I will present why making a game engine from scratch, especially for simpler or 2D focused games can actually save time and resources in a long run, as compared to using existing solutions such as Unity or Unreal.

When planning McPixel 3 development, after careful consideration, I have decided on rolling my own game engine written from scratch in C. All this was done while ensuring that the game remains as portable as possible, but is also relatively scalable and does not limit the scope of the game in any way.

The porting team supposedly had no major issues thanks to how the engine was structured, and I want to talk about not only how building an engine from scratch can be beneficial, but also dive deep into detail how to structure the engine to be able to do that easily.

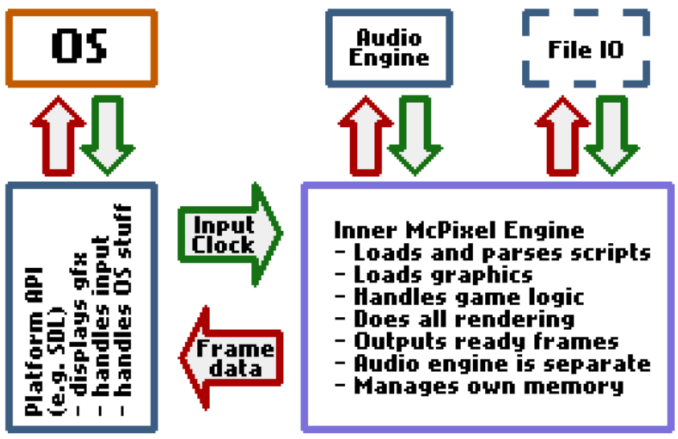
The game was ported to and is now released for:

- Windows
- macOS
- Linux
- Nintendo Switch
- Xbox One
- Xbox Series
- Playstation 4
- Playstation 5
- iOS
- Android
- Windows 95
- Windows NT 3.51
- MS-DOS
- FreeBSD
- Raspberry Pi
- Haiku
- Linux Terminal

Additionally, I was able to run the game on really old hardware, including a 386SX CPU with 8MB of RAM.

I will talk about the engine structure and how to build it for scalability and portability.

- **Software Rendering** – McPixel 3 Engine uses software rendering, so it's independent on hardware. Whatever hardware you can get it to build for, will display exactly the same thing. This saves tons of headaches.
- **Block-based approach** – McPixel 3 engine is a single giant block that you feed with input and it renders images. For most platforms, it is plugged into SDL2, but that can be easily replaced with other libraries. Other platforms use Allegro, WinAPI, or even ncurses, and adding more is as simple as rewriting a single source file. Below is a diagram of the engine to illustrate how it is structured for this:



McPixel Engine - Outer Block Diagram

- **Using C with GNU99 extensions as a language of choice** – using a bit outdated language standard, like C99, or C++98 without STL provides best compiler compatibility. I will talk about how you can utilise these seemingly alien and outdated technologies to have a really smooth coding experience, with things like FOREACH and automatic JSON parsing in C.
- **Memory management** – this is something that people think will cause trouble, but for non-open world level-based games the solution is quite simple: pool all the allocations for a given level together, and then just flush everything when unloading the level. Zero memory leaks, and you don't have to keep track of anything by hand.
- **Research and planning** – this is a very important part. Initial version of McPixel 3 was made using McPixel 1 tech (flash/Adobe Air) and only later the engine was made from scratch. By making a vertical slice prototype game first and only then planning out and making an engine, you make sure about two important points:
    1. You are able to plan the engine accordingly, as the game is already laid out. You do not allow yourself to be surprised, as you know what's in the game, and what may creep in later already.
    2. You are able to assess production time effectively, as you know how the game is going.
- Finally, I will talk about how this approach can cut production costs, as you are able to work on the game on much lower hardware than one would need to work on high-end engines.

During my presentation I will show code snippets from the engine, block diagrams, and pictures of the engine running on different platforms. I will also present testimonies form the console porting team that worked on the game. I will make it clear that Unity's or Unreal's "port the game" buttons will in the end cause more headache than rolling an engine, owning all of your codebase and being able to fix anything without using workarounds.

I will focus on providing project programmers with enough information to be able to consider rolling their own. In the end McPixel 3 engine was ported to pretty much everything with relative ease, and I think the game is an outlier in terms of portability.

I will also want to mention how environmentally friendly this approach is, as an overblown premade game engine is bound to use way more resources than necessary, and using a written-from-scratch focused solution will use way less resources for the end user, not only lowering the environmental impact, but also ensuring long battery life, and thus perhaps making the game a "on-the-go" choice for many.