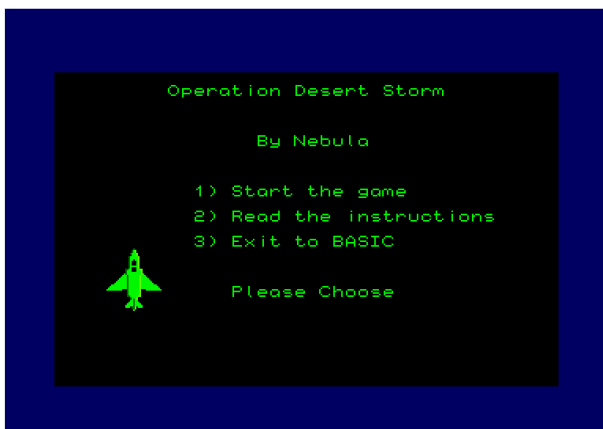


Mimo's Quest

When I was a child of 7 years old, my dad bought my sister and I an Amstrad CPC 464 for Christmas. It became part of our childhood growing up, and we loved to play games on it and type in programs from the User Manual. I was always fascinated at being able to program the computer to do different things. I also loved to type in programs from the Amstrad magazines and see how they worked. Being young I never really grasped much more than the elementary basic commands, though I did attempt to create my own game, the listing for which I proudly printed out on my dot matrix printer. I had read in the manual how it was possible to create your own fonts using the symbol command, and using sheets of graph paper, I painstakingly calculated each letter and typed it out on the computer. I had also read how it was possible using the CHR\$ command to create User Defined Graphics, and I set about creating a sprite of an F4E Phantom, my favourite aircraft. The first Gulf War had just finished at the time, and I thought it would be a great idea to make this into a game. So Operation Desert Storm was born. Sadly I soon realized that User Defined Graphics were not going to be any way fast enough to make a decent game.



I loved to type in programs from the magazines, and remember typing one in which showed the power of sprites. The Batman logo came flying down the screen. I was amazed. There were lots of data statements and funny numbers in the program, and I had no idea how people knew what these numbers did. All I knew was that it was machine code and was far beyond my understanding. I didn't really understand what assembly language was.

Years passed. I got a Sega Megadrive, then an Amiga 500, then a Super Nintendo, then an Amiga 1200, then an Amiga 3000 and eventually a PC. The Amstrad was forgotten about. But I still longed for the days when computing was simpler and they didn't control every aspect of your life. I developed a textviewer for the Amiga called EvenMore in a C-like language. I also used Visual Basic for Access, PHP, Python and a few others languages in my jobs.

I still enjoyed playing the odd Amstrad game through an emulator, and enjoyed reminiscing through old Amstrad magazines. I bought one book about the story of the Oliver Twins, computer game programmers, who learned assembly language programming with the help of an old school teacher using a BBC Micro. I was fascinated to see how they developed the games that I grew up with. I thought it can't be that hard, after all I was used to working in modern computing languages and the Amstrad should be a lot simpler to program. So I searched for examples of assembly programs online and typed them in to WinApe's assembler. It was amazing being able to do things on the emulator that were previously impossible in Basic. I read in the story of the Oliver Twins of a game they had started developing called Excalibar. This game was never released, but it was based around the legend of King Arthur, and used something called pseudo-random numbers to generate a world that the character could explore. This technique had already been used before in games like Elite II to create

```
WinAPE Z80 Assembler
File Edit Assemble Help
;; check keys and update sprite position
call check_keys
ld a,(interruptcounter)
cp 0
jr nz,main_loop
ld a,15
ld (interruptcounter),a
call updatecharactersprites
jr main_loop

interruptcounter:
defb 15

interrupthandler:
ex af,af'
ld a,(interruptcounter)
dec a
ld (interruptcounter),a
ex af,af' ; ENABLE INTERRUPTS AGAIN
ret

;; check if a key has been pressed and perform action if it has
check_keys:
ld a,0 ;; cursor up
call km_read_key
ret nc
cp $?;'
jr z, tilemap_up
cp $?;'
jr z, tilemap_down
cp $?;'
jr z, tilemap_left
cp $?;'
jr z, tilemap_right

tilemap_sprites:
Graphics Speedup Tilegraphics Data Statusbar Populatemap Pseudorandom
37:66
```

a massive game world on computers that had very little memory at the time, and this concept always fascinated me. If only Excalibar had not been shelved! I always wanted to try and create a game like this, so a few months ago I decided to try and cobble a few assembly examples together and try and create one.

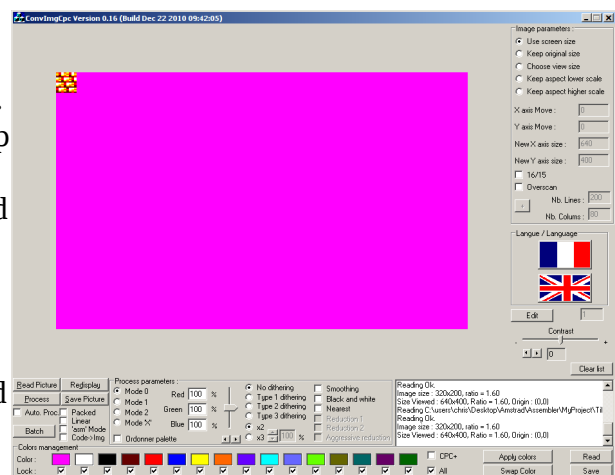
Creating the tilemap

I started with a tile map example by Kevin Thacker. This program builds a tile map and uses a selection of sprites to create a game map. The user can use the arrow keys to move to different rooms which have previously been defined in a grid. The example was sluggish so I tried to improve its speed. I discovered one of the tiles was just empty space and this was taking up some processing time drawing it to the screen, so I made the program clear the entire screen and skip over any tile that was empty space. This speeded up the program.

I found a pseudo-random number generator in assembler and added it into the code. I used this to generate the screen layout rather than rely on the predefined grid. It worked! But each time I moved rooms, the tiles would end up in different places. I needed a way for them to remember their positions. I discovered if I 'reset' the seed of the random number generator each time we moved a screen, the results would stay the same. Adding the room ID number into one of the seed bytes resulted in a unique random pattern for that room that would stay the same even after you reloaded the program! Success!

Now I had to get rid of the room definitions to give us a bigger gaming world. The pseudo-random number generator creates a number between 0 and 255, so I was able to use this to automatically generate a possible 256 rooms - a 16 by 16 square map. This did create a problem in that if you are in room 16 and you move to room 17, you are effectively transported to the left side of the map again, as the rooms are numbered sequentially. So I changed from using a single room ID to a coordinate based system, one byte for X and one byte for Y. This resolved the problem and also increased the world size to 256x256 – 65536 rooms! Or in real world terms, if each tile is a metre square, the world would be 9 square miles! Both of these numbers are added into the pseudo-random number seed to create each unique room.

The next step was learning how the CPC screen worked. I knew the screen was just a portion of memory that the computer displayed on the monitor. But how to write to it? The sprite data in the tile map looked unintelligible. I searched for help on a few websites, and found that the CPC screen is measured in bytes, and that each byte is responsible for displaying one or more pixels depending on the current screen mode. The simplest way of understanding what was happening was to load binary numbers (e.g. %01011010) into a register and write it to the screen to see what it produced. Mode 2 uses a single bit per pixel, Mode 1 uses 2 bits to display each pixel with extra bits used to specify the colour, and Mode 0 uses 4 bits per pixel. These bits are arranged in an interlaced pattern, which is a bit confusing at first. Once I worked out what bits needed to be changed for which pixels, I started to create my own sprites. Now that I understand how the sprites are written to the memory, I have started to use the program ConvImageCPC to create them, and paste the resulting assembly code directly into WinApe. These are saved first as a .SCR file, so I can edit them again if need be. And then, by clicking the Linear

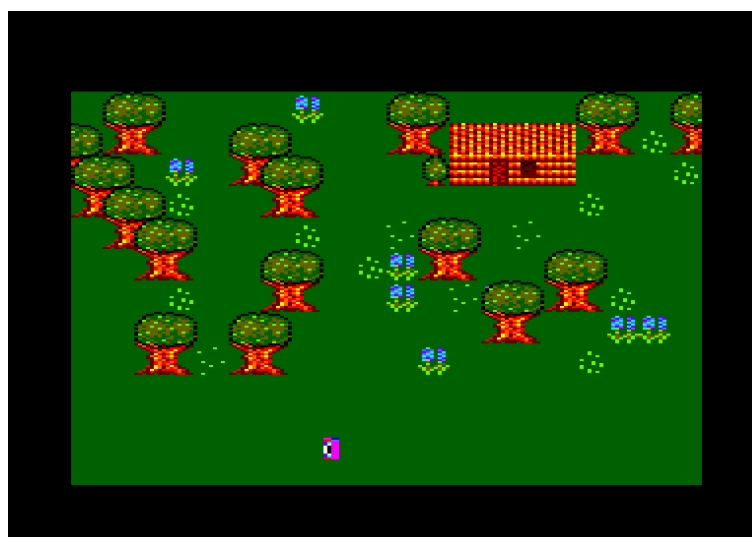


and 'Asm' Mode check boxes and clicking Save again, they are stored in a hexadecimal format that can be pasted into WinApe.

I made another few alterations to the tile map code to speed it up. I created a routine to join tile map graphics together and insert them into the grid automatically. Org'ing the tile map at an even number in memory allows me to transverse it vertically as well as horizontally, which speeds up processing quite a bit. Unfortunately I have discovered that using larger sprites based on several tiles joined together can result in some foreground tiles being accidentally overwritten by ones that should be in the background.



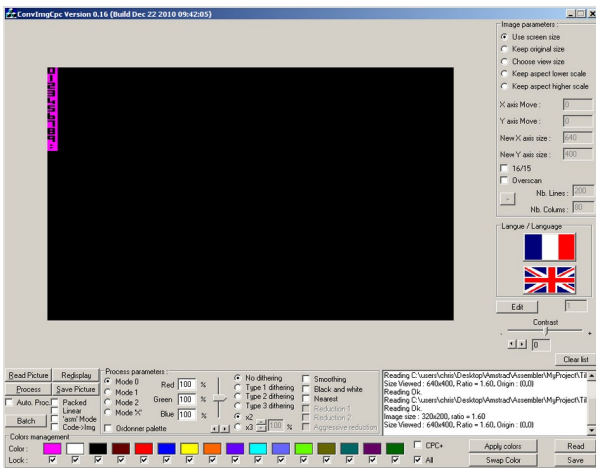
There is a solution, and that is to create an extra tile map grid for each layer in height. Then rendering the tile maps consecutively should prevent foreground tiles from being accidentally overwritten by background ones.



Allowing sprites to overlap one another is a little more difficult. Especially due to the fact that one byte controls the colour of two pixels. It's easy enough to skip any byte that has the colour 0 in it, but if only one of the pixels has the colour 0, how do you separate the pixel bits that should be

drawn to the screen from the ones that shouldn't be? In the end I had to download example code which detects which pixels are colour 0 in a byte while protecting the other pixels through a lookup table. I probably could have figured it out eventually, but it is quite complicated.

I downloaded some example code of how to write to the screen using the Amstrad firmware commands. I will use this to create a status bar, showing the current location and inventory stats. I may need to make a custom routine to do this to make the font smaller if I can't fit all the information on the screen. Either that or do a split screen and have the lower portion of the screen in mode 1 or even 2.



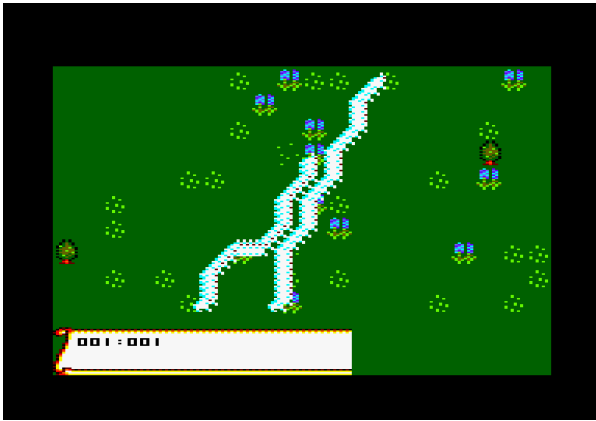
I tried the split screen and it did work, but I didn't like the effect. The change in resolution detracted from the rest of the graphics. So I decided to make a smaller Mode 0 font. This was done by creating a large sprite with all the numbers from 0-9, and using offsets to work out which one to display on the screen. The same is done for the letters, keeping to uppercase to save memory.

As the playing area is so large, I need to display map coordinates so that the player doesn't get lost. The current map location is contained in two bytes, so I needed a routine that could convert a byte to a string. This was done by taking the full number, e.g. 182, and then counting the number of times it took to subtract the 100s, then the 10s and then the 1s and putting the results into a string.

The next stage will be to create a player character that can move around the world, plus some computer controlled characters. The computer controlled characters will be controlled using a data area that will store whether they are active or not, current map location, current grid location, destination map location and destination grid location. The idea will be to randomly spawn a computer controlled character, then update these details at intervals which will give the character a direction. If the character is non essential, like a rabbit, it will be deactivated if it strays too far from the current screen.



Sprite movement



Character movement is quite complicated, due to the fact that the screen is built using tiles. In order for character movement to be smooth, they need to move by pixels. The only solution to this was to create a loop which would permit movement between tiles pixel by pixel. This forces the sprite to remain in the confines of the tile grid, which can be checked to see if it contains an object or not.

Animation is possible by changing the sprite address on each part of the loop. This is done by using a lookup table of sprite images, and cycling the sprite image number each time the sprite is drawn. Moving

left causes the left facing images to be used, and moving right causes the right facing images to be used. I am using a count of five for the movement loop, as this fits nicely into my 8x16 sprite size. The sprite can move 2 pixels at a time horizontally (or one byte in Mode 0), and 4 pixels at a time vertically. So four moves equals the size of one tile, and a new direction is calculated on the fifth part of the loop. At the minute character movement is random, which is a simpler way of creating movement without too much complication. I may need to amend the system to allow characters to go to specific screens and tiles.

The trails are being left by the sprite. I will need to implement some double buffering to erase them. That means taking a copy of the area in which the sprite will be moving to, pasting the sprite into it, drawing any house or tree tops that may obscure the sprite, and then pasting it back to screen memory. It is made more complicated due to the fact that the screen is built using tiles, and so I will need to find a way of working out which tiles are being affected.

Redrawing the tiles around the sprite may not be the best solution, as this might wipe out any sprites that are in the vicinity if they overlap. It may be possible to keep a double buffer for each sprite on screen, and paste it over the sprite whenever we are moving it. After that I would need to redraw the tree tops and house tops in case they are obscuring the sprite. This is complicated though, because I will need to work out how to give each sprite its own double buffer memory address. I could double buffer the entire screen but would lose 16k of ram and it may not be worth it to have smooth scrolling.



Managed to create a buffer for each sprite, and it works well, apart from the fact that erasing the sprite and redrawing it means they tend to flicker a bit. The only solution I can see is keeping a buffer of the entire screen. Then copying the relevant background graphic to a spare buffer, pasting the sprite on top, and writing that to the screen. This way I am only writing to the screen once, so flickering should be eliminated. It does mean taking up an extra 16k of memory, which I am not fussed about doing.

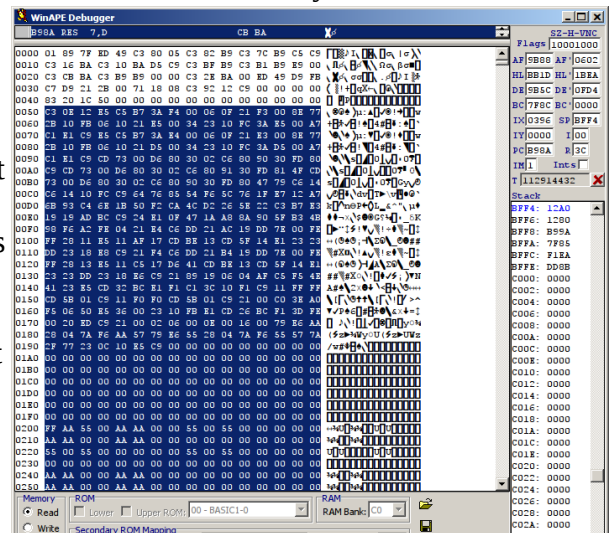
I also discovered that moving sprites outside of the screen will not really work properly, as we only generate an object tilemap for the current screen.

This would mean sprites could be moving over the tops of houses and through trees and although the player wouldn't see this happening outside the screen, if the current screen changes to the screen the sprite is in, it could cause problems. The only solution seems to be deactivating a sprite if it moves outside the screen, and spawning a new one randomly at the edge of the screen with a direction pointing inwards. This will work for non-essential characters, but for game characters, moving out of the screen might be a no-no. Having stationary game characters might be the solution to this.

I managed to make a ticker function to slow down the sprite movement. This was achieved the easy way by repointing BASIC's interrupt function to my sprites function. This is called every 300th of a second, but a countdown variable is used to stop the sprites updating as often as that. I made the clear screen function only clear the gaming area, rather than the status bar at the bottom of the screen, which looks a lot better. Sprite flickering is reduced. I may look at making a double buffer of the gaming area to remove it completely.

I have created teepees and a livery stable. I have divided the map into four main areas. The plains area will have stately homes and farms. The Indian area will have teepees and possibly buffalo. The forest area will have log cabins and lakes. The western area will have shanty towns and possibly horses. I may base the game on the theme of Little House on the Prairie, with the player having to buy and trade, and undertake various quests into each of the zones. There may be a health bar, which will be topped up by buying food, and the possibility of being attacked by wolves. Though that may be difficult.

I reorganized the code so it compiles around the start of the Amstrad memory in one continuous block, to see how much memory I am actually using. As far as I can see I have used about 7k of ram, and have about 42k left. Plenty of room for more sprites, although if I decide to use a double buffer that might reduce somewhat.



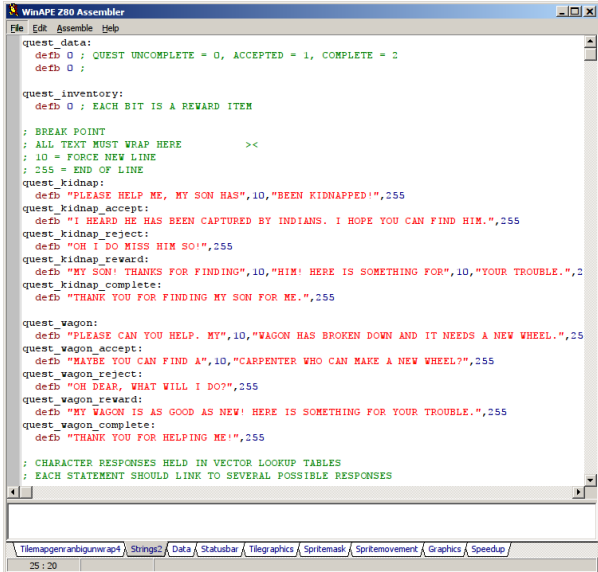
The text window is now working. I combined the numbers and letters sprites together in the same order as they are in the ASCII keymap. A routine subtracts the number 48 from the value of each ASCII character in a string to get the relevant sprite number to print to the screen (as I start with ASCII &32 which is a space). Only uppercase letters are used to save memory.

I will need to develop a routine to read keypresses and enter them onto the text window if necessary. The text window also scrolls. It basically reads the memory locations of four strings. If I want to display a new line of text, the memory contents of lines two and three are copied upwards, and the new line is pasted into the fourth memory location. This will enable the player to interact with game characters. I still need to work out a way for characters to be able to hold a conversation, i.e., linking several statements together based on the players response. These statements may need to include procedurally generated objects and names.



As you can see, I have developed some sprites for the western zone. Though I will need to think of a way of blending the seam between each zone so it appears more natural.

Started working on the quest routines. Each quest will be built up using a set of strings that the player will read when talking to a game character. These will be linked together via a lookup table. The first string is the request for help. The character tells the player what they need. The player can either choose to accept or reject the request, in which case the second response is displayed. If accepted, the character gives more specific information to the player to aid them in their quest. If rejected, the character gives a rejection response, but the player can still choose to talk to them again and accept the request. Once a quest is accepted, a record is kept that it has been accepted, and if the player talks to the character again, a routine will check to see if they have fulfilled the quest, and if so the appropriate reward will be given. The idea is to make some quests dependent on the completion of others, to make it more of a puzzle.



```
File Edit Assemble Help
quest_data:
  defb 0 ; QUEST UNCOMPLETE = 0, ACCEPTED = 1, COMPLETE = 2
  defb 0 ;

quest_inventory:
  defb 0 ; EACH BIT IS A REWARD ITEM

; BREAK POINT
; ALL TEXT MUST WRAP HERE          ><
; 10 = FORCE NEW LINE
; 255 = END OF LINE
quest_kidnap:
  defb "PLEASE HELP ME, MY SON HAS",10,"BEEN KIDNAPPED!",255
quest_kidnap_accept:
  defb "I HEARD HE HAS BEEN CAPTURED BY INDIANS. I HOPE YOU CAN FIND HIM.",255
quest_kidnap_reject:
  defb "OH I DO MISS HIM SO!",255
quest_kidnap_reward:
  defb "MY SON! THANKS FOR FINDING",10,"HIM! HERE IS SOMETHING FOR",10,"YOUR TROUBLE.",2
quest_kidnap_complete:
  defb "THANK YOU FOR FINDING MY SON FOR ME.",255

quest_wagon:
  defb "PLEASE CAN YOU HELP. MY",10,"WAGON HAS BROKEN DOWN AND IT NEEDS A NEW WHEEL.",25
quest_wagon_accept:
  defb "MAYBE YOU CAN FIND A",10,"CARPENTER WHO CAN MAKE A NEW WHEEL?",255
quest_wagon_reject:
  defb "OH DEAR, WHAT WILL I DO?",255
quest_wagon_reward:
  defb "MY WAGON IS AS GOOD AS NEW! HERE IS SOMETHING FOR YOUR TROUBLE.",255
quest_wagon_complete:
  defb "THANK YOU FOR HELPING ME!",255

; CHARACTER RESPONSES HELD IN VECTOR LOOKUP TABLES
; EACH STATEMENT SHOULD LINK TO SEVERAL POSSIBLE RESPONSES
```

If a quest is the search for a specific object, e.g. a lost son, then that object should only be made to appear on the map once the quest has been accepted.



I managed to optimize sprite movement a bit. Sprites will now move pixel by pixel as before, but the tile number they are occupying will be calculated mathematically on each move by dividing the pixel number by the number of tiles on the screen. This number will then be checked against the object tilemap to see if there is an obstruction or if the player has entered a house or wants to talk to a game character. I was trying to work it out before by forcing the player into the confines of the tile grid, but this wasn't going to really work for the player sprite.

I adapted the object tilemap to allow the character to interact with game characters and special locations. Each tile graphic definition has extra numbers at the end, an object id, and an x and y coordinate. The object id will be inserted into the object tile map at the specified location in the tile graphic when it is written to the screen. By calculating the player's current tile position against the object tilemap, the player will be able to talk to game characters to begin quests, to enter houses and shops and pick up objects.

By calculating the player's current tile position on the screen, it is also possible to redraw any obscuring graphics, like tree tops, house tops, mountains, enabling the player to walk behind these. I had to redraw the tiles, both to the left and right of the player as well, as it is possible for the player to be inbetween tiles, and for parts of the player sprite to still show through. It is not perfect, as I am still writing the player character sprite to the screen and then the tree tops on top of this, and sometimes there are occasions when the player character sprite will appear briefly. But it is acceptable, and saves doing a true double buffer and losing around 16k of memory.

Creating a DSK file

```
©1985 Amstrad Consumer Electronics plc
and Locomotive Software Ltd.

BASIC 1.1
Ready
cat
Drive A: user 0
QUEST .BAK 13K QUEST .BIN 13K
QUEST .BAS 1K
151K free
Ready
load "QUEST.BAS"
Ready
list
10 MEMORY &1FFF
20 LOAD "quest.bin",&2000
30 CALL &2000
Ready
```

I managed to create a DSK file with the game on it that can be loaded from BASIC. This was done by creating and formatting blank disk in WinAPE, assembling the code beginning at &2000, and then using WinAPE's memory debugger to see how much space the program takes up. Then by typing SAVE "QUEST.BIN",B,&2000,&3000 (current program length), I was able to save the memory contents to disk. A small BASIC program, "QUEST.BAS", was created to load this content into memory and then CALL the start of the program memory. It will also

be possible to display a picture on the screen while the main game loads. This makes my game feel a little more 'official'.

Creating the quests

I have now created a routine to allow nested quests. The acceptance of one quest, for example, the lumber jack's lost son, results in a second related quest becoming available where it wasn't before. In this case exchanging the lumber jack's son for some desired object. There is a limitation in that I can only have around 250 odd quests. The number is limited by the object tilemap. We can only recognize a number up to 255 in each tile location in the object tilemap. This number is used as a quest ID number. When the player bumps into the specific tile, the number is read from the object tilemap and the corresponding quest is activated if available. I hope eventually to make a random quest generator to make the game endless, but that will require a completely different approach! I would need a way to generate the quest, the map location, the required item and the reward automatically. Completed quests could be overwritten with new ones, but it would need to be done in such a way that it wouldn't seem out of place. I.e, we can't have buildings suddenly appearing and disappearing. It may be possible with character sprites.

My tilemap graphics are inspired by the Legend of Zelda top-down view. This view is not truly three dimensional, more like 2.5D. It can be tricky getting the perspective just right from this viewpoint. In Zelda, especially in the case of walls and mountains and other objects, sometimes the back side of the object is still viewable, although to a lesser degree, to allow the character to be seen most of the time on a flat tilemap. As you can see, the mountain looks a little out of place, being drawn from a head-on view, compared to the buildings in the top-down view. I will have to redraw the mountains and hills to look a little more three dimensional.





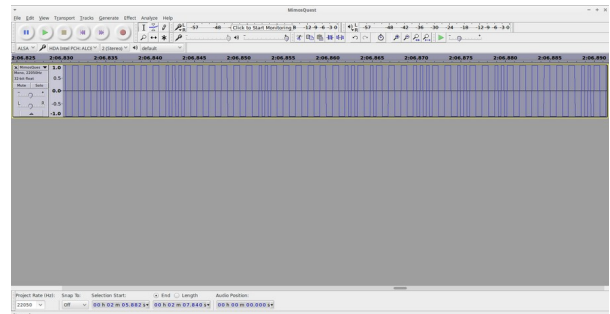
It is now possible to pick up items. This is done using the routine that inserts the main sprite graphics onto the tilemap. An item array contains the screen location, object ID and sprite graphics to insert into the map. Items are checked and added to the screen before it is populated with buildings and foliage. A different object ID is used for the object tilemap (e.g. ID 250 = Gold), which the program recognizes as being an item that can be picked up rather than an ordinary obstacle (ID 1) or empty space (ID 0). The object ID is read and the appropriate action is taken when the player moves over it. The item is then

cleared from the tilemap and from the array of items.

I have now redrawn mountains in the correct aspect. I will still need to think about how to separate the different zones so the seam appears more natural. I was thinking about a mountain range, or a forest that couldn't be crossed without completing a special quest. I will need to see if it looks right.

Creating a tape file

WinApe is able to emulate tapes on the CPC 464 emulation. It can read or write the audio the CPC would produce when loading or saving to tape in a WAV file. I managed to use this to save my game to tape. It took 8 blocks and three and a half minutes to save! I can see now the benefit of having loading screens. The tape emulation is slightly different, in that I will need to have the BASIC loader and the game code in the right order on tape. I hope to be able to produce a physical tape, complete with inlay card, which will work on a real Amstrad. It was great hearing that authentic tape-loading sound again!



Loading screen

I have created a basic loading screen, and a few assembly functions to read the screen data produced by ConvImgCPC and display it on the screen. I will adjust my BASIC loader to run this program first, and then load the main game code. I may produce the loading screen in something like GIMP. It should be based on the main theme of the inlay card. I will then run the image through ConvImgCPC and clean it up.

Creating the inlay card

I have scanned in the inlay card of another Amstrad game, and will use its layout to create my own inlay card, complete with screenshots, the main attraction of any game in the 80s!



Tile animations

Basic tile animations are now possible. This is done using the same lookup table that is used for the sprite animations. A separate data table has also been created specifying whether or not the animation is active, the tile position on screen to use, the ID of the sprite data, and an offset that is used to record which frame of animation to display on the next redraw. The graphic is then drawn immediately after the character sprites are updated. This is very effective for the fire in the Indian village. The added touch of movement gives the scene that extra bit of realism.



As you can see, certain tile maps also have town names associated with them. A routine automatically checks the current screen coordinates against a table of static map definitions. If a match is found, the corresponding code is called to create that particular town map, and the status bar is updated to show the player where they are. A town can be created in just a few bytes of code, which should enable me to add some interesting map locations in the game, and make it a little less 'random'.

The bridge

As you can see, I have redrawn the water sprite, as the previous one was a little flat looking. This was achieved by drawing a pond shape in ConvImgCPC, making sure the edge of the pond touches the corner of each tile so that they fit together in any combination. This image is then split it up into 9 separate tiles. These tiles can then be arranged to make a river shape. These shapes are joined together in a static map definition to make a river than can be meandered around towns and castles. I also created a bridge for crossing the river. This required an extra piece of code to make part of the object passable by the player. A routine fills in the object tile map with the number 254, which I use to define as a passable part of an object. Using a value other than 0 also prevents trees from randomly appearing in the middle of the bridge.



3d tile maps

Drawing in three dimensions can be tricky in a low resolution. I discovered a way of making it easier. Start by drawing a flat tile surface that faces the player, for instance, a simple brick pattern. Then use this pattern for a corner, only shear it, i.e. cut and paste blocks of pixels incrementally narrower and higher towards the edge. This results in a pleasing and easy to produce 3d effect.



Despite its limitations, the Amstrad is capable of producing quite beautiful graphics if the right techniques are used. Below, a simple brick pattern and thatched roof effects for the cottage and the castle produce pleasing results despite using only a handful of colours. Using the same roof images for the castle and cottage saves a little memory too. When you are limited to 64k, and 16k of that is taken up by the screen, every byte counts.

Entering buildings

It is now possible for the player character to enter buildings. This is achieved in a similar manner to the quest routine, by detecting the object ID of the tile the player is passing over in the doorway. If it is a certain number, a variable records that the player has entered a building. The player's current screen coordinates are stored, the tilemap is redrawn for the room, and then the player sprite is relocated next to the doorway inside the room. The process is reversed for exiting a building. I will keep some buildings locked, and others may require the possession of a key to open. It will make the game a lot more interesting by being able to enter buildings and pick up objects or undertake quests.



I found that the wooden walls I had created for the rooms made a really nice roof pattern, so modified the wooden corners so that they are transparent on the top side. This still allowed me to make a room shape by pasting these corners over the top of the back wall of the room. But by themselves they also make a nice roof for my castle. Re-using these tiles saves a bit of memory, and also saves having to draw special graphics for the castle roof. I just need to make a castle entrance now.

I have also modified the quest object code to allow the placement of quest objects inside buildings. An extra variable in each quest object stores the building ID it is associated with.

Worlds

Making the map seamless, blending the graphics style between the forest zones and the western zone proved too difficult, due to limited colours, large map size and limited memory. I have decided that a Narnia-style secret doorway could make an interesting link between the two zones. An old abandoned mine links the forest and western zones together, and a secret tunnel-like room takes the player from one zone to the other. This has effectively doubled the world size as well, and so I will need some way of generating random quests to fill the world.

I have also developed a random seed function, where it will be possible to randomize the world to create new challenges for the player if the first game has been completed. Though I will need to make the map more fully respond to this, by randomly generating towns and other features.

Creating towns

Due to memory constraints, it would be impossible to create lots of statically built towns with their own names. So by using the map coordinates, for example on every coordinate that ends in a 0, a town is randomly generated. This gives a nice space between each of the towns. Giving each town its own name is also done randomly. Each town name is made up of a first and a last name. For example, "Poplar" and "Point", "Woody" and "acres", etc. A first and last name is randomly selected and combined together when the town is generated. Other map features may be spaced at greater intervals, for example, a castle, etc. Map features which span several screens, like a river or lake may have to be statically generated, but given random coordinates when the seed is generated.

Random quests

Generating random quests is a little bit harder. I have to ensure that randomly placing a character on a map will not displace any existing objects, likes trees, houses, etc. The insert quest sprite routine has been modified to check for the next available space. Random quest sprites will be inserted onto



the map in each town, and if the player bumps into them, a random quest will be generated based on their current location. Another participating random character may be placed at a separate location whenever the quest is accepted. It will be up to the player to find that second character and pass on whatever object they have been given or retrieve whatever object they have been assigned to get.

Randomly generated towns now fill the map. Each has its own randomly generated town name. This will save quite a lot of bytes. I will eventually generate the town names based on their locations. E.g if a town has a lake beside it, it will have a water themed name. If it is wild west, etc.. The code is now in place for randomly generated quests. Towns and villages will have characters that will generate a random quest when approached, which you can choose to accept or reject. If accepted, a quest "chain" will be generated. Extra character sprites will be generated at random locations. Each character will possess an object which another character needs. At the end of the chain the player will be rewarded appropriately. This will also save quite a lot of bytes.

Optimizations

A lot of work has been done optimizing the quest code, making sure I use the minimum amount of memory required for each task. The status bar print routine now accepts a placeholder character for inserting a random name into an existing string. This makes generating dialogues for the random quests very easy.

I also decided that scrolling the graphics of the status bar by pixels rather than reprinting the text for all the lines each time was probably a lot quicker and would use less code. I also found room for an extra line of text in the process.



Randomizing the rooms

Pseudo-randomized quests were working well outside of the buildings, but I found that an extra parameter was needed in order for it to work inside buildings too. Normally the X and Y map co-ordinates are used in the pseudo-random number sequence to generate the tile map. But this meant no matter what building you entered into in that same screen, the layout and quest characters in the building would always be the same. I found that by recording the last tile position of the player when he entered a building (i.e. the door way tile), and adding that to the pseudo-random number sequence, I could make each individual building contain a different character and quest to complete. Once the player accepts a random quest, a quest block is filled in, recording which map co-ordinate, building co-ordinate and tile number the quest sprite belonged to when the player accepted the quest. This information is then used to make sure that when the player retrieves the particular object and returns to the quest sprite, that we know which quest it relates to.



Chasing the rabbit

I had decided beforehand that having characters move outside of the screen was too difficult to implement, due to not knowing which objects the character would have to navigate. When a sprite reached the edge of the screen, it would automatically be deactivated, and a new sprite generated in the new screen. This worked, but had the adverse affect, that the sprite was always generated in the same place, so it was apparent to the player that it wasn't the same sprite. Another user on facebook had told me of a method of deactivating the sprite when it reached a certain distance from the player, for instance, two screens away. I thought that might be complicated to achieve, but it is actually not too difficult. When a sprite reaches a screen boundary, a routine is called that subtracts the sprite's screen numbers from the player's screen numbers. If the difference is found to be 2 or 253 (plus or minus the player's screen), the sprite is deactivated. This allows the player to chase a rabbit around the map, and it will remain active until it is more than one screen away from the player. Once this happens it will be deactivated, making room for another sprite to be generated if need be.



I have added four extra coordinate bytes for sprites, to allow a destination map and tile number to be specified. In this case the sprite will automatically pick a direction that will lead it towards the coordinates. This will allow characters to move from town to town, seemingly going about their every day business. Sometimes the characters may get stuck behind an object. In this case randomizing the sprite direction every so often prevents them getting permanently stuck. Once the sprite has found the correct map location, another routine will choose a random doorway tile from the map for the sprite to walk towards. Then the sprite will be given another map coordinate to find.

Flipping sprites!

As the graphics data takes up the majority of the memory space in the game, it is important to make the most efficient use of it as possible. For instance, tiles with a repeating pattern, or the same tiles used for several objects, all helps to cut down the amount of space the graphics takes up. This leaves room for more graphics or game code. Flipping commonly used sprites programmatically, either horizontally or vertically, can also help free up space. I decided to make a routine that would read sprite data and write it to the screen backwards, flipping the sprite horizontally. This is useful for character sprites that can move in multiple directions. It saves having to draw them out facing left and facing right. Especially if they are animated, that can use up quite a lot of space.

Unfortunately the process is not simple, due to the way the screen is laid out in memory. In Mode 0, one byte controls two pixels, and the bits that control them are intertwined, so they need to be separated first. A bit mask is applied that erases every other bit to isolate the left pixel. The result is then shifted to the right one bit. Another bit mask is applied to the original byte of memory to isolate the right pixel and then this is shifted to the left. The results are then merged together. This is repeated for every byte being written to the screen. Thankfully the code is still fast enough that no slowdown is noticeable.

A modified tilemap drawing routine checks to see if a tilemap number is greater than 200. If so, 200 is subtracted from the tilemap number and the tile with the resulting number is drawn horizontally

flipped. This will also save some memory, as quite often tiles used in buildings, etc., use the same graphics only flipped horizontally for opposite ends.

Sprite movement update

I amended the sprite table structure so it is possible to give sprites a destination map screen, as well as a destination tile to head towards in that screen. I was hoping to make it so characters could walk from town to town and enter and leave buildings. That won't be possible. At least, entering buildings is not really possible. If a sprite enters a building, because the room objects are not generated until the player enters the building, the sprite would never be able to leave it again because the exit door tile would not be generated. Walking from town to town is possible if the player follows the sprite around, and it is even possible to make the sprite walk up to other character sprites. But due to objects getting in the way, it doesn't really work well, and the player may not be in the screen long enough to even notice what the sprite is doing. So other than rabbits hopping about randomly, making sprites walk around by themselves is not really that useful. I can see now why game programmers gave sprites a set path to follow, and it was usually pretty limited in range.

Exploring hidden regions of the map

The thrill of exploring previously hidden or inaccessible regions of the map is a major part of the gameplay in the series. So I hope to do the same... Finding the flippers will enable the player to cross rivers and lakes to areas that were previously inaccessible to the player in order to complete more quests. I also decided that having random quests is good to fill up the map, but so the player does not lose interest, I will need to make a set of quests that are tied into the story line of the game to make it more interesting. Like Zelda, random characters around the map will give clues to how to reach inaccessible places.

Drawing six extra sprite images of Mimo swimming when he enters the water would use up a lot of memory (two for each direction, minus the flipped images). So instead, when it is detected that we have moved over a water sprite, and Mimo has the flippers in his possession, an extra parameter is set to only draw the top half of each sprite image. Drawing the sprite is also moved down four pixel lines, so it looks as if he has entered the water and is actually swimming around.

In the same way, finding a climbing rope will enable the player to move over some mountain tiles to access previously unreachable places in the game.



Randomized world generation

I improved the continuity of the randomized world generation routine. Before, screens looked a bit random and did not fit in with each other. Yes, there were specific zones created at the extents of the map, but they didn't really blend together or make the player feel like he was exploring a real world. I have improved the continuity by utilizing the map coordinates. Each of the tens coordinate numbers are associated with a particular type of scenery. For instance, the map coordinate with a 0 tens number is prairie, 2 is prairie, 3 is forest, 4 is mountains, 5 is lakes, 6 is mountains, 7 is forest, 8 is prairie, etc... This pattern is repeated for both the X and Y axis, and where they overlap the scenery is blended together. This will give better continuity in the map, for instance, by making forests extend for tens of screens.



A river runs through it

Having a landscape feature that spans several screens is a good way of keeping continuity. But because there are so many screens (65536), it becomes very difficult to create a large static world without running out of memory and processing power. So procedural generation is used as much as possible. Overlapping repeating features help to mask the randomness of the map, in this case a river, that runs the length of the land at a specific location. This is done using the tens map coordinates. If the tens map coordinate is a six, then this is considered the flood plain zone, and a table is checked to see if part of a river should be created on this particular screen in this zone. The river is designed in such a way as to repeat vertically every five/ten screens. This will also make a handy barrier that the player has to cross during the game. Random houses may also be placed along the river side, giving the impression of settlement, like you would find in real life.



This screen-spanning feature may also be utilized to create paths and roads around the town areas .

Snow

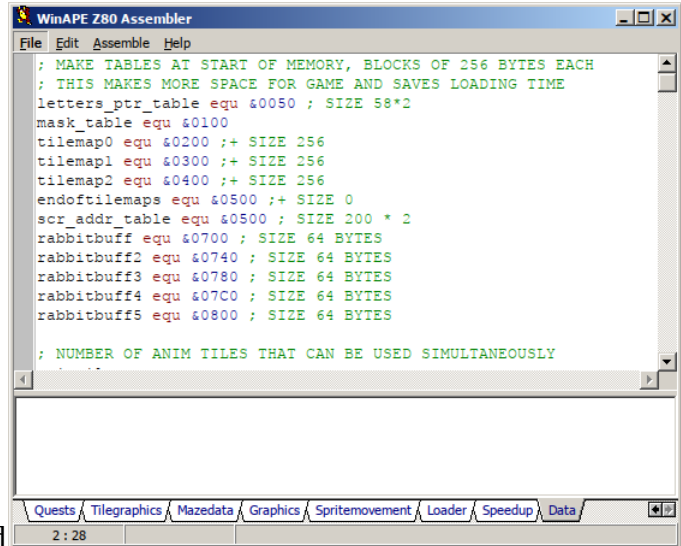
The snow zone on the map. Making the screen entirely white doesn't look right, and is sore on the eyes. Due to the limited colour palette, it is impossible to achieve a pleasing effect. But by changing a few green pixels to white, it is possible to achieve an impression of snow. To save memory, a routine checks to see if we have just changed zones. If so, and if the new zone is the winter one, the sprite data for the trees is directly modified. If



the new zone is not the winter one, any white pixel data is changed back to green. It would have been possible to achieve the same effect by changing the colour palette. But by changing the sprite data directly, I may be able to achieve more effects for different zones. Perhaps an Autumn zone with browns and oranges.

Saving memory

With most of the Amstrad's 64k taken up by sprite data, there is not a whole lot of room left for the actual game, so finding ways to save as much memory as possible is very important. There was a block of around 8k at the start of the Amstrad's memory which I thought I could not overwrite due to some DOS firmware data being stored there which was needed for the loader. But I managed to move some data tables into that area of memory. The tilemap data and some sprite buffers are now located there. This leaves more memory available for the actual game. It also reduces loading time, as before the Amstrad was having to load the empty table data, whereas now it is just defined by the hard-coded memory addresses.



```
WinAPE Z80 Assembler
File Edit Assemble Help
; MAKE TABLES AT START OF MEMORY, BLOCKS OF 256 BYTES EACH
; THIS MAKES MORE SPACE FOR GAME AND SAVES LOADING TIME
letters_ptr_table equ $0050 ; SIZE 58*2
mask_table equ $0100
tilemap0 equ $0200 ;+ SIZE 256
tilemap1 equ $0300 ;+ SIZE 256
tilemap2 equ $0400 ;+ SIZE 256
endoftilemaps equ $0500 ;+ SIZE 0
scr_addr_table equ $0500 ; SIZE 200 * 2
rabbitbuff equ $0700 ; SIZE 64 BYTES
rabbitbuff2 equ $0740 ; SIZE 64 BYTES
rabbitbuff3 equ $0780 ; SIZE 64 BYTES
rabbitbuff4 equ $07C0 ; SIZE 64 BYTES
rabbitbuff5 equ $0800 ; SIZE 64 BYTES

; NUMBER OF ANIM TILES THAT CAN BE USED SIMULTANEOUSLY

Quests Tilegraphics Mazedata Graphics Spritemovement Loader Speedup Data
2:28
```

More flipping sprites!

I decided to try and save a few more bytes by creating a routine to automatically flip sprites vertically. As my sprite numbers only go up to 255, with the last 55 being horizontally flipped versions of the first 55, I had no way of recording whether a sprite was vertically flipped or not. As most of the vertically flipped sprites will be walls, I decided to modify the object tilemap slightly to record if a sprite tile needed to be vertically flipped or not. If the object tilemap contains a 1, the sprite tile is drawn in the normal fashion. If it contains a 4 (water is 2, and mountains are 3), then the sprite tile is still an object to the player but the graphic is drawn vertically flipped. This means an entire room can be drawn with just 3 tiles, a vertical wall tile, a horizontal wall tile and a corner wall tile.

Arrows

Allowing the player to shoot arrows can add to the gameplay by giving the player something else to do other than just completing quests all the time. It will also give the player a skill to master. It should also add to the atmosphere of the game. I was thinking of remoulding the game into more of a Robin Hood type theme. Having a central character similar to that of Robin Hood would help the plot line somewhat, by giving him enemies to overcome, as well as possibly a princess to rescue?

I decided to use the existing sprite movement routines for firing the arrows, as these already contain all the necessary direction logic to control



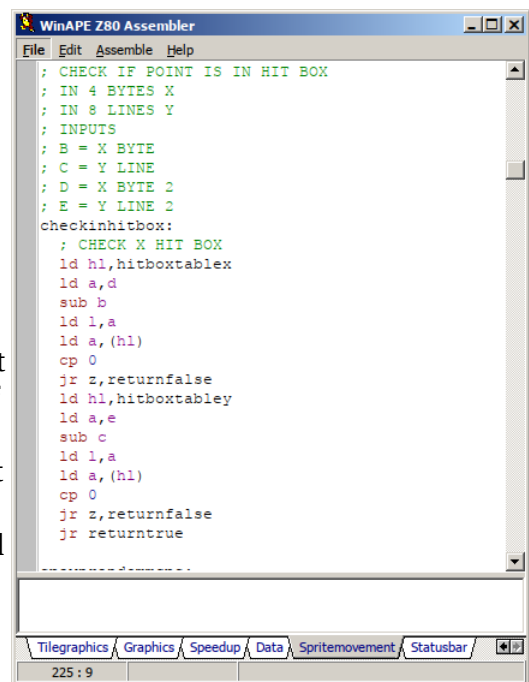
the arrow. It is just a case of modifying the code slightly to determine what happens when an arrow hits an object. If it hits an object the arrow will be disabled. If it hits a game character, the character should be wounded or killed. There should be the possibility of running out of arrows, and buying more arrows, to add an additional challenge to the game.

A table of sprite data enables the game to keep track of many different sprites and their movements on the screen at one time. I found it was possible to allow more than one arrow to be showing on the screen at one time, but I found that when the player fired 3 or more the screen started to slow down quite rapidly. This is because the buffering of the sprite takes up time and the fact that the sprite is using transparency which slows things down a little too. So I decided to restrict the player to firing one arrow at a time.

Collision detection

There are several different methods of collision detection. The one that already exists within the game is the tile grid method, which works very well for buildings and trees which don't move. Object collision detection for game characters that move is a little more difficult. The main problem being that a sprite can be 'between' tiles. I developed a routine that used tile grid collision detection for shooting the arrow, but I found it was not accurate enough for moving sprites. Sometimes it would pass right through a rabbit and sometimes it would hit the tile a rabbit was officially on, but not actually connect with it as part of the rabbit was on a different tile. I thought about different ways of resolving this. Do I record all of the tiles the rabbit is touching in a table, and check each of those to see if it matches the tile the arrow is on? It seemed too complicated to work out.

Eventually I decided to implement a distance check method. The pixel location of the point of the arrow will be checked against the central pixel location of each sprite on the screen. If the distance is within a certain range, allowing for the size of the sprite being collided with, then a hit will be recorded. Using a lookup table to define that range saves a lot of calculations, so hopefully the code should turn out to be quite fast.



```
WinAPE Z80 Assembler
File Edit Assemble Help
; CHECK IF POINT IS IN HIT BOX
; IN 4 BYTES X
; IN 8 LINES Y
; INPUTS
; B = X BYTE
; C = Y LINE
; D = X BYTE 2
; E = Y LINE 2
checkinhitbox:
; CHECK X HIT BOX
ld hl, hitboxtablex
ld a, d
sub b
ld l, a
ld a, (hl)
cp 0
jr z, returnfalse
ld hl, hitboxtabley
ld a, e
sub c
ld l, a
ld a, (hl)
cp 0
jr z, returnfalse
jr returntrue
-----
Tilegraphics Graphics Speedup Data Spritemovement Statusbar
225 : 9
```

Flipping marvellous!

I have been trying to think of a way I could vertically flip a door sprite. The problem with the tile map screen redraw method, as I said before, is that you are limited to a specific number of tiles. Tiles 0-199 are ordinary tiles, tiles 200-255 are horizontally flipped versions of tiles 0-55. These are drawn in reverse on the fly, rather than using up precious memory. I managed to make sprites flip vertically by manipulating the object tile map layer, but it was not possible to make individual tiles within a sprite vertically flipped. And it was not possible to vertically flip anything on the second tile map layer (i.e. tree tops, roof tops), as this layer does not have an object layer behind it.

I managed to come up with a novel solution for the door sprite. It is now possible to vertically flip any sprite within a particular screen. This is done by keeping a blank tile graphic, and programmatically generating a vertically flipped tile and storing it in the memory. If it is kept in the

0-55 tile range, it can also be horizontally flipped on the fly. This will save some memory. Though it does mean it only really works for certain statically generated locations, such as rooms, etc.

Collision detection update

The collision detection routine I mentioned earlier seems to be working very well. This works for all sprites, and all I need to do is specify what will happen when two particular sprites meet, for instance, an arrow and a rabbit. At the minute, the sprite is just prevented from moving further when they collide. I have created an arrow target sprite, and I may create a sub game in which you have to hit several targets in quick succession to win a prize. I may also make a quest where you have to storm a castle and disable several guards who will be firing arrows back at you.



I tried a couple of different methods of cleaning up the sprite collision and stopping orphan pixels being left behind. The first method was to erase all sprites on the screen, grab the background buffer behind each one and redraw them all again. This was slow. Updating the sprite movements in between the redraw was taking too much time, making the sprites flickery.

I also tried a unique method of determining how close a sprite was to another sprite, and if it was close enough, force the two sprites to redraw their background buffer, grab that buffer again, and redraw the sprites in the new position. This worked better, but sprites still flickered on and off when they touched.



In the end I compromised. There are two zones around each sprite. The red zone determines if a sprite has actually collided with something. Eg. an arrow. Arrow sprites ignore the blue zone and only get picked up when they hit the red zone. Ordinary sprites, like the player, other rabbits, etc, will check to see if they are in the blue zone first, and if that is so, the sprite will be stopped from going any further and corrupting the graphic. The benefit of using this system is, it is fast and sprites are less flickery, and it is possible now to make rabbits run away from the player if the player pushes against them!

Foliage generation

Different foliage is generated based on the tens map coordinate. For instance, anything with a 0 in the tens digit is plains, 1 is prairie, 2 is forest, 3 is mountain, etc. These are combined for horizontal and vertical map coordinates to allow blending of scenery to make things more natural looking. I had been adding these numbers together and using a long list of CP instructions to see which unique

landscape I needed to generate. I have now optimised that code by using vector lookup tables instead. The horizontal tens digit will be read, the lookup table checked for the routine to call, and then the foliage added to the game map. Then the vertical tens digit is read, the lookup table checked again, and that particular foliage added to the game map. It should make changing screens a lot faster, and takes up less memory.

I have also modified the random town locations a little more. Previously I had been generating a town if the singles digit was 0. This allowed for plenty of space around each town, but it was a little predictable. Now I add the tens and singles digits together for both the horizontal and vertical map coordinates, and if both add up to 10, a town is built. It makes it a little less predictable, but roughly the same space between each town.

Dictionary

Text takes up an awful lot of space in memory. Each letter takes up one byte, which could be used for better things, such as making more sprites, improving the game play. If words are used more than once, this is an even bigger waste of space, so I decided to create a dictionary. I created a table of commonly used words or phrases, and gave each phrase a number. Then when I am creating the dialogues, a byte 13 indicates that the following byte is the number of the word to look up in the table to print in the string. This should enable me to produce lots of dialogues without using up too much memory.

This is my island in the sun

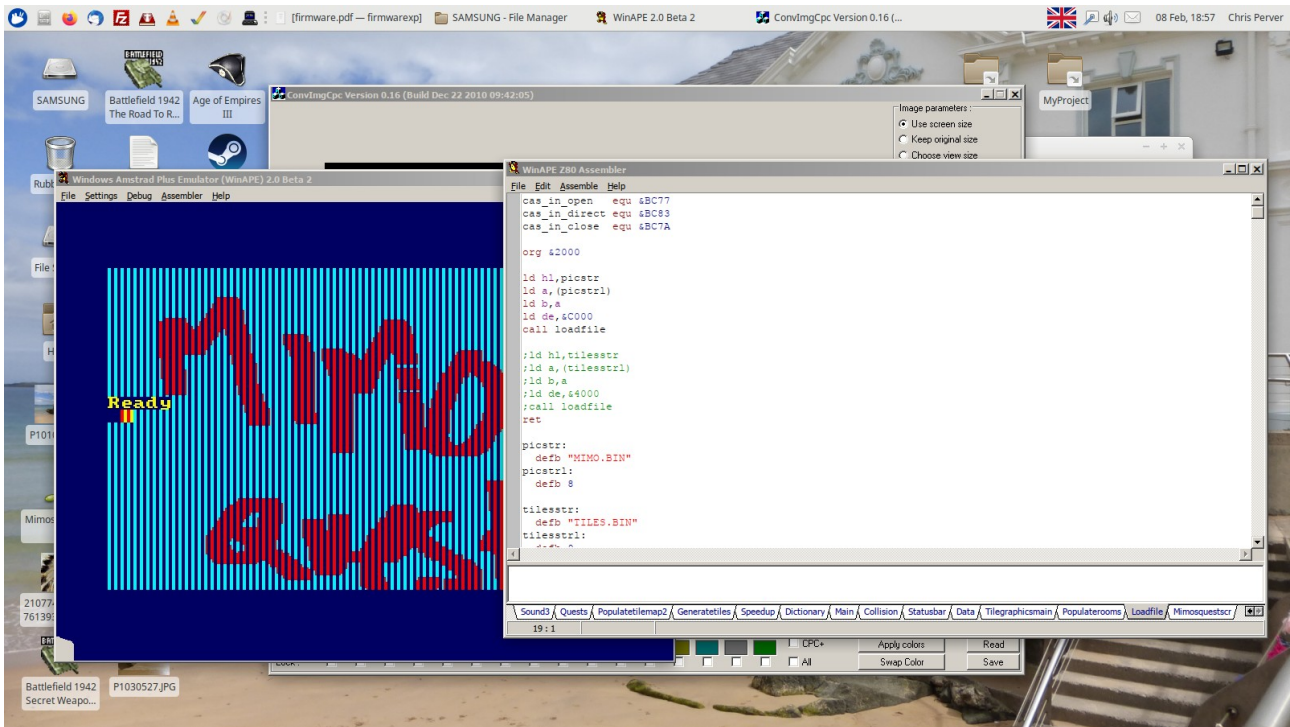
Using a function that can replicate screen rooms across the tens map coordinate, it is possible to create a large lake or sea. This makes the landscape a little more varied, and improves contiguity between screens. I modified the river screens by filling in the outside edges with water, and doing this for the four river bends and the vertical and horizontal river tiles, this allowed me to create a small island in the middle of the lake. On the small island I built a settlement, and will incorporate it into some quest.



64k limits

Despite having used a lot of different techniques to save memory, I am still finding that space is increasingly tight and a lot of the world is yet to be filled. I may have to go to 128k. This may involve redesigning the code a little, as it is only possible to use the extra 64k by swapping out blocks of 16k at a time with the inbuilt memory in the CPC. I will have to make sure no program code exists in the block that I want to swap out so it doesn't crash the computer. It is possible to detect whether or not there is a 64k expansion. If I create a function in the loader to detect this, my program could load the sprite data into one 16k block, tile graphics into another, and the word dictionary into another. These are areas of the game that take up a lot of memory but are kept in one large block, so it should be fairly easy to access them as needed. If there is no 64k expansion, then I could design the program to load up a reduced sprite and tilegraphic set into the correct area of memory. So essentially there would be two versions of the game.

Proper loader



Using firmware functions in assembly language, it is possible to load files directly into memory without having to resort to BASIC. In the picture you can see where I have loaded an SCR file directly into screen memory from the disc. This should enable better memory detection in the main game without having to resort to BASIC peeking and poking of variable locations.

The plan is to run the loader from BASIC. This has the advantage of displaying the loading screen gradually, building the player's anticipation of the game. The loader will detect whether there is a 64k RAM expansion, and if so, load an extended version of the game's quests and graphics into the expansion memory. A variable will be set so the game will know it needs to do bank switching in order to access the data. The loader should be able to give me more space, by loading the game program into memory closer to the start of load memory than BASIC will allow.

Zones

I have decided that the world is much too big to fill, and the current structure, of splitting the world into 9 zones is too uniform. I have decided to change the world so it is divided up into a 10x10 grid. Some blocks in the grid will be different zones of land and some will be water.

The different zones will be randomly allocated in a table at the start of the game. A function will randomly fill this table with a dozen points where the ocean should be. Another function will 'grow' these points until they become a suitable size and perhaps join up in various places.



A second grid of 10x10 screens will divide each block within the first grid. This grid will have to pseudo-randomly generated as soon as the player enters this specific grid. It will be generated in the same way as the first grid. Some blocks will be towns. There will be land and water blocks that will be 'grown' accordingly. Water blocks will need to know how to merge with other water blocks. A

lookup table will be used to know how a partially water-filled screen should join with another partially water-filled screen. The neighbouring parent blocks will also need to be checked to make sure their seam also matches.

As you can see from the picture, the two maps are displayed on the screen by pressing M. The left map is the world map, showing the zones in their different colours. The right map is the local map, showing lakes, towns, etc.

A plot line emerges

I decided to save some more memory by converting my coordinates for the sprites in the statically built rooms into tile numbers. This saves one byte per sprite, and is also faster, as I don't have to convert the coordinates programmatically.

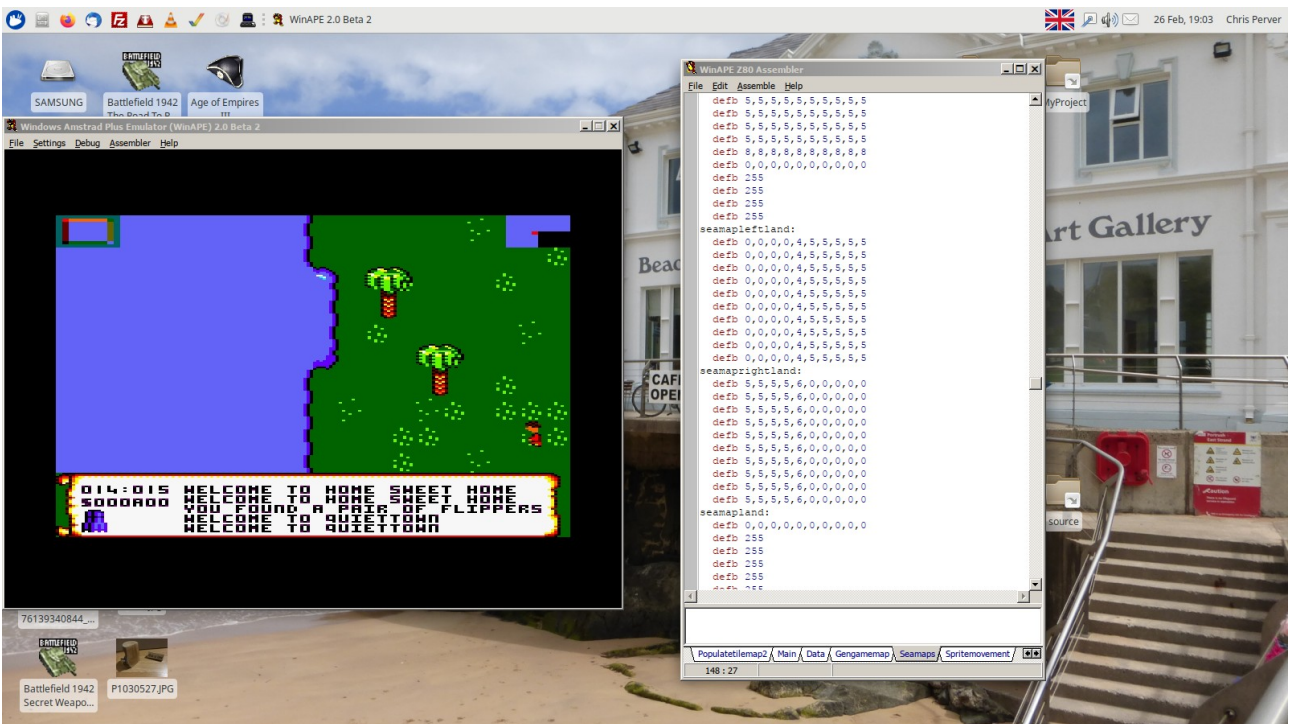


The new map feature in the game got me thinking a little bit about a plot for the game. Mimo should have a location he is trying to get to, for instance, home, and he could be stuck on some sort of an island. Retrieving the map would be one of the key stages in the game that would help him find his way home. He may also need to get the flippers at some point in the game to enable him to swim across the sea.

I also managed to make the seam between the screens a little less obvious by splitting the mountains into a top and bottom half. This will form a natural barrier around Mimo's home,

beyond which will be the sea he has to cross from the island.

Hand-drawn maps



I spent several days thinking about how it would be possible to generate a local map pseudo-randomly. I thought about creating several points on the map, and then 'growing' them into lakes and seas. This is theoretically possible, although it would take a little extra effort to 'round off' the corners of the lakes and islands so they appear natural. What stumped me was the problem of the seam between each local map. If the sea happened to span more than one map block, how would the pseudo-random landscape generator know to make part of a sea adjacent to that block so no seam appeared? It seemed impossible. There is a simple solution, of course, and that is to force each local map to be bounded either by sea or by land. But the results would not be natural looking. You would still have the problem of being forced to have either a small land mass in each block surrounded by sea, or else one big land mass with small seas in each block. In the end I decided the easiest way to create a natural-looking world would be to design each local map by hand.

Around a dozen local map 'types' are defined. For example, one map type has sea on the left, one has sea on the right, one has sea above, one has sea below, etc. These local maps are then joined together in a 10x10 grid to make the world map.

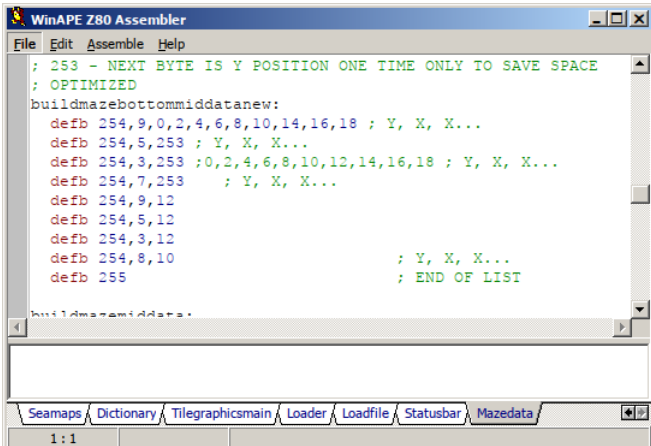
Each local map is made up using a grid of 10x10 bytes. Each byte in the grid references a lookup table, which tells the program what scenery to draw on the screen. Using this method it is possible to manually round off all the corners of the seas or land, and join each local map together with the correct neighbouring local map, so no seams between the blocks appear.

I was worried about using this method, as it would use up quite a lot of memory. But I managed to use a compression technique called RLE (run-length encoding). This involves replacing large sequences of similar bytes with a special character. Then when I am using the map data, my function expands these special characters into the correct sequence again so they can be read. This saves quite a lot of memory, as it is possible to replace 10 or 20 bytes at a time with just one special byte character. In my program, the character 255 tells the program just to copy the same byte sequence on the previous line. The character 254 tells the program to make 4 copies of whatever the previous byte was. In this way it is possible to compress the map data quite a bit.

Previously I had the world map responsible for making the 'zones', for example, the western zone, the forest zone, etc. As I was now using the world map to specify local map 'types', I had to make an additional world map for climate zones. This way I can create islands that are forest, islands that are tropical, desert islands, etc. as needed.

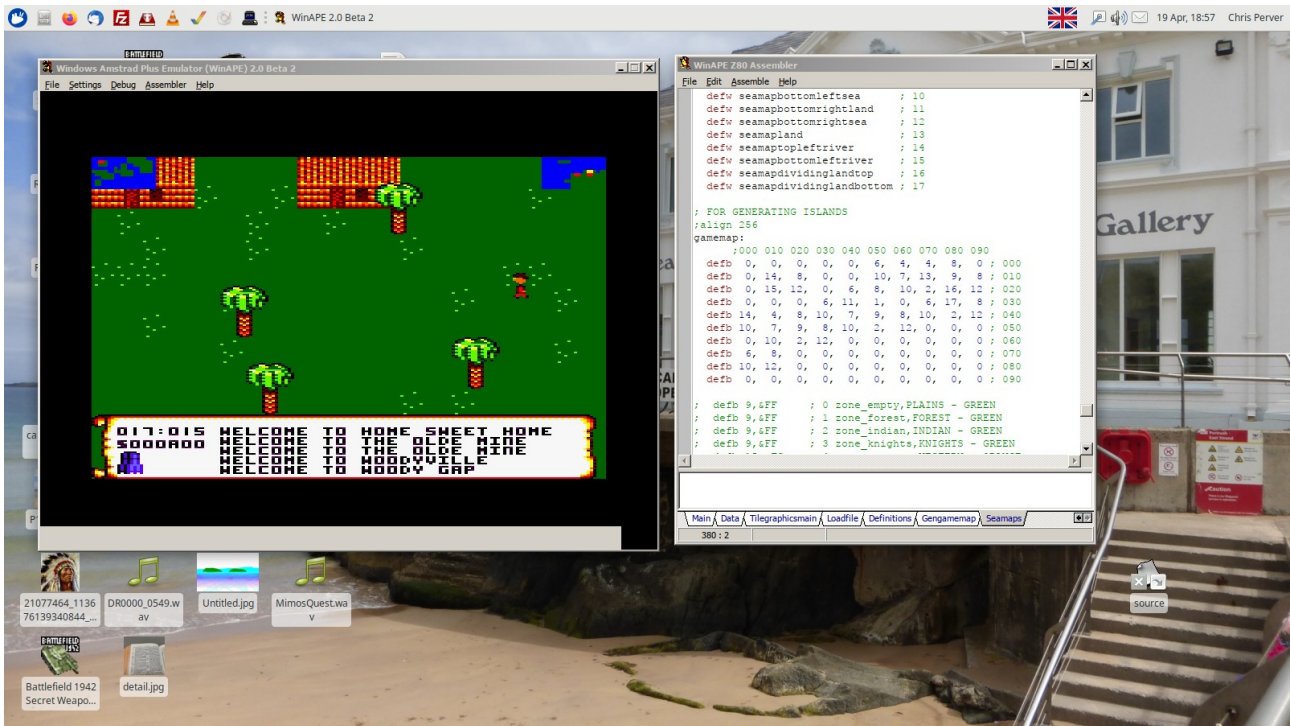
Extra memory found

I had been compiling my start of code at &2000 as I was finding that if I tried to go below that the BASIC launcher would crash the computer. This was leaving quite a large gap of unused memory, and I had been trying to think of ways to use it up. I assigned quite a few lookup tables, quests and character variables to this space, which used up some of the memory, but there was still quite a lot left. But using assembly language to load a file into memory, for example the opening screen, instead of using Locomotive BASIC, I found that I was able to set the start of code at \$1400. This saved quite a bit of memory, around 3000 bytes, which gives me more space for making quests and improving the scenery.



```
WinAPE Z80 Assembler
File Edit Assemble Help
; 253 - NEXT BYTE IS Y POSITION ONE TIME ONLY TO SAVE SPACE
; OPTIMIZED
buildmazebottommidataneu:
defb 254,9,0,2,4,6,8,10,14,16,18 ; Y, X, X...
defb 254,5,253 ; Y, X, X...
defb 254,3,253 ; 0,2,4,6,8,10,12,14,16,18 ; Y, X, X...
defb 254,7,253 ; Y, X, X...
defb 254,9,12
defb 254,5,12
defb 254,3,12
defb 254,8,10 ; Y, X, X...
defb 255 ; END OF LIST
buildmazemiddata:
```

I have also managed to further compress the maze data in a similar way to how I compressed the map data. The byte 253 just repeats the number pattern used in the previous row definition. As the maze is mostly made up of walls of sprites of a similar length, it is very easy to compress them.



Aligning both the local and world maps to boundaries of 256 bytes was taking a lot of memory, seeing the maps are only 10x10 bytes in size now. I am trying to conserve as much memory as possible so I will have more available for the actual quests. In order to unalign these maps, I have to change the lookup function slightly. Instead of just loading the map location into HL, and then changing L to whatever the offset is I need to look up, there are an extra few commands needed to account for the unaligned map. Mainly loading the offset into BC, and adding it to HL, and then looking up the value. This does result in a very small time lag when changing screens, but it is barely noticeable and worth the extra memory it saves – 304 bytes!

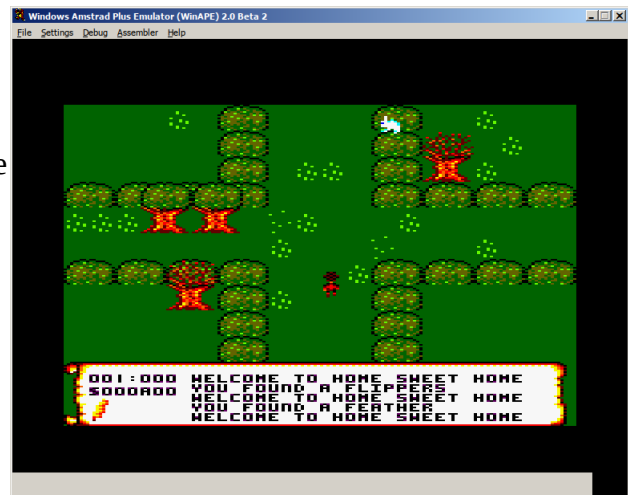
I also managed to save a few bytes in the sea 'tile fill routine'. I use single function to load the tile fill X and Y coordinates, width and height values and tile IDs into the registers from tables, rather than loading them in manually for each screen definition.

Quest objects

A lot of the memory is taken up by the tile graphics, and having a graphic for each quest object is costly. I may decide to make random quests show a generic quest graphic rather than a specific one. This will allow me to make many quest objects, hopefully that can be traded, bought and sold.

Bushes

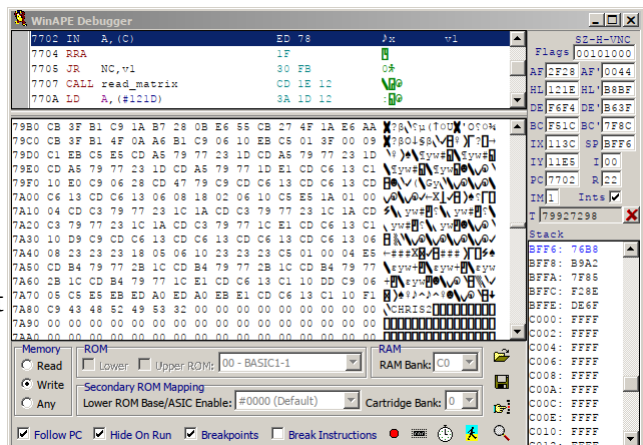
Having large features that span several screens really helps to remove the disjointed feeling between screens, so I decided to try and make some bushes. Taking the top of the big tree, it is possible to make a bush without having to use any extra memory for graphics. I will add random bushes to make the scenery appear a little more natural, as well as making hedge roads to give some structure to the map. Clumping a few bushes together now and again helps the natural look.



Speed up and memory save

Using the assembly commands XOR A instead of LD A,0 (load 0 into A register) and OR A instead of CP 0 (compare A register with 0) offers a significant speed increase and saves a byte each time they occur in the program. I wasn't really convinced of the advantages until I replaced every CP 0 with XOR A and found the tile redraw was probably around a third faster. It does make the program code look a little strange, so it is best to keep in the comments what you are actually doing when replacing these commands. Saved around 20 bytes by doing this.

As I said before, my loader loads the main program code into memory at &1400. I have used the first part of memory for variable storage. Using a linked list of EQU definitions, it is possible to define each variable address in the compiler, and rearrange them as needed. By loading a character into memory at the end of this list of addresses, I discovered I had used up to just under &1200 for variables and lookup tables, so I changed the loader to use &1300 to load the main code in at. I was reserving 512 bytes for my tile lookup table, but I decided to make this a manually defined table, as I was using only around 296 bytes so far. So this has saved me some extra memory too. Using a tag, in this example, "CHRIS2", I can easily see where the end of my program is in the CPC's memory bank. Unfortunately I can't go down to &1200 for the main program code without figuring out how to make a binary file executable.



Binary file executable

I managed to create a binary loader from my assembly code. Looking on the internet, I found the structure for the AMSDOS header. It is this header that tells the Amstrad where to load the file into memory, and which instruction to start at...

LSB = Lowest significant bit, which is the way the Amstrad stores its bytes. This is the way WinAPE stores its variables anyway.

- Byte 00: User number (value from 0 to 15 or #E5 for deleted entries)
- Byte 01 to 08: filename (fill unused char with spaces)
- Byte 09 to 11: Extension (fill unused char with spaces)
- Byte 16: first block (tape only)
- Byte 17: first block (tape only)

Byte 18: file type (0:basic 1:protected 2:binary)
Byte 21 and 22: loading address LSB first
Byte 23: first block (tape only?)
Byte 24 and 25: file length LSB first
Byte 26 and 27: execution address for machine code program LSB first
Byte 64 and 66: 24 bits file length LSB first. Just a copy, not used!
Byte 67 and 68: checksum for bytes 00-66 stored LSB first
Byte 69 to 127: undefined content, free to use

It took a bit of trial and error. Eventually I discovered that the execution address and file length are the length of YOUR code that you want to place in memory. That is, excluding the start and length of the header. Once I figured that out, I could launch the binary file with RUN"". I found I had to add an extra bit on the end when saving the binary file to disc in WinAPE after assembling it.

SAVE "LOADER.BIN",B,&0900,&00FD,&0980

B = Binary
0900 = Start of header code
00FD = Length of code including header
0980 = Execution address for Mimo code

This makes the binary file runnable.

Another problem I faced was when I made the binary file executable, the loader stopped loading the loading screen. Instead of loading it from the disc, it would display the statement "PRESS PLAY THEN ANY KEY:". After asking online, I discovered that when you RUN" a program, all roms are disabled, including the AMSDOS rom. So essentially the 6128 no longer mapped cassette functions to disc functions. In order to overcome this, I had to disable all roms and enable them using the KL_ROM_WALK command at the beginning of the loader code. This enables loading from the disc again. It will be handy to disable all the roms when running Mimo to give us extra memory to work with. Disabling the roms is achieved by calling "ld c,&ff". Although I may need some to do audio.

The next step, once memory starts to get tight, will be to make the loader load the Mimo code into the earliest part of memory as possible. I see a lot of games load at &0040, so I will probably try to do that if possible.

Random Quest Objects

I separated the static quest routines from the random quest routines. The static quest objects were using up a lot of memory for the graphics and I found I was running short on space so I needed a way of having quests that didn't need an image for them and could provide the player with more variety.

As the random quest objects will have different IDs from the static quest objects, I needed to create an extra player inventory for them. Now whenever you bump into a random quest character, they will ask for a random quest object that will eventually be based on the type of zone the player is in. I will need to craft story lines around some of the objects. As the random quest objects can't be picked up on the normal map, it will need to be obvious to the player how to obtain them.



Combining zones

The landscape seemed to be looking a bit homogeneous, with trees everywhere, after creating a manual zone map rather than having bands of overlapping zones based on the map coordinates. So I have decided to make a few extra zones that will combine features of two or more zones again. This is the mountain forest zone which I will put in specific places of the map. I have also made a western mountain zone, the western tropical zone and the western tropical mountain zone. As these zones are defined by lists, it only takes a few extra bytes to combine them. It helps to lift the monotony of the landscape.



From sea to shining sea

I took away the sea sprite, which was just a block of 64 bytes of blue, and replaced it with a command to check the tile number of the tile before it is drawn to the screen. If the tile is the sea zone tile, then a routine will just clear the tile with blue. This makes tile drawing a little slower for landscapes, but actually slightly faster for sea drawing, and saves around 50 bytes.

I also decided to skip tilemap redraw for the screens that are just all sea. Screen redraw is slow when you have to update every tile on the screen. The processor time is being used up by the function to move to the next pixel line in each tile. So it is much quicker to erase the entire screen at once rather than each line in each tile. It could also be done for screens that are part land and part sea but it would be fiddly as you would need separate code for each screen design and the memory could be put to better use.

Quest lists

I figured that if the player accepts a number of different quests at the one time (five can be selected at once), it may become confusing to the player which character is looking for which item. I amended the status bar print line function to be able to convert a map digit to a string using the tag byte 14. If the player presses Q, a list of coordinates and their associated random item will appear on screen. So the player can be reminded what they are meant to be looking for and where to return it to.

I still haven't figured out how these items are going to be acquired if they can't be picked up on the map.



Random quests outdoors

To improve the variety in the western zone, I decided to change the random quests routine slightly so that random quests can be generated outdoors as well as indoors. Sometimes a campfire will be generated instead of a western shanty town or Indian village. When this happens a random quest character will be able to appear. After all, many a good adventure begins with a campfire story!



I have been thinking about how to create more random quest types. As you can see from the status bar, it is possible to count time. If a time byte is added into the random seed generator, it will be possible to make the characters more interactive, by changing what they say every so often.

I was also thinking about the types of quests Elite II Frontier generates. I always wondered that when you accepted a quest that related to a space ship being in a certain time at a certain place, how Elite knew that ship would be there. The only way I can think of it being done is to make a list of quest characters, and when a quest is accepted, a character is added to the list with a specific map coordinate, which will be drawn when the player reaches that particular destination. Of course it will seem a little strange, this character appearing out of nowhere, and then disappearing when the player completes the quest. Obviously with Elite this wasn't a problem, as the player wouldn't know the ship wasn't there previously.

Slowing text display

One of the problems that needs resolving is if you bump into a character they will repeat over and over again the same statement until you stop pressing the direction key. There are a couple of ways this could be overcome. It is possible to put a short delay in the function that writes each letter to the screen. This delay is often used in games to make sure the player has read everything that has been displayed on screen. I tried putting in a couple of `wait_frame_flyback` calls between each letter, but I found that this had the adverse affect of pausing everything in the game, including the player's movements and screen redraw. The text output function would have to be running asynchronously, which would mean making a function that would output text to the screen in much the same way the computer character sprite movements are updated. That would entail storing the current position in the text that is being printed, and keeping a record of what is being printed, etc. It would take a bit of work and use up a bit of memory too. Another possibility is setting a key repeat delay, but this might affect the player's movements. The only other possibility would be to make the character speak only when you press a specific button when you are standing next to them. For instance, the space bar, and make sure there is no key repeat on it. But then I would need a function to check if the player is standing next to a character. So it seems a little complicated at the moment.

Storylines

Another thought I had is regarding the storylines. I used to love reading books where you could choose your own ending. It may be possible to generate certain quests based on the player's previous interaction with game characters. What advice characters tell the player could change based on player responses on previous occasions. This might help to create more of an atmosphere in the game, rather than everything seeming to be a little random. For example, one quest could be to find the lost treasure of Captain Blackbeard. If the player expressed an interest in this quest, other

randomized game characters could give helpful advice on who Blackbeard was, where the treasure is, how to get there, etc. Of course this advice would only happen sometimes, there will still be characters that have no interest in helping the player, and are only looking for their own quest to be solved.

Randomize speedup

Getting a random number between 0 and n is achieved by calculating a random number between 0 and 255 and then subtracting n until the carry flag is set. Then n is added back on to get the random number. This works but the routine is slower the larger the initial random number. I created another random routine, rand32, which calculates a maximum initial random number of 32 before subtracting n. This speeds up random sprite insertion and a few other routines. If we can be sure we won't need a random number greater than 32, for instance, for a tile number (max. 20 wide, 10 high), then I use rand32 instead. Rand32 uses AND %00011111 to blank the high bits of the byte, making sure the maximum number is always 32 and not 256.

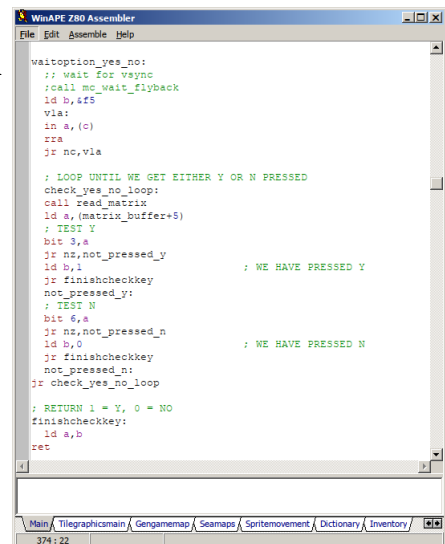
Waiting on key presses

I created a custom routine for waiting on keypresses from the player for Y/N and 1,2,3 for the dialogue routines. While this will take up extra memory, it means I don't have to rely on the ROM routines. This will hopefully mean I will be able to use the memory up until the stack pointer. I may also need to look at making my own routines for the sound effects, as these also currently use the ROM routines, though I am not yet sure how to do this.

Key release

One way of preventing masses of text scrolling when talking to characters is to wait for a key release before printing the text. The keyboard routine on the CPC is quite complicated to understand, as the keyboard is wired in through the sound chip, and you have to talk to the chip to know what is being pressed at any given time. There are 9 'lines' of 8 bits each, making up the full keyboard map. Checking each line in the memory against a bit number tells you whether that particular key is being pressed. This is made more complicated by the fact that the bit will be 1 if the key you are checking for is *not* being pressed. So it all gets rather complicated.

If you bump into a character, you don't want them continually repeating what they have to say for as long as you hold the direction key. It makes the text hard to read and looks silly. So I managed to add a bit to my keyboard scanning routine. An extra subloop does a check against all keyboard lines until they all equal &FF - which means no keys are being pressed. This allows me to make sure the character text is printed to the screen just once rather than hundreds of times as the player holds down the direction key. Checking all the lines to make sure no keys are being pressed takes up much less code than having to check for the specific key the player has just released. Though it does mean control will not return to the player if they happen to be pressing another key at the same time. I have also added a similar routine to the print paragraph function. If there are more than five lines in the paragraph, the player will be asked to press space before the rest of the paragraph is displayed. Again, waiting for key release is necessary to prevent multiple spaces being recorded.



```
WinAPE Z80 Assembler
File Edit Assemble Help
waitoption yes_no:
:: wait_for_waitnc
:: call mc_wait_flyback
ld b,4f5
via:
in a,(c)
rra
jr nc,via

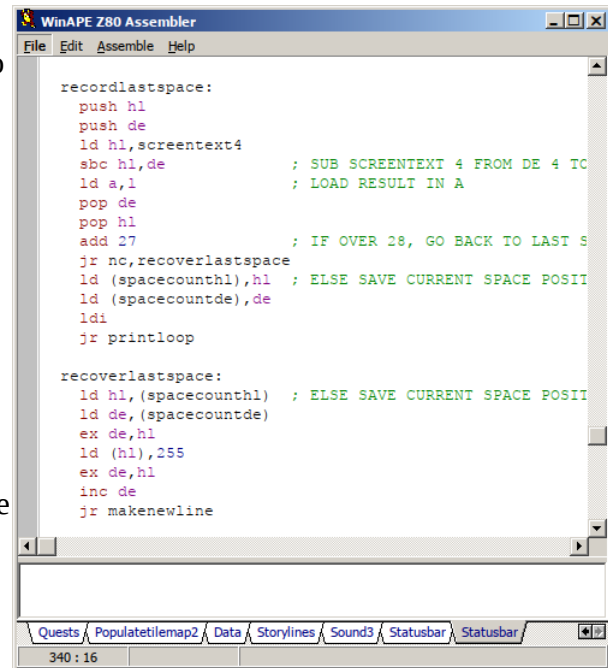
; LOOP UNTIL WE GET EITHER Y OR N PRESSED
check_yes_no_loop:
call read_matrix
ld a,(matrix_buffer+5)
; TEST Y
bit 3,a
jr nz,not_pressed_y
ld b,1
jr finishcheckkey ; WE HAVE PRESSED Y
not_pressed_y:
; TEST N
bit 6,a
jr nz,not_pressed_n
ld b,0
jr finishcheckkey ; WE HAVE PRESSED N
not_pressed_n:
jr check_yes_no_loop

; RETURN 1 = Y, 0 = NO
finishcheckkey:
ld a,b
ret
```

Word wrapping text

If there are going to be a lot of quests, it could be time consuming to have to work out the best place to put line feed markers in the text. I also thought they could end up using up quite a lot of memory which might be put to better use. So I developed a word wrapping routine for the text scroll. I initially thought it would be too complicated to achieve in a small amount of space, due to the fact that I have functions to insert random word objects into the text on the fly, but it has actually worked out quite well and will save memory.

I did this by inserting a new character check routine. Every time a space is encountered, the read and write positions of the string (HL and DE) are recorded. The current line length is calculated by subtracting the write position from the start string position in memory, and if the result is more than the width of the viewer, the previous space position is reloaded and a new line is forced. This achieves a word wrapping function without having to worry about random words that are being inserted on the fly. As there is no SBC DE,HL command, I had to do SBC HL,DEL and expect a negative result for the string length. If adding 27 to this does not set the carry bit, then the line is too long. A separate check also needs to be made on the very last word in the text. Because there is no space at the end of the text, the wordwrap function may not be triggered when it needs to be.



```
recordlastspace:
    push hl
    push de
    ld hl,screentext4
    sbc hl,de          ; SUB SCREENTEXT 4 FROM DE 4 TO
    ld a,l            ; LOAD RESULT IN A
    pop de
    pop hl
    add 27             ; IF OVER 28, GO BACK TO LAST S
    jr nc,recoverlastspace
    ld (spacecounth1),hl ; ELSE SAVE CURRENT SPACE POSIT
    ld (spacecountde),de
    ldi
    jr printloop

recoverlastspace:
    ld hl,(spacecounth1) ; ELSE SAVE CURRENT SPACE POSIT
    ld de,(spacecountde)
    ex de,hl
    ld (hl),255
    ex de,hl
    inc de
    jr makenewline
```

The disappearing Mimo bug

I was trying to work out for a long time why Mimo would sometimes disappear when after talking to a quest character. I first noticed it happening when you approached a character diagonally. So I thought it must be something to do with sprite movement, maybe something to do with the fact that the character moves vertically and then horizontally to achieve a diagonal movement. But I couldn't find anything wrong in the code. I tried disabling the scrolling text, I tried pausing the emulation and examining memory locations associated with the sprites in case they had been corrupted. It made no difference. Then I found a quest character that would cause Mimo to disappear no matter what direction you approached from! Eventually I found that shortening the length of the quest text would stop the disappearing act from happening! So I thought, it must be something to do with word wrapping! Again I tried disabling various functions to no avail.

Eventually I found the bug. The scroll text buffer was located right next to the 'in building' variable, and this variable was accidentally being overwritten by the text. Whenever it was corrupted, the program thought Mimo was in a building and he would disappear from the screen! Or if he was already in a building, suddenly rabbits would start appearing on the screen! So the moral of the story is, always make sure your buffer isn't being overrun! I have moved the 'in building' variable and made extra space for the scroll text buffer, as due to the way the word wrapping works, sometimes it does need to overrun.

Fixed graphic corruption when picking up items

I have been thinking all day about how to solve the problem of graphic corruption when picking up items. Initially I thought if I converted all the tile items to proper sprites like the rabbits, it would be possible to set the collision detection in such a way so they would disappear before the player made contact with the pixels. I set the collision detection on the rabbit meat to 'graze' instead of 'kill' which means Mimo will pick up the rabbit meat before he actually makes contact with it, thus preventing the screen getting trashed.

There are a couple of problems with turning all the tile object items into sprites. One problem is, due to the limited number of buffer slots available, if a rabbit sprite happened to be activated before an object was and there were no more buffer slots left, it could prevent the item from appearing on the screen at all. The code to generate the list of on screen sprite objects would also be more complicated. These would need to be regenerated every time you move screen, and tile locations would need to be converted to pixel locations, etc, etc.

Then I was thinking, even if each tile object had its own background buffer to paste when the item is collected, that still wouldn't resolve the issue. The problem is Mimo's background buffer image. Because I am saving memory by not having a double screen buffer, I have to take a copy of the part of the screen Mimo is moving to before pasting him there. If Mimo happened to be moving over an object, even when the object disappears, parts of it will be left behind when Mimo's buffer is redrawn. Then I realized how to solve the problem!

Even though someone had told me this solution before, I never really thought about it for some reason. To achieve a complete redraw of the original background image before any sprites have been drawn, the sprite buffers need to be pasted to the screen in reverse order. By restoring Mimo's background buffer first, and then redrawing the tile on which the object was placed, graphical corruption is avoided completely. It is definitely a big improvement, and only uses a few extra bytes of code!

Improvements to status bar

The status bar could get rather cluttered looking with old text that the player has already read. I found that by scrolling it up to clear it before any new conversations began made it much easier to read. The player will be asked to press space before responding to a character. This will give them time to read the text. The player will also be asked to press space if there is more text to be displayed. It will not scroll up all the time, as it can get annoying, for example, displaying only a place name or picking up an object.

I have also a good idea of how to do the shop now, with the player being asked whether they want to buy or sell, and then being shown a list of goods. Pressing a number on the keyboard will buy or sell that item. Items will have slightly randomized prices to allow trading. Some items will be able to be bought for fulfilling quests.



Storylines based on zones

I have now segregated the storyline responses based on zones. This will enable me to develop a clearer story line for each island, and also will help to guide the player as to what to do next. This was done using a 'lookup table of lookup tables', i.e., the first lookup table contains a list of table addresses that are selected based on the current zone the player is in. The sub-tables are the ones that contain the actual responses that will be displayed to the player. So far there is a standardized form for the responses, "Tell me more about..." and "Where can I find..." which should cover most situations while providing a bit of variety and interactivity for the player.

Eventually I will segregate the random quest items by zone, which will make things a little more themed for each island.

Themed random quests

Random quest items have now been themed by zone. It's amazing how much difference this makes to the gameplay. For example, Indians will now seek to trade tobacco, and cowboys will ask for beans and shooting irons. It's very easy to add new items per zone. I need a way of being able to purchase these items to fulfil these quests. Once the player is in possession of a few objects, it will make it easier to get started into the game. I will possibly theme random hellos by zone, so that characters give specific help related to the current player location. Another possibility is adding the time factor into the random quest generation, to make the world seem more living.



Shop title

Generating specific signage for the livery stable / shop front / any other building takes up quite a bit of memory, 64 bytes for each tile. So I was thinking if I recorded the position of an inserted sprite on the screen and just used the text printing routine to print the sign in ordinary text, it would save quite a bit of memory.

It was a little complicated to do because of the randomization of the screens. We never know if a shop is going to be generated until it appears, so we need to set a variable that tells the sprite insertion function when we want to record the insertion point of a particular sprite, in this case the shop. After the tile map is drawn to the screen, we check to see if an insertion point was recorded. If it was, we convert that tile number to a pixel location on the screen, print our shop sign, and clear the position so we don't print it again by accident on another screen.



Night time

Some games like Harvest Moon and others use a night time effect to make the game world appear more natural. I thought I would try a night time palette to see what it was like, and it seemed to work well. So I decided to run a timer and make the palette change from day time to night time. It took a bit of fiddling, as I found when I did this, some pens were not changing properly. I had to change the tile map drawing function so it always uses transparent tile drawing. This slowed down tile drawing slightly, especially when there are a lot of sprites, but it also saved me some memory as I was able to remove the tile drawing routines that are no longer used.



I tried to make a graduated effect rather than changing from day to night suddenly like Werewolves of London. The sudden change in that game coupled with the dramatic music still gives me chills! Due to the limited colour palette (15 out of 27 and most look similar), it is not really possible to achieve a graduated darkening effect, but changing one colour at a time does make it look like the sun is setting and the dark palette does seem to work well.

The day and night periods will not be in equal measure of course, as the change could get annoying. So perhaps 2-5 minutes for night time and 10-15 for day time will be ideal. The fact that there is a night time and 'days' can now be counted in the game can help develop the atmosphere. Buildings will close at night. There may be an inn where you can find a place to sleep and speed the passing of the night. And quests will change from day to day, giving a little more variety.

Stone wall

Making small buildings is not really memory efficient as the tiles cannot be reused for other objects. It would be nice to be able to build objects out of a single block like in Minecraft, but as this does not really look great in two dimensional Amstrad graphics, we have to improvise. It is also helpful to build objects that span several screens as this really helps to reduce the disjointed feeling the player gets when moving around randomly generated screens. So I decided to build a natural stone wall that could be used to build castle gardens and estates.



I tried to do this once before but couldn't get the tiles to repeat properly on the back corners. I ended up creating far more tiles than I wanted to and it didn't really look right anyway, so I gave up. This problem was caused because I had made the wall thickness half the width of the tile, which meant I had to make lots of extra tiles to bend the wall left and right. This problem disappeared when I made the wall thickness the entire width of the tile. I can now bend the wall left and right without having to create special tiles to do it. I also made the highlights and shadow on the stones omni-directional, so that when the tiles were flipped, they did not look out of place.

The stone effect was achieved by copying and pasting roughly circular stones, around 4-5 pixels in size, and then randomly altering single pixels here and there without worrying too much about the size of stones that were created. This makes an unevenness which the eye interprets as a natural effect. In order to allow Mimo to enter through the archway, I had to combine two sprite definitions

together. One sprite definition creates the entire back and side walls, with the upper level allowing Mimo to walk behind it. The second sprite definition creates just the front portion of the wall with the archway. This is pasted on top of the first sprite definition, so that the archway appears on the upper tilemap and Mimo can walk under it. I had to do this due to the way my sprite definition function works. This function can only move from the lower object level to the upper level once. To be able to specify single tiles as lower or upper level would take up a lot more memory.

Shop

It took a bit of work and 500 bytes of space, but the shop is now mostly functional. I used a table to record world map ID, local map ID, random stock id, quantity and price. This table is populated with around five items of random stock items when the player enters the shop. There is space in the table to record the stock of a number of shop locations, so the stock should be fairly persistent. If the shop runs out of a product, a new stock inventory will be generated the next day and the out of stock item will be replaced.



I had to use a second lookup table of stock items to record the memory location of the stock item the player is interested in. When the player responds to the shopkeeper by pressing a number, this table is read to get the true memory location in the first table of the item, its price and quantity. If the player has enough money and inventory space, the quantity is deducted and the item added to the player's inventory table.

The player will be able to sell products to the shop. Prices will change depending on demand. If an item is running low in stock, the price will get higher, which will enable the player to build up money by buying cheap products in one shop and selling them to another shop where stock is low.

Random quests changing on day by day basis

With the addition of the day and night times, it is possible to do lots of things that make the environment seem more natural to the player.

When the player completed a random quest, it was quite annoying for them to bump into the quest sprite by accident, only to find they are looking for the exact same quest object again. This happened because I have to clear the memory of completed random quests in order to make room for new ones. The most obvious time to do this was when I was switching screens.

Having day and night periods, it is now possible to clear the completed quest list the following day. This not only makes switching screens far faster, the random quest character will remember for longer that you just completed their quest, and should ask for a different object the next day. A small effect, but makes the game world a lot more believable.

Buying useful items

I thought it would be interesting to allow the player to be able to buy useful items from the shop instead of just picking them up or obtaining them from quests. This will allow the player more scope in the game to do what they want. I have made the arrows one of the items it is possible to buy from the shop. These will be cheap and plentiful of course. I had to amend the arrow routine so it would deduct arrows from the player inventory when they were fired. I will also allow the player to sell arrows back to the shop. I will possibly allow the player to buy the climbing rope and flippers too.

As I no longer needed a space on the status bar to show the current arrow quantity, I removed this, and thought to expand the player's purse from an 8bit number to a 16bit one. This will allow more expensive items to be purchased, which will greatly increase the possibilities for game play. For instance, the player could have a goal to purchase a new house, or island, and allowing the player to trade to achieve this could add an interesting aspect to the game.

Merging static and random quest data

When the player bumps into a quest character, whether they are static or randomly generated, there is a routine that runs through the list of player responses and whether the quest has been completed or not. Originally I created the static quest routine, which worked quite well, but as the game developed it was becoming clear that in order to fill the world, I would need some sort of random quest generation routine.

Having developed that, I was finding that I now had two routines that essentially performed the same function, one for static quests and one for randomly generated ones. My statically generated quest data was also split in two, one table for the sprites and one table for the quest data, whereas my randomly generated quest tables contained both the sprite and the quest data.



I was conscious about doubling up on the static quest and random quest routines, when they were essentially doing the same thing. It seemed to be taking up quite a bit of memory and having to maintain two separate quest routines was a bit of a pain. So I modified the static quest tables into the same format as the randomly generated quest tables, and just pointed the appropriate routines at this data. Seems to be working and saved maybe 100 odd bytes.

I also removed the graphics for some of the static quest items, such as the peace pipe, lost son, etc. I changed the static quest item routine so it uses the random quest object definitions, so it saves having to make images for each quest item.

More tradable items

Local and global maps can now be bought and traded. It should make the game a little more interesting by providing objects that the player can save up for that actually have a use in the game, rather than just being a means to make money with.

Pressing M will now toggle the maps correctly. I had to put the routine in to wait for no keypresses after toggling the map, otherwise it would toggle on and off really fast as long as the player held the key down. The local map should be available to the player on the second or third island, enabling them to find their way around. The global map may become available later on in the game when you have to swim to remote islands to complete some quests.



No western would be complete without a saloon, so I created an extra shop title function to record the last insert position of the saloon sprite. The player should be able to go into the saloon and talk to different characters. Hopefully I will be able to optimize this routine so any shop title can be created with the same function.

Blocked entrances

Due to the random nature of sprite insertion, sometimes it is possible for an entrance, like in the above picture, to be blocked by another sprite. Sometimes a tree or cactus would grow in front of a doorway, and if the building was a shop or something that the player needed access to, it could be very frustrating not being able to get into it.

I amended the sprite insertion routine to cater for this. If a 252 tile number is encountered, the sprite insertion function will create a traversable object of grass or dirt at that location. This means if you position this tile just outside the doorway, the player will still be able to access the door, but as it is marked on the object tilemap as occupied, nothing else will be able to grow in its place.

The random position of the houses also did not look right. For wigwams and stone houses, a random location is suitable, but for the western style houses some sort of street layout is required. So I made a table of acceptable tile locations for town buildings, and the sprite insertion function will use this to plot the locations of the buildings for the western towns. A little variation of one or two tiles is allowed just so it does not appear too fixed.

Running out of quest space

I discovered a bug, that when the player has accepted five random quests and has not yet completed any of them, and there is no more room to create new ones, that random quest characters stopped appearing in the rooms. This was due to the sprite data and quest data being combined in the same table. Since there was no more room to create a quest, no more characters were generated. There are two solutions to the problem.

1) Separate the sprite data from the quest data. Quest data could be generated whenever the player bumps into the quest character, rather than when they enter the room. Then an unlimited amount of quest characters could be generated on a screen without taking up extra memory. It would take quite a bit of reworking the code to do this though.

2) Reserve an overflow, so if all the quest slots are filled, the reserve slot will be used to generate a random quest character that will just speak to the player. This is the solution I have opted for in the meantime as it was the easiest to do. We are not likely to need more than one quest character per screen. It could just become confusing for the player. And we can have other non-quest characters on screen generated using a different method.

Pseudorandom rooms and tilemap bug

Another bug I found which caused a bit of searching to find was causing the rooms to be different depending on which direction the player entered them from...

I discovered a couple of things while debugging this one. I found the quest chooser was being called every time you bumped against an object, even if it was a brick wall. It wasn't causing any slowdown, but it wasn't really efficient either, so I stopped that from happening.

Moving diagonally is achieved by moving a sprite left or right and then moving it up or down. This method saves memory and also allows Mimo to 'slide' against walls rather than just coming to a stop. This caused a problem though. The quest chooser was being run both times when moving diagonally. I had to make a delay by making a trigger, like the tile map redraw trigger, that would run the quest chooser after we had finished moving. I also found that the sprite collision detection routine was also overwriting the registers where I had stored my tile ids, so I couldn't just run the quest chooser function without storing the tile ids as well. Due to the diagonal movement, it was also possible for Mimo to bump against a doorway tile, move on to an adjacent tile, and then trigger the quest chooser routine. This would mean Mimo was on a different tile from the doorway, which caused a different room layout to be generated! So I ended up storing all of the tile data for the doorway and just recalling it when we run the quest chooser function.

I also found loading BC into an adjoining memory address for two different variables was a handy way of saving space and time.

I discovered that preventing the regeneration of shop inventory to the next day was not working when you travelled to more than one shop, so I had to disable that feature. Now shop inventory will be 'topped up' when you speak to the shopkeeper if the inventory is less than 5 products.

Keypresses

Checking for keypresses can use up a lot of bytes in code. When the player is buying or selling, we need to check keypresses for the numbers 1-5, but we also need to ignore certain keypresses if the player happens to have less than five items. Rather than making five different keypress routines for checking these keys, a maximum-input can be specified. Keypresses for 1-5 are checked as normal, but before returning the result, the maximum input byte is subtracted from the value of the key that was pressed. If it carries, then we know we have pressed a key for an item that the player does not hold, and the keypress loop is called again.

Community events

In order to make the game world more feel realistic, a sense of community and atmosphere needs to be formed. In games like Elite II, there are errands that can be run to local systems. In Harvest Moon, there are seasons and festivals on certain days, in which you get to interact with a lot of different characters.

As I already have numbered days, I thought it might be an idea to display this on the status bar, to enable the player to have a sense of the passing of time, and possibly have date sensitive dialogues.

Like Harvest Moon, there will be four 'seasons' of thirty days each. It should be possible to have characters that will say specific things on certain days, for instance a 'town-crier' that could provide the player with bits of information that could form a storyline.

As memory is limited, I may think about developing a system of reading a time sensitive datafile of text from the disk when the player bumps into a certain character. Of course this would make a cassette version of Mimo unworkable. But I will see what happens!

Disk conversations

I have managed to create a routine that can read a block of data into a specific memory address when you bump into a specific character. The filename of the data file contains the season number and the character ID. The file contains a list of pointers to text strings for each of the thirty days in the season. Using this method it will be possible to develop characters that will guide the player in a storyline, to undertake various quests, announce certain events, etc.

I tried the function out on Mimo's mum for the first three days. It worked successfully. There is only one drawback, and that is disk access times. There is a two second lag between bumping into the character and displaying the conversation. I can cache the data, so if Mimo bumps into the character again, disk access won't be needed. So hopefully it shouldn't be too annoying.



I have added another function to the character conversations. A 0 or 1 byte at the start of the file will determine whether or not to choose a phrase based on the day number - for fixed storylines, or whether to choose a random one from the list. The random function will perhaps make some conversations appear more natural to the player, so if they bump into them a couple of times, a different phrase will be displayed.

Weekdays

As I am recording the passing of seasons, I decided to record week days too. Shops and saloons will be closed on a Sunday, which will add a bit of realism for the player.

Conversation loader

After a bit of thought, I have opted for a "day by day" conversation loader. Alternative options were:

- 1) Storing each character's conversations in their own file. This would have worked well, minimizing the number of files I need to create, but disk access was slow and it would be activated each time you bumped into a different character.
- 2) Storing all conversations in the spare 64k and loading it at the start. This would have worked too, but if I am planning to have some 30+ story characters in the game, with a saying for each day, it would have filled up quite quickly.

The day by day loading scenario will save disk access time, and it will only happen once per day, and only if you bump into a story character. It does mean I need to create some 120 data files on the disk though!

Island hopping

Having multiple islands with different topology makes the game more interesting for the player, and it also provides more material for various storylines. But getting from one island to another can be problematic. There is a need to restrict the players movements so they can't explore places that should only be available later on in the game, so we must devise different ways of getting from one island to another.

The abandoned mine, with a tunnel leading under the sea like in the Goonies film, was the first method we used. Then we had a tunnel from a house in the maze to a stone house in medieval world, a bit like the Narnia films. The New World, which has a cowboy and Indian theme, should also have a thematic way of getting there. So I thought of making a boat that would sail there.

Memory constraints make it difficult to create something that looks like a boat and would actually move, but if we make one of these at either end of the journey it should be possible to magically transport the player to their destination some time after they pay the boarding fee. Self modification of the static map will allow me to change which docks are empty and which have the ship at them. The player will be able to ask the captain to go to a certain dock in exchange for a fee.



Dungeons

I created an extra corner stone wall. I already had an inside corner, which I could flip horizontally and vertically to make a room in a stone building. But I had no outside corners, to make corridors and dungeons. So I erased the rainbow sprites to make room, and I will see if it is possible to have a dungeon spanning several screens.

At present a dungeon is not possible, due to the way the static map works. As the static map is a flat map, it is only possible to define one level. The insides of the buildings are single rooms, they do not allow you to move screens. The code only checks if you are inside a building, and if so, it draws the room no matter what map location you happen to be in.

I would need to create two static map tables, one for ground level and one for below ground. The code would check to see if the player is above or below ground, and pick the appropriate static table to use. It should not be too difficult to accomplish.

Further tile compression

I have implemented a further level of compression to the sprite tile definitions. If 5 is found in the row, the previous row is copied from the screen to make the new row. This will allow 'long' sprites to be compressed a bit. As there was a little extra code needed to accomplish this, I am not sure how much memory it will actually save. It may also not work too well on sprites that have gaps in them, as the routine works by copying what is on the screen rather than copying the sprite definition data. I had to do it this way, in case there are numerous rows of 5s, and it would be time consuming to move backwards until you find the row data you are looking for.

Dungeons

I have created an extra corner wall graphic. Before I could only have a square room shape in the stone graphic, but now I will be able to bend the wall to make corridors. This should enable the creation of dungeons, like in Zelda.

The plan is to make two static map tables, one for buildings above ground and the other for below ground. Then Mimo should be able to discover hidden treasures and visit secret locations when the dungeon entrance is uncovered.



Room levels

After creating the dungeons, I decided to add basements to some of the houses, where I could hide various items. But this complicated things a little when entering and exiting buildings. When Mimo enters a building, the program remembers his last position outside, as well as the tile location of the front door for pseudo-randomly generating the building contents. As I was adding an extra level to the house, I therefore needed to remember two locations, one for outside the house, and one for inside. Rather than use a lot of extra code to remember these positions, I made a table of 'levels' containing the X/Y pixel position of Mimo as well as the door tile location. Each time Mimo enters a building, the program stores this information in the table. And whenever Mimo leaves a building, the last entry is removed and the one before it is recalled. It saves a lot of memory and complicated code, as the same routine can be used to remember any number of levels we need.

Speed up using self modifying code

Previously I had tried to minimize the amount of code needed by reusing the transparent tile draw function for both the lower and upper tilemaps. Only the upper tilemap really needs transparency, so using the same code for both saves memory but also slows the screen redraw slightly.

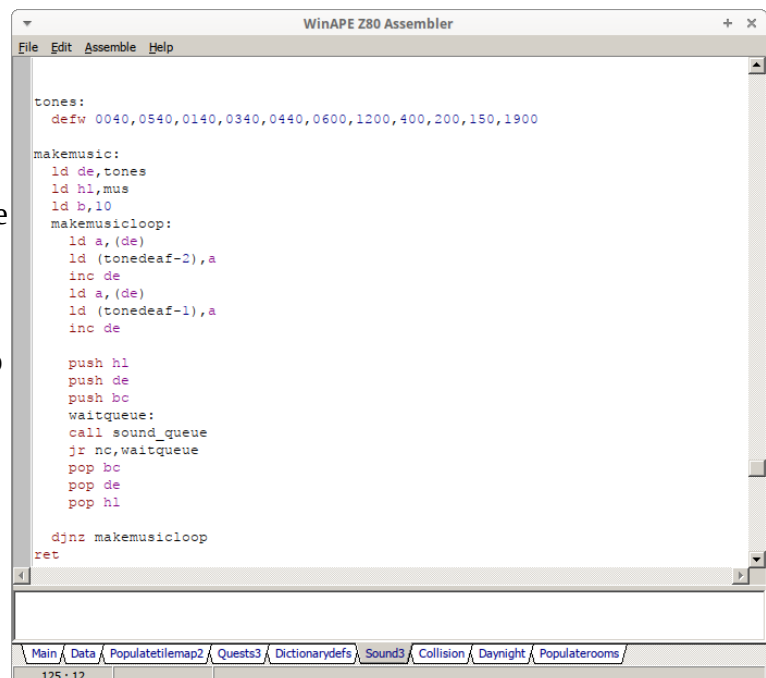
I was trying to think of a way to reverse that, and have the lower tilemap drawn without transparency again but without using up too much processing power to do the change. I discovered that I can switch the tile drawing function using self-modifying code before I draw the tilemaps. Loading the function pointer we want to use into BC, and then pasting it over the top of the function pointer in the CALL command used to draw the tiles, it is possible to change the function used before the tilemap is drawn.

Sounds good

I have finally managed to understand how sound works on the CPC. I have made a table of tone envelopes, amplitudes and tone periods for each sound effect needed, for example, firing an arrow or knocking a door. The routine works out which sound to use and loads it into the sound queue.

Tone and amplitude envelopes are set up as follows...

- 1st Byte – Number of sections
- 2nd Byte – Step count
- 3rd Byte – Step size
- 4th Byte – Pause time
- ... repeat above three bytes for number of sections required.



```
WinAPE Z80 Assembler
File Edit Assemble Help

tones:
  defw 0040,0540,0140,0340,0440,0600,1200,400,200,150,1900

makemusic:
  ld de,tones
  ld hl,mus
  ld b,10
makemusicloop:
  ld a,(de)
  ld (tonedeaf-2),a
  inc de
  ld a,(de)
  ld (tonedeaf-1),a
  inc de

  push hl
  push de
  push bc
waitqueue:
  call sound_queue
  jr nc,waitqueue
  pop bc
  pop de
  pop hl

  djnz makemusicloop
ret
```

These define the waveform of the sound, attack, decay, etc. Then the pitch of the sound is defined below.

- 1st Byte - CHANNEL STATUS
- 2nd Byte - VOL ENV

3rd Byte - TONE ENV
4th and 5th Byte - TONE PERIOD
6th Byte - NOISE PERIOD
7th Byte - START VOL
8th and 9th Byte – DURATION

These are loaded into the queue by calling `sound_tone_envelope`, `sound_ampl_envelope`, and then `sound_queue`. If the sound queue is full, we need to wait until the queue is free otherwise the sound doesn't get played.

It is possible to create music by stringing a number of pitches together, and adding them to the sound queue. A function would need to be made to loop through these pitches and add them to the appropriate sound channel queues as needed. Obviously that can slow a game down a bit if you are having to continually check if the sound queue needs to be filled. Working out the pitches and durations will also be difficult enough in order to create a good tune.

Character spawning

Spawning the rabbits and tumble-weed was a little inefficient. I was calling a randomize function for X and Y pixel coordinates, and not checking to see if they were spawning on top of a building or in the sea. It didn't really look very good, and it is surprising just how unnatural the game world seems when things don't spawn correctly. It breaks the illusion.

Instead I call one randomize function based on tile numbers. I then check to see if the tile is empty space. If not, then we don't spawn anything. Just in case there are actually no empty spaces. We could get stuck in an endless loop on a screen with mostly ocean tiles. If empty space has been found, I convert the tile number into pixel coordinates. At least the rabbits don't drown now. Remember, no animals were harmed during the making of this game!

Inlay card

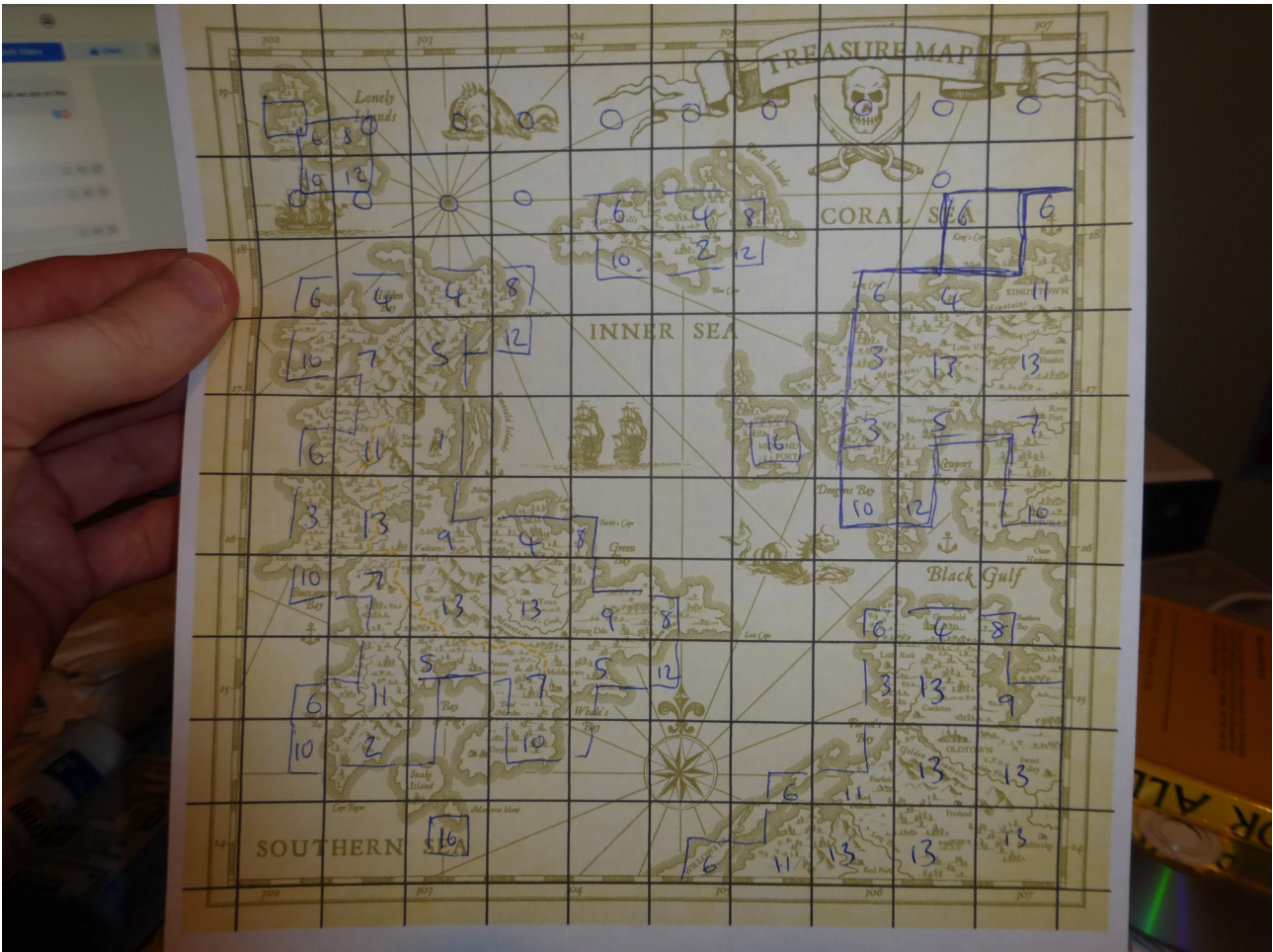
I lost the old inlay card design on my harddrive, so I had to start from scratch again. I was trying to create a cover design which would be eye-catching and would convey something of what the game is about. It needs to draw the player in to the game. So I tried designing a nice landscape scene in Gimp, but without much success. It wasn't really working. You couldn't really tell anything about what the game was about by a landscape, and I wasn't very good at the designing part either.

I was messing about with the mouse, and happened to paste a green brush all over the inlay cover. The result happened to look a bit like a map, so I thought, what a great idea. Get some sort of old map and put it on the cover. It will draw the player into the game and also provide a little backstory for the game itself.



Treasure map

I found the treasure map on a clipart site on the internet. I bought the scalable version, and scaled it up really big. I drew a 10x10 grid on it in GIMP and printed it out. I need to assign tiles to each of the squares in order to recreate the map in the game. I did try a couple of different grid sizes, but I found that changing from 10x10 caused problems with my map coordinate functions. So I decided to keep the 10x10 size. Once I recreate the map in tiles, I can concentrate on building the towns at the correct locations.



Artificial intelligence

I was watching a video on Youtube on the Little Computer People (it's a game I never played!), about how they did the little man that types letters to you.

So I am working a little bit on the random conversations. I think having a fully fledged story line for each character for each day in the year will be too cumbersome, so having a little randomized conversation that makes some sense might be better.



So far I have combined the inventory list with a few choice phrases based on whether it is a food item or a tool. I may add other criteria to make it a little more informative for specific items that can be used in the game.

I also tidied up the random quest conversations. Having a specific set of phrases for each tradable object was not working out. For instance, if a cowboy was looking for an object that was normally associated with Indians, he would speak like an Indian. Now the set of phrases for tradable items is based on the zone the player is in. So if a cowboy is looking for arrows he will use cowboy speech, and if an Indian is looking for arrows he will use Indian speech. This also means I can have tradable items that overlap, i.e., can be traded in multiple zones.

Random towns

I had been looking at how Elite created its universe, using a Fibonacci numbering system. I wanted to try and expand the horizons a little bit without using up too much memory. I was amazed at how David Braben managed to squeeze the Elite universe into 22kb of code. But the randomization routine doesn't really suit my grid method of creating the world. In Elite, the galaxy is created using a 6 byte seed, and these bytes are 'twisted' for every new star system that is to be created. Using this method it is possible to create a lot of systems together, but in order to get the 'seed' of the location you are in, the bytes have to be twisted for the correct number of iterations each time. This wasn't practical for the grid system. In order to get my town seed, I take the coordinates and mix them into the world seed. This is a much quicker method, although it does mean that game characters won't know the specific location of other towns without keeping a static list of them. I would like to be able to create quests where you have to deliver items to other towns, though memory is very tight.

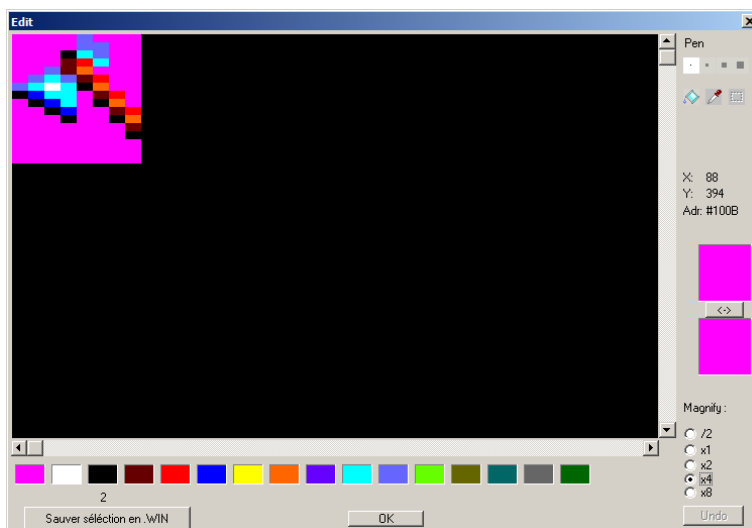
Chopping trees

I thought of a way to make the world more interactive, and that is to be able to cut down trees using an axe, as in Harvest Moon. This does raise a few difficulties though. A few extra sprites may need to be created to show Mimo using the axe. An extra collision detection routine will need to be added to detect if Mimo is facing a tree. And then a list of damaged trees will need to be stored, and each tree's hit count depleted accordingly. When the hit count for a tree reaches a certain level, the tree will have to be removed and converted into lumber that can be picked up. I have made 6 and 7 on the object map relate to each side of a tree trunk, so that we know which tree we are dealing with when Mimo starts chopping. I will see how much memory we have spare to do all this.

Axe the way to do it

I decided it would be simpler to make the axe work in a similar way to the arrow. Rather than making a new set of Mimo sprites holding an axe, making an axe fire out of Mimo for a few pixels looks just as good.

To do this I had to develop a new sprite movement routine. This routine allows the axe to move for five pixels and then forces it to redraw its buffer clearing the screen and deactivates it.



I also decided to implement a proper sprite flipping routine for arrows and axes. This enables me to have only a single image for a left facing axe and an upward facing axe. The routine will work out which direction the axe is travelling and flip it on the correct axis accordingly.

Delivery quests

It is now possible to cut down trees. There are a number of sprite 'slots' set aside which get filled with a lumber sprite when you cut a tree down. This allows persistence, so if you walk out of the screen and back in again, most of them will still be there... up to around 10 sprites, though I may make that number higher later.

Tree damage is working. You have to use the axe 10 times on a tree to cut it down. If you use it 6 times and move to another tree, it will remember, and if you go back to that tree, you will only need to chop it 4 more times. Though it will only remember the damage on a couple of trees, to save memory.

Delivery quests are now possible, like in Elite. The coordinates of around 50 towns that the player has visited are stored in a list. If the player bumps into a delivery quest sprite, one of those towns in the list will be picked, and the player can deliver the item to a character that will appear to receive it. A cash reward will be given on delivery. What you do with the item is up to you, but if you don't deliver it, the quest slot will never be freed again.

Destination

I managed to fix a few bugs in the logic of the sprite direction routine, when a character needs to head towards a specific screen coordinate or player coordinate. I have also enabled human characters to spawn. These characters will be given a set of coordinates from the discovered towns list. If the player bumps into them, they will either give a random greeting, or they will tell the player a little bit about the town they are heading to. The player can follow them to the town if they like. Once the character reaches his destination, he will walk around randomly and give random greetings. Characters are only spawned during the day, to add some realism.



Moveable characters can now be killed by the player. If they are killed, the sprite will be flipped upside down, and will remain as an obstacle on the screen until the next day. If you kill a character, it may be possible to have the player be given a bad reputation, and have other characters try to exact revenge.

Jail-house rock

To develop the Western theme, I have decided to try and create a bounty hunter system. If the player kills a good game character, they will get a bad reputation, and a 'Wanted' poster will appear at the town jail. The poster will show the 'danger' level of Mimo. If Mimo is a wanted man, some of the bounty hunter characters will try to kill him to claim the reward. If however you kill a bad game character, you will be able to claim a reward yourself, and your reputation will improve. If you have a bad reputation, certain characters like the town shopkeeper may not want to have dealings with you.



Bitwise

In order to save space, I have started trying to use bits to store certain character data, rather than bytes. Bits are often used for 'settings' by programmers. Obviously a byte can store 0-255 numbers, and if you use bits, then you will only be able to store 0-7. But it is quite handy for recording boolean numbers, for instance, if your character is killed or disabled you are only going to need a 0 or a 1. Checking the bit values also saves space and is faster. You don't have to load the result into into the A register to check its value, so this could speed up some routines significantly.

Bounty our kid!

I have managed to adapt the arrow firing routine so that computer players can now fire arrows. I had to add an extra bit to the character data block to specify whether the character should shoot arrows when its next move is being calculated. I also added a dedicated computer arrow sprite data block, so that it can be updated and move quickly the same as the player arrow sprite.

When a player kills a good game character, all the active characters will be turned into bounty hunters and will have their sense of direction modified so they seek the player out. As long as the player has a 'wanted' status, bounty hunters will continue to spawn periodically.

Sprite collision detection was modified. I needed to check when an arrow collides with the player or with a game character, which sprite it was fired by. I needed to prevent the player being harmed by their own arrows when they are fired. The same for the computer characters. So when an arrow is spawned, it records the sprite ID of the character that fired it. We then check this ID when a collision is detected.

The firing logic is fairly simple at the moment, they just aim in the player's general direction. They don't wait for line of sight. Their arrows can also damage trees and kill other characters if they get in the way! I thought about doing line of sight calculations, but it probably wouldn't make a lot of difference, and it makes it more interesting if arrows are flying everywhere.

I think perhaps if the player gets hit, they will end up in jail with all their money gone, and they will have a chance to redeem themselves.

Optimization of tables

Looking around on the internet as to how I might save more memory, I came across a page that recommended dividing bytes up into bits in an array of data, such as a tile map, if the data numbers in the array do not exceed a certain value. Using this technique I was able to save around 30 bytes or so in the maze generation function.

I was using byte 253 as a marker to tell the function to repeat the last row of data. I was also using another byte, 254, to tell the function that the following byte is the Y axis position where we are wanting to place our sprites.

First I converted all the Y axis numbers from integers to binary numbers. I then removed the 254 bytes from the tables and instead set bit 7 of the Y axis binary numbers as a marker for where we want to begin a new line. I then modified the routine to check for this bit being set, and if it is, the bit is cleared and the resulting value is our Y axis position. This seemed to work well, so I repeated the process for the other marker, setting bit 6 of the binary number every time we want to duplicate the previous row.

Love and marriage

Like Harvest Moon, I wanted a way that the storyline could develop based on choices the player made, like who to marry, etc. So I needed a way that a character could become interested in Mimo and a relationship could develop. In order to do this I would need to keep a table of quests that the player has completed for specific characters, recording their map location, tile location and building ID. As I was already recording this for the tree damage list, I found that I could use the same functions and table for both. When a player completes a quest for a character, their details will be recorded in the list, and after 10 quests are completed, the character will fall in love. The player should then be able to buy a wedding ring in the shop, which will be super expensive, get married in the church, and cut down trees to help build a big house wherever the character lives.



Bank heist

It's so easy to develop plot lines for the Western zone, as the stereotype is so well known. I have created a bank building, and if the player enters it, they will see that the bank's secret vault is stocked full of gold. Of course the banker will deny any knowledge of it. As I am already using the axe to chop down trees, I thought it might be interesting if the player had to do something in order to get at the gold. If you use the axe on the wall, it will be broken down into lumber, and you can start grabbing the bags.



Of course the banker will raise the alarm and you will become wanted as soon as you start grabbing the gold. So watch out when you exit the building, as you may get shot at by local bounty hunters.

Carry flag

Looking at some source code of the AMSDOS rom, I found that it sets the carry flag if a function is completed successfully. I had been setting the A register with a 1 or 0 and checking its value when the function returned. By using SCF to set the carry flag if a function is successful, and OR A to clear it if it is not successful, I can do a jump or return straight away without having to check the value of the A register. This saves space and is much quicker too.

Hooks

If the player moves out of the screen or into a building, then a variable is set which tells the main program loop that the screen needs to be redrawn. This worked well, but it is wasting an awful lot of processing power, having to repeatedly check this variable when it is not set. So I was thinking there must be a faster and better way to do this. While programming on the Amiga, there was a program called Magic User Interface which used 'hooks' to call its gadget functions. I don't really know how hooks work, but they work really fast because there is no main loop that the program needs to run through to check if each gadget has been pressed. The gadget's function is called directly when it is needed.

I had an idea of replacing the 'call check_tilemap' line in the main program loop with NOP instructions so that it would not be called again until it was needed. Then I amended the 'mark tilemap changed' function so that it changes the NOP instructions back to 'call check_tilemap' using self-modifying code. This saves an awful lot of processing power, as we no longer need to check to see if the tilemap needs redrawn.

Marriage

The marriage code is now working as it should. If you complete ten quests for a random character, and they are a girl, the next time you bump into them they will ask you if you love them. If you say yes, then you will become engaged. You will then get special quests to complete to prove your love.

Firstly the girl will suggest to you how beautiful flowers are, and it will be up to you to procure one for her. When you give it to her, her love will increase. Then she will suggest you get married, and if you purchase a ring, she will tell you to meet her at the church. The girl you were speaking to will be at a specific church building, and if you bump into the minister, he will ask the appropriate question, to which you can reply yes or no.

Once this question has been answered, the girl will reappear back at her original home, and will suggest that you buy a bigger house for your family. You will need a lot of money and logs saved up. When you bring these to a builder, he will build a big house for you and give you the key, which you can return to your wife.

In order for all of this to work, I have to record the sprite location and tile graphic details of the girl you have been speaking to, as well as house location if we want to build a new house. Once a new house is bought, then it should be just a case of inserting it into the map before the original random house is generated.

Joystick menu controls

With a number of different 8-bit computers using the Z80 as a processor, I was thinking about the possibility of converting Mimo to other systems. It should be easy enough to do if the code is arranged properly, and if screen drawing routines and music are adapted.

I was specifically thinking about the Mastersystem, although I don't know much about programming the graphics on it yet. But with no keyboard on the consoles, it is important to be able to control the game from the joystick or joypad. My self-modifying code would also have to be changed if I was running from a Mastersystem rom.



I changed the menu system to use the joystick instead of the keyboard. You can navigate the menu using the up and down directions, and the fire button selects an option.

Memory expansion

I have compressed the code as much as I know how, but I am now starting to run out of memory. My code was ending around &9600, which leaves a little under 1 kilobyte of memory for the rest of the game. ROM functions start around \$A300. The world is still feels empty and there was not much room left for graphics, so I ended up having to move to 128k. Detecting whether there is 64k or 128k installed is not a problem, but catering for both versions in WinAPE in a single program was proving too difficult. There were several possible solutions...

1) Copy in functions from the external memory bank to a fixed location in the 64k as they are called. This would have solved the problem of accidentally switching out essential code, but it could be time consuming and messy. Each 'module' would need to be compiled separately in WinAPE, and then merged into one big file, to ensure function pointers remained correct.

2) Build a jump table. A jump table at the start of a 'module' would make it easy to call the functions in the module without having to 'know' the function addresses. It takes some extra effort to compile it in WinAPE, as function pointers outside the module will not be known to the module unless they are calculated manually, for instance, the function for updating the status bar.

3) Load all the tile graphics into external memory at the start, and switch banks whenever we redraw the screen. Make sure we keep all essential code out of \$4000-\$7FFF.

In the end I went for option 3 . It is the easiest to implement as I don't need to remember any function pointers. Though if I need more room for code, I may use a jump table like in option 2. I will still try to maintain a 64k version until I run out of memory.

You got to know when to hold 'em

What broke the monotony in Zelda were the various sub-games you could play. In one of the games the player has to choose between three pots, and is given a reward if they chose correctly. It is an easy type of game to program, so I thought I might make a similar game for the saloon.

You can play the game Paper Scissors Rock with one of the people in the saloon. You can choose your stakes, and if you win then you can get more money. If you lose then you will lose actual money. If you don't have enough money to pay then you will become wanted. It makes the game a little more interesting to play, and gives the player a little more freedom from a fixed storyline.



Sega MasterSystem

As the Sega MasterSystem and Amstrad CPC use the same Z80 processor, I thought it might be an idea to try and port the code to the MasterSystem. Using WinAPE's compiler, it is possible to compile SMS roms, save the memory to a file and load it into the Fusion MasterSystem emulator. Experimenting with code downloaded from the Chibiakumas assembly coding website, I was able to start compiling a Sega version.



There are quite a few differences between coding on the Amstrad and the Sega. The MasterSystem has a 'VDP' graphics co-processor which handles writing to the screen. On the Amstrad I have to do all the tile drawing, flipping, scrolling and collision detection manually, but this can be done by the Sega's VDP in hardware. On the Amstrad I could write to the screen memory directly, but not on the Sega. Instead, the screen is divided up into 8x8 tiles. While this is useful for creating and maintaining tilemaps, it does limit the size of font that can be used. I may need to adjust the status bar functions to accommodate the loss of space.

The Sega's memory is also divided up differently. Due to the fact that the Sega uses ROM cartridges instead of loading the entire program into memory, it is not possible to use self-modifying code. I did manage to get the randomization code working for the grass, but this involved copying the random letter string from ROM to RAM, and then pointing the function to the area of RAM where it is stored. As ROMs are read-only, any variables need to be stored in a separate memory bank in the cartridge.

Debugging may be more difficult as well, as there is no way of pausing the emulation at certain points to check CPU registers. Having a dedicated development environment for the MasterSystem would be handy.

Sprites are also stored in bitplanes instead of the convoluted way they are written to the CPC screen. This means I won't be able to use ConvImgCPC to design them, but I will be able to generate them by hand in binary. Of course it will take a while to redraw the 150 or so tiles for the Sega.

Each bit represents one pixel, and each bitplane is stored consecutively. So one 8x8 pixel sprite looks like this...

```
# BITPLANE 1,      2,      3,      4
# PIXEL 1,2,3,4,5,6,7,8
defb %00000000, %00000000, %00000000, %00000000
```

I managed to get a decent colour palette selected and create my first few sprites for the grass. So I should be able to build up a working screen. Then I will remap my variables, fix any self-modifying code, recode joystick input, sprite movement, collision detection, and hopefully eventually I should have a working game on the MasterSystem.

Debugging the Sega

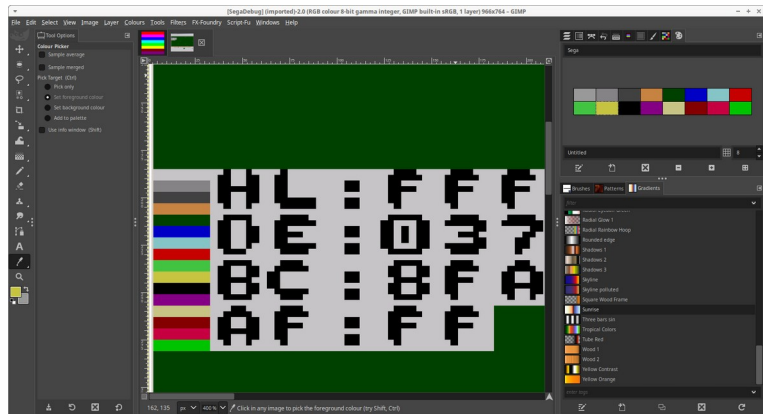
Since we can't debug in the usual fashion through WinAPE by pausing the emulation, I managed to create a function that displays the contents of the CPU registers on the screen. I will be able to call this function whenever I need more information about what is going on during the game.

I also managed to get sprite movement and joystick control working. Hardware sprites work by sending data to the VDP, specifying the X and Y location and sprite ID. Though so far I have not managed to set the size of the sprites.



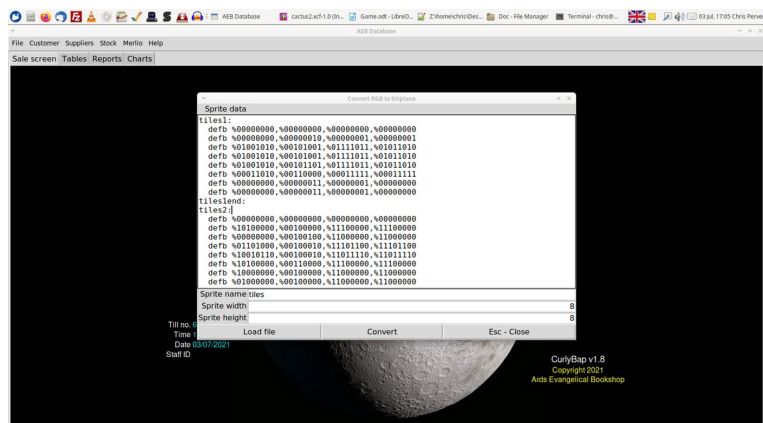
Making the sprite editor

ConvImgCPC was handy for quickly creating sprites for the Amstrad and exporting them to the WinAPE assembler. Unfortunately due to the fact that the MasterSystem uses bitplanes to define graphics, I will not be able to use it to create my sprites. So I will use GIMP to draw the sprites. I created a palette in GIMP with the same colours that I have defined in my code. I will then export each sprite I draw as an indexed RGB image. This will save the pixels of the graphic as palette numbers. I will then need to convert these numbers to bitplanes. I will need to create a Python program that will convert the palette numbers into bitplane code. That shouldn't be too difficult, as long as I know which palette numbers refer to which bitplanes.

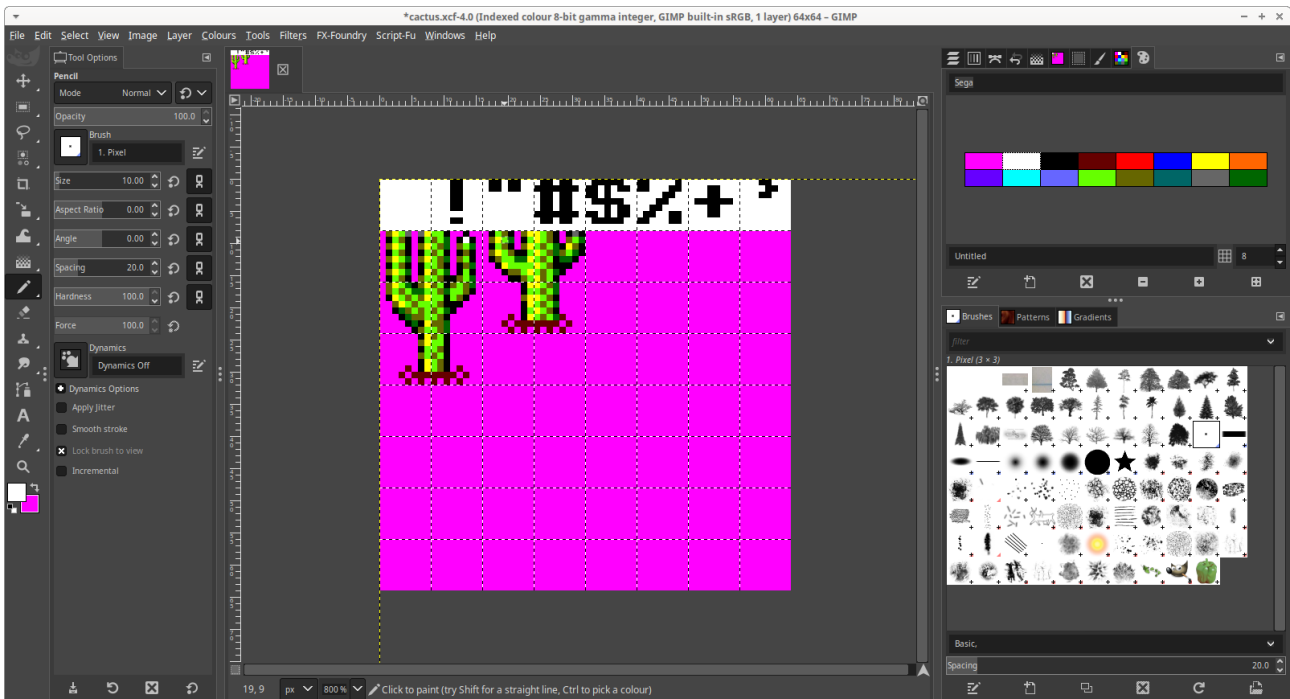


The Python script

I managed to create a Python script that reads a RAW image created by GIMP using the planar export function. This script breaks up the image into 8x8 tiles, which I can then copy directly into the WinAPE assembler. This is useful because it means I can create one single image with all my tiles / sprites on it, and have a function that iterates through each image and copies it to Vram.



ConvImgCPC is actually able to save my existing tiles as BMP, so there is the possibility of loading in my graphics directly into GIMP and saving them as planar images. The only problem is, CPC mode 0 uses 8x16 pixels per 'tile' and my MasterSystem uses 8x8 pixels per tile. These means the aspect ratio is completely different, so they will need to be drawn out again.



I have now created a proper font and a few sprite routines for the MasterSystem version, but due to the complications of different screen dimensions and pixel aspect ratio, it is going to take quite a lot of work just to get a working game on it. All the tiles will need redrawn, and I am not even sure it could cope with many tree tops or building tops on the screen at once due to the limited number of sprites on screen at once (64). Sprites is the only way to get tile transparency, so this will take a bit of thought.

Under the sea

Taking a leaf out of Treasure Island Dizzy, I decided to make a snorkel item. Using the flippers item, it is possible to swim across water and the sea. I thought it would make the world a little more interesting if the player is also able to explore under the sea. It will allow a more varied storyline to develop. The player could look for sunken treasure, or explore hidden cave systems, or discover underwater civilizations.

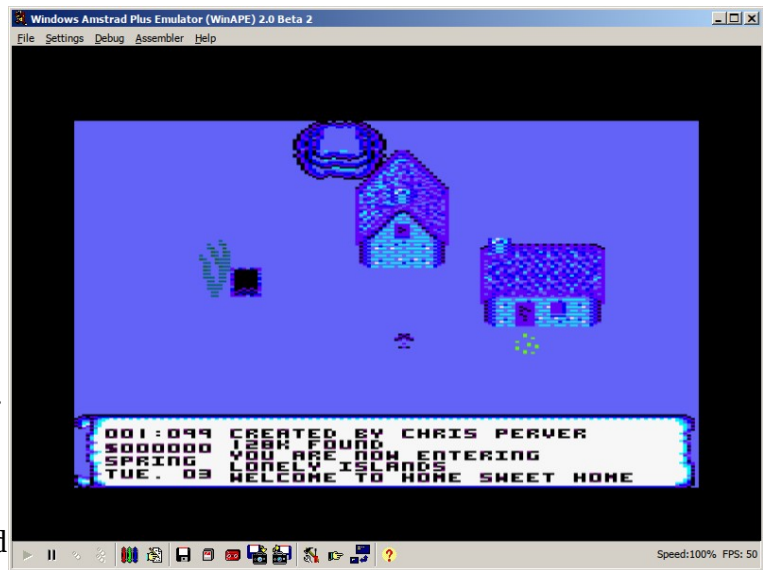
If you have the snorkel in your possession you will be able to view objects underneath the water. I drew a giant kelp sprite to make it look like the player is viewing the bottom of the sea. I had to adapt the sprite insertion routine so it would recognize water tiles as being empty space. I also changed the generate local map function to look up an additional underwater static map table to mark the extra locations. These locations will only become visible if



the player has the snorkel in possession. This should make the player feel that they are exploring a new frontier.

Sunken city

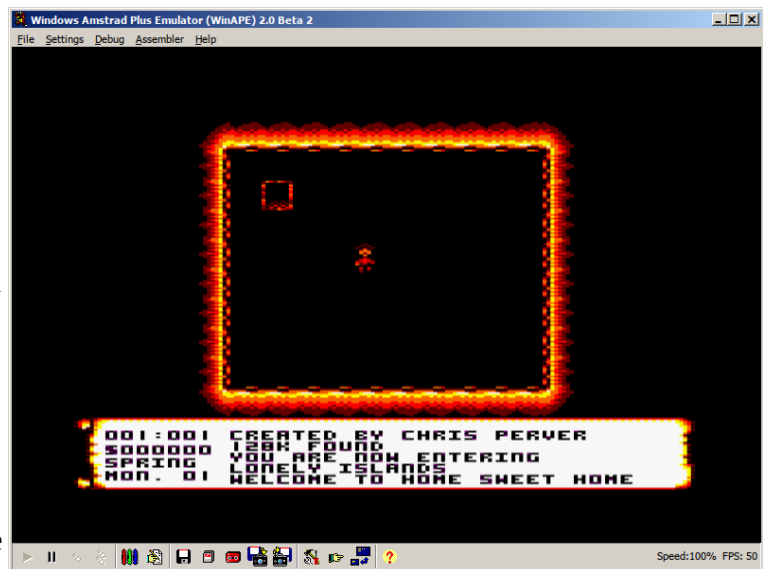
I was trying to figure out a way of having locations beneath the sea without having to draw new sprites and use additional memory. I needed them to look like they were under water. I tried several different methods. Erasing every other line with blue after the screen is drawn would have worked, but not for the sprite transparency if Mimo walks behind an object. The object would be redrawn normally and it would look bad. I would have to amend the tile drawing routines to fix that, and that was a little complicated. In the end I opted for a palette shift. If Mimo has the snorkel and flippers, every colour will become a shade of blue, making it look submerged. I will be able to create a mysterious sunken city, which may have peculiar inhabitants and treasure to find



Caves

In order to create some variety in the game, and give the player the feeling that they are really exploring different lands, I thought about making some caves.

As I had already made the dungeons from the stone wall sprites, it would be a waste of memory to try and create another set of rooms with the same layout but with a different sprite set. So instead I created a small routine that is called after the dungeon tilemap is created. This routine takes two tables of tilemap numbers. It then checks each tile number on the screen and if it finds a tile that matches the first table, it inserts the corresponding tile number from the second table. This routine works quite fast using the CPDR (Compare Decrement Repeat) command to work out which tiles on the screen match the one in the table.



I will use this routine to create some caves and possibly rewrite the wigwam room code so it doesn't use up so much memory. The same routine could be used to create wooden rooms as well.

Story lines

In order to make an immersive game, we need a believable story line. The classic kidnapped princess story line never grows old, and as I already had a castle, I decided to make a king whose daughter has been kidnapped. Your quest will be to find this princess and return her to her father. The princess could be captured by the evil sheriff. Luckily the sheriff's plan was foiled and the princess is actually safe and sound in the old abbey. You need to return proof of her well being to her father in order for them to be happily reunited. I will need to think up some in-between quests to keep the player interested. Maybe involving the evil sheriff and his men.

I had to make some modifications to the static quest system to allow quests to be created in the dungeons.

No wife

I decided to remove the code for marrying a game character. That story line doesn't fit in with the overall theme of the game, which is completing quests. There also isn't enough memory to develop a goal or purpose for getting married. So I am focusing on making more quests and locations, and trying to develop a storyline which the player should be able to logically follow.

I have modified the static quests code so that locked quests will automatically be unlocked when a preceding quest is completed. Locked quest characters will also not be visible until the quest is unlocked. This means the princess will only appear when you accept the quest from the king to rescue her, so the player will not know in advance where she might be.

Merging bytes

As there are so many sprites in the game, it is useful to try to compress that data as much as possible to save memory. I managed to save about 100 bytes by merging the sprite width and height definitions into a single byte. The sprite width and height definitions are only used for determining whether a randomly placed sprite can fit inside the screen before placing it. There is a slight slow down as I need to unpack the width and height data in order to do that. Also, sprites over 16 tiles in size cannot be compressed in this way. Though this does not pose a problem for sprites that are not randomly placed.

Wanted man in...

To make the game a little more interesting, I used some of the spare bytes of the sprite data to create names for the sprites. One byte is used to store a first name and surname from random name tables. The first 4 bits represent the ID of the first name in the first names table, the last 4 bits represent the ID of the last name in the last names table. Now when the player bumps into a character, they will tell them their name. I also modified the text printing routine to extract the first and last names from this byte when it occurs in the status bar.

This also made it possible to have a real 'wanted person' that could be captured in exchange for a reward. When Mimo reads the wanted poster, the random name generated will be stored in a separate byte. If Mimo attacks a person whose name byte matches this byte, then Mimo will be rewarded with bounty. I am not sure whether or not to have a table of remembered wanted names in case the player visits more than one town.

More memory saved

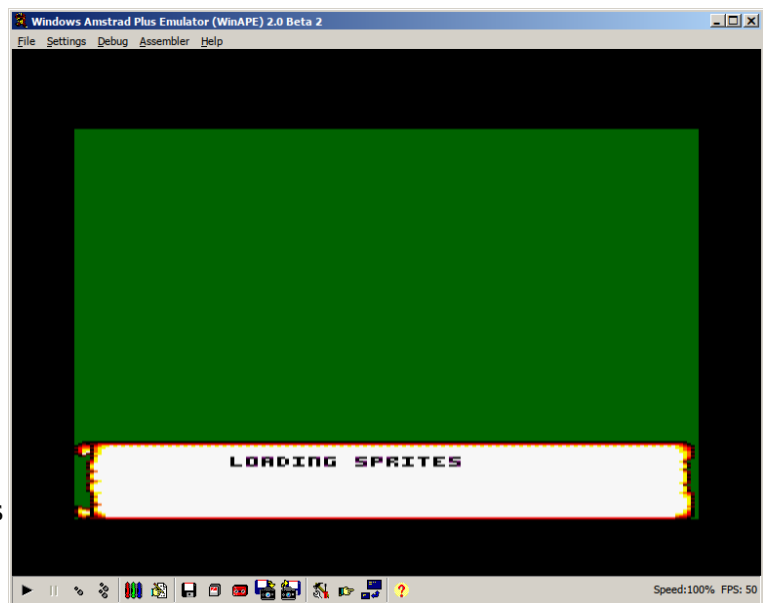
I managed to save about 100 bytes. This was achieved by overlapping parts of some of the tile graphics that use the same pattern. For example if a sprite graphic does not use all 16 rows of bytes, there will be some rows that are empty, either at the top or bottom of the sprite. By moving this tile data block next to another tile that also has empty space at the top or bottom of the sprite, it is possible to merge the rows they have in common together and adjust the memory pointer of the second tile accordingly.

I also had to modify my tile drawing function to accommodate this change, as I was previously relying on every tile being 64 byte aligned. I was only incrementing the low register to move about the data block to increase speed. This had to be changed to increment both the high and low registers.

Two memory banks now in use

As I am running low in memory in the 64k version of the game and still haven't a fully developed story line, I thought I would try to increase the amount of memory available in the 128k version of the game. I had already moved the tile graphics into a separate file on the disc which I am able to load into one of the banks of the memory expansions. I decided to try to do the same for the sprite definitions as this is mainly data and so is well structured, which makes it easily accessible from the banks.

As I was referencing the sprite definitions directly in my map and room generating code, this had to be changed to reference a lookup table. I created a table of lookups for all the sprite definitions and put this at a fixed location, &4000, which marks the start of the second bank. This table is followed by the actual sprite definitions themselves. I then had to work through all my code, changing the direct sprite references to their position in the lookup table. Then I needed to add an extension at the start of each of my sprite functions to get the reference from the lookup table before performing any insertion.



That was the easy part. The complicated part was ensuring all of my critical screen generating code was safely out of the &4000-&7FFF range. I had to use breakpoints in WinApe to see where the program was crashing, and then move certain parts of code into the &0000-&3FFF range and into the &8000 range. Another bug was caused by my function that modifies tile graphics progamatically, to add snow effects or create dirt for the desert. I had forgotten to update the pointers to these tiles for this function, and I spent hours trying to figure out why the 64k version worked and the 128k version had corrupted graphics and kept crashing!

I adjusted the 128k loader to load the extra sprite data into the second memory bank. So far so good.

ROM

While talking with a few people on facebook, somebody mentioned that it might be an idea to create a ROM version of my game, seeing I was already doing bank switching. I checked the internet to see how to compile a ROM in WinApe, and sure enough, I found a nice example. It had a few bugs, but once I got those straightened out, it was very satisfying to be able to make a 'ROM'. I always felt that was something magical. No tape loading necessary, just type the RSX command and instant loading! Brilliant.

So basically to create a ROM in WinApe, we first need to make a 16k file full of zeros. This is achieved in Linux using the following command...

```
dd if=/dev/zero of=Empty.ROM count=1 bs=16384
```

Once this is created, go into the WinApe memory settings and select the empty file as ROM 1 in the upper ROM bank. An additional command in the assembly code compiles the program to ROM memory instead of the standard 64k. There is a short header for the ROM so the CPC will recognize it. After that, the program code can be inserted. This code makes a few palette changes to the CPC on boot...

```
org #C000

write direct -1,1,#c0

defb 1
defb 1,1,1

defw nametable
call initialize
call prog1

ret

.nametable
defb "Setup Palett"
defb "e"+#80
defb "PA"
defb "L"+#80
defb 0

.initialize
push hl
push de

ld hl,message
call print

pop de
pop hl

scf
ret
```

```

.message
  defb " Palette Installed",10,13,0

.print
  ld a,(hl)
  cp 0
  ret z
  call #bb5a
  inc hl
  jr print

.prog1
  push hl
  push de

  ld a,0
  ld bc,#0b0b
  call #bc32
  ld a,1
  ld bc,#1a1a
  call #bc32
  ld bc,#0b0b
  call #bc38

  pop de
  pop hl

  ret

```

There is a limitation, and that is, ROMs are limited to 16k in size. So if I was to make my program into a ROM, I would probably need to compile several of these files into a cartridge file for the 464+. There is a tool on the internet, MakeCart, which may be able to do this. Then it should be just a case of making a small program that runs on boot and copies the contents of the ROM into 64k. I need to do this as a lot of my code is self-modifying, and ROM is read only, so self-modifying code and variables won't work.

Creating my first cartridge

I downloaded two tools from CPCWiki for creating cartridges. One tool, RomInject, by Kevin Thacker, allows you to insert 16k roms into a binary file. Then using BuildCPR, it is possible to convert this binary file into a CPR cartridge file that can be read by the emulator.

There are three methods of rom creation. One uses a directory method for storing the files, though this is not recommended



if you need to know exactly where your data is being stored. One uses a direct copy method, which copies a single file from ROM to RAM before launching it. And a third allows you to 'inject' 16k ROMS into a single file at precise locations. I initially tried the single file method, but for some reason it would not copy my file correctly. It is quite large, so perhaps that has something to do with it. So I settled for the inject method. Initially I just got a black screen. When I paused the emulator, I saw that the data had been copied correctly, so I thought perhaps the palette hadn't been set. As there is no firmware when booting from cartridge, the hardware has to be set up manually. So I checked on ChibiAkumas website to see how to set the palette, and added it onto the boot.asm code. And sure enough, a picture appeared! As I have a few firmware calls in my code, I will need to try to find ways of replacing them with the hardware equivalents.

Firmware

If I want Mimo to run as a cartridge on the plus machines, I will have to remove reliance on firmware commands. Thankfully I don't have many firmware commands to remove, just palette, screenmode, sound, and the timer device. I removed the KL_TIME_PLEASE call and replaced it with a simple byte that is incremented roughly every second. As my main function already has a loop that runs every second to play sound, it was just a case of incrementing a byte in this loop and using it for the time of day. I was also running a lot of code each second to change the palette to night or day whether it was the correct time to do so or not. So I made a second byte that is checked at the beginning of this code which is set when the palette is being updated, and will let me know whether or not to call the palette changing function.

If I manage to get Mimo running on the plus machines, I should be able to dispense with a lot of this pen by pen palette changing code and do a proper palette fade.

Plus dilemma

In the 128k version of the game, I had reserved &4000-&7fff as the area in which the tiles and sprites would be paged into. The problem is, it doesn't seem to be possible to page the roms in the cartridge in at this place. There are only two options, low memory (&0000-&3fff) or high memory (&c000-&ffff).

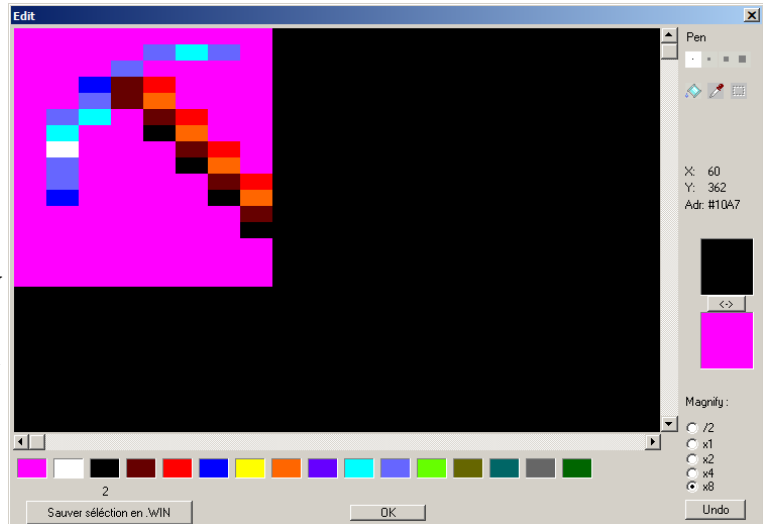
High memory occupies the same space as screen memory. If you write to it, the changes are made to screen memory, but if you read from it the bytes will come from the rom. This is handy enough for drawing sprites but this presents a problem with doing transparency. In order to do transparency I need to read from the screen memory to see what pixels are there before reading from the rom and writing it back to the screen. This could become very cumbersome and slow.

Low memory would work for transparency, but this occupies the same space as my data variables. So I can't be populating the screen and access the variables containing room location, etc. at the same time. The plus machines also have an 'ASIC' memory which controls the graphics hardware, and this gets paged in at &4000-&7fff. So it's all very complicated.

Possibly if you were designing a game from scratch, it would be much easier to start building your code in rom, and then work out what data you need to store in memory.

Pick axe

I am struggling to get a plus cartridge compiled due to memory management complications. So I decided to work again on the actual game. I have separated the screen population function into a sprite population function and a screen draw function. I realized that toggling the map on and off was actually regenerating the screen rather than just redrawing it. If you had chopped down a tree, it would magically reappear when you toggled the map.



I decided to create a pick axe graphic and inventory item. This will behave much like the axe, only the pick axe will be capable of breaking rocks. This will enable the player to gain access to additional locations, such as the city of Atlantis, whose entrance has been blocked by rocks. This is achieved by making walls and rock sprites with an object ID of 8 (destructible with the pick axe), rather than 1 (solid object) or 3 (mountain object), or 6 and 7 (wooden tile destructible with the axe).

It may be possible to reduce these to rocks, or to the occasional gold nugget, though this would use extra graphics memory. It may also be possible to use the pick axe to open some hidden doorways, by making a specific wall or door object change into a room entrance tile.

Inventory blues

As I now had three items that can be 'fired', the fire method was becoming a bit problematic. If the player has the axe, the pickaxe, and the bow, and is stuck in a cave somewhere and needs to use a specific item and is unable to drop any, it could be game over. Before I was just assuming that the player would probably use the axe before he would need to use the bow. So I needed a way of selecting which item is the active one.



I extended the inventory to record not only how many of each item a player has, but whether it is currently 'active'. This state can be changed by going into the menu and pressing fire against the item you want to make active. An asterisk appears beside the active entry. Clicking on it again makes it inactive.

I also modified the local and global map code to use this active state instead of using the toggle method.

Improved character AI

Enemy characters in games like Zelda have an 'aware' zone. If you came too close to a marching soldier, for instance, he would become aware of you and would chase you. I was thinking about making a particular quest where you have to get past a guard to get a key. So I needed to make a few new sprite movement and collision routines to do this.

First of all, I modified the sprite spawning routine to allow certain sprites to spawn at specific screen locations, as opposed to just randomly based on the map zone.



Then I created an extra movement routine, so that a character could move so many steps in one direction, and then move the same number of steps in the opposite direction. This was actually quite easy to do in a very small amount of code.

I created an extra variable in the sprite data to allow a sprite to be designated as a 'guard'. Then I modified the X and Y hit box tables. I already had byte 1 representing a direct hit, and 2 representing a graze, which is used for speaking to a character. I had to create an extra boundary. 3 marks an 'aware' zone. If the player moves into this zone and it belongs to a 'guard' character, then the guard's movement pattern gets changed.

This extra zone actually exposed a bug in my sprite collision code. For some reason a 'graze' was being triggered whenever the player entered the 'aware' zone. It turned out I hadn't been checking my X and Y hitbox results against each other. I was just returning the Y hitbox result if X was not zero. So if X was 3 (in the aware zone) and Y was 1 (a direct hit), the result returned would be 1 (a direct hit). I modified the code slightly by comparing the two results and returning the largest one (furthest away). This seems to have solved the problem.

Some bizarre Z80 optimizations

I was checking out some Z80 programming websites to see if there was any way of further optimizing my code. I came across a few bizarre optimizations...

I do a lot of comparing of byte 255 to check for the end of data lists. It turns out 'CP 255' can be changed to 'INC a'. This saves 1 byte and is actually faster, so long as you don't mind the A register being modified. A list of CPs can also be changed to 'DEC a', again so long as you don't mind the A register being modified.

For instance...

CP 0
JR z,someroutine
CP 1
JR z,someroutine
CP 2
JR z,someroutine

...can be changed to...

OR a
JR z,someroutine
DEC a
JR z,someroutine
DEC a
JR z,someroutine

It does look a little strange, so you do need to be careful. But this has massively sped up my text printing function as well as my room drawing routines by cycling through data lists in a more efficient way.

Better opening plot

I decided to modified the opening game plot. As we already have a mine in the opening part of the game, I thought it would be good to use to pickaxe in some way. So the pickaxe can now be found in Mimo's basement. A large boulder blocks the entrance of the mine, so the player can get the axe and use it on the boulder. This will help the player get used to using the inventory. I also modified the mine tunnel to use the dungeon map, rather than just the single vertical corridor. This actually saved some memory, and it makes the mine more interesting to walk through for the player as they will be wondering where it leads to.

