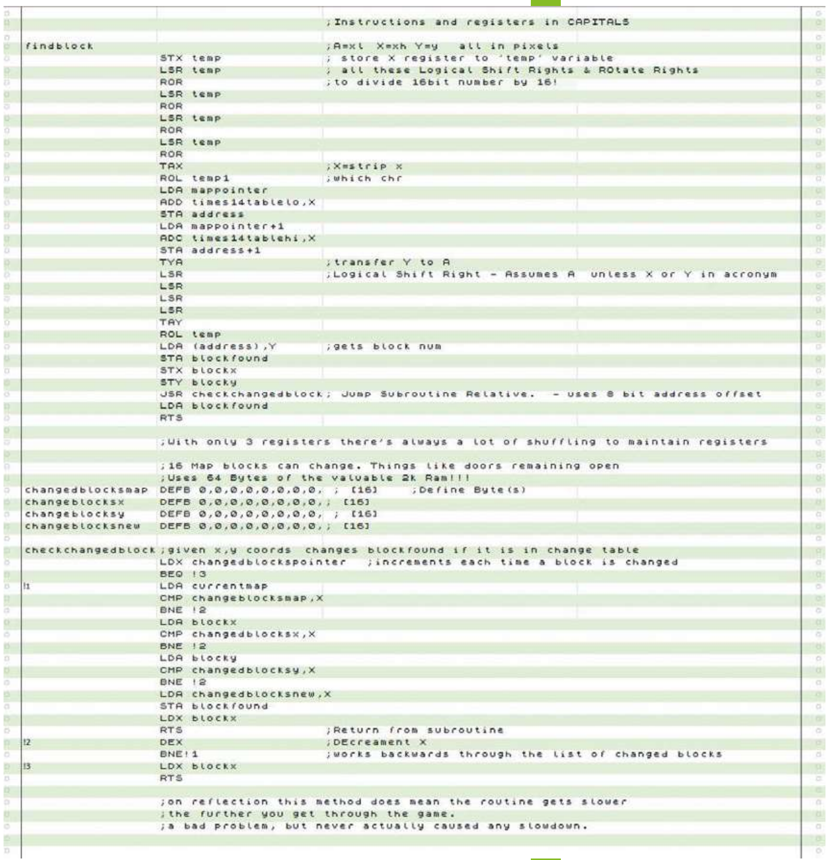
assembly, you can find more information at **wfmag.cc/6502**, and you can even take a look through *Super Robin Hood*'s complete source code at **wfmag.cc/wfmag34**.

RELEASE

Sadly, *Super Robin Hood* was released over a year late, and amid some thorny distribution issues – essentially, Codemasters didn't have official Nintendo approval for publishing games on the NES. Eventually, *Super Robin Hood* was sold as part of a collection of games called *Quattro Adventures*, released in 1993, and it didn't sell very well or earn us much money as a result. Still, we were proud of what we created, and hired some developers to convert the game to the Atari ST and Commodore Amiga, and even to the ZX Spectrum and Amstrad CPCs, where it was retitled *Robin Hood: Legend Quest*.

Writing code in 6502, on the NES in such small amounts of memory was a challenge, but also hugely satisfying. We're still proud of the final game to this day; it was a fine example of elegant design, code, and art all coming together beautifully to create a fun, slick adventure.

Figure 4: The entire code base of *Super Robin Hood* is 8kB. Instructions vary between 1–3 bytes. So, assuming an average of 2 bytes, this means there are around 4000 lines of code and it really does look like code, which is how the publisher got its name: Codemasters.



```
                                            ;Instructions and registers in CAPITALS

findblock                                   ;A=xl  X=xh Y=y   all in pixels
                  STX temp                  ; store X register to 'temp' variable
                  LSR temp                  ; all these Logical Shift Rights & ROtate Rights
                  ROR                       ;to divide 16bit number by 16!
                  LSR temp
                  ROR
                  LSR temp
                  ROR
                  LSR temp
                  ROR
                  TAX                       ;X=strip x
                  ROL temp1                 ;which chr
                  LDA mappointer
                  ADD times14tablelo,X
                  STA address
                  LDA mappointer+1
                  ADC times14tablehi,X
                  STA address+1
                  TYA                       ;transfer Y to A
                  LSR                       ;Logical Shift Right - Assumes A  unless X or Y in acronym
                  LSR
                  LSR
                  LSR
                  TAY
                  ROL temp
                  LDA (address),Y           ;gets block num
                  STA blockfound
                  STX blockx
                  STY blocky
                  JSR checkchangedblock; Jump Subroutine Relative.  - uses 8 bit address offset
                  LDA blockfound
                  RTS

                  ;With only 3 registers there's always a lot of shuffling to maintain registers

                  ;16 Map blocks can change. Things like doors remaining open
                  ;Uses 64 Bytes of the valuable 2k Ram!!!
changedblocksmap  DEFB 0,0,0,0,0,0,0,0, ; [16]     ;Define Byte(s)
changeblocksx     DEFB 0,0,0,0,0,0,0,0,; [16]
changeblocksy     DEFB 0,0,0,0,0,0,0,0, ; [16]
changeblocksnew   DEFB 0,0,0,0,0,0,0,0,; [16]

checkchangedblock ;given x,y coords  changes blockfound if it is in change table
                  LDX changedblockspointer   ;increments each time a block is changed
                  BEQ !3
!1                LDA currentmap
                  CMP changeblocksmap,X
                  BNE !2
                  LDA blockx
                  CMP changedblocksx,X
                  BNE !2
                  LDA blocky
                  CMP changedblocksy,X
                  BNE !2
                  LDA changedblocksnew,X
                  STA blockfound
                  LDX blockx
                  RTS                       ;Return from subroutine
!2                DEX                       ;DEcreament X
                  BNE!1                     ;works backwards through the list of changed blocks
!3                LDX blockx
                  RTS

                  ;on reflection this method does mean the routine gets slower
                  ;the further you get through the game.
                  ;a bad problem, but never actually caused any slowdown.
```