CSE 111 Team 9 Report

Contributions and Highlights

Ryan Welter

OS Placeholder Text

Disassembler Placeholder Text

Troubleshooting/Cleanup Placeholder Text

Wyatt Avilla

CPU I'm particularly proud of the CPU, where I was able to offload a significant amount of computation to compile time while still maintaining readability and adherence to the spec.

Specifically, I used `contexpr` to build a jump table of the various CPU methods. Avoiding run-time initialization meant that the compiler was able to help us catch bugs with the table implementation that startup time for the emulator was faster.

In terms of readability, I used `std::variant` to allow methods with different signatures to be placed into the same jump table. This meant that the CPU method implementations matched the spec exactly while still being callable in a general way. For specific details, see `cpu.h` and `cpu.cpp`.

GitHub Actions Our project has a robust set of checks actualized by GitHub Actions. Firstly, everything in our repository is formatted. YAML and Markdown files are formatted with Prettier, and C++ source files are formatted with clang-format. Additionally, the correctness of our emulator is assessed with a compilation check followed by a set of tests against the provided "hello world" slug files. Then, our project is compiled with `-Wall -Wextra -Werror` to ensure that no compiler warnings are present. Finally, clang-tidy is run with close-to every lint rule enabled to help guard against potential bugs.

Build Reproducibility We used Git submodules to pin the exact version of the provided starter code and each of our dependencies. Although Cmake provides a `find_package` function, Git submodules allow for a finer-grained control over libraries that's agnostic to distro package managers. This approach ensures that all teammates (and potential users) utilize the same library versions and build the same binary.

Additionally, building from source allows us to manipulate the build parameters of our dependencies. If necessary, we could compile our dependencies with

debug flags to give us richer stack traces or with thread sanitization to ensure concurrent safety.

GitHub Issues Integration I used GitHub issues to keep track of bugs, in-progress features, and to-dos. The "tags" and "assignees" features made it extremely easy to see everyone's contributions. For example, you're able to see who worked on what features by filtering closed issues by the "extra features" tag.

Another highlight of using GitHub issues was how easy it made checkoff 1. Opening an issue for the checkoff meant that Charles could directly reference places in our codebase that needed to be changed. Further, it gave us a centralized place for a to-do list that all team members could modify and allowed us to track the changes related to the review.

Monisha Garika

Controller Placeholder Text

GUI Color Selector Placeholder Text

Problem With Color Setting Placeholder Text

Michelle Gurovith

Extra Feature: Video Recorder I am extremely proud of implementing the video recorder feature, which allowed the game to be recorded and viewed as a video. I used SDL(Simple DirectMedia Link), which is a type of graphics library that we discussed that renders VRAM content. The way this works is that it creates an SDL window, and `startRecording()` starts enabling recording, and `stopRecording()` disables recording. Then, `saveRecording()` and `loadRecording()` allow the video recording to be saved and watched.

My favorite part of this feature was creating a movable progress bar, where the user could move a yellow box at any point in game execution. This works because when a user clicks the progress bar, the `dragging_process` is set to true, which allows the user to adjust the progress box to their liking. Additionally, I added a button called View Recording, which, after the execution of the game, allows the user to press the button to see the video. The View Recording button gets temporarily disabled after the user confirms that the next slug file that they choose is the one that they want. Once the game execution is finished, the View Recording button is enabled.

I had to fix an issue regarding chosen colors not showing up as needed when the game video was played. I realized that the issue was due to the `VideoRecorder` class not storing information regarding color, and there wasn't a way for the color information to go from GPU to the video recorder. The way I fixed this was by creating `filter.cpp` and `filter.h`, where I made a separate Filter class where color information was stored. Then I made sure convertFrameToRGBA() uses the filter's colors in the code. Then, in the `loadRecording` and `saveRecording` methods, I made code to save and load the color information. By fixing this issue, when a person picks a color from the catalog, it is reflected correctly in the video recording.

GPU I was responsible for making the GPU portion of the Banana Emulator. For the emulator to show the games in motion, I used SDL. The way I created the GPU included the initialization process, which included initializing SDL for rendering, creating a window, and creating a renderer and texture to hold pixel data. After this, the rendering process involves checking for interactions with the user, such as checking if the window is closed. It also included getting pixel data from VRAM, converting grayscale pixels into ARGB format, and rendering each frame onto the screen.

GUI and Pixel Filters I helped enhance a couple of GUI and pixel-related parts of the final project. One thing I decided to do was to disable the color black from being used in the game. The reason why I wanted to do this is because I noticed that whenever I would apply a solid black color to be my pixel color, the color-changing pixels would fully blend in with the black background, which was problematic. I also ensured that the default color would be grey on the color catalog so that if a person didn't choose a specific color, grey would automatically be used in the game. Additionally, I created a visual message that the solid black color cannot be used in the game if someone decided to press on solid black. The message then redirects the user to the color catalog to choose a different color for the game.

Michael Kamensky

GUI (My Proudest Work) Placeholder Text

Memory Placeholder Text

Build Reproducibility Placeholder Text

Extra Features

Disassembler

Sample of disassembled Slug instructions.

```
8200        ADDI ZERO,r1,128
8204        AND r4,r1,r2
8208        JR r31,ZERO,ZERO
820c        SLL ZERO,ZERO,ZERO
8210        ADDI ZERO,r1,64
8214        AND r4,r1,r2
8218        JR r31,ZERO,ZERO
821c        SLL ZERO,ZERO,ZERO
8220        ADDI ZERO,r1,32
8224        AND r4,r1,r2
```

Graphical User Interface (GUI)

Figure 1: Graphical User Interface

Pixel Filter

Game Recorder and Playback

File Organization

Our file organization strategy is straightforward. From the top down, we used the `build/` directory to organize cmake-related build files and to keep them easily git-ignoreable. Next, we used the `external/` directory to keep the git submodules for the external libraries we depend on. The `src/` directory (of course) contains our source files. Also, we used the `report/` directory to organize the automatically generated report pdf and it's associated images. Finally, the starter code for the project is included in `starter-code/` as a git submodule for easy reference and use by the CI.

File Tree

```
.
├── build
```

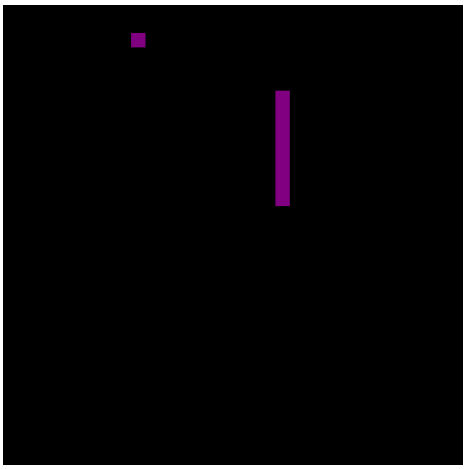

Figure 2: Snake With a Purple Pixel Filter Applied

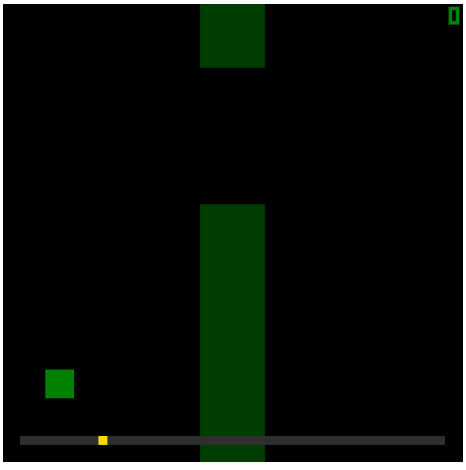

Figure 3: Playback of a Recorded Flappy Bird Game

```
│   ├── CMakeCache.txt
│   ├── CMakeFiles
│   ├── cmake_install.cmake
│   ├── compile_commands.json
│   ├── emulator
│   ├── external
│   └── Makefile
├── CMakeLists.txt
├── external
│   ├── SDL
│   └── wxWidgets
├── helper_tools
│   └── opcode_analyzer.py
├── report
│   ├── report.md
│   └── report.pdf
├── readme.md
├── src
│   ├── banana.png
│   ├── bit_definitions.h
│   ├── console.cpp
│   ├── console.h
│   ├── controller.cpp
│   ├── controller.h
│   ├── cpu.cpp
│   ├── cpu.h
│   ├── disassembler.cpp
│   ├── disassembler.h
│   ├── filter.cpp
│   ├── filter.h
│   ├── gpu.cpp
│   ├── gpu.h
│   ├── gui.cpp
│   ├── gui.h
│   ├── main.cpp
│   ├── memory.cpp
│   ├── memory.h
│   ├── os.cpp
│   ├── os.h
│   ├── vr.cpp
│   └── vr.h
└── starter-code
    ├── bananaslug_documentation.pdf
    ├── games
    ├── gpu
    ├── hws
```

```
├── LICENSE
├── README.md
└── tests
```

Working Slug Files

TODO

Cmake Flags

The CMAKE_BUILD_TYPE flag allows users to select the desired build mode. Setting it to Debug enables debugging symbols (`-ggdb -O0`), making it easier to troubleshoot issues. The Release mode applies high-level optimizations (`-O3 -Werror`) for maximum performance while treating warnings as errors. For a balance between debugging and optimization, RelWithDebInfo (`-O2 -g`) retains debug symbols without sacrificing too much speed. MinSizeRel (`-Os`) optimizes for smaller binary size. Additionally, we provide a Headless mode (`-O3 -Werror -DHEADLESS_BUILD`), which removes the GUI components for systems that don't require a graphical interface. By default, if no build type is specified, we set it to Release to ensure the best runtime performance.