

Rust on the Nintendo 64



Melbourne Rust Meetup
8th May 2023

How to use an N64 controller:



Overview

- Introduction
- Reality Co-Processor (RCP)
 - Reality Signal Processor (RSP)
 - Reality Display Processor (RDP)
- Microcode
- Old School C
- Calling Rust from C
- Booting Rust
- Graphics and Text

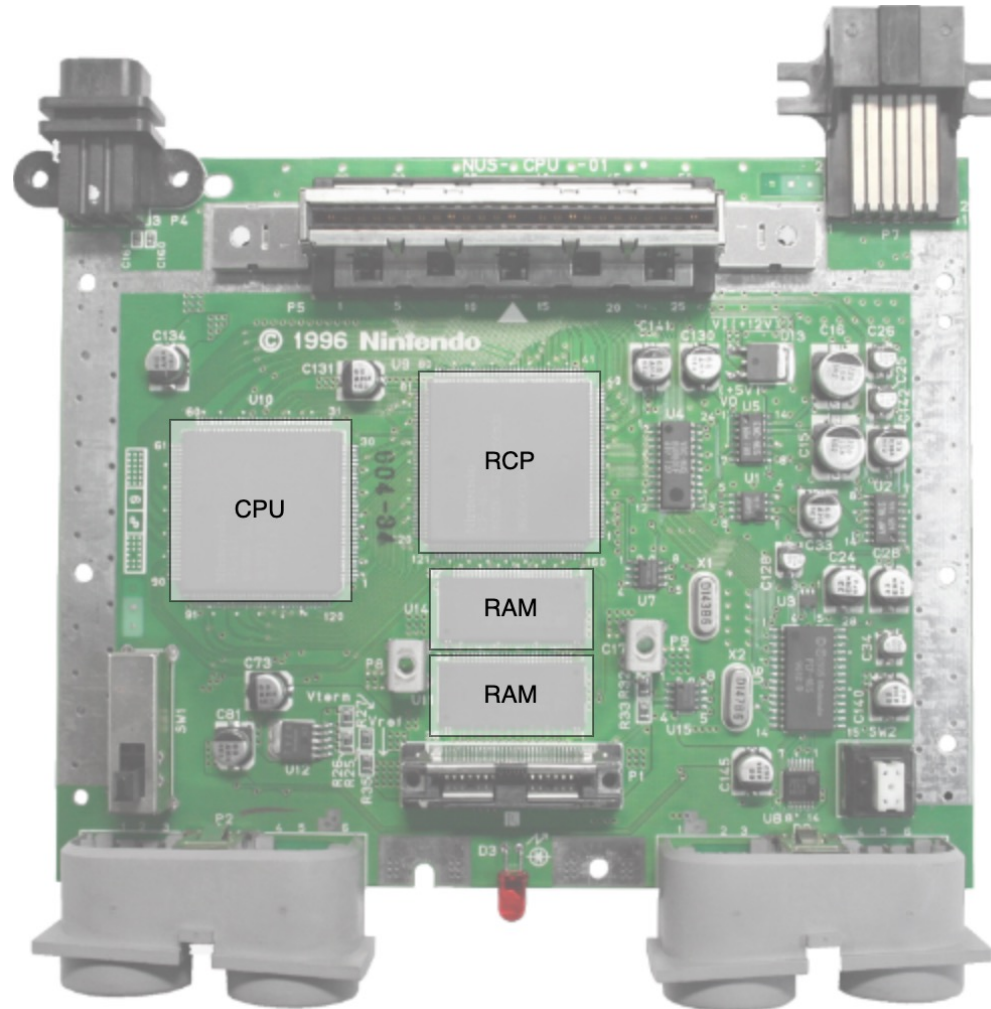
About Me

- I'm a Software Engineer, working at an EdTech startup called Vivi, where I do a fair bit of low-level and embedded programming
- Retro game programming is a hobby I picked up in 2020, during lockdown
- I've dabbled with:
 - Dreamcast
 - Gameboy
 - Mega Drive
 - Nintendo 64



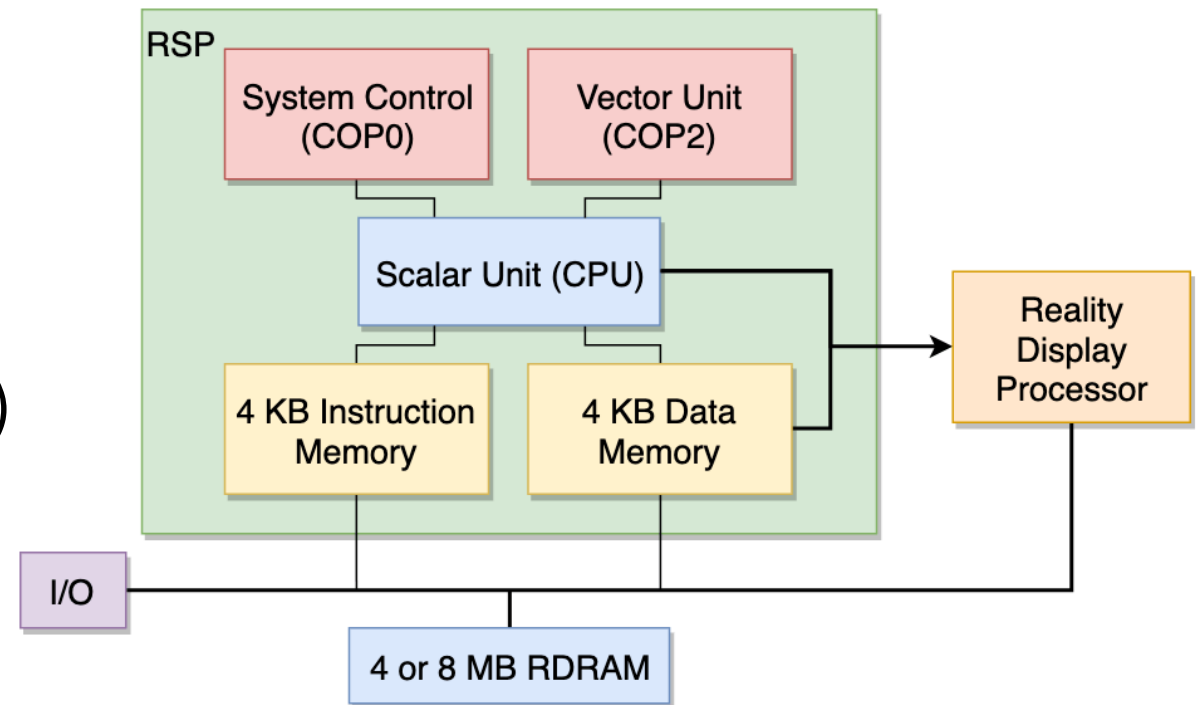
Feel The Power

- The main processor is a **64-bit NEC VR4300**, with 24KB of L1 cache
 - Based on MIPS R4300i, clocked at 93.75 MHz
- This is augmented by the **Reality Co-Processor (RCP)**, a chip designed by Nintendo, consisting of two main components:
 - the **Reality Signal Processor (RSP)** - vector processing
 - the **Reality Display Processor (RDP)** - rasterisation
- The system includes **4.5MB of Rambus RDRAM**, expandable to 9MB
 - 4MB of this is accessible by the CPU (or 8MB expanded)
 - 0.5MB (1MB expanded) is reserved for the RCP for tasks such as Z-buffering



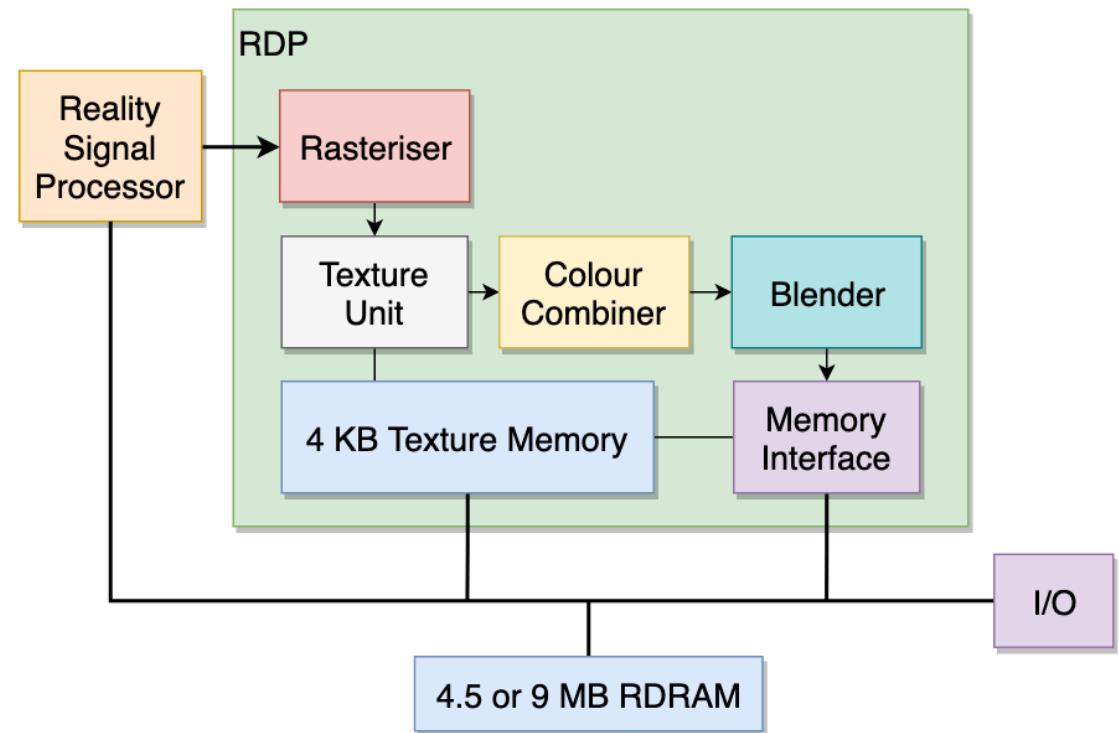
Reality Signal Processor (RSP)

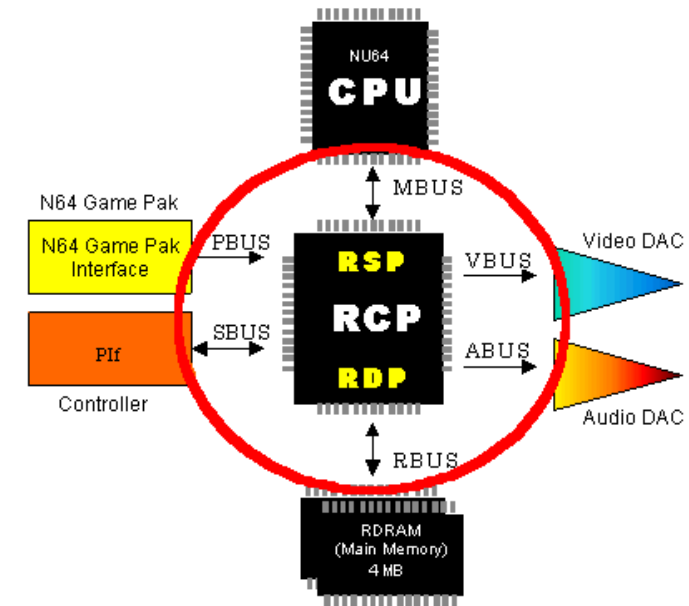
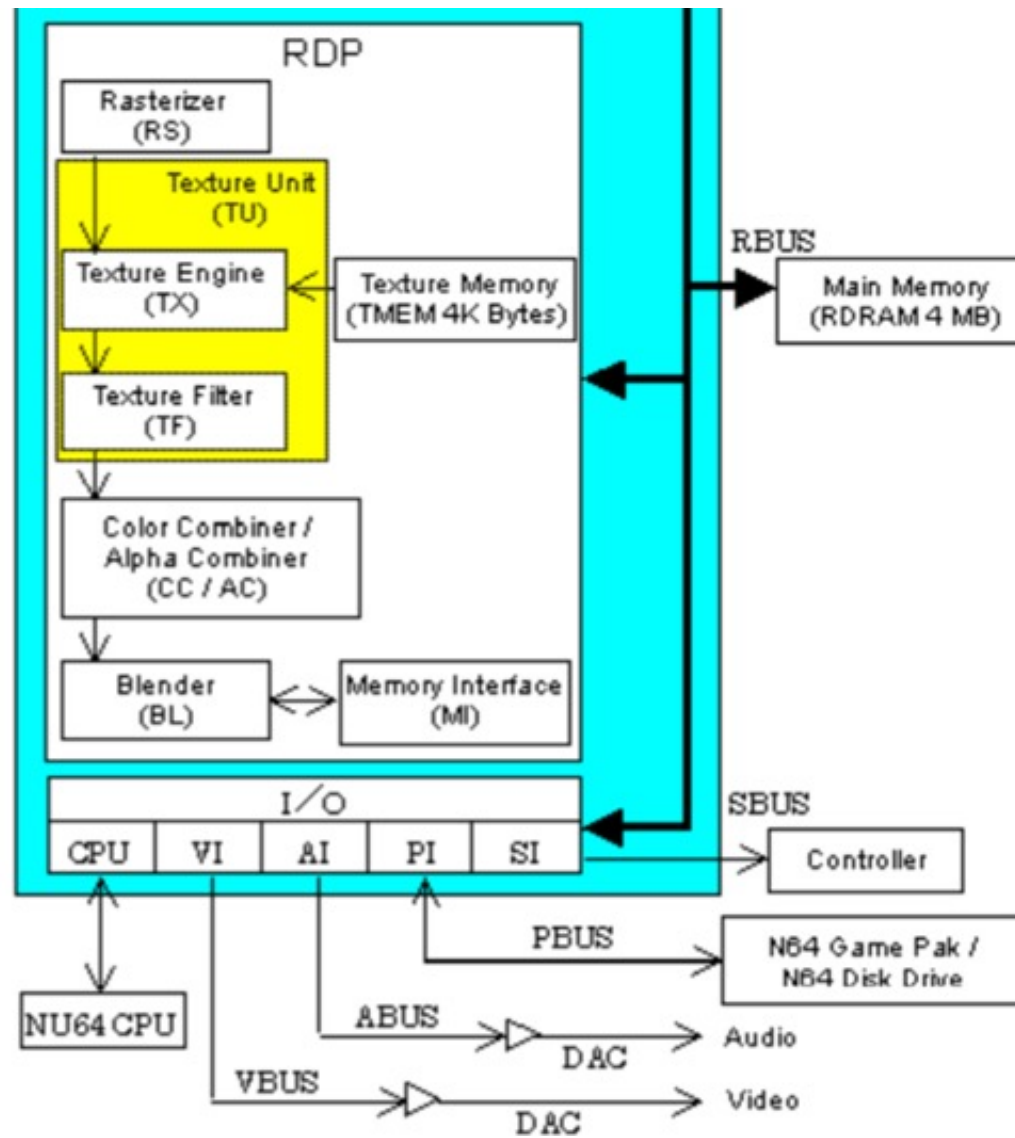
- The **Reality Signal Processor (RSP)** is a highly programmable vector processor
 - Programmable via **Microcode**
- Three main components
 - **Scalar Unit** (cut down MIPS R4000)
 - **Vector Unit** (SIMD style instructions)
 - **System Control Unit** (DMA, etc.)
- On-chip memory is very limited
 - 4KB for data, 4KB for instructions



Reality Display Processor (RDP)

- The **RDP** is a fixed function rasterization engine
- Responsibilities include
 - Converting polygons into pixels
 - Texture mapping (4KB RAM)
 - Colour mixing and blending
- Driven by the RSP
 - Commands are sent via XBUS, or read from main memory





<https://ultra64.ca/files/documentation/online-manuals/man/kantan/step1/2-3.html>

Microcode

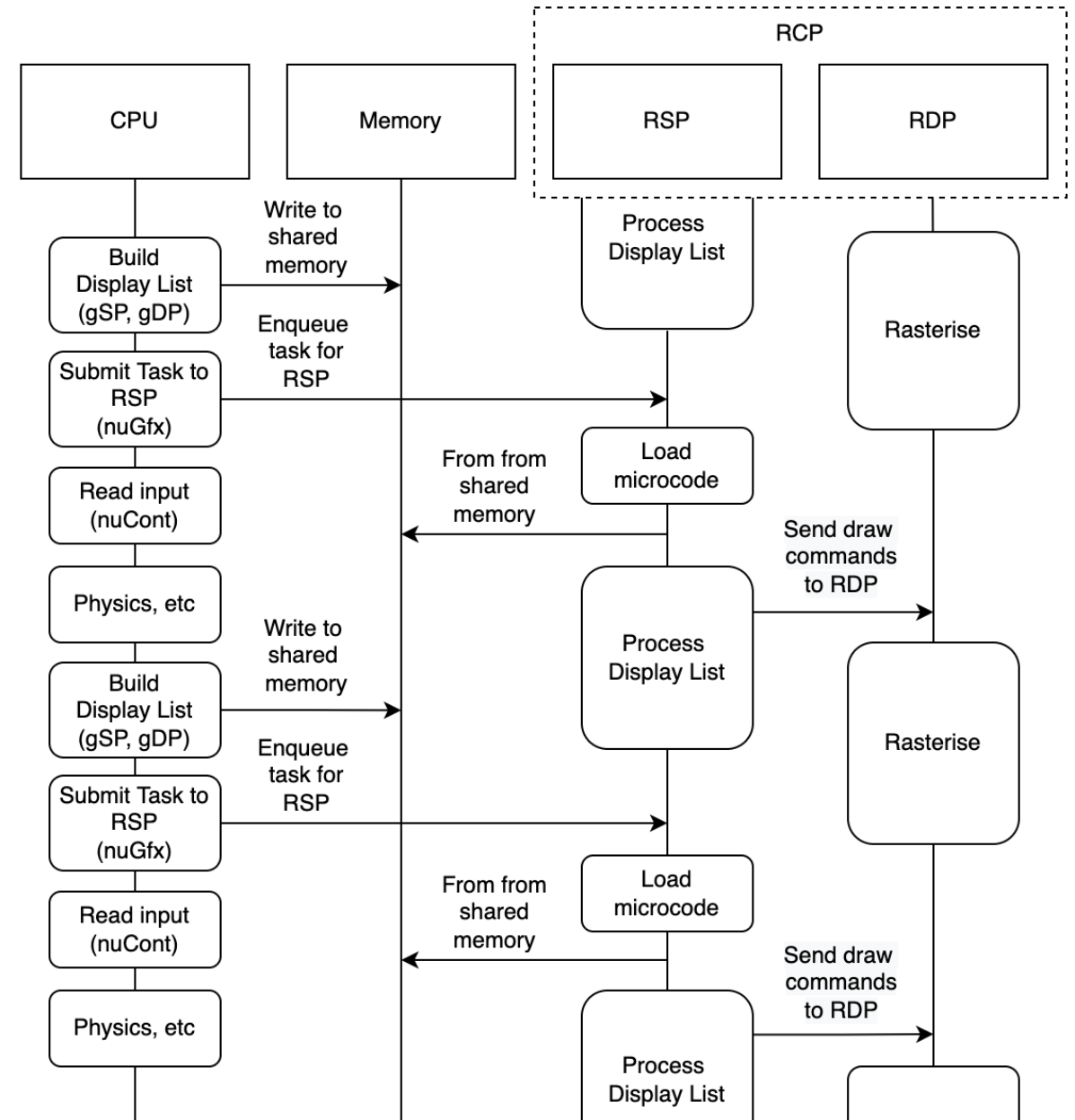
- What is Microcode?
 - Microcode is a small program (max. 1000 instructions) that serves as a graphics pipeline or audio mixer/synthesizer
 - Microcode is loaded on to the RSP prior to processing Display Lists or Audio Lists
- Audio/Display Lists
 - These contain the data/instructions that the CPU can use to control the behaviour of microcode running on the RSP
- Fast3D was the original graphics supplied by Nintendo

RSP Tasks

- A program that runs on the RSP is called a Task, defined as a Task Header
 - Contains pointers to the microcode and data to load
- Task invocation:
 - RSP is halted
 - Main CPU DMA's boot microcode into RSP (IMEM)
 - Main CPU DMA's task header into RSP (DMEM)
 - RSP Program Counter (PC) is set to zero
- Boot microcode loads the task-specific microcode, and the data specified in the Task Header

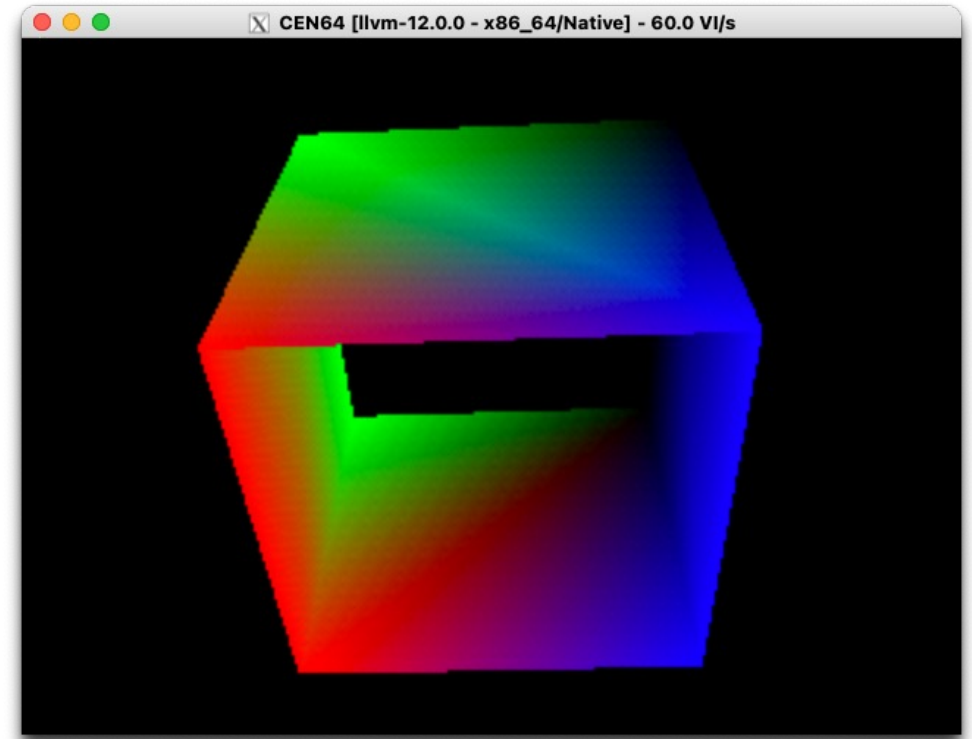
N64 SDK and OS

- Much of this is abstracted away by the N64 SDK and OS, which handles the scheduling and execution of tasks on the RSP
- Games can use various high-level APIs to construct Display Lists (or Audio Lists), and prepare tasks for the RCP
- The relationship between game code and the RCP is illustrated in this sequence diagram...



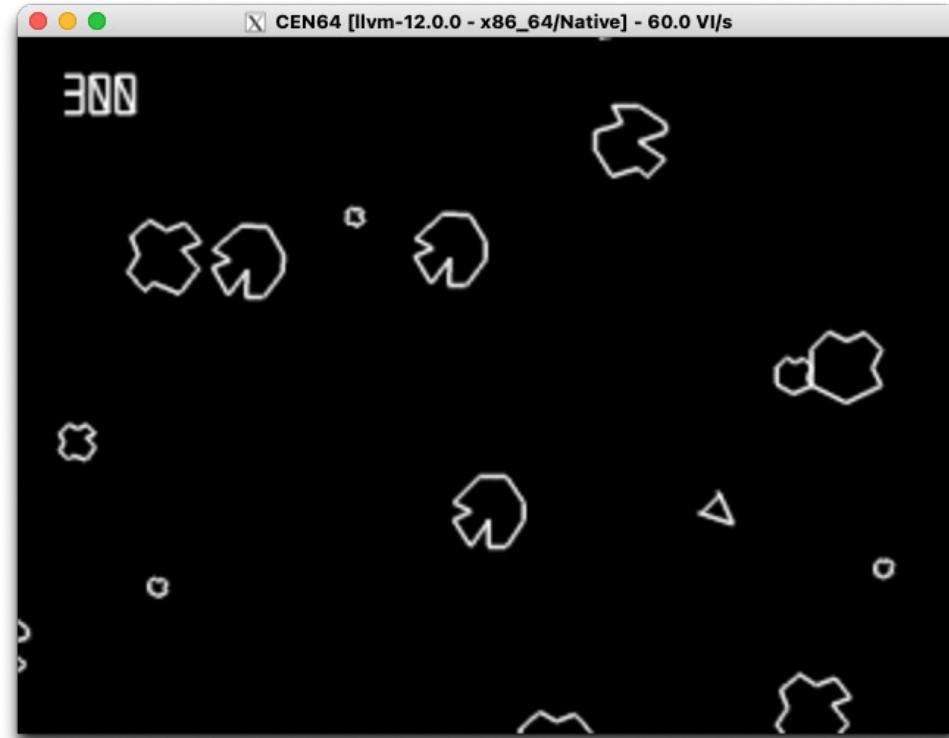
Example 1: Old School C

- Example contains two *stages*, which are toggled via a `#define`
 - Stage 0 is just boilerplate
 - Stage 1 renders a spinning cube
- Code is compiled using GCC
 - Generic Makefiles for legacy and modern SDKs
 - Linker scripts that include common RSP microcode binary blobs



<https://github.com/tristanpenman/n64-heart-rust/tree/master/examples/01-old-school-c>

Shameless Plug: Asteroids64



<https://github.com/tristanpenman/asteroids64>

Developing on Real Hardware

Made possible by flash-carts,
such as the EverDrive-64 and 64drive!



UNFLoader

- Using an SD card is SLOW
- UNFLoader (Universal Nintendo 64 Flashcart Loader) makes it easy to load ROMs on to flash carts via USB
- Supports debugging via USB

Demo: Running Code on the Nintendo 64

Rust and MIPS

- Rust on the RSP?
 - MIPS Assembly is the only viable way to fit useful code into the RSP's limited memory (1000 instruction limit)
 - The reality is that we don't need to develop our own microcode for the RSP, and we certainly won't be doing so in Rust
- ***We just want to program the CPU***
 - The VR4300 uses the MIPS III instruction set, which can be most certainly be produced by the Rust compiler.




Producing MIPS III Binaries

- We need to tell **rustc** how to cross-compile for the MIPS III architecture
- We do this using **xargo**, passing in a JSON file that describes the target architecture

```
1  {
2    "arch": "mips",
3    "cpu": "mips3",
4    "data-layout": "E-m:m-p:32:32-i8:8:32-i16:16:32-i64:64-n32-S64",
5    "dynamic-linking": true,
6    "env": "",
7    "executables": true,
8    "features": "+mips3,+noabicalls",
9    "has-rpath": true,
10   "linker": "rust-lld",
11   "linker-flavor": "ld.lld",
12   "llvm-target": "mips-unknown-elf",
13   "max-atomic-width": 32,
14   "os": "none",
15   "panic-strategy": "abort",
16   "position-independent-executables": true,
17   "relocation-model": "static",
18   "relro-level": "full",
19   "target-c-int-width": "32",
20   "target-endian": "big",
21   "target-family": "unix",
22   "target-pointer-width": "32",
23   "vendor": "unknown"
24 }
```

mips-n64-elf.json

Example 2: Calling Rust from C

- Our first taste of Rust on the Nintendo 64 will allow us to call Rust code from C
- We'll need the **xargo** target file
- A few other updates are required...
 - Cargo.toml file to describe a Rust library (minimal / libminimal)
 - Makefile rules for Rust files
 - Changes to linker script
- Not realistic to support Legacy SDK 

```
1  #![no_std]
2
3  use core::panic::PanicInfo;
4
5  #[panic_handler]
6  fn panic(_info: &PanicInfo) -> ! {
7      loop {}
8  }
9
10 #[no_mangle]
11 extern "C" fn call_rust() -> i32 {
12     1
13 }
```

src/lib.rs

Demo: Running *Rust* code on the Nintendo 64

How far do we go?

- So far, we've shown that it is possible to write some code in Rust
- Pros/cons
 - Combining C & Rust gives us access to the **N64 SDK**, or **Libdragon**
 - Get the OS *for free*
 - Sacrifices some purity
 - Incrementally adoptable
- But we're not here because Rust is easy...
 - Can we do everything with Rust, right down to the OS?
 - Can we use Cargo to build a ROM?

Existing Projects: Loka64

- There are several projects out there that use Rust from scratch:
 - **n64-systemtest**
 - Comprehensive test suite intended for N64 emulator authors
 - **rrt0**
 - Basis of example 3...
 - **Loka64**
 - Somewhat playable on both cen64 and real hardware



Cargo N64

- cargo-n64 provides a subcommand for cargo, that knows how to build N64 ROMs
 - Provides default linker and cross-compilation settings
 - Basic FAT file system support (appended to end of ROM)
- **Install from source to support Rust 2022-06-21**
- Easy to integrate... projects built using cargo-n64 can simply run:
 - `cargo n64 build --path-to-ipl3 ../../common/ipl3.bin`

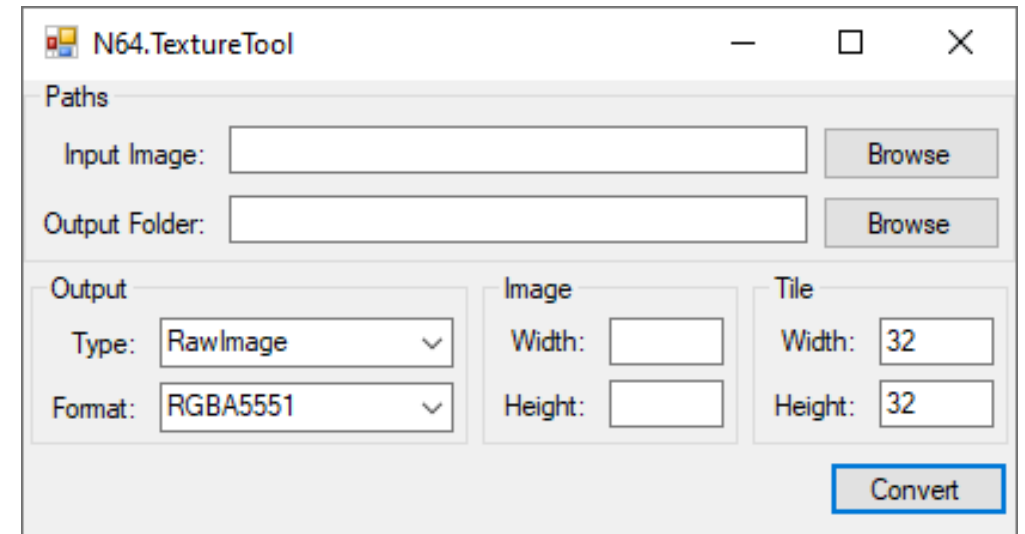
Example 3: Booting Rust

- Project Structure (based on rrt0 example):
 - **n64lib**
 - Video interface (VI definitions)
 - IPL3 Font Renderer
 - **rrt0**
 - Minimal Rust runtime, with MIPS assembly entry point for N64
 - **src/main.rs**
 - Uses n64lib to render text using IPL3 font

Demo: Booting *Rust* code on the Nintendo 64

Minimal Graphics

- While Loka64 provides an example of full asset pipeline support, what's the least we can do to display images on the N64? **Write to the framebuffer!**
- In this example, the framebuffer is configured to be 320x240 with a RGBA 5551 pixel format
 - Images are converted from PNG using a small Windows app called N64TextureTool
- These are then appended to the end of the ROM
- ***Rewrite it in Rust?***



Example 4: Graphics in Rust

- Main ROM is built using cargo-n64:
 - **cargo n64 build --path-to-ipl3 ../../common/ipl3.bin**
- We then concatenate images:
 - **cd fs; cat index.txt | xargs cat > fs.bin**
- Append fs.bin to the ROM
- Update checksum using rs64romtool:
 - **rs64romtool chksum rom.bin rom-final.bin**
- This is all encapsulated in a Makefile

Demo: Graphics in Rust

Future Work

- Port Asteroids clone to Rust
- Potentially explore using Libdragon via Rust bindings
- Rewrite N64TextureTool in Rust ;)
- Continue fleshing out templates
 - One for combining C and Rust, because this feels like a more sustainable way to get started with N64 Homebrew + Rust
 - Another for vanilla Rust, by extracting useful pieces from loka64

Questions

References

- All the code for this talk is available on GitHub:
 - <https://github.com/tristanpenman/n64-heart-rust>
 - This repo contains many useful links
- If you're interested in Nintendo 64 homebrew, here are some great places to start:
 - Awesome N64 Dev (<https://n64.dev>)
 - N64brew Discord (<https://discord.gg/WqFgNWf>)