ECE241 Design Project: Game & Watch Gallery

James Pan & Bradley Stone

Introduction

The goal of this design project was to apply and evaluate what we have learned throughout the semester in our ECE241 Digital Systems course. Over the span of the past four weeks, we have designed and created a fully functional system that uses keyboard inputs, and outputs to a LED monitor and a speaker. Our design idea is based on the collection of video games popular in the 1980s and 1990s known as Nintendo Game & Watch. We decided to recreate many of these games using the DE1-SOC board and some of the resources available to us online.

Our aim was to create a splash screen (introduction page), menu, and as many games as our time constraints would allow. In the time give, we created four different games, each with two levels of difficulty. The games that we created were *Fire*, *Octopus*, *Flagman*, and *Super Mario Bros*. The player chooses the game and difficulty to on the menu screen and can return back to the menu at any time while playing. For the games, Fire requires the player to juggle characters jumping out of a burning building to help them reach the ambulance; *Octopus* requires the player to avoid the octopus tentacles and take treasure from a sunken ship; *Flagman* requires the player to press certain keys in sequence or as they show up on the screen; and *Super Mario Bros.* requires the player to move up and down in order to pass through holes in the wall.

The Games

Fire A and B game will spawn jumpers that will fall from the building and require the player to bounce the jumpers back up with a trampoline. If the player is at the correct position and time, the jumper will bounce up and move forward to the next position. Each jumper will require to be bounced up three times before reaching the ambulance. The game is designed so that no two jumpers will reach the ground at the same time, which would make it impossible to save both of them. Each jumper that successfully makes it to the ambulance will increment the player's score. If a jumper reaches the ground without the player in the position to bounce it, the player will lose a life. Upon losing all three lives, the game will end and the player will be returned to the menu screen. The A game spawns a new jumper when the previous jumper is at a set location. The B game can spawn a new jumper at any point.

Octopus A and B game has an octopus that extends and retracts its tentacles. If the player is at a position where the tentacle fully extends, the player will lose a life and restart on the boat. By successfully avoiding the tentacles and reaching the treasure chest, the player will pick up the treasure and must make it back to the boat. Each treasure brought back to the boat will increment the player's score. Upon losing all three lives, the game will end and the player will be returned to the menu screen. The octopus's tentacles in the A game move in a predictable pattern, while the tentacles in the B game they move more randomly.

Flagman A and B are two different games requiring the player to press keys (A, S, Z, X) on the keyboard at different times during the game. *Flagman* A is a version of the game Simon Says. You must repeat the random sequence that is shown on the screen. Each success will increase the length of the sequence by 1 and increment the player's score. Failing to complete the sequence will result in the player losing a life and the game will restart. Flag B is a version of a common typing game. The player must press the key shown on the screen before the timer bar disappears. Each correct key will decrease the time for the next key press and increase the player's score by one. Missing a key or pressing the wrong key will cause the player to lose a life and reset the timer. Upon losing all three lives, the game will end and the player will be returned to the main menu.

Super Mario Bros. A and B requires the player to move the character up and down in order to move past walls. Each wall in the game has one hole in different possible positions where the player must match in order to avoid losing a life. Bros A and B are the same game with differently coloured sprites. At the start of the game the player starts on the far right side of the screen. Upon losing a life, the player will move to the middle, therefore being able to see one wall ahead (making the game easier). Upon losing a second life, the player will move to the far left side of the screen, therefore being able to see two walls ahead (even easier). Losing three lives will cause the game to end and return the player to the menu.

The Design

(Refer to Appendix A for block diagram)

Top Level Design (GameAndWatch.v)

- Our top level design contains most of the base code of our design. In this module we instantiate the VGA adapter, keyboard controller, an instance for splash, menu, and each game in our design, and HEX displays used for keeping track of score and lives. We also use a finite state machine and a multiplexer to choose between which images to draw on the screen. The FSM is linked to the menu module which selects the state that corresponds to the game that the player has chosen. The multiplexer uses the state of the FSM as a select that picks what to draw on the display. At the start of executing the program, the FSM is set so that the splash screen will be shown.
- A Random Number Generator is also implemented here that will be used in the games.
- We designed our code in a way that we feed VGAx and VGAy into the modules for the splash/game/menu. As the VGAx and VGAy are constantly looped, the modules return a colour value that will be used by the multiplexer.

Splash (Splash.v)

- When the code is uploaded to the DE1-SOC board and reset is pressed to start the program, the player will be presented with a splash screen. This is the introduction page that shows a sprite (see Mem for how sprites are implemented) with the title of our project and our names along with a picture of the Mr. Game & Watch character. By pressing Enter on this screen, the quitSplash output will be driven high to the top level and will set the state to stateMenu.
- If the Konami code (Up -> Up -> Down -> Down -> Left -> Right -> Left -> Right -> B -> A -> Enter) is inputted during the splash screen, the player will be able to access a different version of the hidden game (*Super Mario Bros.*) that can be found in our design (See Menu).

- The splash screen also instantiates an audio controller that will play a sound when the program is reset.

Menu (Menu.v)

- The menu uses the inputs of the keyboard to allow the player to select a game to play. By pressing the Left/Right arrow keys, Enter, and ESC, the player can pick the game and difficulty to play. By clicking Enter after selecting a game and difficulty, the state corresponding to the game/difficulty will be returned to the top level FSM and will choose that game to show on the VGA display.
- We included a hidden game (*Super Mario Bros.*) that can be accessed by pressing Ctrl + Alt + Del during the menu screen.
- The menu screen uses different sprites shown at different areas of the VGA display. To do this we used if-statements with the VGAx and VGAy inputs and a multiplexer to display the sprites in their correct places.

Games

- A counter is used to keep track of the player's lives and score. If the register Lives equals three, a quit value is returned to the top level. This will return the FSM to stateMenu.
- The randomness is applied using the Random Number Generator created in the top level of the design.
- A multiplexer is used to determine which sprite to show on different parts of the display. The position of the player character and the computer characters determine the position of the sprites to show.

Fire A/B

- A 2-bit register is used to keep track of the position of the character. A 22-bit register is used to keep track of the position of the jumpers. Since the jumpers bounce up, there are 22 possible positions that jumpers can be in (LSB being the building, and MSB being the ambulance)
- If the correct position of the player character (positions: 00 left, 01 centre, 10 right) matches the jumpers' position (jumpers[4], jumpers[12], jumpers[18], respectively), then the jumpers register will shift left by 1 bit.
- The Lives register is incremented if the player position is incorrect.
- The RNG is used for the falling jumper animation sprites.
- In Fire B, the RNG is used to determine when jumpers are spawned. Conditions must be met for a jumper to spawn so that no two jumpers will land at the same time.

Octo A/B

- A 3-bit register is used to keep track of the position of the character. A 1-bit register is used to show if the player is holding the treasure. Each tentacle has a 2-bit or 3-bit register to keep track of its length and in game A, a 1-bit register used to keep track of the direction of each tentacle.
- Lives is incremented if the tentacle at the player's position is fully extended.
- In Octo A, the direction register will change upon fulling extending/retracting the direction.
- In Octo B, the direction register will change based on the RNG.

Flag A

- An additional random number generator is used to create the sequence required for the player to enter, which is clocked once per input, rather than on the 5MHz clock. This causes the sequence to stay the same when increasing the length and makes it easier to check if the player entered the sequence correctly, without needing a large register to store the values. The additional random number generator is seeded from the primary random number generator at the start of each life.
- A 2-bit register is used to keep track of the player's key press.
- If the player's key press does not match the corresponding value in the sequence, the lives register is incremented.

Flag B

- A 2-bit register is used to keep track of the player's key press. A counter is used to keep track of how much time the player as to press the correct key.
- The RNG is used to generate the key that is required for the player to press.
- If the player's key press does not match the key shown on the screen, the lives register is incremented.

Bros A/B

- Based partially on L1-4 from the *Super Mario Bros.* Game & Watch and the *Squish* Game & Watch
- Four 2-bit registers are used to keep track of the holes in the walls.
- The RNG is used to generate the position of the holes.
- To shift the walls, the values in each wall resister is passed to the wall to the left, with the rightmost wall getting a value from the RNG.
- The lives register is incremented if the player does not match the position of the holes.

Mem Files (eg. menuMem.v, fireMem.v)

- Modules are instantiated here to create ROM for the sprites.
- Each ROM module is used to store the values for the sprites using .mif files.
- The splashMem.v file also creates a RAM to store the data for the .mif file containing audio data.
- The colour sprites were stored with a custom 3-bit palette.
- The black and white sprites were stored as a single bit, and were resized to 3 bits right before being sent to the VGA controller.

Keyboard Controller (Keyboard.v)

- Used logic statements so that upon key press, the value is returned once for that clock edge. After that value is returned, the key output is reset to 0. This is so that by pressing and holding a key, only the initial signal from that key will be used.

VGA Controller (vga_controller.v)

- The provided VGA controller was used, changed only to allow a custom palette (see Appendix B).
- Black and White were assigned to 3'b000 and 3'b001, respectively, so that the game that only use black and white could store the sprites using one bit per pixel. When the colour of the pixel

was passed to the VGA controller, two zeros were concatenated to the front of the colour values returned from all black and white games.

- The last four colours are ordered such that the LSB corresponds to the game mode for *Super Mario Bros.* (i.e. 0 for game A, 1 for game B). This allows easy palette swaps, such that the LSB is inverted if the MSB is high and the alternate palette swap is needed.
- Each colour code has a corresponding RBG colour (e.g. 3'b100 corresponds to Red - #FF0000). The sprites had to be coloured this way before being converted to a .mif using *img2mif.exe* (see .

Random Number Generator (RNG.v)

- Uses a 16 bit linear feedback shift register with taps on the 11th, 13th, 14th and 16th clocked on the 50MHz clock to produce a sequence of pseudorandom numbers.

Report on Success

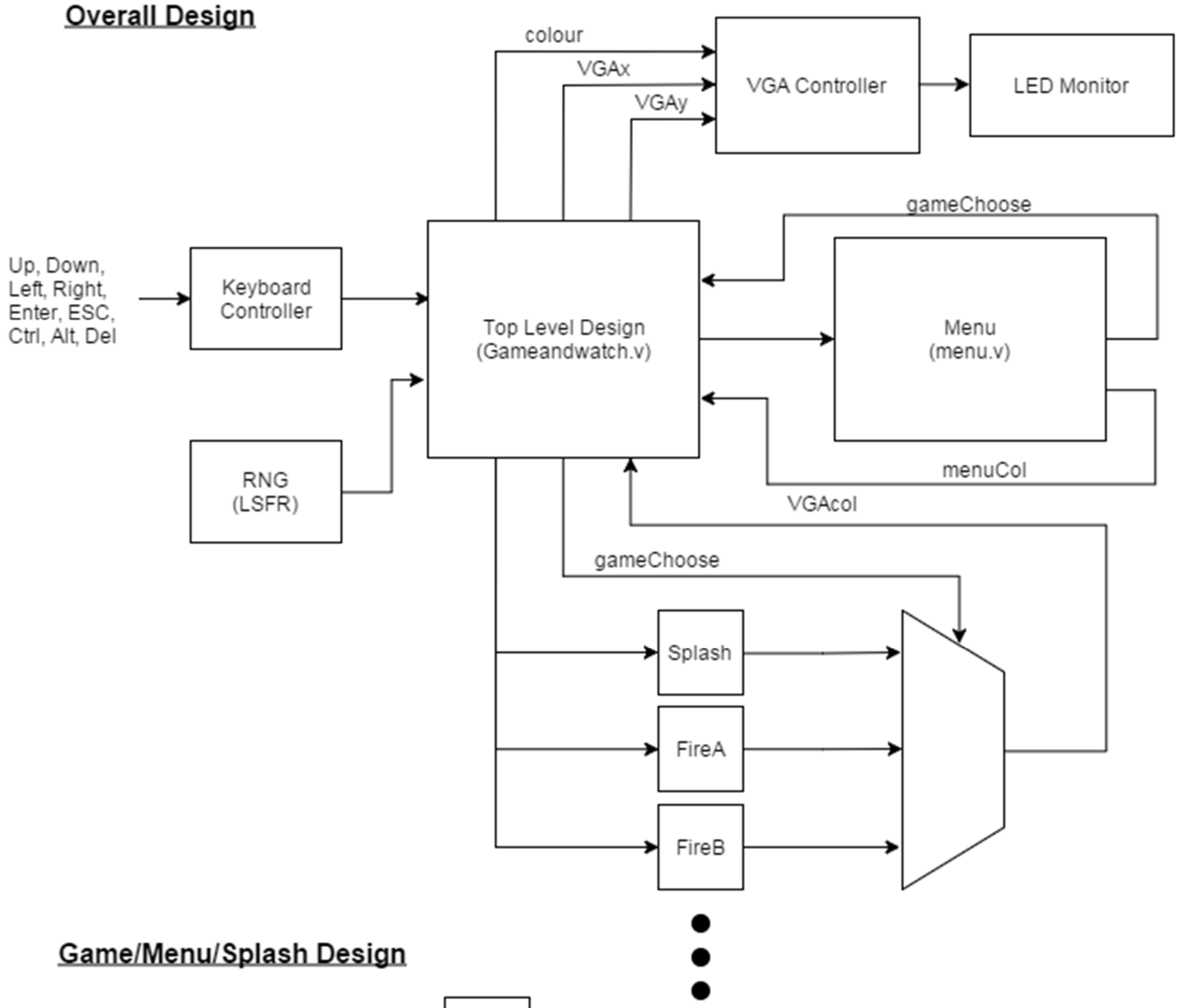
We managed to create 4 distinct, fully functional games in the given time that we had. In addition to these games, we have also created a few other games but did not have to time to implement these games into our design. The code can be found in our project but is not incorporated into our final product. From what we have seen, only a few small glitches appeared in our final project. Upon returning to the menu screen after ending a game, the program returns the player to the difficulty selection for Fire instead of fully returning to the original state of the menu. For one of the tentacles in OctoA, the tentacle fails to change direction after fully extending. When our sprites overlap, small black pixels occasionally appears at the borders of the sprite. The first two issues were probably the result of minor details with our code that we may have overlooked. For the third issue however, we have checked our code for the VGA but were unable to find a reason why the black dots were showing.

What Would You Do Differently

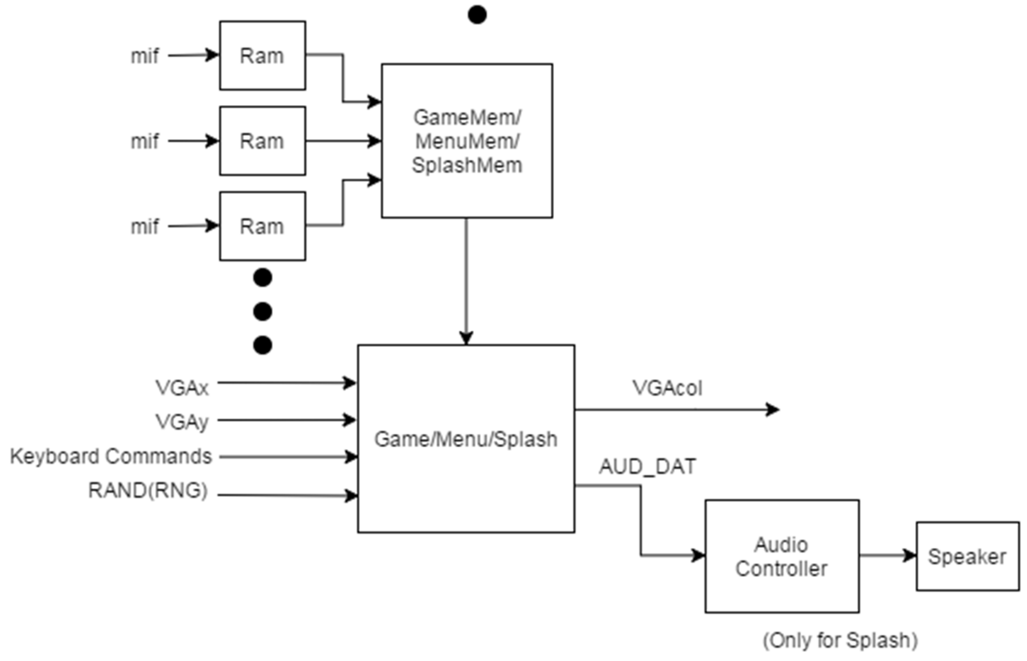
If we were to start over again we would have paid more attention to the code so minor glitches would not appear in our final product. When we started this project, we were not planning on incorporating an audio controller into our design. After trying it out with the splash screen and making it functional, we realized that this could have been implemented with the rest of the code. Using the audio controller more frequently could be a goal for future projects.

Appendix A: Block Diagram

Overall Design

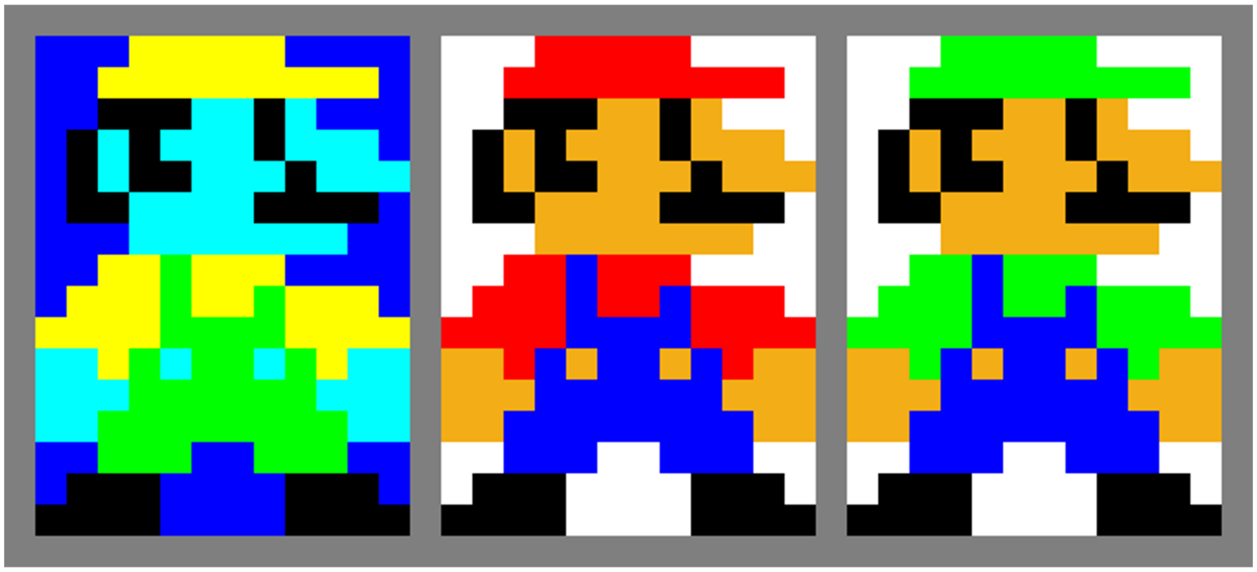


Game/Menu/Splash Design

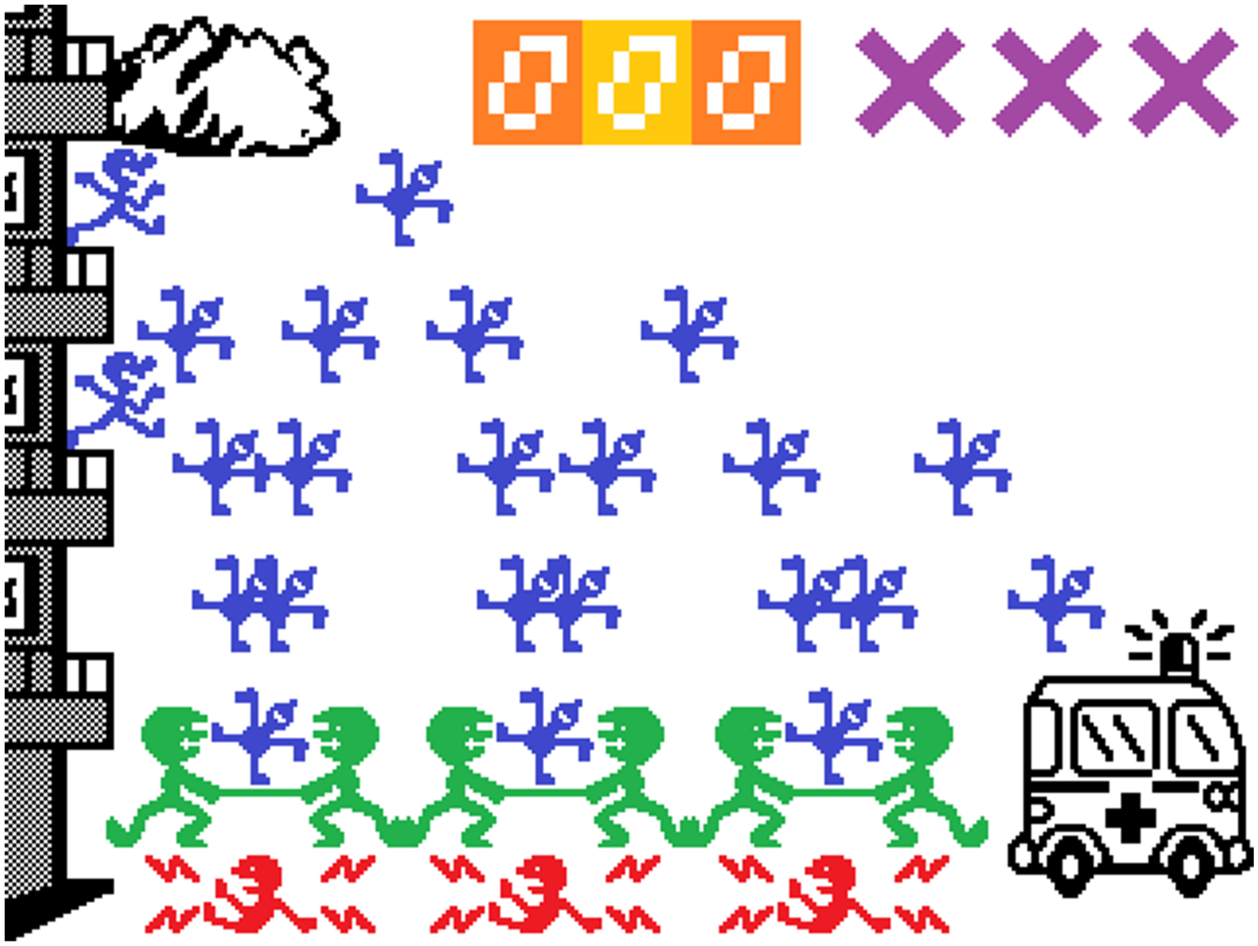


Appendix B: Custom Palette w/ Example Sprite

Code	Colour Name	Colour	RGB Colour
3’b000	Black	⬛	⬛
3’b001	White	⬜	🟦
3’b010	Blue	🟦	🟩
3’b011	Buttercup	🟧	🟦
3’b100	Tia Maria	🟫	🟥
3’b101	Dark Cyan	🟦	🟪
3’b110	Red	🟥	🟨
3’b111	Green	🟩	⬜



Appendix C: Layout of Fire Game

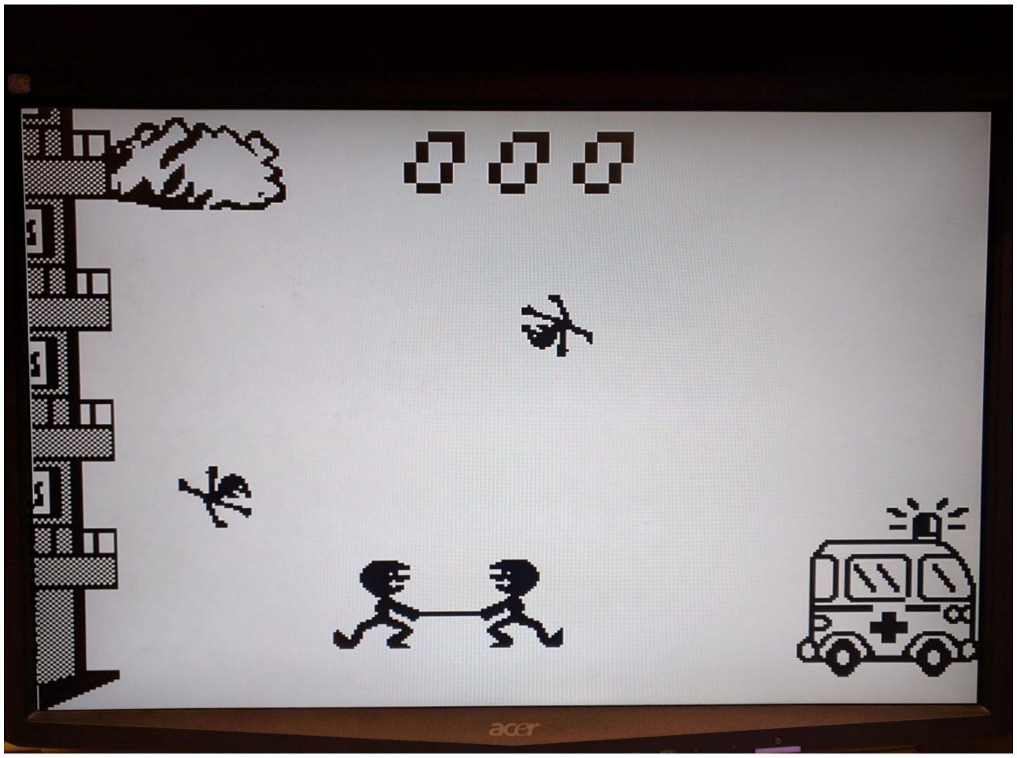


Appendix D: Sprite Positioning of Fire

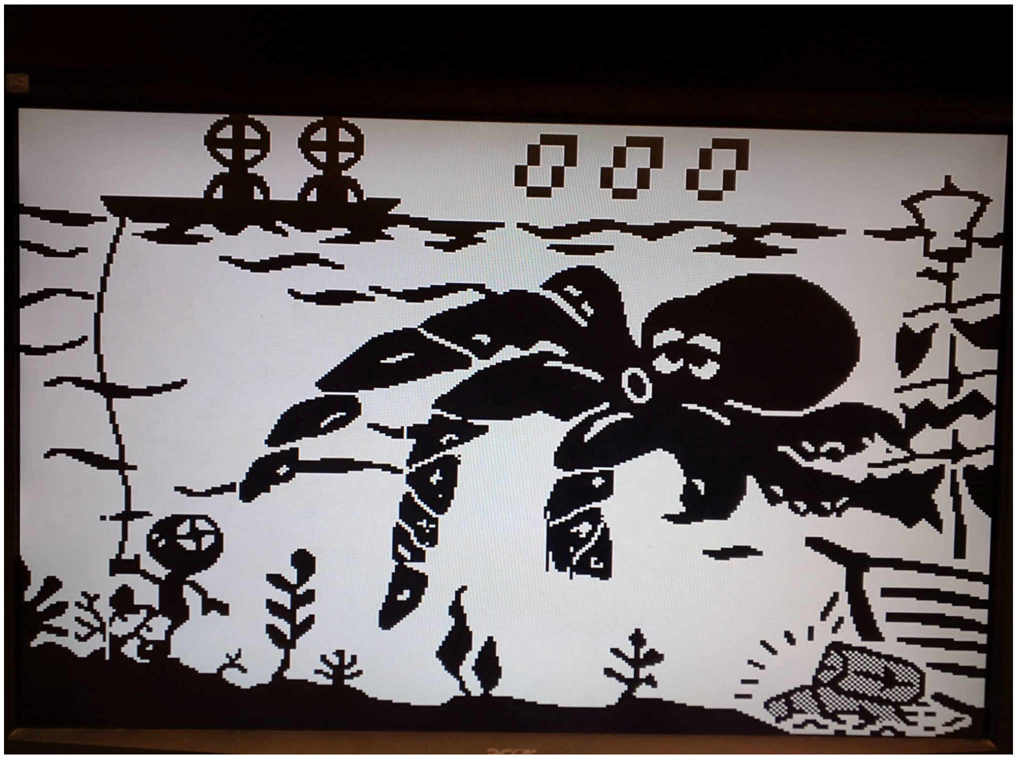
Figure	Left >=	Right <	Width	Top >=	Bottom <	Height
Jumper 2b	16	41	25	89	114	25
Jumper 1	16	41	25	37	62	25
Jumper 2	34	59	25	72	97	25
Jumper 3	42	67	25	106	131	25
Jumper 4	48	73	25	141	166	25
Jumper 5	53	78	25	175	200	25
Jumper 6	58	83	25	141	166	25
Jumper 7	64	89	25	106	131	25
Jumper 8	71	96	25	72	97	25
Jumper 9	90	115	25	37	62	25
Jumper 10	108	133	25	72	97	25
Jumper 11	116	141	25	106	131	25
Jumper 12	121	146	25	141	166	25
Jumper 13	126	151	25	175	200	25
Jumper 14	133	158	25	141	166	25
Jumper 15	142	167	25	106	131	25
Jumper 16	163	188	25	72	97	25
Jumper 17	184	209	25	106	131	25
Jumper 18	193	218	25	141	166	25
Jumper 19	200	225	25	175	200	25
Jumper 20	209	234	25	141	166	25
Jumper 21	233	258	25	106	131	25
Jumper 22	257	282	25	141	166	25
Trampoline 1	26	105	79	180	216	36
Trampoline 2	99	178	79	180	216	36
Trampoline 3	173	252	79	180	216	36
Corpse 1	36	95	59	218	238	20
Corpse 2	109	168	59	218	238	20
Corpse 3	183	242	59	218	238	20
Ambulance	257	320	63	155	229	74
Smoke	0	86	86	0	39	39
Balcony	0	28	28	39	240	201
Score 1	120	148	28	4	36	32
Score 2	148	176	28	4	36	32
Score 3	176	204	28	4	36	32
Miss 1	218	246	28	6	34	28
Miss 2	253	281	28	6	34	28
Miss 3	288	316	28	6	34	28

Appendix E: Pictures of Final Product

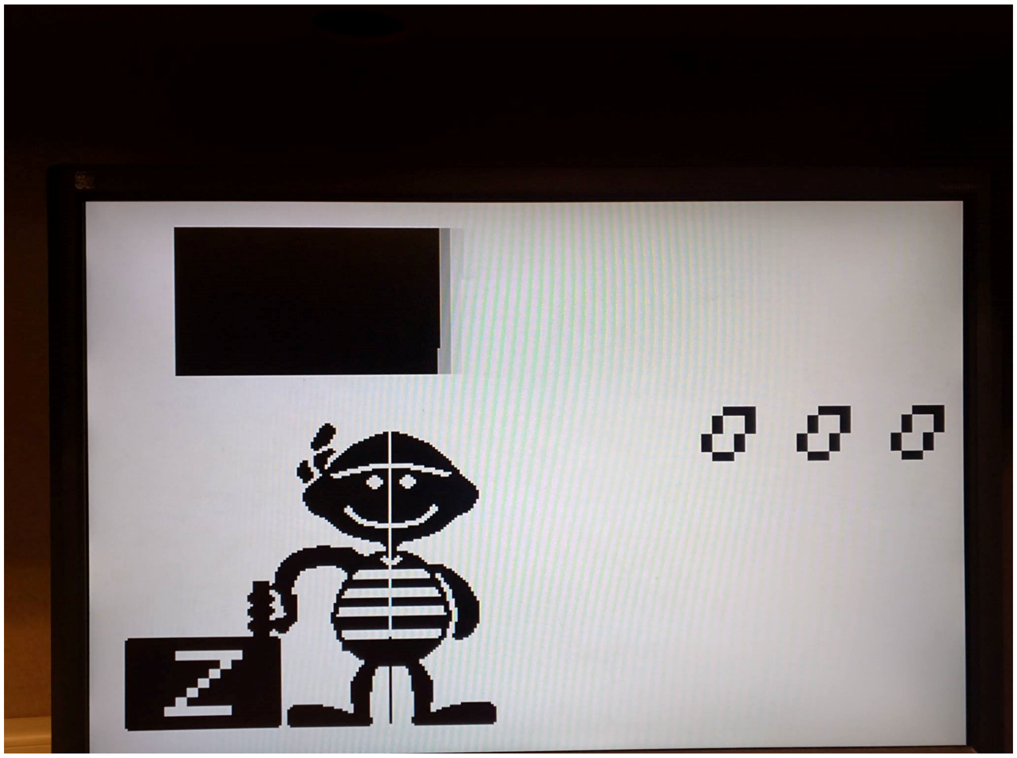
Splash Screen



FireA Game



OctoA Game



FlagB Game