Slide 1 - Title

Hello everyone. I thought it’d be interesting to make this presentation on the console itself, because I like to be a pain in the ass for the organizers, and because it will allow me to demonstrate things working on the system directly instead of you just taking my word for it. If anything goes wrong, I’ve also prepared a backup PowerPoint on my laptop, and if that decides to not work, I’ve also prepared an interpretive dance. So let’s dive into the wonderful world of Nintendo 64 homebrew.

# Slide 2 - Intro

First a very brief introduction; my name is Lourenço Soares, but you might find it easier to call me by my handle “Buu342” (pronounce that however you find most fun). I’m a computer science student who jumped into the world of N64 development in early 2018. I have one fully finished N64 game that I released at the end of that year, and I have since then been trying to improve the general development scene by producing guides and tools. I’ve written a fully detailed multipart series about the development of a little WarioWare clone game of mine called “Pyoro64”, tools and libraries for USB debugging, texture and model converters, tutorial ROMs and documents, among other things. Yes, I open source everything.

I really want to make more fully fledged games for the system, but my main priority has ended up been trying to make N64 development more accessible.

The key word there being **accessible**, not easier!

# Slide 3 - Hansei

The Nintendo 64 is a notoriously complex system, so much so that, Genyo Takeda, the Hardware Development Chief at the time, referred to the system’s programming challenges with the word “Hansei”, which translates to “Reflective Regret” (hence the very specific choice of words for this presentation’s title). Now, it’s probably not as difficult as say, the PlayStation 2 with its 10 different processors, or the Sega Saturn with its quadrilateral meshes and insanely tight timings between its 8 processors, but it is still very challenging for someone who has not had any sort of embedded systems development experience. And the difficulty has been getting higher with time, for reasons I’ll get to. Frankly, I’m sure that there’s an alternative universe out there where instead of giving you this presentation, I’d be giving:

# Slide 4 – Joke Title

So let’s talk about why this is the case, what sort of stuff you need to deal with, the history of N64 homebrew, and its future. Obviously, you’re going to be hearing this from someone who not only drank the Kool-Aid, but is actively helping produce it.

# Slide 5 – Why do this?

After this introduction, you’re probably wondering why on earth anyone would do this to themselves.

<Read the slides>

Slide 6 – The Hardware

So to better understand its complexity, let’s begin by talking about the N64’s hardware (Sorry guys, I know it’s boring but you do need to know this to work on the system). The hardware was designed by Silicon Graphics, the leaders of computer graphics at the time. It was pretty ahead of the curve, especially retailing at 250$. Now, I could spend years talking about every minute detail because the N64 community has been doing a fantastic job of reverse engineering protocols and decapping chips, but I’m going to focus on the 3 main components of the system, which is pretty much the only stuff programmers would need to worry about anyway.

The CPU – The Father

The RAM – The Son

And the RCP – The Holy Spirit

# Slide 7 – The CPU

CPU is a derivative of the MIPS R4300 with some slightly modified pin-outs and clockrate.

<Read the slides>

# Slide 8 – The RAM

<Read the slides>

# Slide 9 – The RCP

<Read the slides>

It’s a common misconception for people to say the RCP is split into just the RSP and RDP, with IO being a part of the RSP, but that’s not true. The interfaces have their own cache/buffers, their own registers, and physically-speaking, their own sections of the RCP. However, I’m not going to be talking about the IO interfaces simply because they aren’t super important for the developer and this was purely a side rant to waste your time.

# Slide 10 – Reality Display Processor

<Read the slides>

# Slide 11 – Reality Signal Processor

<Read the slides>

# Slide 12 – Layout, and why you should care

Putting it all together, the console, at its most bare, looks like this. You can see that the RCP is at the center, and good thing too because it runs at the fastest clockspe-…

I’m not sure what they were thinking… I mean, I get **why** the RCP is at the center: because the RAM contains everything related to drawing (like your framebuffer), the RCP needs to have access to it all the time. Giving it DMA was a smart choice, but cheaping out on its clockrate put the rest of the system at its mercy. For instance, those 500MB/s theoretical RAM peak?

HA!

With the RCP running at 60MHz, you’ll be lucky if you manage to hit half of that, especially because half of the DMA requests are being occupied by audio.

I’m being a bit mean because the N64 was incredibly tightly designed, and getting that much power in a single chip at such a low price is fantastic (after all, the RCP was made from scratch for the N64). But that obviously came at a cost to developers, and it isn’t really helped by the fact that Nintendo did not want to expose a lot of how it works for fears of it being copied by competitors.

Regardless, the CPU not having DMA to RAM is probably going to be a bit annoying, we’ll get to that in a bit.

# Slide 13 – Cartridges

<Read the slides>

Game size addendum – The largest commercial games were 64MB in size, and there were only two: Conker’s Bad Fur Day and Resident Evil 2. RE2 had FMV, while CBFD had fully voiced cutscenes. The CPU can technically address 250MB in total (some recent discussions have found that the CPU can actually address 1,967MB, but this hasn't been physically tested yet), but if you custom make carts with bank switching then your game size is potentially infinite.

# Slide 14 – Official Dev Kits

Alright so, you’re a brand new game dev and you want to try out the new shiny Ultra64 that Nintendo’s been harking about at E3. Here’s some reasons as to why you’d want to become a Nintendo partner: <Read the slides>

# Slide 15 – Official Dev Kits (Cont.)

<Read the slides>

# Slide 16 – Official Dev Kits (Cont.)

<Read the slides>

Obviously, in these prices, I assumed you already own an N64. If not, it’s relatively easy to get one nowadays with a controller for $50. These prices also don’t take inflation into account, so they hurt the devs a lot more back then than they do now.

# Slide 17 – SDK Options

<Read the slides>

# Slide 18 - Flashcarts

I know I’m repeating myself but I want to drive this point home. Get a flashcart. Like, really. With USB. Yes, it costs a bit, but you will save yourself frustrations (at least a lot of it, not all of it). I personally recommend a 64Drive (which I own) because of its highly documented hardware, incredibly fast USB, but also because the guy who makes it has done so much for the community that he deserves it for his hard work. Unfortunately, due to production issues they’re unobtanium until the backlog is dealt with. So your other commercial option is the EverDrive X7 (or if you can find a second hand one, the 3.0). Their USB is slower (it took 1 minute to upload a 64MB ROM in one of my tests), but they will work just as well. If you’d rather build your own, you can look at open source carts such as the SummerCart64.

I’m gonna shill a bit, but I wrote an open source multi-platform USB tool and library that supports all 3 of these options called UNFLoader, which supports both Libultra and Libdragon. You might find it handy for your projects, especially if you have a development team with varying hardware.

Back on topic, never trust an N64 emulator. Just because your game runs on one, there’s no guarantee that it will work on hardware. Even CEN64, the cycle accurate emulator, can’t be trusted, heck, there is not a single N64 emulator out there (as of making this presentation) that properly emulates the VI. I’m not just talking about “oh this pixel color is a little off”, I’m talking about emulating its quirks, which makes developing stuff without a flashcart almost impossible without hardware checking. Plus, if your game only works on an emulator, well you haven’t really made an N64 game, you’ve made a PC game with extra steps.

# Slide 19 – Programming Languages

Game Development on the N64 is pretty much all C. C++ was seldom used, but it’s entirely possible, you just won’t be able to rely on having STL available. You can develop entirely in assembly but it’s not required nor recommended. Official support for ASM was incredibly poor, but you insist (for writing some optimized functions), you can use BASS, armips, or GAS as your assembler. If you belong to one of the cults, some folk have gotten Rust and Zig compilers working for the system, but don’t expect any community support or examples for it.

# Slide 20 – Other Development Tools

One main point of contention with developers getting into N64 homebrew development is the fact that they have to DIY everything. Most of the old official tools are very difficult to find and, even if you do get them, are useless without the hardware to accompany it. For both libraries, you’re going to need to end up writing your own tools, or relying on community ones. Luckily for you, the number of tools has grown significantly since when I first joined.

Hell, if you wanted to get animated 3D models in your game, you had to write your own tool, library, and whatnot, unless you decided to use the same code as Super Mario 64. Your other option would have been to grab a copy of SoftImage 3D and tear your hair our trying to figure out how to do anything basic. You have quite a few options for static models, but no one really went the extra step of making tools for animated models. Now? Just this month we saw the release of 3 new tools for Blender that support different types of model styles (such as skinned meshes or sausage link style models). One of which was mine, obviously.

There are tools for music and sound, heck Libdragon recently saw the release of a microcode assisted XM player. For Libultra users, there are tools as well but they’re either command line only (which might not be very friendly for your artists) or the official SoundTools <Shudder>. The SoundTools are quite powerful, but without the official hardware they can be very annoying to use. Tackling audio tools is something I intend on doing in the near future.

Anything you need though, just visit <https://n64.dev> for all your resource needs, the page is community run and still updated. Just don’t expect to find premade game engines for the system.

Slide 21 – Development Process

Ok so, that was all the boring but necessary stuff out of the way. Now I’m going to get to the development process itself, aka actually making games. I’m primarily a Libultra developer, but I have consulted with Libdragon people to ensure this can be relevant to both sides.

The N64 only contains a very small kernel, everything else is made by you, tailored to how you best see fit for your project. If you’re using Libdragon, a lot of the Operating System overhead is handled for you. Heck, with Libdragon you don’t even need to worry about what a thread is. For Libultra developers, your options are to write the OS yourself, or to use Nintendo’s Nusys library to handle a lot of the first time setup busywork for you. Regardless of which route you choose, there’s no harm in understanding the operating system behind the game itself.

# Slide 22 – Threads & Tasks

The N64’s CPU is single-core, so threading is handled entirely by you. By default, Libultra exposes a preemptive scheduler system where threads are given priorities, and only yield to the next highest priority thread when they choose to do so. However, a running thread can be interrupted at any time. The CPU gives the RCP things to do (for instance, render a frame), and leaves the RCP to do that on its own. When the RCP finishes their task, it will generate an interrupt, which the CPU can then process and give the next task. Threads can communicate with one another using a message system, and if needed they can queue messages to deal with them all later. Threads can be paused until they receive messages from another thread, or receive an event/interrupt. Libdragon doesn’t support threads, but it is intended to in the near future. Libdragon doesn’t really use threads, but it does offer an asynchronous programming paradigm where you can write an interrupt handler to be notified of when RSP tasks finish (so that you can schedule another). There are plans to introduce them, however, pending a pull request on a recent Kernel rewrite.

# Slide 23 – Microcode

When creating RSP tasks, the CPU decides what microcode will be utilized. The microcode itself is linked in the ROM as a binary blob. There’s a few options available, I’ll go over some of them briefly. For 3D games, you’ll pretty much want to stick to F3DEX2. Fast3D was the original microcode written by SGI and it wasn’t very good, because it was designed to render high quality triangles (which for a game application, not a great choice). The F3DEX series replaced it later in the console’s lifespan and it’s worth using. For open source options, you have ugfx which was released last month (but is currently having its API changed completely), as well as the unfinished but promising libhfx. For 2D, you have the S2DEX microcode, but if you’re not doing anything super fancy then you can just use the things provided by F3DEX, they’re basic but they work. Libdragon doesn’t use any 2D microcode because the CPU issues RDP commands directly. For audio you have naudio or libdragon’s new RSP Mixer. I’ll cover audio a bit later. And if you’re so inclined, you also have options for video decoding on the RSP.

During the N64’s lifespan, there were a select number of highly talented devs like Rare and Factor5 that were given the source code for the graphics microcodes, along with their very poor documentation. These companies probably went through hell to get those microcodes working, as their later games look fantastic (heck, Conker almost looks like a GameCube title). I personally don’t have much experience with writing microcode, but from what I’ve been told it has actually become easier to do thanks to hard work on an interactive RSP disassembler called r64emu. Developers back then didn’t really have access to a tool as powerful as it.

# Slide 24 – The Framebuffer

Remember how I mentioned that the N64 uses a Unified Memory Architecture? You see, the RCP has very small memory registers, heck its instruction cache is only 4KB, so large microcode needs to be split into overlays. 4MB of RAM might seem like plenty, but it’s battling between all the other things that need to be there, such as your framebuffer and Z-buffer. The size of which, will depend on the region and bit depth. The most common size of framebuffer was 320x240 pixels (or 320x288 for Europe’s PAL users) although you can set it to whatever value you see fit, and typically you’d have two framebuffers so that you can utilize the double buffering technique to give your game a small framerate boost. If you do the math, that’s 300KB of your memory used up for the two framebuffer alone (double that if using 32-Bit color) on top of 150KB used up by the Z-Buffer. Half a meg just gone like that. You can cheat a little for PAL users, as the Video Interface allows you to stretch the Y scale of each horizontal scanline, while keeping the framebuffer’s size unchanged, but this will result in blurry video and make me cry.

Worth noting, because RAM is made up of 1MB banks, putting the two framebuffers in different banks will improve your game’s performance. Each time you read or write to a bank, you have to wait for a period of time for the operation to finish before you can access the same bank again. Since the RAM Interface can have multiple banks open at once, you can effectively read from a separate bank while the other is being used. Remember, you have full control of where things get placed in memory, alternating your data between memory banks helps immensely.

# Slide 25 – 1MB limit on code

On the topic of Memory Management, when you start your ROM, the bootcode will copy the first megabyte of code from your ROM and execute it. You can’t really change this aspect without modifying the bootcode, which is incredibly non-trivial for reasons I won’t get into. Essentially, this means you should avoid putting all your game assets in the codesegment or you’ll very quickly surpass this limit. Remember that the cartridge is super-fast, so you can load assets from it as your game is running. In the event that you do hit that 1MB limit with pure code, then you’ll need to apply some classic Operating System techniques to work around this limitation. You’re not just limited on streaming assets from your cartridge, you can stream code as well (just remember to fix the location of your pointers when you do). Of course, when we start talking about reading from the Cartridge and placing stuff in memory, we are effectively talking about a filesystem. You’re not forced into structuring stuff in a specific way, so the way you go about this is entirely up to you. If you’re using Libdragon, however, it has its own built in filesystem that you might find incredibly handy.

# Slide 26 – Explicit CPU Cache Flushing

And because we’re not done managing our memory, I think it’s also important to mention that because the CPU runs in Kernel mode, you are expected to do quite a bit of legwork yourself. The CPU typically is running your game in the KSEG0 address space, which uses direct address mapping, and is cached. The RCP expects that the addresses that you give them correspond to physical addresses, so you’ll need to correct those.

The CPU transparently caches data accesses on an onboard data cache. However, once something like a DMA operation occurs (to get stuff from ROM), the data is held in the cache until the entire cache line is written back, usually because a cache miss occurs which requires the same cache line. Since the cache and physical memory will no longer agree on the contents, you’ll need to manually writeback the data cache lines, or invalidate the cache when more recent data exists in physical memory. This is one of many annoying little thing that developers had to be aware of.

# Slide 27 – Graphics Programming

Alright so, let’s talk about graphics. First and foremost, this question gets asked a lot, but no, the Nintend64 does not use OpenGL. Yes, the OpenGL API was also created by Silicon Graphics, and their graphic engines were running off their proprietary IRIS GL, but the N64 uses its own thing (we sometimes call it N64GL (not an official name)). That’s not to say that knowing OpenGL is useless! The concepts that you’d learn from it (especially regarding matrix calculations, lighting, etc…) will definitely help you out, but there is differences in the details. Regardless, if you happen to have a copy the *OpenGL SuperBible: First* Edition (specifically about the 1.0 iterations of OpenGL), it will be a great resource for you.

The N64 (and early OpenGL) uses display lists, back when graphics cards were fixed function pipelines. You can dynamically create display lists as your application is running, or use static, premade display lists for things that contain static data (for instance, your character models). In OpenGL, you have something called the Post Transform Cache, which is a memory buffer with vertices that have been processed, but has not yet rendered. The N64 has an explicitly managed version of this, with the size of the cache depending on the microcode that you are using. The tricky part of the vertex cache is that the vertices have to be loaded in contiguous order. So for instance, if the vertex cache has a size of 32, you can’t selectively load like vertex 4 and then vertex 64 of your model, you’ll need to order your vertices so they’re together, or duplicate them if that’s not possible. There are plenty of algorithms for this, such as Tipsify or Forsyth, but you can also easily write your own. Preventing cache misses when loading your verts is pretty easy, and a great way to improve your performance.

The N64GL API provides a 10 element modeling matrix stack, like you’d expect any modern graphics API to have. The catch is that the matrix stacking is done only when the RSP is processing the display list tasks, so you can’t just call a matrix push operation and then reuse that matrix for something else, you need to leave that matrix untouched in RAM so that the RSP can load it during processing (which also implies you’ll want to keep a set of matricies for each framebuffer to prevent graphical issues). When a group of vertices is loaded, they are first transformed by the matrix stack. All vertex transformations are done only when they are loaded; sending a new matrix down later will not change any verts already in the vertex buffer. There is also support for vertex colors (which are used to great extent in a lot of titles), however be aware that you cannot mix vertex colors and lighting, because lighting calculations are actually modifying the vertex colors themselves. You can, however, mix primitive colors with lighting, just not individual vertices. The largest point of contention of the system, however, is the fact that texture memory is tiny. 4 kilobytes is almost nothing. For reference, if you have an 8-bit color image, the largest size you can manage is 32x32 pixels. If you’re using mip-mapping, then you’ll lose half the size of your texture memory. There is **no way to bypass this limitation**. The best thing you can do for rendering higher quality textures is to either reduce the bit depth (if you use 4 bit black and white images, you can fit almost 128x128 textures, with the caveat being you need to color them via vertex colors), or you chunk your image into smaller bits loading them part by part (which obviously means you might need to increase the number of triangles on your model).

# Slide 28 – Color Combiner

Not having access to shaders seems like a large limitation, but you are allowed to modify certain things in the RDP. One of these things is the Color Combiner, which is essentially like a fragment shader. The CC has a very simple equation that it uses to color every single pixel, where each variable is a different source: such as the texture color, the vertex color, noise, primitive color, environment color, shading color, or just a 1 or 0. The CC is also used for Alpha, and it uses the exact same equation. The cool thing about the CC is that you do it in two passes, putting the calculated color from the first pass as a variable for this equation. This will allow you to perform three color multiplications (for instance, vertex colors with the texture color and with the environment color). Lemme demonstrate:

# Slide 29 – Blender

The Blender (not the modeling program) is used to combine the colors from the CC with the ones in the framebuffer, it is essentially what shades the final pixel. It also calculates the Anti-Aliasing and Z-Buffering in the process. I didn’t demonstrate it in the previous slide, but usually the CC and Blender need to work together for the output to be correct. The Blender is a pretty complex beast, with more than one equation for calculating which pixel gets outputted and how its combined with others. I’d be here all day talking about all the different intricacies, so I would highly recommend reading the manual pages on it. There’s even a massive matrix of what the different blender mode combinations will affect what. They’re large but will cover it a lot better than I could.

# Slide 30 – Model Animation

It helps to animate your models to make your game interesting, but the fact that the N64 doesn’t provide any built-in ways of doing this usually overwhelms a lot of newcomers, who’ve never had to think about how to implement something like this before. The most common method used by titles on the system is what we call the “sausage link” system (I’m not sure if there’s an official name for this, but even if there is I doubt it sounds as fun as this one). In the sausage link system, models are made up of a bunch of uncommented segments, which are individually manipulated by the matrix stack. This method is popular since it’s relatively lightweight in terms of memory (you only need to store final position/rotation/scale values of each keyframe in your animation), CPU computation if you intend to lerp two different frames, and because the vertex transformations are offloaded to the RSP. If you want to have fully skinned models (like you do on modern games), you can store multiple copies of a model transformed per frame and just load them one by one between frames (remember, the DMA is very fast). You can take a hybrid approach to save memory by cutting tween frames, and then linearly interpolating on the CPU, but this will become very heavy as the number of vertices increases. At that point, you might as well implement a skeleton system. There is no microcode for skinned models, but it’s theoretically possible and a much smarter solution than simply doing it on the CPU.

A cool trick that the Super Mario 64 modding community uses to fake skinned models is by abusing the fact that vertices are transformed by the matrix stack when they’re sent to the cache. This means that, with careful ordering of your vertices, you can effectively fake skinned meshes. The idea is that you load one bone’s worth of vertices, ensuring that you load the vertices at the bone’s tail **last**. Then, you push the next bone’s matrix and you load its corresponding vertices (ensuring that you don’t overwrite the previous ones). Now, because you have both sets of vertex edges, you can effectively dynamically render some new triangles to connect them. The display list construction for this is done by hand as no one has written any tools to automate it, so this method only works well for meshes that don’t have UV coordinates, since getting the textures to look right is going to be tricky. The other caveat is that this only supports vertex weights of 1 or 0, you won’t be able to have any in-between weights.

# Slide 31 – Sound and Music

Audio development largely depends on the microcode you’re using, because the N64 doesn’t have a dedicated audio chip. The N64 isn’t locked to specific waveforms like the NES was, instead you place a 16-Bit audio buffer somewhere in RAM that, with the combined effort of the CPU and RSP, you write waveform data into. You then send these buffers to the Audio Interface in a fixed time interval. Because PAL and NTSC N64’s run at different clock rates, the audio will differ ever so slightly between the two regions, so it’s important that developers properly test their stuff on both systems. It’s also important to give audio as much priority as you can, because a pop or click in audio is much more noticeable than a dropped frame or two (and more annoying). So it will help to write a scheduler to interleave audio and graphic tasks properly.

Thankfully, a lot of the hard audio work has been done for you, so you can instead utilize the provided audio libraries for your SDK of choice. For Libultra, you have the naudio microcode which is used by both the SGI audio tools and the PC SoundTools. The libraries work with a custom Nintendo ADPCM format, and music is typically MIDI. The RSP is used to decompress the audio, apply effects, and generate the music notes from your MIDI by using soundbanks that you give it. The MIDI themselves don’t have any channel limits per se, because in the end everything gets crushed into one buffer, so that information would be lost anyway. Libdragon recently got new audio microcode which doesn’t have an official name as far as I know; it’s just referred to as RSP Mixer. Unlike the Nintendo microcode, it’s designed for XM modules instead and uncompressed audio samples (such as wav files)

If you really don’t care about your ROM size, you can store and stream uncompressed audio from the cartridge. One highly popular method for compression though is to store your sounds sped up in ROM, and then slowing them down in the RSP (because sped up sounds are shorter). Obviously the harder you do this, the worse the outputted audio quality will be. Regarding MP3 playback (since I know someone will ask about it), it is possible (for instance, Conker’s Bad Fur Day). MP3 is a bit heavier on the CPU than MP2 audio, however I don’t know of any public microcode or library for either audio format.

# Slide 32 – Performance

Putting all of that information together, you’ve coded your game but made a horrible discovery! It runs like ass! Yep, that’s right, get used to it. F-Zero X is the only commercial game for the system that runs at a near stable 60 FPS. I’ve mentioned it before, and I’ll mention it again, most games for the system are fillrate limited, so use the CPU as much as you can to reduce overdraw, clip out triangles, etc… I’ve heard people talk about overclocking the CPU, but please don’t do this. Not only will it not benefit the performance of 90% of the games out there, you can potentially mess with the timings of the RCP, VI, or even break games in the process. This will also obviously generate more heat, which isn’t a good idea of a passively cooled console.

One of the best ways to improve your performance is to reduce texture loading calls. There is someone who made Mario Kart 64 run 40% times faster by simply reducing texture loading (IE, rendering everything that uses the same texture at once so you don’t need to load it again). Another common technique which is also used a lot nowadays is to draw from front to back, because pixels that are drawn behind a triangle can be skipped early. Also an interesting tidbit, the Z-Buffer is atrociously slow, but not everything that you draw requires it! You can selectively disable the z-buffer to maximize your performance. Heck, you can even toggle Anti-Aliasing between frames (as in render one frame with, one without).

More obvious things are to use culling algorithms on the CPU, like binary space partitioning, and to use level of detail models. I’ve mentioned it plenty, but since the CPU bus is 32-Bits wide, try to have your data aligned to it to prevent cache misses, as well as keeping your data between different RAM banks.

I know using compiler optimizations sounds obvious, but Super Mario 64 shipped in debug mode and the performance difference is very noticeable in CPU bottlenecked areas. Also, read the manual. Like really. Do it. Yeah I know its 30 chapters long, and very dry, but there’s so much important stuff in it.

Slide 33 – The Now, and the Future

Alright, to wrap up this presentation, I want to talk a little bit about the beginning and future of N64 homebrew development. If anyone follows gaming news, you might be aware that Nintendo has a notoriously, er, crappy attitude towards fan games, shutting down projects that have had years of work put into them. I’d like to put your worries to rest by mentioning the fact that, historically, Nintendo has **never** gone after homebrew games. Nintendo only acts on fan games, because people are using their IP/characters without permission. Heck, the Super Mario 64 decompilation project is still up after 3 years, simply because it contains no game assets in its repository.

# Slide 34 – The Humble Beginnings: Dextrose

The year is 1997, a user by the name of Actraiser registers the website dextrose.com to collect information about backup devices. It quickly becomes a place where people gather to discuss and develop homebrew for the N64. Unfortunately though, most of this website has been lost to time, but it can still be partially surfed through the web archive. However, in 1998, dextrose in collaboration with Bung Enterprises, who developed backup and development units for the N64, announced a demo competition called Presence of mind, with the winners taking in some sponsored hardware from Bung (every competitor also got a free t-shirt). There was a second, and final, PoM in 1999. During this timeframe, since the system was hot, we saw the release of a ton of demos, intros, and homebrew.

# Slide 35 – The Great Depression

In 1999, an SGI employee leaks internal documents, Verilog code, and much more in a big archive known as the Oman archive. This has unfortunately tainted the N64 scene since, as the forbidden knowledge within was used by some emulator devs and has propagated throughout the rest of the scene ever since. From the many emulators, Project64 is released in 2001 and it rises above the rest as one of the best N64 emulators. Eventually though, it’s time for the new console generation, so the N64 is discontinued in 2002. At this point, interest in the system started to fall, and it was kept alive by a small handful of users, mostly who were emulator developers, hackers working to translate Japan only games to English, or people interested in taking the system apart and figuring it out, bit by bit. The emulation scene, unfortunately, also saw a lot of infighting and elitism which still lingers to this day.

# Slide 36 – The Renaissance

Eventually several years have passed and several game studios fell. Many of these liquidated their assets in auctions, which led to the SDK disks, among other things, ending up in the hands of collectors. In 2008, however, through the collaborative work of many users, a free and open source Library for the N64 called Alt\_libn64 is released, and it eventually becomes what is now known as Libdragon. 6 years later, Discord appears and the N64brew Discord channel is created, which I end up joining and releasing my first homebrew project in 2018. It probably feels a little cheeky putting my own project in this timeline, but you do have to realize that Pyoro64 was probably first “properly” documented N64 game, at least no one before me had gone to such an extent to make a writeup about how to make N64 games from scratch. So much so, that sometimes I can spot a little bit of my Pyoro code in other people’s projects. Super Mario 64 is kept relevant by its insane speedrunning community, but in 2019 with the release of the decompilation project, interest in hacks and general N64 development explodes. And last year, N64brew hosts the very first N64 homebrew competition, 20 years after the last Presence of Mind. The N64brew wiki is also created, which now serves as a hub for accurate N64 hardware and software documentation. In case you’re wondering, yes we’re hosting another game jam right now, although it’s a little late to sign up!

# Slide 37 – Charts

This chart isn’t probably super comprehensive, but I’ve tried to the best of my ability to ensure its correct. The numbers in the Dextrose years is a little inflated due to me not differentiating between intros, demos and games. I’m not including hacks or translations in here, this is strictly homebrew. <talk a bit about the chart>

# Slide 38 – Communities

There are a few active communities worth noting regarding N64 homebrew. The N64dev IRC channel is very old (it was around at least since Dextrose) and it contains most of the old timers and super knowledgeable folk. The N64brew Discord is probably the go-to place for N64 homebrew development, with Discord64 being the other option (which serves as an all-around pit stop for N64 development). You have iQueBrew which is homebrew but for the Chinese version of the console, and a myriad of hacking communities for different titles. And, while not super active, you also have the N64Homebrew subreddit.

# Slide 39 – Thanks for putting up with me!

Alright, that’s it! You can reach me on all the places listed on the slide. I’d like to give a special thank you to: <Read the slides>. Do we have time for questions?