

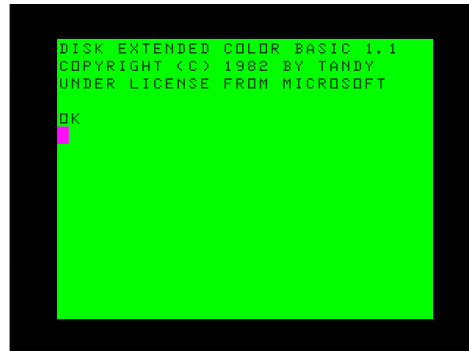
White Rock Ingotz

INTRODUCTION

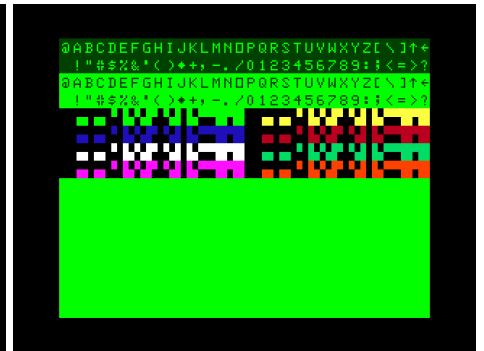
The TRS-80 Color Computer (COCO) boots on its most standard text resolution sporting 32 columns by 16 lines and a single font with dark green over green. There are 64 text characters and 64 inverted ones. There are also semigraphics characters (SG4) consisting of 8 colors, each with its own 16 character variations with black.



TRS-80 Color Computer



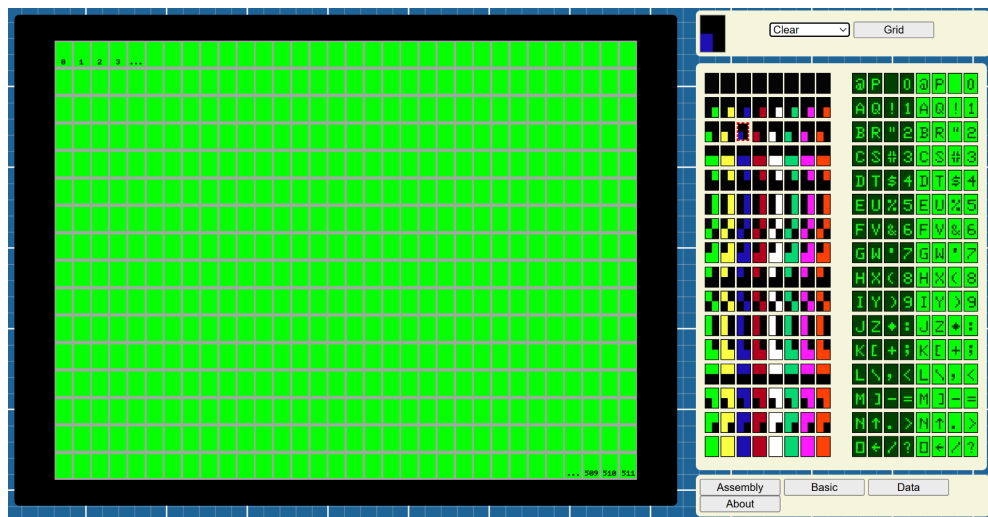
Boot screen, ready to game dev



All text and SG4 characters (ID 0-255)

At 32 by 16, the screen effectively becomes a grid of 512 possible character positions, and these can be assessed by a single space coordinate starting with 0 on the top left to 511 on the bottom right; it works as if reading a text, when reaching the right screen border, continue on the next line.

Using Simon Jonassen's SGEitor (<https://daftspaniel.neocities.org/tools/sgeditremix/>), the text screen can be understood as a tile system where you pick tiles on the right (or use WASD) and mouse plot on the left. The web (online / offline) application can save and load data as well as generate working Assembly or Basic code.



SGEitor's interface, character position (0-511) added for reference

The single font graphics (8x12 pixels) can not be trivially altered but access to a variation with dark red over light yellow is possible. Using the semigraphics, as they are split into 4 parts, conveys the impression of a 64x32 pixel display.



SG4 examples at finest resolution

BLITTING CHARACTERS

There are 2 main ways of blitting characters on the text screen using Basic, one with POKE, another with PRINT:

POKE location, characterID Location refers to the screen position 0-511 but here it must be shifted to 1024-1535.
CharacterID ranges from 0-255 where the first 128 are text and the later are semigraphics.

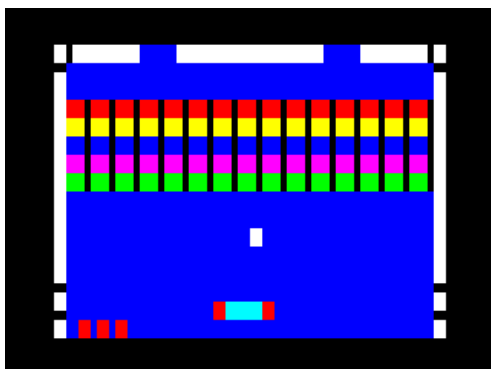
PRINT @location, content Location refers to the screen position 0-511.
Content can be quoted characters, strings or CHR\$(characterID).

POKE is faster on a single character blit context and is also able to blit on the last character position (511) without triggering a screen scroll, on another hand, its position is shifted to start at value 1024 and only a single character can be blitted by one command. PRINT is faster on multi character blit as it can stack them horizontally, it is fast enough to instantly blit half the screen with a single command.

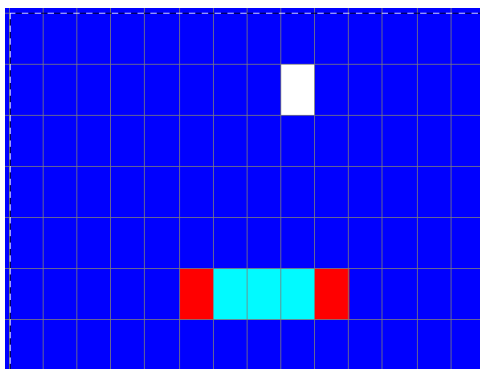
IN GAME DEV

In a game dev context, considering a fast action game, it is safe to generalize that having 6-10 blit commands on a cycle is a limit for smooth flow. Here is where PRINT shines, other than blitting more characters, its edges can contain a background character, thus auto erasing itself when moving horizontally.

An Arkanoid (1986) example port would require 2 POKEs for drawing and erasing the ball, and a single PRINT containing the bat and blue characters on its edges for auto erase, hence 3 blit commands. If using POKEs for the bat it would require 6 of them, summing up to 8 blit commands, dangerously close to the general limit.

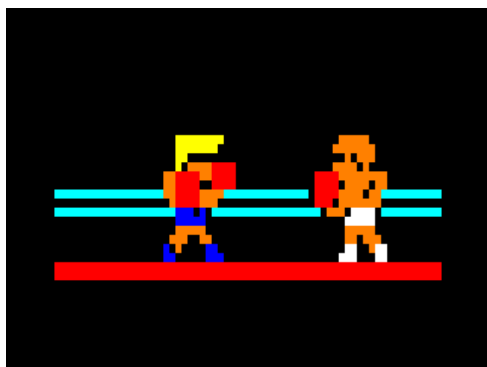


Arkanoid mockup

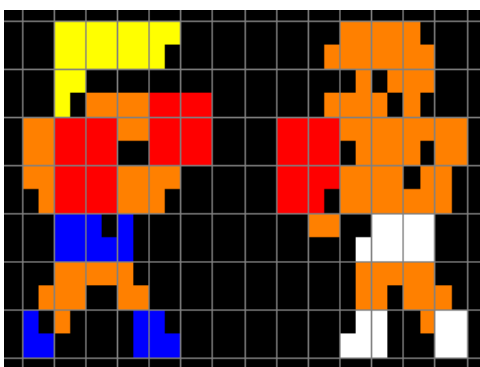


POKE for the ball, PRINT for the bat

A Ring King (1985) example port, with only side motions, would require 7 PRINTs for each player. The total 14 blitting commands are very taxing and exceed the general limit. The speed might be acceptable but adding controls and game logic may compromise it. Using POKE is unfeasible as it would require about 64 blits.



Ring King mockup



Seven PRINTs for each player

SPARK

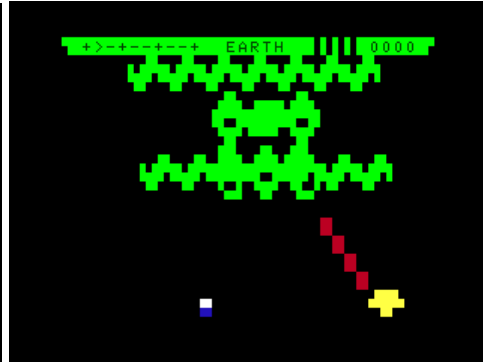
PRINT can be understood as a slice blit, if it could stack characters vertically as well, it would perform as a partial software sprite. One primary benefit is that the Ring King example would drop from 14 blit calls to only 2. But how to do it?

It is currently possible to stack characters vertically with PRINT by adding 32 characters to it. The next character will be underneath the first one because the border of the screen wraps to the next line. There are 2 obvious costs: the blitted content becomes quite big, other objects blitted alongside either blink or disappear.

On the vertical shoot 'em up The Sedna Strain, the almost half screen bosses are blitted with a single PRINT and are capable of moving sideways. They contain the green alien and enough black characters to wrap the screen. It won't suffer the costs because its content is built dynamically and there are no objects moving on its sides.



The Sedna Strain title screen



Moving boss

Considering this existing technique, a theoretical simple solution would be to tell the computer to skip blitting an specific character ID. By trade, each blitted character within a PRINT also advances the cursor 1 position to the right, this behaviour is to be kept, leading to a cursor advance without disturbing a pre-existing character in the background. If a string containing 32 of these specific characters is added to a PRINT, it will automatically move the cursor down.

SOCIAL

Early November 2025 the theoretical case was posted on the COCO related socials and groups in hope to either find an existing Basic friendly solution or check the feasibility of an Assembly routine solution.

At this point, Allen Hoffman, from Subhetha Software, stepped in and declared he once converted VIC-20 PRINT's control code to Color Basic on the Coco, it reaches the same goal but with much more control and elegance, also zero overhead (no need to use 32 characters). Allen decided to revisit his past code and eventually evolved his solution to prime level (Kudos to Sean Connery for low level help). The saga is a must read and can be found here:

<https://subethasoftware.com/2025/10/16/hacking-the-color-basic-print-command-part-1/>

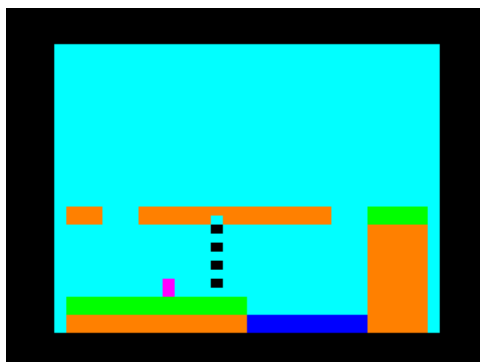
Allen's Assembly routine is converted to Basic and can be pre installed or attached to the final game code. Upon execution the following lower case letters u, d, l, r, when in a PRINT content, will move the cursor up, down, left or right, effectively turning PRINT into a partial software sprite system with mask, lacking only background preservation.

The performance is outstanding, objects containing multiple blits are now resolved in a single blit, this alone can speed a game up more than 10x. It excels over POKE when dealing a single character because if it is in motion, this PRINT will blit and erase in a single command regardless of the object's direction. A single string can be used to store an object's frame making it easy to organize and use semigraphic data. You also gain memory for using less code and another bit of performance boost for not having to use calculations on extra PRINTs positions related to the same object.

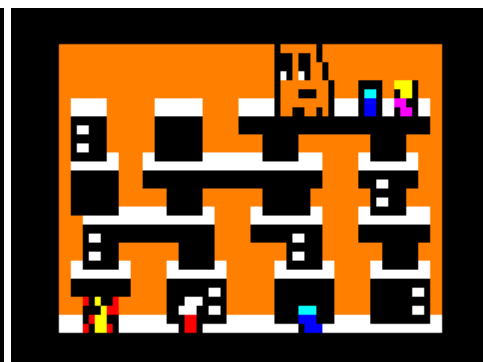
These are just highlights, the routine is a game changer when developing in native Color Basic. It opens the possibility to quickly and easily make a vast gamma of games that were not quite feasible mostly because of blitting performances.

PLATFORM FRAMEWORK

A development article going through the process of making a platform game framework started taking shape. This is a subject for a future issue but as the framework matured, so did the need to put it to a real test, which brings us to the subject of this report article.



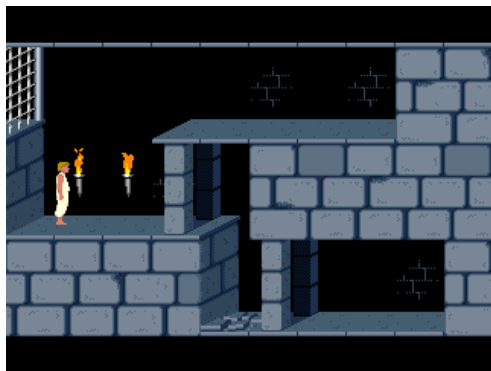
Single character Pitfall like mockup



Donkey Pug with 2x2 character objects

BACKGROUND

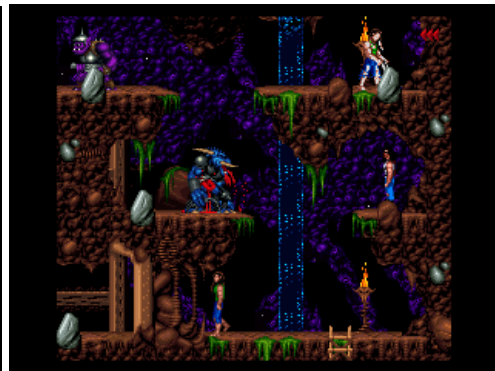
Platformers are popular game genres, if not the most. It sports plenty of sub-genres, many that have one game defining it. Prince of Persia (1989) is one of them, it introduces realism to actions. Subsequent games like Flashback (1992) and Blackthorne (1994) iterated on the formula.



Classic climb and sword



Roll, shoot and shield



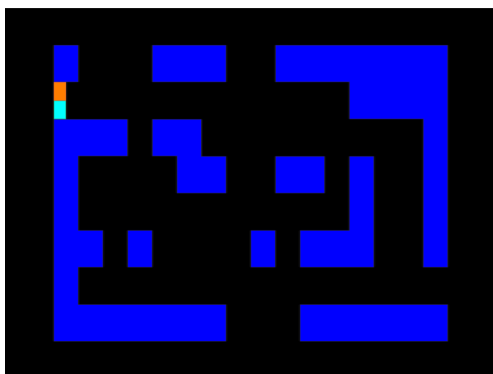
Shot gun (without looking) and stealth

These are static screen platformers with a generous world size. Different from the likes of Donkey Kong (1981), where the player can move at the pixel level and perform high diagonal jumps, the player motion is in sync with the realistic animation frames, leading to a tile based kind of motion. This takes away motion freedom, and to compensate, the player sports 2 walking speeds together with 2 side jump distances. Jumping up is only relevant to climbing platforms. You can also hang on ledges or grab them while falling, duck and duck walk.

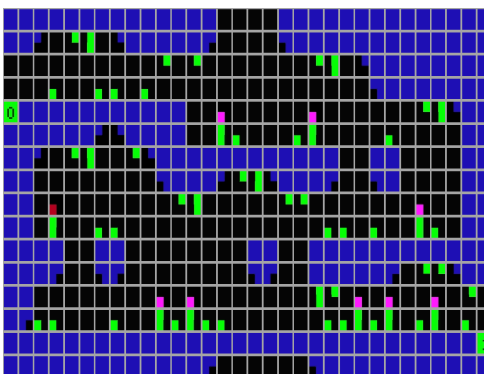
Both graphic fluidity and that peculiar player mechanics are the main references for the framework game test.

LAYOUT

Considering the single screen nature of the references, the SG4 can be made to accommodate more horizontal platforms. Its 32x16 character's screen is divided into tiles of 2x2. Mockups are made in Photoshop to create the tile's art. SGEitor is used to later convert the characters into code data and also test ongoing results on the machine.

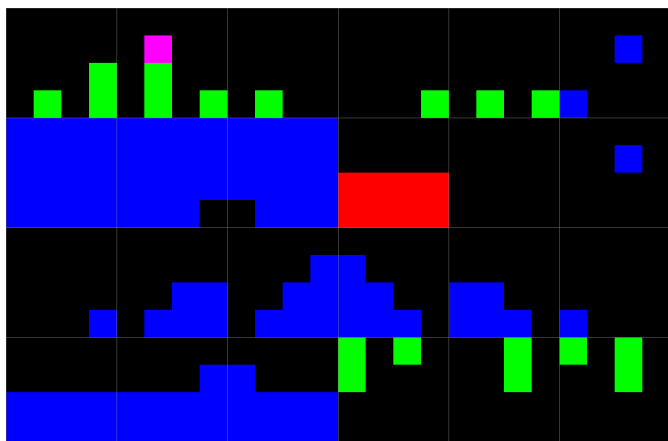


Simple blocked mockup

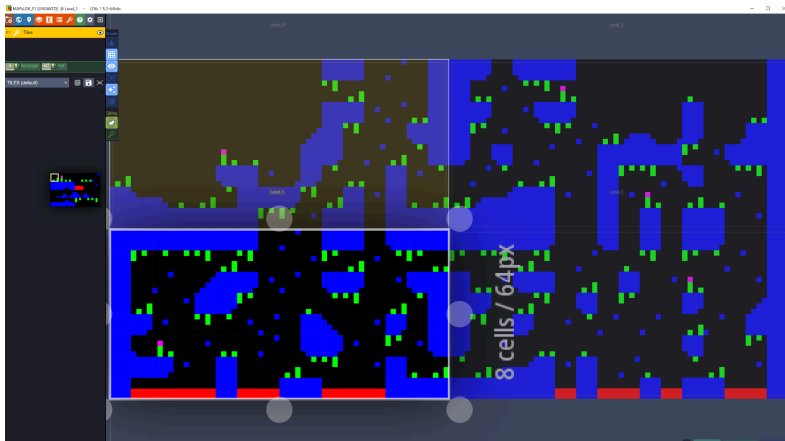


Tile candidates for ground and background

With the tiles set, they are exported as a PNG tile-sheet and are imported into the LDtk map editor (<https://ldtk.io/>). The editor can function with auto-filling considering set rules but in this case, the maps were hand placed. Some ground tiles are half height, those permit even more horizontal platforms for the player to stand. Overall, this layout can support a much larger and more complex single screen maze. Four connected screens were assembled depicting a cave and its lava underground lake.



Tiles as a PNG image



LDtk with the tile set and the world's map screens

There are 24 tiles each with its own number and LDtk can export the screens in CSV style. Those can be accommodated in data statements in code where each screen has 128 tile locations. The technique helps both drawing the screen faster and using less memory to store them, where originally it would cost 512 character entries.

REM --- STAGE MAP--	
DATA 11,11,11,11,11,11,11,11,7 ,7 ,11,6 ,11,7,4~	DATA 7 ,8 ,11,9 ,7 ,7 ,11,11,11,11,11,11,11,11,7,4~
DATA 11,11,11,11,11,11,11,11,24,1 ,11,11,11,7,4~	DATA 7 ,5 ,12,23,24,2 ,11,11,11,11,11,3 ,11,11,11,7,4~
DATA 11,11,11,11,11,11,11,11,11,7 ,7 ,12,6 ,5 ,7,4~	DATA 7 ,8 ,11,12,9 ,8 ,11,11,11,11,11,7 ,11,11,5 ,7 ,4~
DATA 11,11,11,11,11,11,11,11,15,7 ,24,11,11,9 ,7,4~	DATA 7 ,22,12,11,23,22,6 ,1 ,14,17,13,8 ,11,13,7 ,7,4~
DATA 11,11,11,11,11,2 ,4 ,11,11,7 ,8 ,11,6 ,1 ,23,7,4~	DATA 8 ,11,9 ,11,6 ,11,11,7 ,7 ,7 ,7 ,22,12,9 ,7 ,7,4~
DATA 1 ,11,11,11,11,9 ,7 ,12,13,8 ,23,5 ,11,7 ,12,23,8~	DATA 11,6 ,23,3 ,1 ,11,12,7 ,22,24,6 ,9 ,11,23,24,7,4~
DATA 7 ,21,17,3 ,1 ,23,7 ,7 ,8 ,24,1 ,7 ,12,7 ,3 ,1,4~	DATA 1 ,11,15,7 ,7 ,12,11,7 ,12,2 ,11,1 ,4 ,12,11,7,4~
DATA 7 ,7 ,7 ,7 ,7 ,7 ,22,22,12,7 ,7 ,11,7 ,7 ,7,4~	DATA 7 ,7 ,7 ,7 ,7 ,12,11,7 ,11,7 ,11,7 ,7 ,11,12,7 ,7,4~
DATA 7 ,7 ,7 ,7 ,7 ,7 ,11,6 ,11,7 ,7 ,11,9 ,7 ,7,4~	DATA 7 ,7 ,7 ,8 ,11,11,9 ,11,7 ,11,7 ,8 ,11,11,7,4~
DATA 7 ,22,23,24,1 ,24,11,11,12,22 ,1,12,23,22,5,4~	DATA 22,23,24,1 ,24,11,6 ,11,3 ,24,11,22,23,6 ,11,7,4~
DATA 7 ,11,6 ,11,15,7 ,8 ,12,6 ,11,9 ,8 ,11,12,9 ,7,4~	DATA 7 ,5 ,12,9 ,7 ,11,1 ,11,7 ,11,12,2 ,11,11,12,7,4~
DATA 7 ,1 ,11,12,7 ,8 ,22,11,11,3 ,23,24,11,1 ,23,7,4~	DATA 7 ,8 ,11,1 ,24,11,9 ,11,22,11,11,7 ,8 ,11,1 ,7,4~
DATA 7 ,8 ,12,11,24,23,12,6 ,11,7 ,16,11,12,8 ,11,7,4~	DATA 7 ,24,11,8 ,11,12,23,6 ,11,6 ,11,1 ,6 ,11,7 ,7,4~
DATA 7 ,18,2 ,11,12,5 ,11,6 ,11,9 ,8 ,12,11,22,12,9,4~	DATA 8 ,11,12,11,5 ,11,3 ,11,11,11,11,9 ,11,11,22,7,4~
DATA 7 ,7 ,8 ,12,11,7 ,11,11,1 ,23,24,11,12,7 ,3 ,1,4~	DATA 23,1 ,14,20,7 ,12,7 ,11,6 ,3 ,11,23,1 ,11,12,7,4~
DATA 7 ,10,10,10,10,7 ,10,10,7 ,7 ,10,10,10,7 ,7 ,7,4~	DATA 7 ,7 ,7 ,7 ,7 ,10,7 ,10,10,7 ,10,7 ,7 ,10,10,7,4~

Four screens with DATA entries containing the numbered tiles

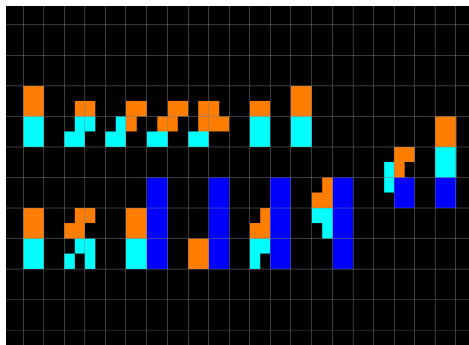
Each tile is stored on a single dimension string array and takes advantage of Allen's routine, meaning a single PRINT can blit a single tile, therefore looping the 128 map data while blitting each tile is 4x faster than drawing every single character by POKE. Each array entry is composed of 2 upper characters (CHR\$), "dl" which makes the cursor go down once and left twice, 2 lower characters, thus making the 2x2 character tile.

```
REM --- GRAPHICS MAP TILE SET ---
TI$(1)=CHR$(128)+CHR$(128)+"d11"+CHR$(129)+CHR$(133) : TI$(2)=CHR$(225)+CHR$(128)+"d11"+CHR$(133)+CHR$(129)
TI$(5)=CHR$(128)+CHR$(128)+"d11"+CHR$(129)+CHR$(129) : TI$(6)=CHR$(128)+CHR$(162)+"d11"+CHR$(162)+CHR$(128)
TI$(9)=CHR$(175)+CHR$(175)+"d11"+CHR$(173)+CHR$(175) : TI$(10)=CHR$(128)+CHR$(128)+"d11"+CHR$(191)+CHR$(191)
TI$(13)=CHR$(128)+CHR$(128)+"d11"+CHR$(128)+CHR$(161) : TI$(14)=CHR$(128)+CHR$(128)+"d11"+CHR$(161)+CHR$(175)
TI$(17)=CHR$(128)+CHR$(128)+"d11"+CHR$(175)+CHR$(162) : TI$(18)=CHR$(128)+CHR$(128)+"d11"+CHR$(162)+CHR$(162)
TI$(21)=CHR$(162)+CHR$(128)+"d11"+CHR$(175)+CHR$(175) : TI$(22)=CHR$(138)+CHR$(136)+"d11"+CHR$(128)+CHR$(128)
```

String array containing each 2x2 character tile

PLAYER AND OBJECTS

Game object and player frames are a mix of 1x2 and 2x2 character sizes depending on the motion. The variation is used to provide smooth per pixel animation where needed. Each frame data is stored as a standard string, this way it has faster access than an array string for these are constantly being used inside the main game loop. This also takes advantage of Allen's routine so the player can be blitted by a single PRINT.



Mockup player frames

```
REM ----- GRAPHICS PLAYER AND OBJECTS -----
P0$=CHR$(255)+"d1"+CHR$(223) :REM PLAYER STAND-
P1$=CHR$(247)+CHR$(248)+"d11"+CHR$(214)+CHR$(218) :REM PLAYER WALK R/L
P2$=CHR$(244)+CHR$(251)+"d11"+CHR$(213)+CHR$(217) :REM PLAYER WALK L/R
P3$=CHR$(241)+CHR$(242)+"d11"+CHR$(215)+CHR$(216) :REM PLAYER JUMP R DUCK/L
P4$=CHR$(209)+CHR$(254)+"d11"+CHR$(220) :REM PLAYER JUMP R/L
P5$="d"+CHR$(255) :REM PLAYER JUMP R/L LAND/L
P6$=CHR$(241)+CHR$(242)+"d11"+CHR$(212)+CHR$(219) :REM PLAYER JUMP L DUCK/L
P7$=CHR$(253)+CHR$(210)+"d1"+CHR$(220) :REM PLAYER JUMP L/R
P8$=CHR$(247)+"d1"+CHR$(222) :REM PLAYER JUMP UP / FALL/L
P9$=CHR$(247)+"d11"+CHR$(221) :REM PLAYER GRAB R/L
Q0$=CHR$(251)+"d1"+CHR$(222) :REM PLAYER GRAB L/R
Q1$=CHR$(209)+CHR$(254)+"d11"+CHR$(212) :REM PLAYER CLIMB R/L
Q2$=CHR$(253)+CHR$(210)+"d1"+CHR$(216) :REM PLAYER CLIMB L/R
Q3$="d"+CHR$(151) :REM OBJECT GOLD INGTOT-
Q4$="r"+CHR$(138)+"d1"+CHR$(152) :REM OBJECT GOLD FLOWER-
```

Player and object's strings

The game is joystick(right) controlled and while the color computer sports a single button analog joystick, no advantage is taken from the analog nature, so it is recommended to use a joypad when possible. Pushing the joystick left or right will make the player walk until you either hit a wall or fall off an edge, if you walk over a single character obstacle, you will automatically climb it. Pushing up will perform a vertical jump and you can use that to grab a higher edge, if touching a 2 character's wall, you will automatically climb it. Grabbing on an edge is performed by holding the button and can be done when jumping or falling. Ultimately, when standing, the player can perform side jumps by holding the button and pushing the joystick sideways.



Color computer analog joystick

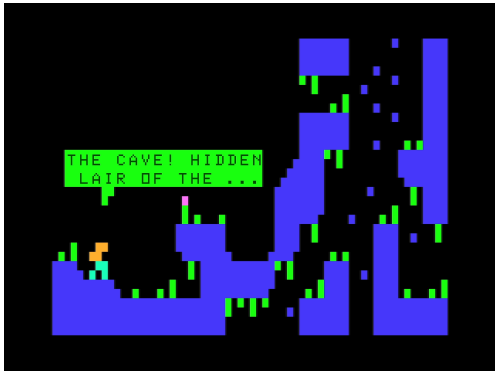
STORY

White Rock mountain (https://pt.wikipedia.org/wiki/Morro_da_Pedra_Branca) is a real geographic location close to the Atlantic shore in the state of Santa Catarina, Brazil. Its prominent feature is a table like rock at its summit, the white characteristic is probably because it gets covered in clouds and fog every now and then. It is a location with rich folklore tales of fantastic creatures and witch meetings, there are also stories of hidden pirate treasures.



Summit of White Rock mountain

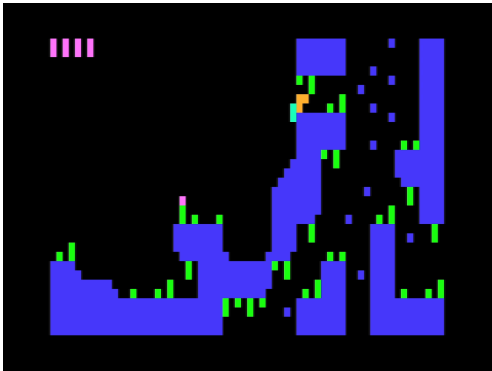
You are an adventurer in search of those treasures, the tale says there are 7 gold ingots to be found inside a secret cave. The last ingot is more of a mystical kind and only the most persevering hero will ever find it. While exploring, you should be aware of high falls, they can hurt and you can only take enough, touching the lava is an instant death. Every ingot collected will raise your health. Finding some treasures is only the first part, you also have to escape the cave and leave the area with them.



Intro story



Title



HUD with 4 health bars, player entering the cave

CODE STRUCTURE



The code is first created for functionality and follows a linear structure akin to the development process. Variables are defined at BOOT DATA as they are required by the GAME LOOP. The only optimization at this point is the order of variable declaration, keeping the more used first speeds up the game.

The GAME LOOP contains the player state machine consisting of 10 states: Stand A, Walk Right, Stand B, Walk Left, Stand C, Jump Right, Jump Left, Jump Up, Fall, Grab, Climb.

Each state has its own timer, animation calls, collision checks and joystick read, used only when necessary. The order of state is set in a jump-forward fashion, that means any outcome of a state will be solved in the following code lines. That brings fast action response on controls and animations, it also speeds up the game for not having to jump backwards in code.

Game sound effects and short tunes are added directly in code using the PLAY command and occasional looping of it with a control variable. There are sound effects for walking, jumping, landing, grabbing, getting hurt, burned and for picking treasure. Short tunes will play on the title, game over and won. The adventurer speaks through comic balloons, and when doing so, a shuffly sound similar to the adults from "Peanuts" gets played.

SUBS contain Allen's routine, collision detection, exit stage detector, game over and won, endings, and a few other game related elements not needed inside the GAME LOOP.

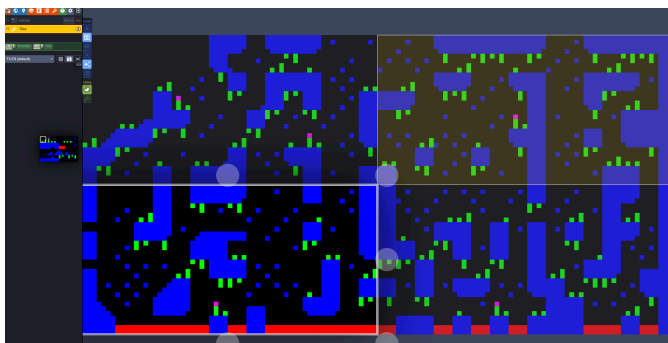
SG4 has a built-in alpha channel, so only visible character pixels are blitted while the background remains. To preserve it, the rendered tile screen is copied into a 512 character array. When the player moves, the previous position is restored from this array. Using PRINT with @ (0-511), you can place and erase characters in one efficient call using the same position index. The drawback is that capturing the background is slow, so the screen and HUD are built and stored together with sound effects to speed up stage transitions. Fast speed POKE is used during this procedure.

```
REM --- 2-STAND R---
PRINT @P,"r"P0$ "u11"CHR$(B(P))"d1"CHR$(B(P+32));:REM DRAW STAND + ERASE---
P=P+1:S=0:REM MOVE RIGHT / RESET STATE---
IF B(P+64)<>175 THEN 350:REM NO GROUND > FALL---
GOTO 99:REM END TURN---
```

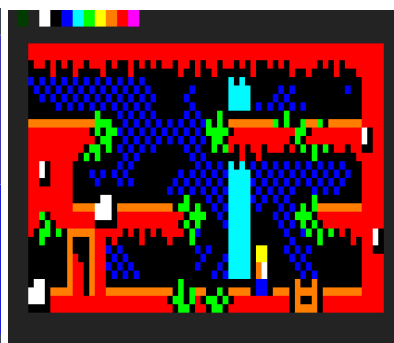
Stand right state prints the player at P while erasing the background with CHR\$(B(P))

CONCLUSION

White Rock Ingotz is a small experiment that shows BASIC can still handle simple, interesting platformers. The final code is compact, about 280 lines, and could be reduced even more. With Allen's routines, there's still performance left for adding enemies, obstacles, or even music. Using a background array also improves how visuals are handled, allowing proper screen restoration. Together, these techniques form a simple but effective software sprite system for SG4.



A different cave map



Blackthorne tile set

If you enjoy this kind of content, consider buying one of FUED's games or donate a few quids to support the effort.
Have fun!



FUED.NET