

6.115 Final Project Proposal

Yohan Guyomard
Massachusetts Institute of Technology
Cambridge, MA
yohang@mit.edu

Abstract— This project will push the boundaries of embedded hardware through the implementation of a real-time 3D video-game. At its core is the Cypress PSoC 5LP responsible for all game logic and user inputs. In addition to this, an RP2040 microcontroller serves as a hardware accelerator to offload the task of rendering and pushing frames to the display.

I. INTRODUCTION

Modern video-games simulate and display complex environments in realtime, necessitating high-end hardware with several gigabytes of volatile memory and a considerable multicore processor. In comparison, microcontrollers rarely push a megabyte of random-access memory (RAM) with one or two underpowered cores (in comparison to a desktop system). The PSoC 5 boasts 64kB of system RAM [1], or just under half a second of uncompressed audio as stored on a CD; its single-core ARM Cortex-M3 processor would not even compare to any desktop CPU, as it lacks a floating-point unit [2] (FLOPS do not apply here!).

Despite this, game designers have been creating compelling experiences on low-end hardware for decades (or in the case of retro consoles, it was the best they had at the time)! In employing software tricks or being clever with their artistic directions, titles like *Zelda: Breath of the Wild* (2017) and *DOOM* (1993) deliver much more than what their “minimum specs” permit at face value. Likewise, implementing a 3D video-game on hardware that costs several dollars poses an interesting challenge of the same order.

For this final project, I propose the implementation of a dogfighting video-game similar to the gamemode in *Star Wars: Battlefront* (2015), but entirely on embedded hardware. It will, of course, not be as content-rich as its AAA counterpart but still feature the basic gameplay elements like controlling your spaceship, shooting laser beams at AI turrets and avoiding crash collisions with Imperial Star Destroyers and asteroids. An additional microcontroller, the RP2040, is introduced to offload the task of rendering 3D graphics; its processor is comparable to the PSoC but features slightly more SRAM to account for frame and depth buffers (more on this later).

II. HARDWARE OVERVIEW

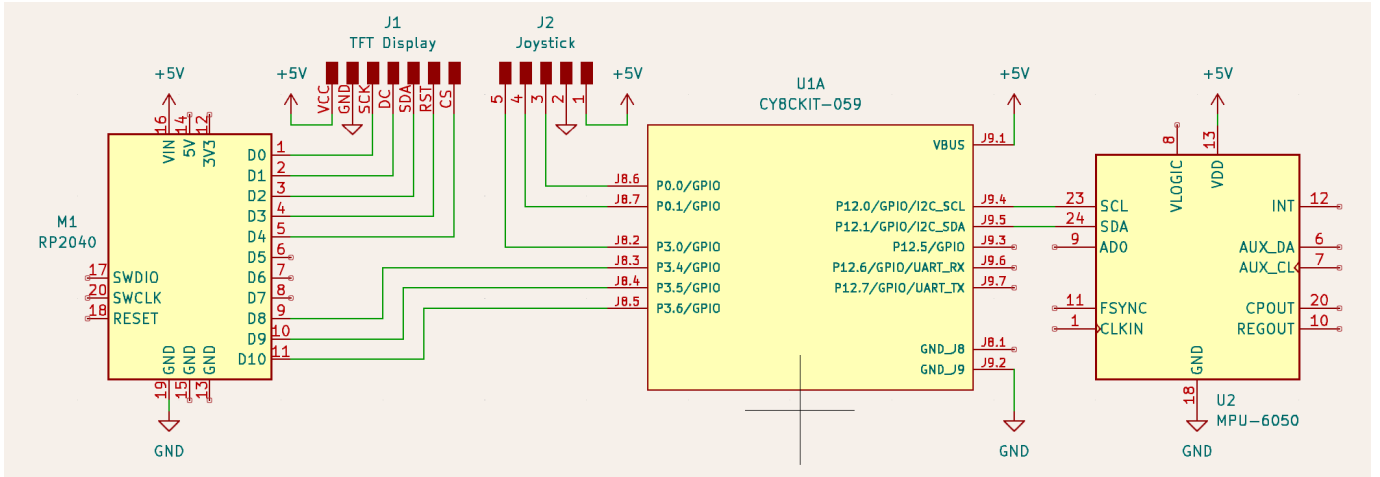


Figure 1: Circuit elements required for the project (second joystick and push-buttons not shown)

This project is software-heavy, however there are still some hardware considerations. Of course, the PSoC (CY8CKIT) is the centerpiece and will power everything from its USB VUSB 5V bus. Connected to this are the user inputs, including two analog joysticks and push buttons; the PSoC’s configurable hardware makes the pin selection fairly simple, as I can just route a SAR ADC to the GPIOs of choice. The PSoC communicates with its graphics accelerator (RP2040) via SPI, a three-wire interface consisting of a clock (driven by it) and two wires for full-duplex communication. That is, the PSoC is the SPI “master” and RP2040 is the “slave.” The RP2040 communicates directly with a TFT display (it would be wasteful to send the frame buffer back to the PSoC) using a separate SPI bus. The TFT display is driven by the ST7735 chip, whose protocol lies outside the scope of this proposal but I have written drivers for it in the past (and third-party libraries also exist). Also included is an IMU (MPU6050) connected to the PSoC via I²C for 3DOF look-around (more on this later).

A. Analog Joysticks

I’ll be using two joysticks for turn, pitch and throttle. The exact model isn’t specified yet, but in principle these joysticks are all the same; both the x and y directions are potentiometers that spring back to the “zero” position. So, reading the voltage at these two pins gives a vector proportional to the stick’s position. There are additional considerations like dead-zones and drift, but these are easily corrected in hardware.

B. Push Buttons

Two push buttons are included for firing laser beams and ion torpedoes. These need to be debounced, which can be done using ButtonSw32 (Button Switch Debouncer) on the PSoC.

C. SPI Communications

The PSoC will be sending large amounts of the data to the RP2040, and relatively frequently (each frame, probably around a kilobyte). This data must be received quickly so it can be processed, so slow protocols like I²C aren’t an option; SPI is also incredibly simple to debug. Internally, the RP2040 stores data transfers directly to a buffer in RAM using one of its Direct Memory Access (DMA) so the CPU can keep busy (i.e. it doesn’t need to read data as soon as it is available). This bus is also full-duplex since the PSoC needs to know when whatever operation is just requested has been completed, though

this may change through the course of this project because an entire protocol for 1 (Ready) / 0 (Not Ready) is overkill.

D. TFT Display

As mentioned, I plan to use an LCD display driven by the ST7735. This chip has an internal frame buffer and works with the SPI at interface at relatively high speeds. Using DMA, the RP2040 can send entire frame buffers without intervention from the CPU other than the nanoseconds it takes to setup a transfer. This is because in addition to SPI, the ST7735 has a Data/Command (DC) pin so pixels can be sent all at once. There are several modules available online that use this chip, but the rest of this proposal assumes the ST7735 “Red” 160x128 LCD display.

III. SOFTWARE OVERVIEW

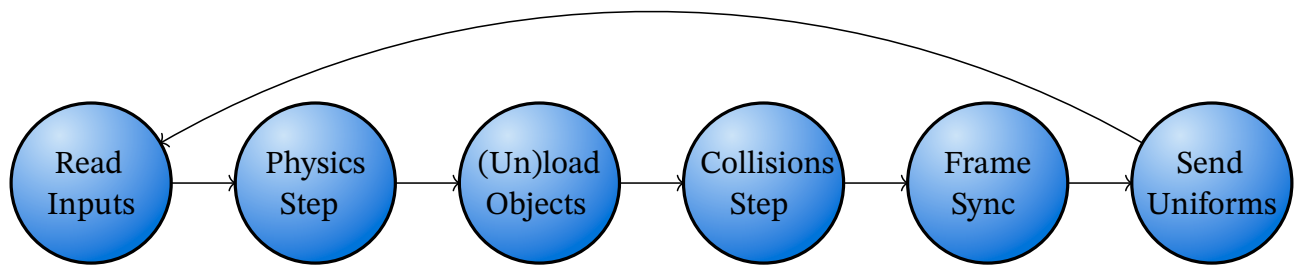


Figure 2: High-level view of the software on both the PSoC and RP2040

A. Read Inputs

The user inputs are buffered using the PSoC’s configurable hardware modules (e.g. ButtonSw32) and this stage concatenates all the readings since the last frame, to be processed by this one.

B. Physics Step

At this stage, every object that moves (whether it is controlled by the player or deterministically) receives an update to its position. This also includes AI entities, such as turrets that automatically target the player.

C. (Un)load Objects

This step is a combination of frustrum culling and chunk management. The former involves checking that objects aren’t behind the camera before asking the graphics processor to draw them (and saves time). The latter is because I’m anticipating a “large” playable area (i.e. the player is small compared to the world) and it’s likely the system won’t be able to simulate or render this many objects. Thus, objects that are too far away become “dormant” and stop being updated. I anticipate that this will be a massive speed-up since the game-world (outer space in a galaxy far, far, away) is massive but mostly empty space.

D. Collisions Step

The physics and collision engine occur at separate steps because unlike a normal physics engine, there is no need for a “resolver” in this game; when a fast-moving spaceship collides with something [anything], it generally crashes and explodes. This makes the project simpler because physics engines are really complicated. Rather, this step compares the axis-aligned bounding boxes (AABB) of the player and laser-beam objects with the environment (asteroid, static spaceships, etc.). More compli-

cated geometry is broken down into several AABBs, which is still easier (and likely faster) than mesh colliders.

E. Frame Sync

There's no point in simulating the game world as fast as possible if the system can't display it. This step prematurely assumes that rendering will be the bottleneck and waits for it to complete before moving on. This means waiting for a "Ready" signal from the RP2040.

F. Send Uniforms

Term borrowed from (most) graphics APIs corresponding to data that applies to a block of vertices, e.g. the position and rotation of a mesh will apply to all its vertices. At this stage, the PSoC sends all the updated transforms/properties that it just calculated to the RP2040 so that it can process the next frame. It immediately goes to the next frame so that rendering and game logic can happen in parallel; the frame displayed will always "lag" by at most one frame.

IV. SOFTWARE RASTERIZER

Thus far, the RP2040 processor has been used as though it were actually a specialized graphics co-processor. However, it is actually a general-purpose dual-core ARM Cortex M0+ microcontroller. In some ways, it is "less powerful" than the PSoC but due to its higher-available SRAM (264kb) it is more suited for this task than a second PSoC. That being said, I propose this chip because its instruction set and planned optimizations are *very* familiar to the PSoC (both are ARM chips).

The purpose of the software rasterizer is to listen to commands from the "host" PSoC, much like a desktop GPU and graphics APIs like OpenGL or Vulkan. It will have support for loading meshes (collections of vertices and their mapping to triangles to form the surface representation of an object), rendering them to a screen buffer and pushing that to the display; nothing more. All the gameplay, physics logic and game loop management happens on the PSoC. More details will be provided on the final report.

V. RISK MANAGEMENT

A. Minimum Viable Product

It's possible to scale down this project to a 2D "top-down" version of what I've proposed. In this scenario, the game is still a "massive" world with all the gameplay elements proposed.

B. Desired Product

A considerable step-up from the 2D spaceship game is bringing it to 3D. This would include everything I've described so far

C. High-Risk Product

I propose two improvements over the base game that would require substantial work but would greatly improve the quality of the project. The first is adding multiplayer support, where each player has their own console (a PSoC, LCD display and inputs) and all share a game world. This would require a refactor of the game code and additional considerations on synchronization. Perhaps I could add another coprocessor like an ESP32 to would act as the PSoC's "network card." Making multiplayer games is an entire engineering problem, but I would be making key assumptions about latency (e.g. normally its dozens of milliseconds whereas here it would be near instantaneous), anti-cheating, etc.

Another expansion upon this project is adapting the graphics to a head-mounted, stereoscopic display. In other words, the world's first microcontroller Virtual Reality (VR) headset. This would involve doubling the TFT displays and likely adding another RP2040; it would also have to perform barrel lens distortion. On the hardware side, the PSoC would need to interface with an IMU (MPU6050 on the schematics above). Note that this would only provide 3DOF, and I would start from pre-existing VR optics "hardware" like the Google Cardboard.

VI. COMPONENTS NEEDED

All of these should be found in EDS, but here is the bill of materials anyways:

- PSoC 5LP
- TFT 128x160 Display
- Raspberry Pi Pico
- 2x Push Buttons
- 2x Analog Joysticks

VII. TIMETABLE

A. *Week of April 15*

Implement a "headless" version of the game on the PSoC, where entities are simulated but not yet displayed. Use print statements and the debugger to verify that this works. This should include user inputs, moving the player and asteroid entities, and collisions.

B. *Week of April 22*

Implement the software rasterizer. By the end of the week, I will have the classic teapot rendering in circles on a TFT display.

C. *Week of April 29*

Integrate the PSoC with the RP2040. This is the "frame sync" and "send uniforms" part of the software diagram above.

D. *Week of May 6*

Debugging (possibly catching up on previous weeks). If time permits, I will also have completed the non-essential gameplay elements like the AI turrets or other non-playable spaceships.

E. *Week of May 13*

Debugging (possibly stretch goals).

REFERENCES

- [1] Cypress, "CY8C58LP Family Datasheet."
- [2] "ARM Cortex-M - Wikipedia — en.wikipedia.org."