Dominic Szablewski, @phoboslab
— Tuesday, August 4th 2026

How to Make a Nintendo 64 Game in 2026

Two years ago, I was Porting my JavaScript Game Engine to C for No Reason. I have since found a reason: making a new N64 game!

The result is Xibalba 64 – a Wolfenstein 3D-like FPS. Modretro agreed to publish the game as a physical launch title for their M64 (a modern N64 clone), complete with cartridge, packaging and manual!

Xibalba 64 on Modretro.com

This, to my knowledge, is only the second physical release of any *new* N64 game since the end of the console's original commercial life. The infamous Xeno Crisis by Bitmap Bureau – originally a new game for the Sega Mega Drive and subsequently released on many, many more consoles – came to the N64 in 2023. No other new games have been published for the N64 since Tony Hawk's Pro Skater 3 in 2002.

The Engine

Impact was a JavaScript game engine I developed back in 2010. It was tailored for 2D action games, handling tile sheets, background maps, sprites and collision detection. It's very simple, but still a sound foundation for whatever you want to throw at it.

Two years ago I rewrote Impact in C. Why? I don't know. It was fun.

This C port, high_impact, has a notion of a “platform backend”. The platform handles the low-level plumbing – opening a window, creating a drawing surface, reading input, etc. Out of the box, high_impact comes with two platform backends (SDL2 and Sokol), and you can compile your game for either one. This already enables high_impact games to run on many different devices.

The rendering backend in high_impact is also modular. You can compile your game with a software renderer, OpenGL or Metal (for iOS/macOS). Support for new platform backends or rendering backends can be added without modifying any other part of the engine.

A perfect starting point for an N64 game.

N64 Hardware and Platform Library

The N64 is a quirky beast. In addition to the 93 MHz MIPS CPU (big-endian!), it has two coprocessors for handling graphics, sound and more:

- “Reality Display Processor” (RDP) – a fixed-function graphics processor
- “Reality Signal Processor” (RSP) – a programmable vector processor

Both of these live in the same physical package, commonly called the “Reality Coprocessor” (RCP).

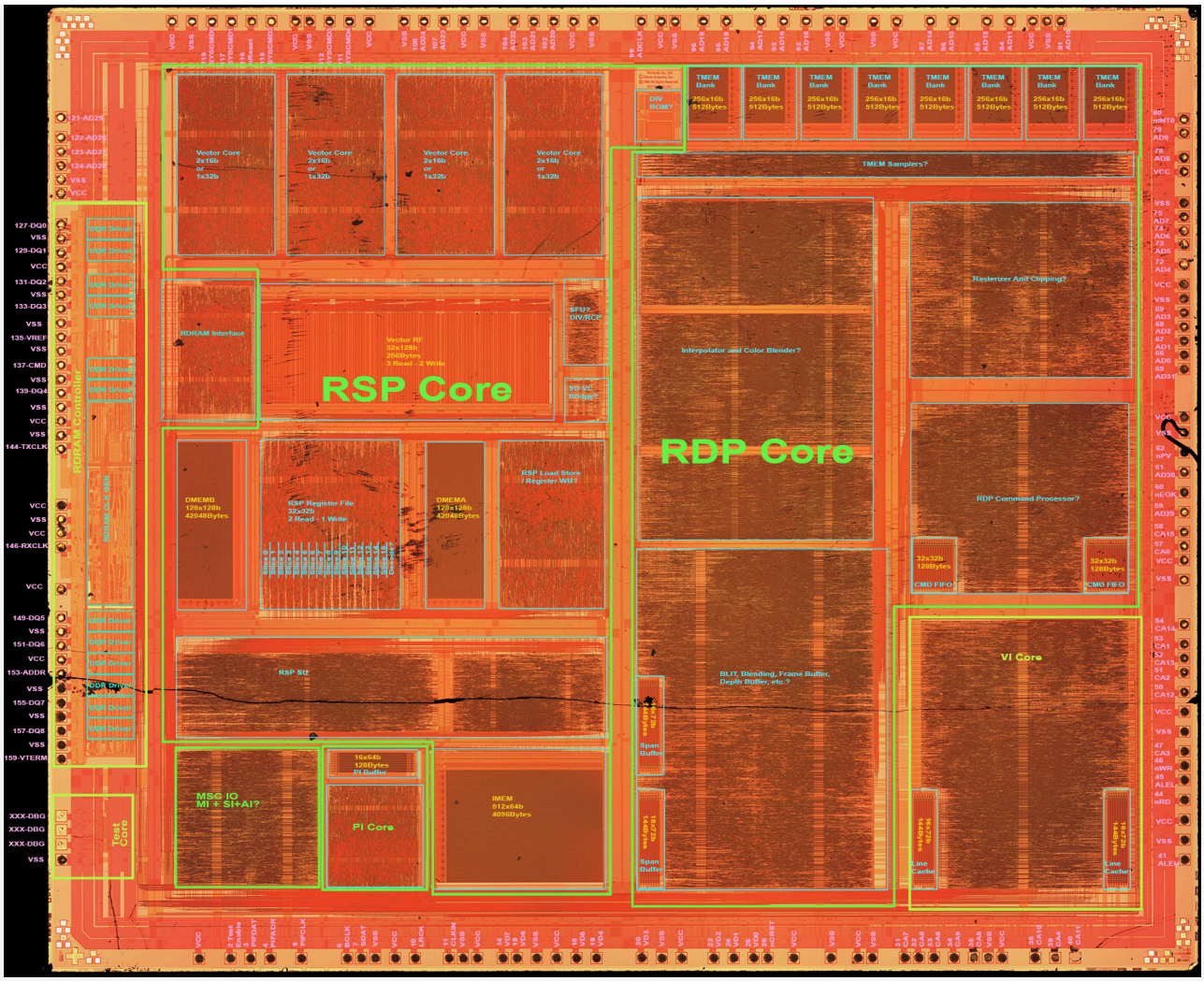


Image from the N64Brew Wiki

For the first few years of the N64's life, Nintendo closely guarded access to the RSP. It was exclusively used by Nintendo's officially sanctioned platform library, “libultra”. Only later did Nintendo allow game studios to write custom “microcode” (actually just MIPS assembly) for the RSP.

Keeping the hardware happy is no simple feat, and the instructions for the RDP are quirky and complicated. Programming your game on *bare metal* is pretty much out of the question. In recent years, Nintendo's official “libultra” has made its way onto the internet, but using it would risk a copyright lawsuit.

Luckily, the N64 homebrew scene has picked up a lot of steam in the last few years and we have a very capable alternative now: Libdragon.

Libdragon is basically SDL for the N64. It provides facilities for drawing sprites and triangles, sound output, controller input and much more.

It took me only a few evenings to build a new platform backend for high_impact on top of libdragon. I tested this with Biolab Disaster. The game code remained unmodified; performance was *meh*, but I was using the N64 hardware in the most naive way possible.

Video from my post on X, Sep 17, 2025 showing Biolab Disaster running in an N64 emulator

Dev Environment

Libdragon provides the compilers and everything else that's necessary to build a ROM file for the N64. The installation instructions and all other documentation are comprehensive and well-written. The library comes with many examples to get you started.

In general, it was a pleasure to work with Libdragon. Just a heads up: you probably want to use the `preview` branch as the “stable” `trunk` branch has hopelessly fallen behind.

For testing, a good emulator is invaluable. For the longest time, N64 emulation was extremely inaccurate. Lackluster emulation of the RSP and RDP coprocessors, in particular, was the cause of most problems.

Most emulators just emulated Nintendo's platform library, libultra. They emulated the *intent* to draw a triangle, not what the hardware would actually do. While inaccurate, this made emulation possible *at all* in the early days. Famously, UltraHLE ("Ultra High Level Emulator") was released well within the lifetime of the N64 and caused a lot of headaches and subsequent lawsuits.

These days the N64 core in Ares is much closer to the actual hardware – the RDP and RSP are fully emulated, including accurate timing for the RSP. The infamous slow memory bandwidth of the N64, however, can still only be tested on real hardware (which recently caused me some disappointment).

So you need a real N64 and a cartridge that lets you play arbitrary `.z64` ROM files. The open-source SummerCart64 is excellent and available from many different manufacturers. Be aware: some manufacturers (especially on AliExpress) cheap out on the components of the board.

SummerCart64 has the usual SD card slot to store your ROMs, but what makes it great for development is its USB-C port: you can directly connect it to your PC and upload a ROM as part of your build process using sc64deployer.

I ended up with the N64 next to my PC, connected via USB, and used a cheap $10 USB analog capture card to display its video output in a window on my desktop. On Linux, it took some fiddling with `mpv` to get low-latency output; here's the script I used.

With this setup, iterating on real hardware was just a matter of compiling and pushing the N64 reset button.

The Game

I originally made Xibalba as a demo for my JavaScript game engine in 2014. WebGL was still the hot new thing back then; a 3D game in a browser was quite a novelty. The game was very short, featuring only a handful of levels, weapons and enemy types.

In contrast, I wanted Xibalba 64 to be a *real* game, not just a demo. So I not only needed to port the game to C and high_impact, but also expand on it with more levels, more enemy types and more weapons.

`high_impact` is a 2D game engine, but Xibalba 64 is clearly 3D. Well, not quite. Since the game has no elevation, it can be *mostly* treated as 2D. You could conceptually play Xibalba 64 from a 2D top-down perspective. Of course, that wouldn't be as exciting, but all the physics, movement and shooting would work the same way. In this regard, the game is very similar to Wolfenstein 3D.

Many of high_impact's physics functions expect a `vec2_t` argument with `.x` and `.y` components. But for drawing, I absolutely needed a 3D position, so I came up with this definition for a `vec3_t` type and changed the `entity_t` type:

```
typedef struct {
    float x, y;
} vec2_t;

typedef union {
    vec2_t xy;
    struct {
        float x, y, z;
    };
} vec3_t;

typedef struct {
    // ...
    vec3_t pos;
    vec3_t vel;
    // ...
} entity_t;
```

Now, whenever I need to call a function that accepts a `vec2_t` , I can “convert” from `vec3_t` for free:

```
trace_t res = trace(collision_map, entity->pos.xy, entity->vel.xy);
```

Since the inner `vec3_t` `struct` is “anonymous”, I can still access all values directly; i.e., `entity->pos.z` works just fine.

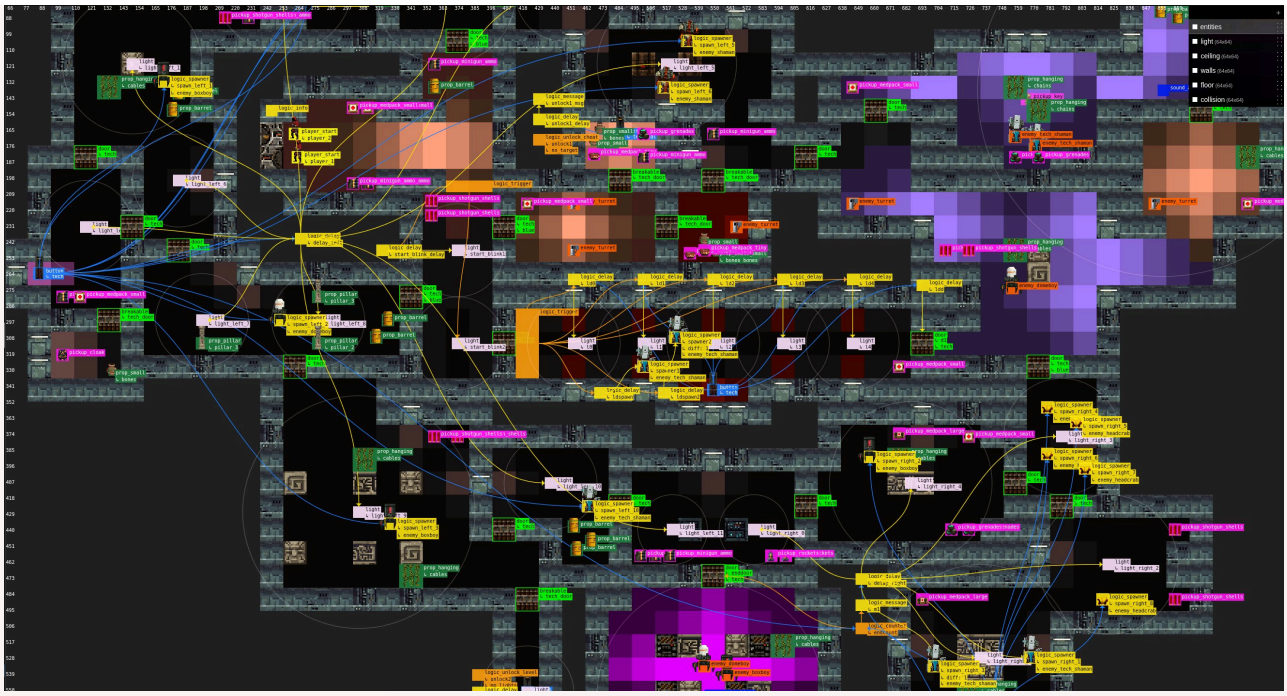
The initial port of the existing levels and enemy types went quite smoothly and was finished in about two weeks. I then spent another few months on extending the game and optimizing the renderer.

Most of Libdragon's functions fit naturally into a new platform and rendering backend, though I had to change some other parts of high_impact to bypass its mixer (Libdragon has its own, accelerated by the RSP) and image loader.

Throughout the whole process, I retained the ability to build the game with the SDL2 or Sokol backends. This was great for playtesting game logic and enemy behavior. To make levels, I also implemented a simple hot-reload mechanism triggered whenever a level file changed.

Video from my post on X, Jan 25, 2026 showing the level editor and hot-reloading

The level editor, bundled with high_impact, is a single self-contained HTML file. I ended up extending it quite a bit to add better support for lightmaps, display actual sprites for entities (instead of just boxes), add descriptions for entity settings and provide other small features. The single source of truth is still the C source code – the level editor reads it and extracts the entity types and supported settings automatically.



Since the level editor still works with JSON files, I built a small map compiler that reads the JSON and emits binary data. While loading JSON on the N64 is of course possible, it added some unnecessary ~100 ms of load time. So during the build process, each JSON level file is converted into a struct that essentially looks like this:

```
typedef struct {
    uint16_t magic;
    uint16_t entities_len;
    uint16_t map_width;
    uint16_t map_height;

    struct {
        uint16_t type_id;
        uint16_t x;
        uint16_t y;
        uint16_t settings_len;
        struct {
            uint16_t setting_type; // such as "name", "target", "size", ...
            union {
                float16_t float_value;
                int16_t int_value;
                struct {
                    int16_t string_len;
                    char string_value;
                };
            } value;
        } settings[settings_len];
    } entities[entities_len];

    uint16_t collision_map[map_width * map_height];
    uint16_t floor_map[map_width * map_height];
    uint16_t wall_map[map_width * map_height];
```

```
    uint16_t ceiling_map[map_width * map_height];
    uint16_t light_map[map_width * map_height];
} level_t;
```

The level compiler writes those values in big-endian format for the N64 and little-endian format for x86 (SDL2, Sokol, WASM), so we can easily read everything on all platforms without byte swapping.

Rendering

Libdragon itself has a function for drawing triangles: `rdpq_triangle()` inserts a single triangle draw call into the RDP queue. While this works, what you really want to do is submit your draw calls to the RSP, have some custom microcode to perform transformations, lighting, depth calculations, etc., and then let the RSP instruct the RDP to ultimately draw the triangle.

The intricacies of the RDP and RSP were still new to me, but luckily another outstanding open-source library, Tiny3D, handles all this and more with a simple-to-use API. Getting something on the screen was the easy part; making it performant was a whole other endeavor.

The N64 infamously only has 4 KB of texture memory. The largest textures you can upload are just a meager 64×64 pixels. Even worse, the memory latency for a texture upload is atrocious. One solution, used by Mario 64 and many other titles, is to render untextured polygons whenever you can.

This wouldn't really fly with the style of my game, so instead I had to be really careful with the draw order of level tiles to minimize texture uploads. On top of that, Tiny3D can load and submit up to 17 quads at once, and it would be wasteful not to use that. So I ended up collecting batches of triangles with the same texture in 64-bit draw calls:

```
typedef union render_call {
    uint64_t packed;
    uint32_t hashable;
    uint64_t ident : 46;
    struct {
        uint64_t translucent : 1;
```

```
        uint64_t texture_index : 9;
        uint64_t x : 10;
        uint64_t y : 10;
        uint64_t w : 8;
        uint64_t h : 8;
        uint64_t vbi : 14;
        uint64_t len : 4;
    };
} render_call_t;
```

Here `vbi` is the accompanying vertex buffer index, and `len` is the number of quads in this call. Since every call is just 64 bits wide, we can efficiently sort them at the end of the frame and issue them to Tiny3D.

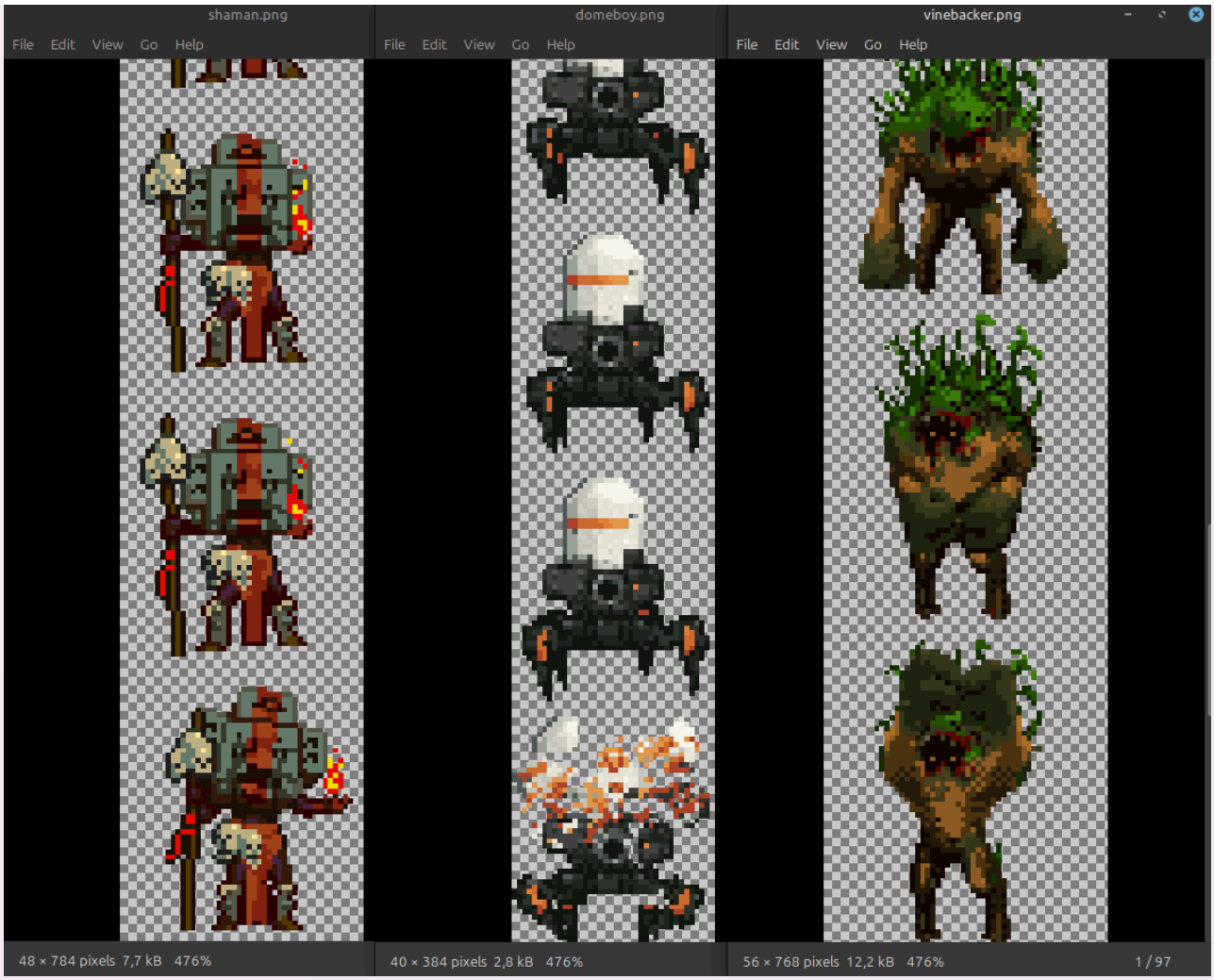
But before I could do all this, I had to first figure out which parts of a level are actually visible. The original JavaScript Xibalba used a portal system, dividing each level into sectors and precomputing which sectors were visible from the current one. This worked fine, but produced a bit more overdraw than I would have liked.

So I opted for another approach: raycasting. Yes, the game is just casting 320 rays into the scene, covering the whole field of view. Each ray marks traversed tiles in a bitmap for submission to the renderer.

Later, I optimized the raycasting a bit more by recursively dividing the 320-pixel field of view until two rays hit the same tile. In this process, I also check whether any of the traversed tiles are missing a ceiling – if so, we have to draw a skybox.

Fun fact: the skybox in Xibalba 64 is just a single 32×32-pixel texture that is beautifully smeared across the horizon.

As another optimization, I arranged each tile sheet that wouldn't fit in a single upload into a single column and tried to use just 4-bit indexed colors wherever possible. Using fewer colors allowed more pixels to fit into texture memory, and the column layout ensured that each tile could be uploaded as a single, continuous chunk of memory.



With all of this, the game runs at a stable 60 FPS. Something not many other N64 games can claim!

The four-player split-screen mode doesn't quite hit the 60 FPS mark at all times, but still remains fluid. In contrast, GoldenEye 007 infamously often dropped into single-digit frame rates here.

Sound & Music

As with all my other games, my good friend Andreas Lösch produced some outstanding music. You can listen to the whole Xibalba 64 Soundtrack on Bandcamp.

The game itself also has a built-in music player that you can unlock in the single-player campaign.

Cartridge space is tight, and even compressed audio is typically either quite large or too expensive to decode.

In a heroic effort, Giovanni Bajo – one of the maintainers of Libdragon and an absolute wizard when it comes to anything N64 – implemented an RSP-accelerated Opus decoder. For context: Opus is an audio codec first published in 2012. That's 16 years(!) after the N64. It's a marvel that it works at all, but sadly it's still a bit too computationally expensive to be used during gameplay.

(*Aside:* Giovanni Bajo went on to implement a real-time H.264 decoder for the N64, too.)

The better option for now is a simple 4-bit VADPCM format that Libdragon transparently decodes on the RSP during playback. The compression ratio, of course, isn't great, but it's way better than uncompressed WAV. About 31 MB of the 32 MB ROM is used by sound and music.

After the release of Xibalba 64, I started to implement another audio compression format that would better address the space/quality/complexity trade-off. More on that in the next blog post!

Publishing & Sales

Modretro previously made a Game Boy Color clone – the Chromatic – compatible with all existing Game Boy and Game Boy Color games, and they were quite eager to publish many new games by hobbyist developers, too.

When they announced the M64, I thought this would be a great chance to get on board. As soon as I had a working prototype ready and was confident that I would be able to build the whole game (and make it *good*), I wrote an email to the generic Modretro customer service address and, to my surprise, heard back from the head of publishing within a day.

Bureaucracy was minimal and the contract was straightforward. I requested some changes that would allow me to release the game engine as open source later on, which Modretro was happy to accommodate.

Of course, as it goes with these projects, there were some delays and I only received a pre-production M64 late in the development process. But it didn't matter – the M64 worked as advertised; no changes to the game for the M64 were necessary.

Modretro also offered help with the box art, but another friend of mine was eager to do this instead. He later told me that, back in the '90s, he was responsible for the packaging of many titles published by Sierra in Germany, including Half-Life. So it's no surprise that the Xibalba 64 box art turned out great!

For the manual, I supplied text and illustrations to Modretro, and they laid it out for print. Smooth sailing. You can have a look at the PDF manual on the Xibalba 64 store page.

I'm not able to talk about sales numbers or my contract with Modretro in detail, and the game was released only a few days ago, but so far it's looking quite good. Of course, making N64 games will probably not let you quit your job, but it looks like it will pay for a few nice holidays.

Huge thanks to Giovanni Bajo of Libdragon, Max Bebök of Tiny3D, the entire N64brew Discord community and, of course, the Modretro team for pulling this off!

Here's the final trailer for the game.

Resources

If you're looking to get started with N64 development, these resources may help:

- n64.dev – a collection of *everything* related to N64 development
- N64brew Discord – a very friendly and helpful community; many of the library developers are here
- Libdragon – *the* system library for the N64
- Tiny3D – a simple and fast 3D graphics library
- Pyrite64 – a visual editor and runtime for creating 3D games