Geometry Boy Project Report

Ben Agro: 1005857020 Jerry Chen: 1006269654 Shardul Ghuge: 1005964495

April 12, 2022

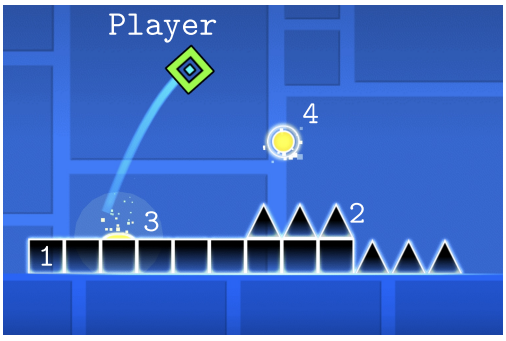
Introduction

The purpose of this project was to develop a version of the popular mobile game Geometry Dash called *Geometry Boy* that can run on the original *Game Boy Dot Matrix Game* (DMG). Geometry Dash is a horizontal auto-scrolling platform game available on iOS and Android [1]. The player taps the screen to control one of several vehicles while avoiding obstacles in an attempt to reach the end of the level.

This document summarizes the original proposal, our final design, an overview of relevant Game Boy hardware, software tools used during development, implementation details, and next steps for Geometry Boy.

Project Proposal Summary

We planned that Geometry Boy would start as a minimum viable version of Geometry Dash: the player controls a vehicle at a fixed horizontal position on the screen. Obstacles and objects the player can interact with scroll into frame from right to left. This basic version has only one vehicle, a square sprite, which the user controls with **one input** - the `Up` button on the Game Boy joypad (see figure 1b) - which makes the player jump a fixed height if the square is touching the ground. The game has four main objects: (1) **Blocks** that are safe to stand on, (2) **Spikes** that will end the current attempt, (3) **Jump Tiles** that automatically cause a large jump, and (4) **Jump Circles** that allow the player to jump in mid-air if the jump button is pressed (see figure 1a).



(a) Annotated screenshot from Geometry Dash. The pictured player is mid-jump, triggered by a jump tile.

(b) Game Boy console with circled jump control input.

The main shortcoming with the original proposal was that it only focused on gameplay and not on supporting structures (player selection, level selection, saving progress, etc.,). In retrospect, it was not ambitious enough because we under-estimated the Game Boy's capabilities (e.g., available code memory via Memory Bank Controllers), and was missing some key features we included in our final design (music, different vehicles, more obstacle types).

Final Design and High Level Program Structure

Our program was written in C and compiled for the Game Boy with the Game Boy Development Kit (GBDK). The high-level structure of the Geometry Boy program can be summarized by a state machine depicted in figure 2. The high-level state is encoded by three variables:

- `current_screen` $\in$ {`TITLE`, `GAME`, `LEVEL_SELECT`, `PLAYER_SELECT`}: Dictates which "main" function is running. These functions include `title()`, `game()`, `level_select()`, and `player_select()`, each of which return the next `current_screen` value on user input (causing a state transition).

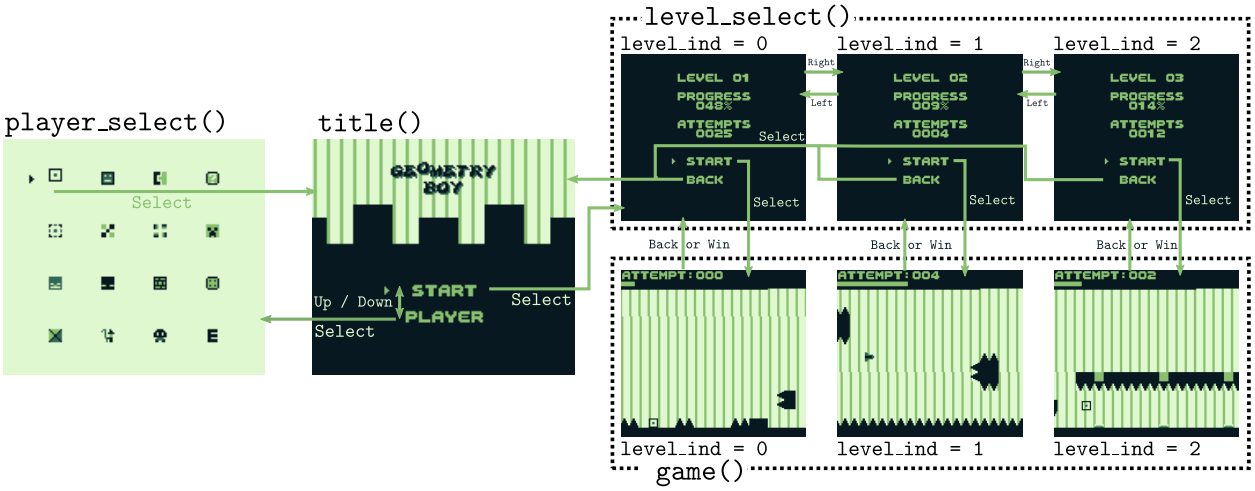


Figure 2: Block diagram of the high level state machine of Geometry Boy. Arrows denote state transition based on the listed user input.

- `level_ind` $\in \{0, 1, \ldots,$ `NUM_LEVELS` $- 1\}$: indicates the level (map and song) that will be played in the `game()` function and which level meta-data is displayed in the `level_select()` function. This state is only modified in the `level_select()` state/function when the player presses `Left` or `Right` on the joypad.
- `player_sprite_num` $\in \{0, 1, \ldots,$ `NUM_PLAYER_SPRITES` $- 1\}$: dictates which player sprite (8×8 pixel image representing the player) will be rendered in the `game()` function.

The **title screen** consists of a `while(1)` loop that implements an "infinitely" scrolling background with parallax (implementation described below) and a title with periodically "bouncing" letters. On this screen, the user can input `Up` or `Down` to move the *cursor* (green triangle) to point to the `START` or `PLAYER` letters, respectively. Pressing `Select` when the cursor is pointing to the `PLAYER` letters will cause a transition to `player_select()`, while it will cause a transition to the `level_select()` screen when pointing to the `START` letters.

The **player select screen** consists of a list of player sprites and a cursor that points to the current selection. Inputting `Up`, `Down`, `Left`, or `Right` will move the cursor to another sprite, with wrap-around on the rows and columns. The sprite that the cursor points to bounces up and down. Pressing `Select` will set the `player_sprite_num` variable such that the selected sprite is used in-game. It also causes a transition back to the title screen.

The **level select screen** consists of text indicating the selected level number, the saved progress (as a percentage of level completion) and the saved number of attempts (saving data is described below). The cursor points to either `START` or `BACK`, and is moved with the `Up` and `Down` joypad inputs. Pressing `Select` when the cursor points to `START` will begin the actual gameplay. If the cursor points to `BACK`, the screen transitions back to the title screen. Pressing `Right` or `Left` will increase or decrease the `level_ind` (respectively) and update the screen.

The **game screen** is where the actual gameplay takes place. The level map and player sprite are loaded according to `level_ind` and `player_sprite_num`, respectively. The game controls and features are described below. For implementation details see Implementing Geometry Boy.

Gameplay

The finished game is as described in project propsal with all four types of obstacles, with the addition of "vehicle" transitions. A variety of spike and block obstacles were added (different

orientations and sizes) but their function stays the same.

An additional *space ship* vehicle is added. Partway through certain levels, the player will pass through a "portal" and transition from a block to a spacecraft or vice versa. To control the spaceship vehicle, hold the `Up` button to move the spaceship up. Releasing the `Up` button lets the spaceship fall at a constant speed (see the image under `game()`, `level_ind = 1` in figure 2).

The game play is enhanced by additional features, including parallax and music (unique to each level).

Game Boy Hardware

This section details the Game Boy hardware *relevant to the project*; features we used and limitations imposed by the hardware. This information will be relevant in the subsequent description of the software implementation of Geometry Boy.

Memory Map

Figure 3 provides a diagram of Game Boy memory (simplified to only include low memory portions relevant to our discussion) [2]. The code read-only memory (ROM) stored in the cartridge is mapped to the lowest 32 kB in memory [2]. While the bottom 16 kB of this ROM is fixed (*memory bank* 0), the top 16 kB bank can be swapped for another 16 kB *memory bank* in the cartridge to allow for ROMs larger than 32 kB [2]. Switching ROM banks is achieved by a *Memory Bank Controller* (MBC) chip that is located in the game cartridge itself [2]. Bank 0 also stores the *cartridge header*, which declares information like the entry point to the program and the cartridge type [2].

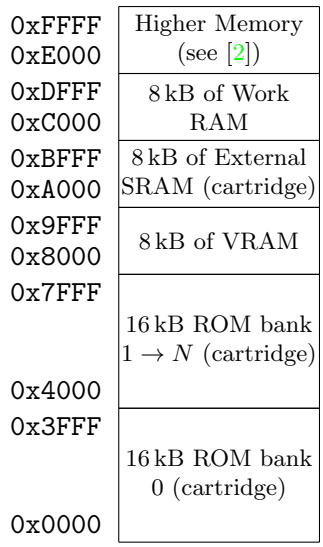


Figure 3: A portion of the memory map of the Game Boy [2].

Cartridge Type

The cartridge type specifies which MBC is used (if any) and if other external hardware exists in the cartridge [2]. For Geometry Boy, we used a type 0x1B cartridge. This type specifies that our cartridge uses memory bank controller chip `MBC5`, and that the cartridge has battery-backed SRAM (which is mapped to `0xA000 - 0xBFFF`, see figure 3) [2]. `MBC5` can control a total of 8 MB of ROM and 128 kB of external SRAM. This extra ROM is necessary to store long level maps. The battery-backed SRAM allows for the storage of game data (level progress, number of attempts) while the Game Boy is turned off. `MBC5` comes with memory-mapped registers for enabling RAM prior to reading or writing, and selecting the current ROM bank number mapped to `0x4000 - 0x7FFF` (whose functionality is abstracted into macros by GBDK) [2] [3]. Other MBCs have similar memory capabilities (e.g., `MBC1`), but we choose `MBC5` because it is the newest cartridge type and thus the easiest to source if we wanted to run Geometry Boy on a real Game Boy [4].

Rendering

The VRAM in figure 3 is shared between the CPU and the Picture Processing Unit (PPU), but cannot be accessed by both at the same time [2]. The CPU does the graphics calculations to update VRAM, while the PPU renders the image to the `160 x 144` pixel (width, height) LCD screen [2]. Each pixel can display four shades of grey [2]. The LCD screen is drawn row-by-row in pixel rows called scan-lines [2].

To reduce computational load, the Game Boy's pixels are not manipulated individually [2]. Instead, pixels are grouped into 8 by 8 squares called **tiles** [2]. A tile assigns a color ID to each

of its pixels, ranging from 0 to 3 [2], so each tile takes (2)(8)(8) bits, or 16 bytes of memory. Tile data is stored in VRAM at `0x8000 - 0x97FF`, enough space for 384 tiles [2].

The PPU draws three layers of tiles to the screen: background, objects, and window (from back to front) [2]. The **background layer** is composed of a `32 x 32` tile (`256 x 256` pixel) map called a **tile map**, only a portion of which is displayed to the `160 x 144` pixel screen [2]. The tile map does not explicitly store tile data, but just an index of the tile in VRAM [2]. There is space for two tile maps in VRAM, located in memory ranges `0x9800 - 0x9BFF` and `0x9C00 - 0x9FFF` [2]. Scrolling is achieved by writing to the `SCX` and `SCY` registers of the LCD, which specify the top left coordinates of the visible `160 x 144` pixel area in the `256 x 256` tile map [2]. These registers are memory mapped to `0xFF43` and `0xFF42`, respectively [2].

The **object layer** contains tiles that can move independently around the screen, commonly called sprites [2]. Instead of four shades of grey, sprites have one transparent color, and three shades of grey [2]. Their tile data is located in VRAM at `0x8000 - 0x8FFF`, and their attributes (x position, y position, and tile index) are located in Object Attribute Memory (OAM) located at `0xF000 - 0xFE9F` [2]. One main limitation of the Game Boy hardware is that the PPU *can only display up to 40 sprites with a maximum of 10 per scan-line*, so they were used sparingly in Geometry Boy [2].

The **window layer** is a non-scrollable background-like layer (using tile map data), without any transparency (see the STAT interrupt for details on how this is used) [2]. Various bits of the **LCD control register (LCDC)** (memory mapped to `0xFF40`) determine if the window layer, background layer, and object layers are drawn on the LCD [2].

Interrupts

Here we describe the interrupts used in Geometry Boy. Details on how they are used can be found below.

When all of the LCD scan-lines are drawn (which occurs at $\sim 60\,\text{Hz}$) a **vertical blank interrupt (VBlank)** is generated [2]. After this, the CPU can update the VRAM with the next frame of the game (e.g., updated sprite position, background scrolling) [2].

The **LCD Y Coordinate (LY)** register (memory mapped to `0xFF44`) holds the current scan-line [2]. The **LY Compare (LYC)** register (memory mapped to `0xFF45`) is constantly compared with the `LY` register [2]. If they are equal, then the `LY = LYC` flag (bit 2) of the **LCD Status Register (STAT)** is set [2]. Various LCD related interrupts can be enabled by setting bits in the `STAT` register [2]. Setting bit 6 enables the **LCY=LY STAT interrupt**, which is generated when `LCY = LY` [2].

Joypad

The Game Boy joypad has four **action buttons** (`Down`, `Up`, `Left`, `Right`) and four **select buttons** (`Start`, `Select`, `B`, `A`), as pictured in figure 1b. The **JOYP** register (memory mapped to `0xFF00`) has two bits for choosing if the action buttons or direction buttons are read, and then four bits indicating if any of the four selected buttons are pressed.

Sound Controller

Audio from the Game Boy is controlled by the Audio Processing Unit (APU), which can produce sound in four channels, each with a different waveform:

- Tone and Sweep (CH1): Distinct beep sounds for melodies, implemented with *quadrangular wave patterns* with sweep (changing frequency) and envelope (changing amplitude) functions [2] [5].
- Tone (CH2): Same as CH1 but without the frequency sweep functionality [2].

- Wave Output (CH3): A four bit digital to analog converter (DAC) that plays custom wave forms specified by 32 four bit samples [5].
- Noise (CH4): A set of white noise wave-forms that sound like static (percussion and ambient sounds) [2].

Each channel has four to five memory mapped control registers, and CH3 has RAM for storing the 4 bit samples [2]. There are "global" sound registers for volume and turning sound on/off [2]. Each channel circuit produces and digital value in the range `0x0 - 0xF`, which is then passed through a DAC (one for each channel) to create an analog value [2]. These analog values are mixed, scaled, and sent to the output (amplifier, filter, and speaker) [2].

Toolchain

GBDK

The Game Boy Development Kit (GBDK) provides a C compiler and linker based on the Small Device C Compiler (SDCC) to generate `.gb` ROM files from C code. The Geometry Boy project uses the GBDK front-end compiler `lcc` in a `Makefile` to generate object files like:

```
lcc -Wf-bo# -c -o object.o source.c
```

where `#` is replaced with a number indicating the ROM bank where the code data will reside. In the link stage (where the `.o` files are combined into a `.gb` ROM file), we can specify a type 0x1B cartridge by passing the flag `-Wl-yt0x1B` and the number of required ROM/RAM banks with `-Wl-yo#`/`-Wl-ya#`, where `#` is a power of 2. Geometry Boy uses 5 total ROM banks and 1 RAM bank, so we pass `-Wl-yo8` and `-Wl-ya1`, respectively.

Apart from compiling higher-level C code into assembly for the Game Boy, GBDK also provides various libraries to abstract the details Game Boy hardware from the programmer. Geometry Boy used various GBDK marcros and functions:

- `SHOW_BKG`, `HIDE_WIN`, `SHOW_WIN`, `HIDE_SPRITES`, and `SHOW_SPRITES` set/unset bits of the LCD status register (STAT) to show/hide the layers rendered by the PPU.
- `set_bkg_data`, `set_sprite_data`: load tile data into the background/window tile VRAM or object tile VRAM.
- `set_win_tile_xy`, `set_bkg_tile_xy`, `get_bkg_tile_xy`, `set_bkg_tiles`: Get/set background tilemap VRAM data with individual tile indices or from arrays of tile indices.
- `SCX_REG`: The SCX register, used to scroll the Geometry Boy level in the x-direction.
- `SWITCH_ROM_MBC5`: Switch the ROM bank that MBC5 is mapping to `0x4000 - 0x7FFF`.
- `ENABLE_RAM_MBC5`: Enable external cartridge SRAM to read/write game data to.
- `move_sprite`, `set_sprite_tile`: set the sprite pixel (x, y) position or tile index (appearance) by modifying the OAM.
- `disable_interrupts`, `enable_interrupts`, `add_LCD`, `add_VBL`: enable/disable interrupts, add interrupt handlers for LCD interrupts and VBlank interrupts. The `LYC` register and `STAT` register can be accessed with the macros `LYC_REG` and `STAT_REG`, which are used to set up the LCD scan line interrupt.
- `wait_vbl_done()`: halts the CPU until the VBlank interrupt is done, prevents the updating the VRAM more frequently than the PPU can draw it to the LCD.
- `joypad()`: Reads and returns the state of the `JOYP` register as per Nintendo's specifications (debounces the input) [3]. GBDK provides macros to identify which button is pressed (returned from `joypad()`): `J_START`, `J_SELECT`, `J_UP` etc.

GBMD and GBTD

The Game Boy Tile Designer (GBTD) and Game Boy Map Designer (GBMD) are applications written by Harry Mulder that use a GUI to design tiles and maps for the Game Boy. With the GBTD, you can design sets of `8 x 8` pixel tiles, and export them to `.c/.h` files with an array of `char`s representing the tiles. This tile data can be loaded into VRAM with the GBDK functions `set_bkg_data` and `set_sprite_data`. Using a previously designed tileset, the GBMD allows you to draw maps and export `.c/.h` files with an array of chars, each char representing a tile in the map by the index of that tile in the tileset. The GBDK provides the function `set_bkg_tiles` to set multiple tiles in the background tile map VRAM from these map arrays.

We encountered difficulties when designing letter tiles - which are challenging to make readable and appealing given limited pixel art experience. Thus, we wrote a python program `img_to_tile.py` that parses any image with less than four colors into a grid of tiles encoded as character arrays. We used this program to parse 8x8 font tilesets sourced online [6] into tile data for in Geometry Boy.

OpenMPT and GBT-player

OpenMPT is a popular music sequencing software used to create the background music for Geometry Boy [7]. OpenMPT allows for four channel audio editing, aligning with the Game Boy's APU, and it can export `.mod` sound samples used by the **GBT-Player library** [7]. GBT-Player is an open-source music player library for the Game Boy. It has a **mod2gbt** executable that converts a `.mod` file into `.c` arrays representing the song [8]. Further, it has a library of assembly language functions to play the song on the Game Boy [8] that decode the array of music pattern data row by row and load data into the registers/RAM areas of the corresponding APU channels [8].

Virtual Platform

Without access to a real Game Boy, this project used the **BGB** emulator to run the `geometry_boy.gb` ROM. This emulator was chosen for its high accuracy, debugging features (e.g., VRAM viewer), and cross-platform compatibility (can run on Linux with `wine`). The VRAM viewer debugging feature proved essential for learning how to program for the Game Boy, and allowed us to expand the scope of the project based (e.g., parallax, "infinite scrolling").

BGB emulates many features of the real Game Boy that are listed on the BGB website [9], including clock-exact LCD and sprite emulation, and accurate sound output. It has been extensively tested on many Game Boy ROMs; *"if your ROM works in BGB, it will most likely work on hardware too"* [9].

To **run this project on a real Game Boy**, we need three pieces of hardware: a Game Boy, a type 0x1B flash cartridge, and a flash cartridge reader/writer. Re-programmable type 0x1B cartridges (with MBC5 and SRAM) are available from insideGadgets for 22 USD [10].

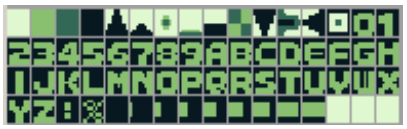
A flash cartridge reader/writer like the GBxCart RW (30 USD from insideGadgets, cross-platform compatible) provides an interface between the computer and the flash cartridge, and a GUI to allow `geometry_boy.gb` to be "flashed" onto the cartridge [11]. The cartridge can then be inserted into the Game Boy and run like any other game. The Game Boy itself is the most expensive piece of hardware; we found listings on amazon for 100 USD [12].

Implementing Geometry Boy

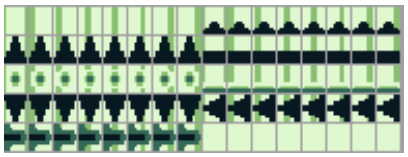
Below we describe the implementation of various features of the Geometry Boy, with reference to the Game Boy's hardware capabilities described above.

Tiles, Maps, and Parallax

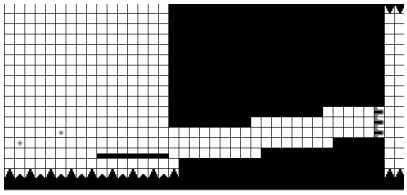
Figure 4a shows the tiles that are loaded into background VRAM while the game is running. These are the tiles that are used to construct the level maps (figure 4c) and the on-screen text and progress bar (figure 2).



(a) Tile VRAM.



(b) Parallax Tiles.



(c) A portion of Level 1 in GBMD.

Figure 4: Tile VRAM and maps in Geometry Boy. Note that the parallax tiles are not all loaded into VRAM at one time, instead they are stored in code ROM.

Parallax is a technique where some background objects scroll slower so they appear further away [13]. Implementing parallax on the Game Boy is difficult because there is only one background layer, and the level obstacles use this layer [13]. (Due to hardware limitations - max 40 sprites, 10 per LCD line - we cannot use the sprite layer for this purpose). To generate the illusion of parallax, for each tile in figure 4a that the player should be able to "see behind" (e.g., the spike tiles) we construct a corresponding *parallax tileset* shown in figure 4b. Each parallax tileset consists of eight tiles with a green line that moves left to right behind the main object, opposite to the direction the game scrolls. After constructing the map using the tileset in figure 4a, we can simply substitute the tiles from the parallax tilesets into the index of the original tile in tile VRAM to make it appear where the original tile was placed in the map (recall that the map just stores tile indices which are loaded into background VRAM). Beginning from the 0^{th} tile in each parallax tileset, each frame of the game we substitute the next tile from each parallax tileset into the same index in tile VRAM so the green line moves against the scrolling direction, providing the illusion that it is scrolling more slowly than the rest of the background.

ROM Bank organization

All the code for the main game functionality is in `geometry_boy.c`, which is stored in bank 0 (so its code data is always accessible). From `romusage`, an open-source tool for estimating the ROM usage of a `.gb` ROM file [14], we can see that this code takes up 14689 bytes, or 89% of the ROM bank. All the code from the `gbt_player` library is stored in bank 1. If the project were to continue, we would include all code in banks 0 and 1. All the music and tile data is stored in bank 2.

There are four maps used in Geometry Boy, one for the title screen and one for each of the three levels. The title screen map is `60 x 18` tiles (width, height), while each of the level maps is `455 x 18` tiles. With each tile index in the map taking 1 byte, the title map uses 1080 bytes, while each level map uses 8190 bytes (a level width of 455 was chosen so that two level maps could fit into one bank). The maps for the title screen and level 1 are stored in bank 3, while the maps for the levels 2 and 3 are stored in bank 4. More levels can be easily be added by using higher banks.

Scrolling

As aforementioned, the background map stored in VRAM is `32 x 32` tiles, too small for our `455 x 18` tile levels. Scrolling in the x-direction with `SCX_REG` sets the pixel x coordinate of the left edge of the view-able `20 x 18` tiles on the background tile map (`SCX_REG` $\in [0, 32(8) - 1] = [0, 255]$, see figure 5).

To allow for scrolling with maps larger than those that fit into memory, we use the background VRAM as a *rolling buffer* for the map data, which is dynamically loaded from code ROM (implemented in the function `scroll_bkg_x`). As the viewable area moves left to right (and then wraps around) in VRAM, we over-write the VRAM column that has just moved off the left edge of the viewable window with the tile indices of background map data 32 columns ahead. For example,

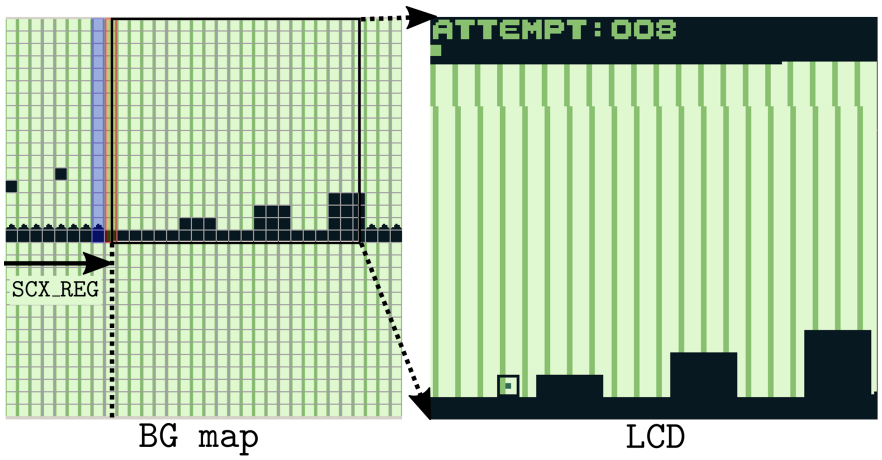


Figure 5: Scrolling in Geometry Boy

in figure 5, the blue column has just been hidden from view, and over-written with tile indices 32 columns ahead in the background map array. The red column is the next to be over-written. In this way, the player never sees the tiles changing, and the map appears to scroll contiguously as the viewable area wraps around to reveal these tiles.

Main Game Loop

The main game loop in `game()` looks like

```
1  while(1){
2      if (vbl_count == 0){ // global uint8_t
3          wait_vbl_done();
4      }
5      vbl_count = 0;
6      // ... game logic below, including background_x_shift += player_dx ...
7  }
```

`vbl_count` is incremented in a VBlank interrupt handler shown below, counting the number of VBlank interrupts that occurred during the last iteration of the game loop. The if statement on line 2 only waits for the VBlank interrupt (so the CPU can update the graphics for the next frame) if it has not yet already occurred during the last game loop, speeding up execution [15].

The game logic runs at a slower rate than the LCD screen updates (60 Hz). However, to keep scrolling smooth, we update the `SCX_REG` in the VBlank interrupt handler based on a stored global scroll variable `background_x_shift` [15]. We register a VBlank interrupt handler using `add_VBL` that has code like:

```
1  void vbl_interrupt_game() {
2      SHOW_WIN;
3      vbl_count++; // global uint8_t
4      old_scroll_x += (background_x_shift - old_scroll_x + 1) >> 1; // global uint16_t
5      // vbl_number | old_scroll_x | background_x_shift | SCX_REG |
6      //          1 |            0 |                   3 |       2 | (old_scroll_x += 4/2 = 2)
7      //          2 |            2 |                   3 |       3 | (old_scroll_x += 2/2 = 1)
8      //          3 |            3 |                   6 |       5 | (note player_dx = 3)
9      SCX_REG = old_scroll_x & 255; // equivalent to modulo 255
10 }
```

Line 4 interpolates `old_scoll_x` so that if this interrupt runs multiple times without `background_x_shift` changing, `SCX_REG` is updated more smoothly (with smaller increments), as shown in the comments below line 4 [15].

Notice that the VBlank interrupt handler shows the window layer. This means that the next frame will start drawing with the window shown. We include this so that we can display a progress bar and attempt counter on the window layer which will not scroll with the background layer. To ensure that the window layer does not cover the entire screen, we set `LYC_REG = 16` and `STAT_REG |= 0x40` (set bit 6) so that an interrupt occurs when the PPU draws the 16^{th} scan line (two tiles from the top of the LCD screen). We register a handler for this interrupt that hides the window layer, so the window layer is only displayed for the top two tiles of the screen (figure 5).

Saving Data

At the beginning of execution, we enable external SRAM with `ENABLE_RAM_MBC5`. At the start of SRAM (`0xA000`), we store a flag character `'s'` if the cartridge has saved data. If it is not `'s'`, we initialize two arrays of zeros of length `num_levels` to store the attempts and progress per level, and write `'s'` to the start of SRAM to indicate there is now saved data there. Whenever the player loses or wins a level, these arrays are updated with the number of attempts and best level progress.

Physics

While general to most platformer games, here we give an overview of our implementation of physics. The y-position and velocity of the player are tracked with the global variables `player_y` and `player_dy`. A variable `on_ground` tracks if the player is on the ground or not and `lose/win` track if the player has lost or won. The physics update is as follows:

1. If `Up` is pressed and `on_ground = 1`, give the player an upward jump velocity `player_dy = -PLAYER_JUMP_VEL` (the y-coordinate increases down the LCD screen).
2. If the player is not on the ground, increase the players downward velocity `player_dy += GRAVITY` to make it fall.
3. Check collisions in the x-direction (accounting for the new `background_scroll_x`, but without accounting for the new `player_dy`). Here we can check if the player has run horizontally into an obstacle or the finish line, setting `lose = 1` or `win = 1`, respectively.
4. Update `player_y += player_dy`. Assume the player is not on the ground by setting `on_ground = 0`.
5. Check collisions in the y-direction (accounting for the player's new y-velocity and position). Here we can check if the player is falling onto, or already on, a block and set `on_ground = 1` `player_dy = 0` and `player_y` accordingly (so the player rests on-top of and not inside a block). We also check if the new player position results in collisions with any other obstacles.

We check for collisions by iterating over the four tiles the player sprite is touching. For each of these tiles, we determine its index by reading background VRAM. Based on the tile index, we check if the collision box of the player overlaps with the collision box(es) of object under consideration by checking for rectangle overlap. This is illustrated in figure 6, where the player's red collision box is checked for overlap with the four red collision boxes of the spike. As shown, collision boxes are pixel perfect.

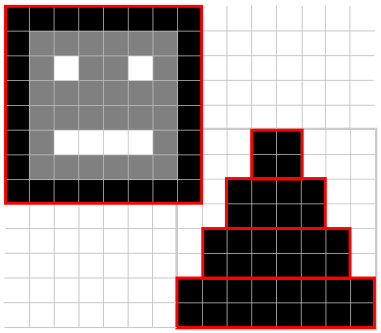


Figure 6: Collision checking the tile in the player's bottom right corner.

Music

First, a `.mod` file is generated for each of the songs via the OpenMPT software. Then, using the **mod2gbt** executable, the `.mod` file is converted to a `.c` file which has all the patterns of a particular

song stored as an array of chars. In the same `.c` file, these patterns are combined together in a larger array `song_Data` (one for each song) that orders the patterns of a song (i.e., orders the "verses" of the song). This procedure is repeated for all four songs (one for the main screen and three for the levels). A combined music file `music_sample.c` was created which stores the data for all the songs, which is compiled to `bank 2`.

Then, GBT-Player functions abstract the Game Boy music hardware details and are used to play the song. `gbt_play(song_Data, storage_bank)` is used to initialize a particular song in the music player by passing the array of `song_Data`, the location of the song via `storage_bank` (`bank 2` in our case). Furthermore, the songs were made to loop via a call to `gbt_loop(1)`. Lastly, the `gbt_update()` function is called in the game loop, which runs a single row of the music pattern in `song_Data` per call.

Other Optimizations

Functions that are called a few times but run many times are made **inline**. The function `rect_collision_with_player` checks if a collision box overlaps with the player's collision box. It is called more than a few times; however, we decided to trade larger code memory for reduced execution time (because collision checking is an expensive part of the game loop).

We used **minimal variable widths** to reduce code/stack memory usage [3]. We also used unsigned variables where possible to reduce the execution time of various operations (e.g., comparison) [3]. We used global variables where possible, which use code memory instead of stack memory, and are generally more efficient on the Game Boy [3].

We used multiplication and division by powers of two where possible, and implemented these as bit-shifts. Whenever performing a modulo by a power of two (e.g., when setting the player's y-position to be tile-aligned), we instead used a bit mask (e.g., `x % 32 == x & 31`)[1].

Next Steps

Geometry Boy is a fully functional game that runs smoothly in an emulator. Our primary next step would be to test it on original Game Boy hardware to ensure compatibility. We expect few complications since the BGB emulator provides an authentic representation of the Game Boy.

Additionally, we would like to write the game **without the use of GBDK**, instead writing in assembly and using the *Rednex Game Boy Development System* (RGBDS) toolchain/linker to increase speed and control of the low-level logic (GBDK is known to be the simplest but certainly not the most optimized or transparent compiler) [16].

Some additional game features we want to add are:

- More vehicles: Geometry Dash currently has eight different vehicles, of which Geometry Boy only includes two [17].
- Gravity and scrolling inversion: Some sections of levels in Geometry Dash invert gravity or invert the scrolling direction [17].
- Custom level designer: Geometry Dash allows users to design, play, and share custom levels [17]. Sharing could be accomplished with an as yet unused Game Boy feature: **serial data transfer via a link cable**, which uses an interface similar to SPI.
- Timing music and levels: Geometry Dash doesn't just have music, the beats of the music are timed to the required jumps in the level.

[1] newer versions of SDCC will automatically make some of these optimizations [3], but we include them explicitly.

References

[1] Hackey5, *Geometry dash wiki*, Oct. 2021. [Online]. Available: https://geometry-dash.fandom.com/wiki/Geometry_Dash_Wiki.

[2] A. N. Díaz, A. Vivace, Beannaich, *et al.*, *Pan docs*, 2022. [Online]. Available: https://gbdev.io/pandocs/Specifications.html.

[3] P. Felber, L. Malmborg, M. Hope, and D. Galloway, *Gbdk 2020 docs*, 2020. [Online]. Available: https://gbdk-2020.github.io/gbdk-2020/docs/api/.

[4] *Choosing a flash cart*, 2022. [Online]. Available: https://shop.insidegadgets.com/choosing-a-flash-cart/.

[5] A. Bourque, *Game boy advance sound channel 1*. [Online]. Available: http://belogic.com/gba/channel1.shtml.

[6] WinTakeAll, *File:atari st character set 8x8.png*, 1985. [Online]. Available: https://commons.wikimedia.org/wiki/File:Atari_ST_character_set_8x8.png.

[7] *Openmpt*, 2022. [Online]. Available: https://openmpt.org/.

[8] A. N. Díaz, *Gbt player v3.0.8*, 2020. [Online]. Available: https://github.com/AntonioND/gbt-player.

[9] Lyth, *Bgb*, 2021. [Online]. Available: https://bgb.bircd.org/.

[10] *Gameboy 1mb, 32kb fram flash cart (ultra low power, custom boot logo, mbc1 mode)*, 2022. [Online]. Available: https://shop.insidegadgets.com/product/gameboy-2mb-32kb-fram-flash-cart-ultra-low-power/.

[11] *Gbxcart rw (gameboy/gbc/gba cart reader, writer & flasher)*, 2022. [Online]. Available: https://shop.insidegadgets.com/product/gbxcart-rw/.

[12] *Original game boy console*, 2022. [Online]. Available: https://www.amazon.ca/Nintendo-DMG-01-Original-Game-Console/dp/B000R08L7M.

[13] *Nes background parallax explained - audiovisual effects pt. 03*, Aug. 2020. [Online]. Available: https://www.youtube.com/watch?v=wt73KPS_23w&t=200s.

[14] bbbbbr, *Romusage*, Jan. 2022. [Online]. Available: https://github.com/bbbbbr/romusage/.

[15] Zalo, *Zalo ds blog*, Jun. 2016. [Online]. Available: http://zalods.blogspot.com/2016/07/game-boy-development-tips-and-tricks-ii.html.

[16] R. Eldred Habert, *Rgbds*, 2022. [Online]. Available: https://rgbds.gbdev.io/.

[17] *Portals — geometry dash*, Oct. 2021. [Online]. Available: https://geometry-dash.fandom.com/wiki/Portals.