ViewTube System Design Document

Ben Allison (bja2142) Jared Gonzales (jrg2221) Lynsey Haynes (lah2224)

Spring 2022

Table of Contents

Introduction

ViewTube will be a real-time display of the NYC subway train data. It will use a software API to fetch the subway data every 30 seconds, then display the information using a custom hardware module on a VGA display.

ViewTube will give a user interface so that users can switch train lines dynamically and scroll around a real-time display with trains depicted on a realistic map with an estimated real-time position displayed on the map. Users can scroll around the map to view their preferred area and change the line that is displayed via a keyboard button. A main thread will monitor the keyboard for input and update the display based on user input.

Bresenham's Line Algorithm will be used to display the subway lines between the subway systems.

System Block Diagram

Software Userspace

The MTA provides an API interface for free to registered users that provides the ability to query train status in real time. A python3 program will query this API in near-real time and keep a local cache of the train status for all lines.

A compiled binary will use a kernel driver to configure the initial display information and default line. Upon initialization, it will allocate data structures for the maximum number of required trains and initialize the display background and sprite graphics for each train via calls to the kernel driver. It will then launch a second thread and move into the keyboard routine.

The main thread will monitor the keyboard and update the line and display based on user input. If the user updates a line, it will update a global variable. If the user moves the display, it will send a call to the driver to scroll the display. If the user quits, it will close all threads and safely exit.

The second, launched thread will query the local python3 program over a pipe and request a list of the status of all trains for the selected line. For each train in the line, it will use the kernel driver to update the sprite at that position.

Software Kernel Component

The kernel driver interfaces with the hardware component via functions that access the MMIO memory registers via the avalon bus interface. The function prototypes are listed in the block diagram for this module and provide a concise summary of the interface between the hardware and software

Hardware Component

Using the avalon bus, the hardware provides an interface for various actions to the kernel module.

The background layer provides a canvas that the kernel can draw on to draw the map for a given line, including the lines and the subway stations. The entire layer has a single color. A bitmap of the layer is tracked in SRAM such that each bit corresponds to a 4x4 block of 16 pixels. There are a maximum of 9 total panels of 640x480 pixels allocated to this layer, with only 540x440 pixels visible at any given time.

The status bar and menu modules track and output the menu and status bar. The sprites track individual trains. The z-index pixel merger layers graphics on top of the background layer, such that the lowest index is closer to the back and the highest index is closest to the front.

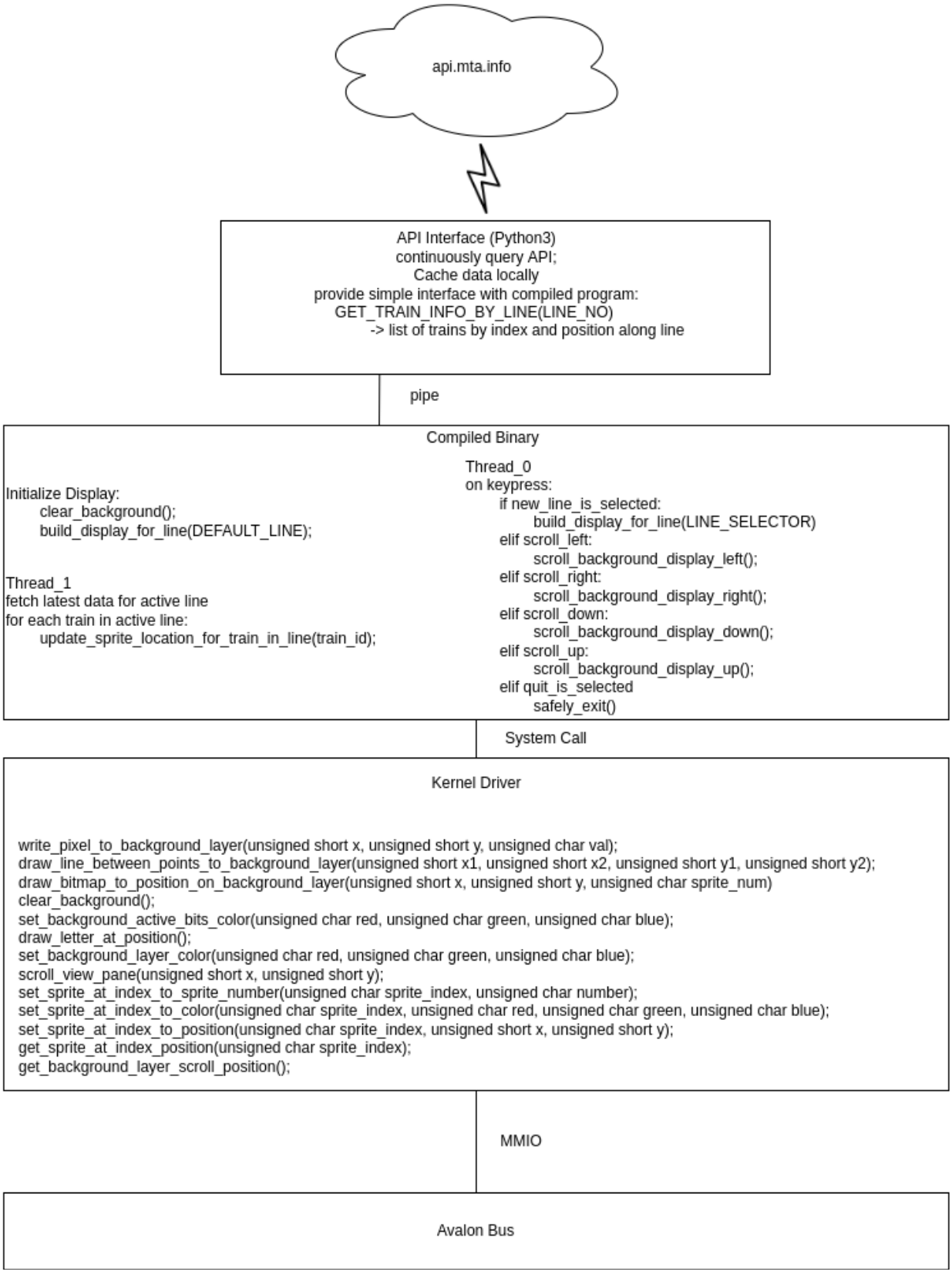

Figure 1: Software Component (User and Kernel Land)

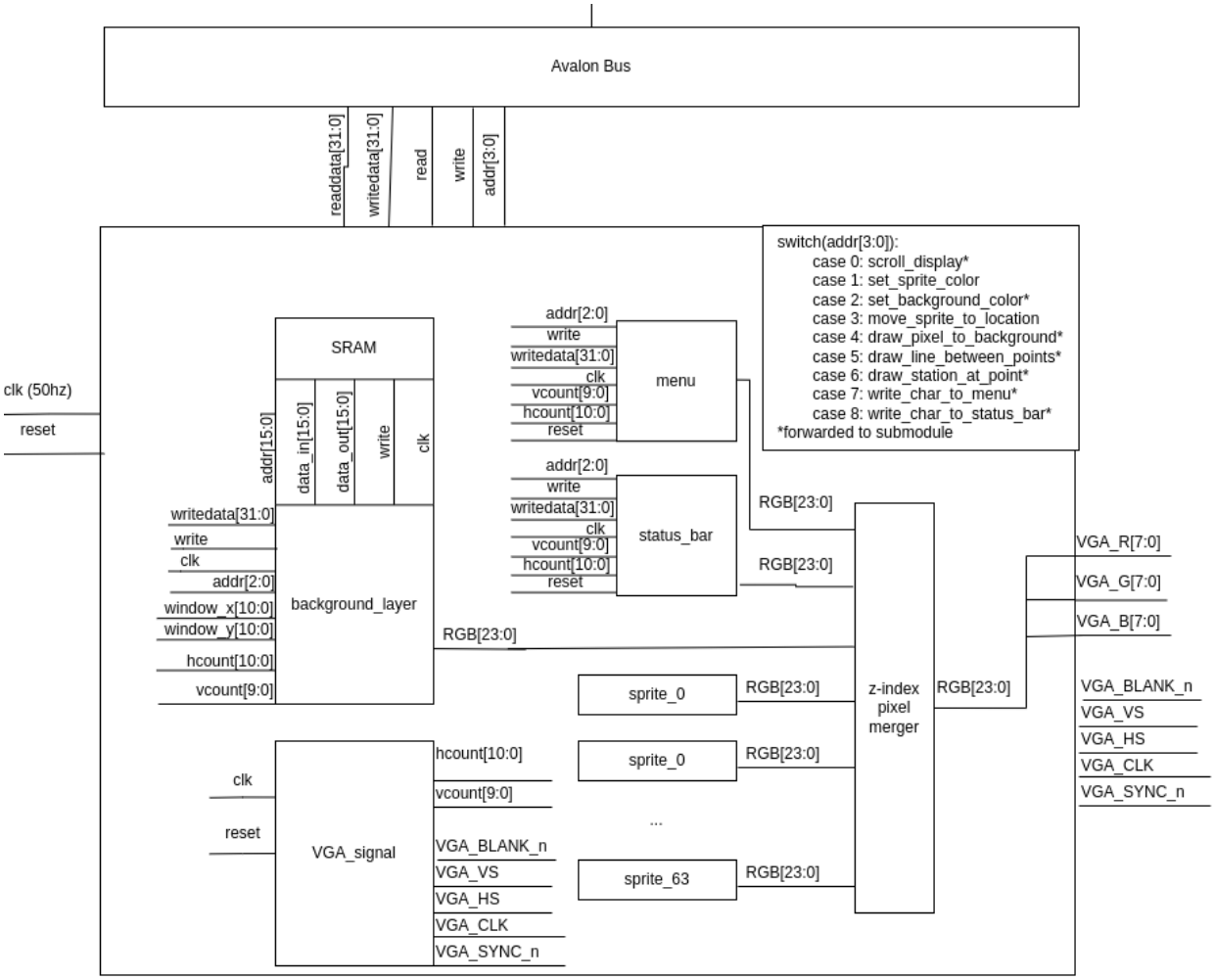

Figure 2: Hardware Component

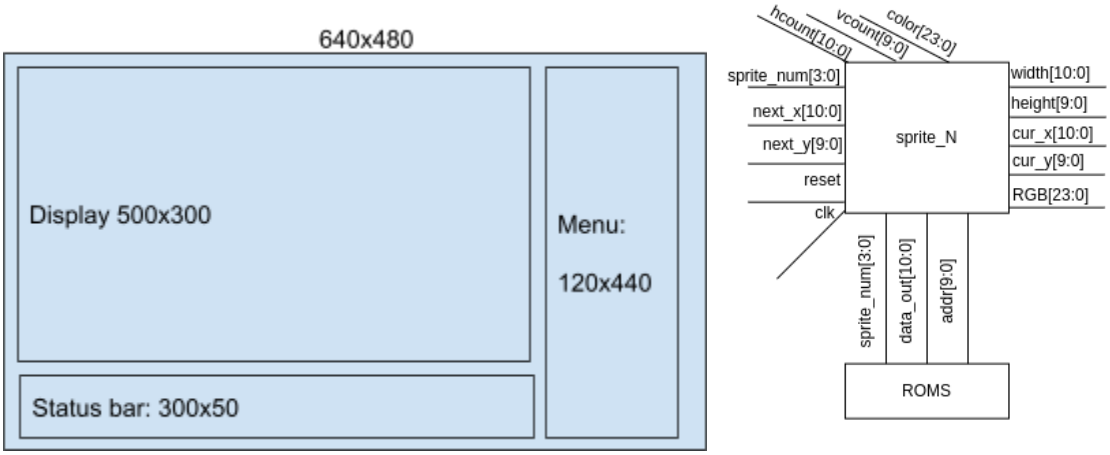

Figure 3a (left): Concept Sketch of physical display
Figure 3b (right): internals of a single sprite module

Algorithms

Bresenham's Line Algorithm

- Because the subway maps are too large to store individually in memory all at once, and reading from storage adds significant complexity and limited flexibility, the hardware component provides the kernel a canvas to draw maps dynamically. As part of that, the canvas provides the option to draw lines from two points.
- In practice, this is used to draw the lines of the Subway map to connect the stations
- The design follows the algorithm as defined in the lecture here: http://www.cs.columbia.edu/~sedwards/classes/2022/4840-spring/lines.pdf

The algorithms for compiled and scripted code in software land are characterized as necessary in the software hardware interface.

Resource Budgets

- **Background display buffer:** (16 pixels per bit (¼) * 4 bits per pixel * 640 x 480)/8 bits per byte * 9 panels = 86,400 bytes.
- **Train Sprite:** 40x40 bits (single color) = 200 bytes
- **Sprites for 70 characters** (single color)
    - 8 pixels x 16 pixels (single color = 128 bits == 16 Bytes
    - 70 * 16 = 1,120 bytes
- **Status Bar display buffer:** (50 pixel x 300 pixels x 1 bit per pixel)/8 bits per byte = 1,875 bytes
- **Menu display buffer:** 120 pixels x 440 pixels x 8 bit color = 52,800 bytes

Total: 142,395 Bytes, rounding up to 524,288 Bytes (150KB) which is less than 496KB with a large margin.

The Software/Hardware Interface

Pipe Between Python3 and compiled component

To communicate with the compiled userland program, a simple pipe is implemented from stdout of a second thread, running the python program to query the MTA API, to the stdin of the compiled userland program. The python program will use the requests library to send HTTP GET requests to the API endpoint. It will do basic parsing to make sure that only relevant data is passed to the userspace program that will update the display with the appropriate information.

API data from api.wheresthefuckingtrain.com/by-route/{ROUTE NUMBER}

- Data is separated by station and further categorized by North/South trains
- Example:


```
▼0: {,…}
  ▼N: [{route: "1", time: "2022-03-30T13:57:47-04:00"}, {route: "1", time: "2022-03-30T13:59:59-04:00"},…]
    ▶0: {route: "1", time: "2022-03-30T13:57:47-04:00"}
    ▶1: {route: "1", time: "2022-03-30T13:59:59-04:00"}
    ▶2: {route: "1", time: "2022-03-30T14:00:06-04:00"}
    ▶3: {route: "1", time: "2022-03-30T14:02:07-04:00"}
    ▶4: {route: "1", time: "2022-03-30T14:07:43-04:00"}
    ▶5: {route: "1", time: "2022-03-30T14:13:20-04:00"}
    ▶6: {route: "1", time: "2022-03-30T14:20:03-04:00"}
  ▼S: [{route: "1", time: "2022-03-30T13:52:49-04:00"}, {route: "1", time: "2022-03-30T13:57:06-04:00"},…]
    ▶0: {route: "1", time: "2022-03-30T13:52:49-04:00"}
    ▶1: {route: "1", time: "2022-03-30T13:57:06-04:00"}
    ▶2: {route: "1", time: "2022-03-30T14:02:32-04:00"}
    ▶3: {route: "1", time: "2022-03-30T14:09:43-04:00"}
    ▶4: {route: "1", time: "2022-03-30T14:16:43-04:00"}
    id: "07e1"
    last_update: "2022-03-30T13:52:38-04:00"
  ▶location: [40.799446, -73.968379]
    name: "103 St"
  ▶routes: ["1"]
  ▼stops: {119: [40.799446, -73.968379]}
    ▶119: [40.799446, -73.968379]
▼1: {,…}
  ▶N: [{route: "1", time: "2022-03-30T13:59:47-04:00"}, {route: "1", time: "2022-03-30T14:01:59-04:00"},…]
  ▶S: [{route: "1", time: "2022-03-30T13:55:06-04:00"}, {route: "1", time: "2022-03-30T14:00:32-04:00"},…]
    id: "0d51"
    last_update: "2022-03-30T13:52:38-04:00"
  ▶location: [40.807722, -73.96411]
    name: "116 St - Columbia University"
  ▶routes: ["1"]
  ▶stops: {117: [40.807722, -73.96411]}
```


- We want to track each individual train so we test whether arrivals are after the earliest arrival times at the previous station. If this is the case, then the train must be at the previous station and is excluded from that segment of the line.

  Pseudocode:

```
for arrival in current_station.north_arrivals:
    if (arrival.time - prev_station.north_next_arrival.time) > 0:
        add_train_to_segment_from(prev_station, current_station)
```

The API only needs to communicate information for each active train and can cache the rest of the information to be referenced later to keep information up to date and account for delays. Between the API and the userland we pass the number of trains, their current position on the map (as a segment between two stations), and the expected arrival time at the next station.

Train_Info_By_Line (struct)
- — Northbound_train_count (int)
- — Southbound_train_count (int)
- — Northbound_trains (array)
  - — Train0 (struct)
    - — Current_position_segment (int)
    - — Expected_arrival_time_in_seconds (int)
  - — Train1 (struct)
    - — Current_position_segment (int)
    - — Expected_arrival_time_in_seconds (int)
  - — TrainN (struct)
    - — Current_position_segment (int)
    - — Expected_arrival_time_in_seconds (int)
- — Southbound_trains (array)
  - — Train0 (struct)
    - — Current_position_segment (int)
    - — Expected_arrival_time_in_seconds (int)
  - — Train1 (struct)
    - — Current_position_segment (int)
    - — Expected_arrival_time_in_seconds (int)
  - — TrainN (struct)
    - — Current_position_segment (int)
    - — Expected_arrival_time_in_seconds (int)

Interface between userland and kernel

The main userland program will be running in Thread0 which will wait for keypresses on the keyboard. When a key is pressed, there are five possible outcomes. Four outcomes are moving the screen depending on which arrow key was pressed. The fifth outcome is that a key which corresponds to a subway line was pressed. Screen movement sends the direction using 1 bit to determine the x direction and a second bit to determine the y direction.

This program also knows the Subway lines and stops which may be hard coded because this is still an easy piece of code to update if there ever were changes to the infrastructure. Taking the input from the Python program running in Thread1 it parses this data while drawing the lines to also draw train sprites on top of the line. Each segment is calculated in the same way as in the Python program, from the furthest station south at index 0 and the northernmost station at index N. The expected arrival time is used to determine the progress towards the next stop that the train has made and how fast it will be arriving at the next station as an approximation for the speed that it is traveling.

Functions:

- clear_background()
  - System Call to clear_background() which will remove sprites, lines, and background colors.
- build_display_for_line(LINE_SELECTOR)

    - Sets background color, draws stations and lines, initializes active sprites
- update_sprite_location_for_train_in_line(train_id)
    - Updates the location of a sprite based on the index/position (starting from furthest south)
- scroll_background_display_up()
- scroll_background_dsiplay_right()
- scroll_background_display_down()
- scroll_background_display_left()
    - Move screen with scroll_view_pane() and updating the position of the sprites opposite of the scroll direction

Interface between kernel and FPGA

Each register is 32 bits long. A register corresponds to a write to the FPGA over the avalon bus using writedata, where the addr[3:0] value is equal to the register number.

Register 0: Scroll Display

(MSB)	31:17	16	15:1	0
	padding	Scroll y 0 = down 1 = up	padding	Scroll x 0 = left 1 = right

Padding is added to align data along unsigned short

Register 1: Set Sprite Color

(MSB)	31:24	23:16	15:8	7:0
	Sprite #	Red	Green	Blue

Set the color for a given sprite number.

Register 2: Set Display Background Color

(MSB)	31:28	27:24	23:16	15:8	7:0

	padding	index	R	G	B

Set the display buffer bitmap color at an index to the given color

Register 3: Move Sprite to Location

(MSB)	31:24	23:21	20:11	10:0
	Sprite #	padding	Y Position	X position

Move sprite to a location in the visible display.

Register 4: Draw Pixel to Background

(MSB)	31:24	23:16	15:8	7:0
	padding	Y Position	padding	X Position

Draw a pixel on the background display buffer. Note that the resolution is reduced to fit a single byte.

Register 5: Draw Line Between Points

(MSB)	31:24	23:16	15:8	7:0
	X1 position	Y1 position	X2 Position	Y2 position

Note here that this draws a line to the background layer display buffer, and can draw anywhere in the visible or hidden portions of the canvas. The x/y positions have reduced resolution to fit in a single byte.

Register 6: Draw Station at Point

(MSB)	31:26	25:16	15:11	10:0
	padding	Y Position	padding	X position

Draws a station at a position on the background layer display buffer.

Register 7: Write Char to Menu

(MSB)	31:24	23:16	15:8	7:0
	padding	char	Y Position	X Position

Write a character to a position within the menu display buffer.

Register 8: Write Char to Status Bar

(MSB)	31:24	23:16	15:8	7:0
	padding	char	Y Position	X Position

Write a character to a position within the status bar display buffer.

Other (Reserved)

Registers 9-15 reserved for future use

Keyboard Controls

Selecting lines is done with the keyboard (1 Line by pressing “1”, A Line by pressing “a”, etc)
Scrolling up/down/left/right is done by the arrow keys.

Milestones

25% Milestone: scrolling hardware display buffer

Implement the background display buffer with ability to draw shapes from usermode into the display buffer and control scrolling in the buffer with the keyboard.

50% Milestone: animated trains

Successfully animate a train along all stations of a single line, with the ability to scroll the display along the line to follow the train as it moves through each station.

75% Milestone: live data on a single line

Successfully display all trains in a single line with real-time data from the MTA API. Menus correctly display text.

Final Milestone: multiple train lines

Ability to toggle through all train lines, successfully clearing and redrawing the entire display for each new line.