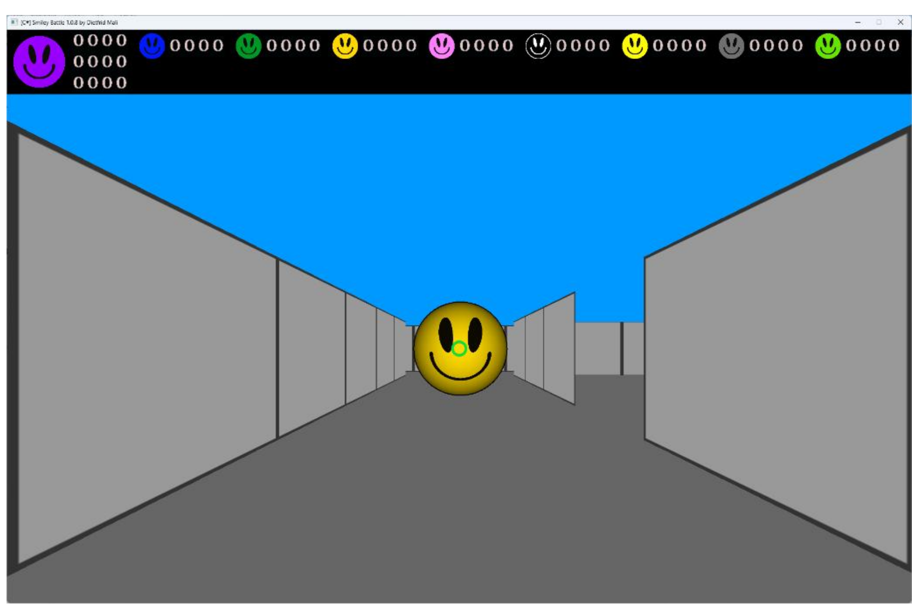
The Making Of

The Beginnings

It all started 40 years ago, when I had the chance to participate in a LAN party setup by students at my university, connecting a bunch of Atari ST computers via their Midi interfaces and playing the first 3D deathmatch shooter with fluid movement and visible shots ever: Midimaze.

The Prototype

Many, many, many! years later I decided to recreate Midimaze in Python, C# and c++20 as a coding exercise and to hone my coding skills in these languages. The first version was written in Python, complete with working multiplayer networking code and OpenGL renderer. From there, I ported it to C# to learn the language better. Working with C# is awesome – it's all the power of c++ you actually need without any of its hassle. Finally, I ported it to c++20 because that imvho still is the holy grail of programming languages, the one that solves it all, or simply the "42" of the coding universe. 😊 (I wonder how many of you will understand that reference). You could also say the answer to a question nobody knows …



First Attempts

The first version looked just as bland as the original: No textures, just color, the only improvement being way higher resolution and smoothed images thanks to modern graphics hardware. Midimaze ran on a very low resolution software rasterization – there were no graphics acceleration and rather any mathematical coprocessors available at that time. The Atari ST had at least a super-modern and very powerful Motorola 68000 16 bit CPU at its time, which was way better than the very common Intel 6502 8 bit processor. Unfortunately, Intel won the race for CPU dominance, despite every other competitor having way better hardware …

Where To Go Now?

A few years later (pronounce with a French accent as in all the shorts where this audio meme is being used), after having lost my job and not finding a new one, I decided to make something of what I thought I was pretty good at – coding – and build and sell a computer game.

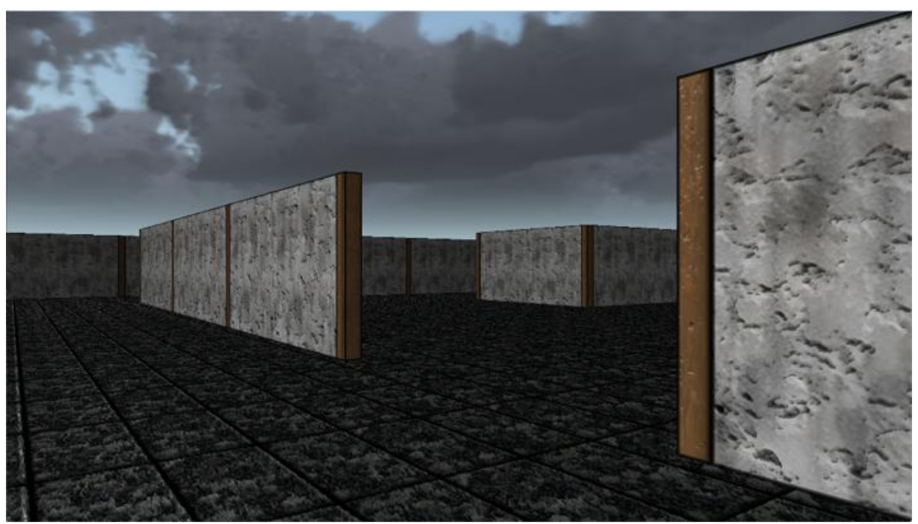
The thing is: Every day, over 40 computer games are being published on Steam, the biggest digital game distribution platform - alone every day. Many of these follow the same pattern of being quickly hacked together to satisfy the desire for entertainment or distraction with some game principle repeated a hundred thousand times already. Quick, simple, assembly line produced pieces of software "to go". Of course, there are also gems on Steam, but most game releases will just drown in the mass of other games, many resemble each other and are just made in a few months to generate an income.

Striving For Quality

I never wanted to build something like that. I always was very much into quality and polish, even in a simple game. So when my son mentioned to me that my game would look much better with textures, I added to start these. The next thing he told me was "If you have textured walls and floor, you should also have a textured sky!" At that point, the sky still was a flat, blue plain with some color gradient in the distance to give it a little more depth. He was right, so I used some old cloud textures I still had from another project, rendered them at three different heights and animated them to make them look more like real clouds. The result looked awful.

Moving On To 3D Geometry

Another thing I had to change was the geometry – the walls and doors. In the beginning, these had no volume: They were flat as a sheet of paper, true 2D. This did not fit the direction my game had taken. However, there was a twist to having 3D geometry: All walls in my game were made procedurally, i.e. by code, and not modeled by a 3D modeling program like e.g. Blender. It took me a while to build that as well; far not as long as clouds and decals though.

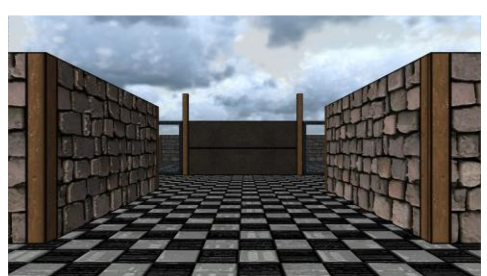


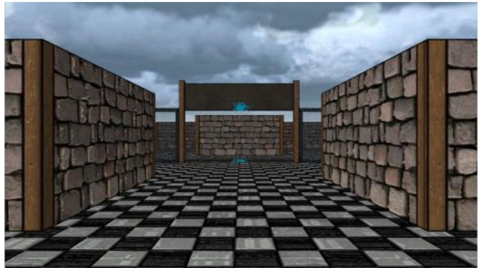
Shadows

Next thing that was missing were shadows. There are various approaches, each with their pros and cons. I went for so-called shadow maps. However, these have the problem that they cannot be arbitrarily large, and if they are too small create horrible visual artifacts like jagged edges or even big spikes. After a while of pondering on this, I found a very viable solution for this that works even for rather large scenes, like the game's biggest levels.

Sliding Doors

Once I had gotten that far, smaller flaws became obvious to me, like the vertically sliding doors sliding above the surrounding walls and hovering in the air. Since my geometry consists of wall and door segments between posts with a quadratic cross section, the simple solution was to make the wall posts next to doors higher. Now it looks like the doors move between two door posts containing some mechanic (which I did not model).



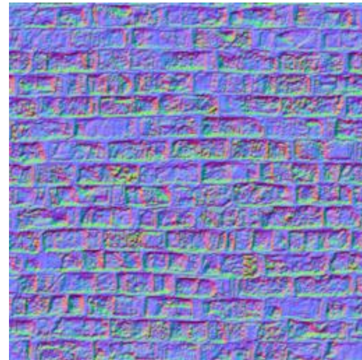


Paint Splats

The next thing I wanted to have was paint splats where paint bags thrown by the players hit walls or doors. This requires projecting "decals" on the walls. This also is a well researched and documented area of computer graphics, but again I had to fiddle everything together – and I didn't like the standard solution to this, so I implemented my own approach to putting decals on moving surfaces. The highlight is paint splats on doors that get divided when the door opens. I have a very elegant method to handle that; way more elegant than what I have learned is the usual way to do it.

Geometry Detail

After I had shadows and paint splats, I had to return to the wall, door and floor texturing in my game. There are techniques to give it plasticity and detail, but I lacked the data to create the required data (normal, displacement, ambient occlusion and albedo maps if you care to know … you probably don't). However, the latest technical marvel of this world called "AI" has blessed us with free and pretty good tools to create that information just from the textures themselves. Once I had that data, implementing normal mapping was simple, and it looked so much better: Walls now have real structure, small shadows created by the pits and bumps in them, and look way more realistic.

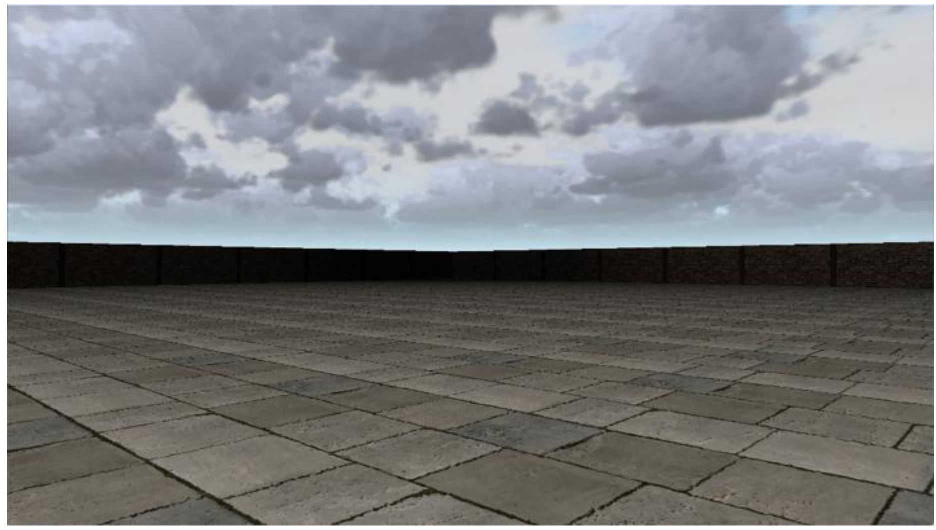


Volumetric Clouds

A while after that I stumbled across a video about volumetric cloud rendering, which means rendering true 3D clouds, simulating the way clouds appear in the real world. My problem was that I didn't find an easily accessible way to implement them. There is a lot of theoretical work about how to do it, and there is an established standard, but you need to put together quite a few pieces to make it work. It took me weeks to have at least something looking half way like

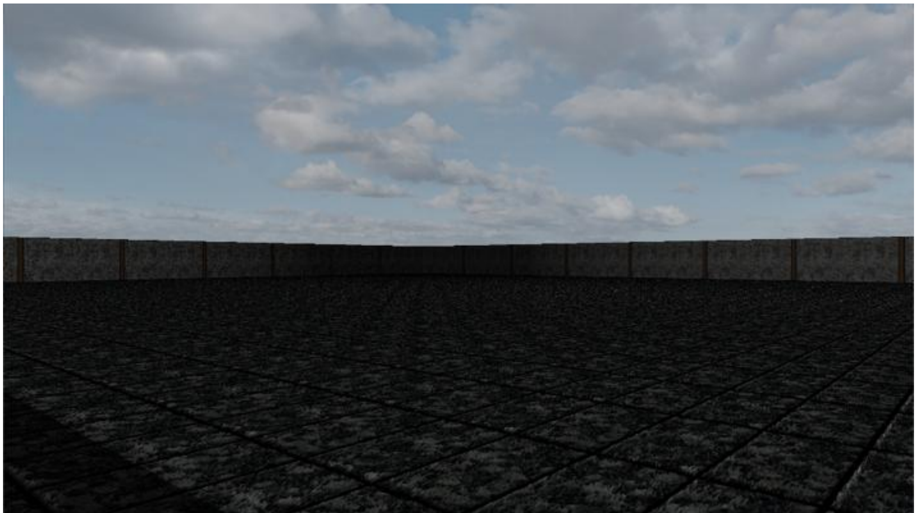
clouds, and I was returning to my cloud renderer again and again, even to the very day before I started writing this history of my game.

After a lot of tinkering with volumetric cloud rendering I finally arrived at something good enough to be put in my game. They do not look super realistic (I have a version that can do that), but the way they look they fit perfectly to the visual style of Smiley-Battle.



Sky Box

Still, my volumetric cloud implementation was by far worse than fully professional ones. Particularly the speed was too low, which shows on weaker graphics hardware. Until that moment, I had never considered adding a so called “sky box”, which is exactly what it sounds like: A huge box around the scene with a sky imagery painted on its inside. If the box is huge enough, it looks very convincing. It turned out that creating the skybox geometry cost me almost nothing.



The problem was getting good sky images that were laid out in 6 rectangles (actually 5, floor isn’t needed). There were fantastic free HDR sky surround photographs available, but not in

that format. But alas, AI came to my help again: There are free online tools that map a regular surround photograph to a cube's sides.

Powerups

At this point, I still by far wasn't done. I didn't just want to offer the decades old, extremely simple gameplay of Midimaze. A modern game in its spirit would need to offer more. So I started adding a complete powerup mechanic to the game. Devising, building and testing that took weeks and months as well.

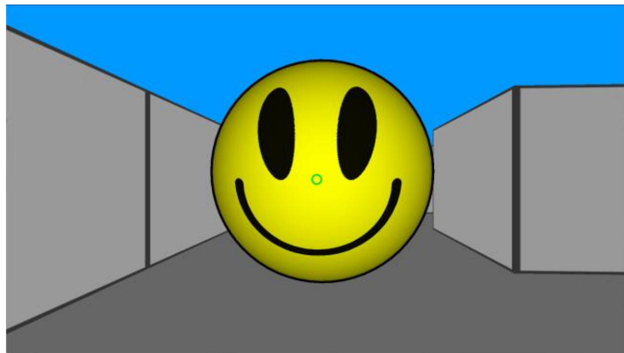
User Interface

A very big deal in building my game was creating the user interface. The initial prototype of the game had no UI at all: It was controlled by a settings file containing all information needed to set it up and run a multiplayer game.

It took me months to create the user interface, and it had a lot of iterations. The final result was very pleasing to me: Everything just works with icons describing the function of its UI element. No text and hence no localization. The only text are arabic numerals required to enter private match ids and IP addresses. All buttons are round, just like thee smileys. There is a consistent design language throughout the entire game. The entire game can be fully controlled with a gamepad, which also is the preferred input device for controlling your smiley in a match.

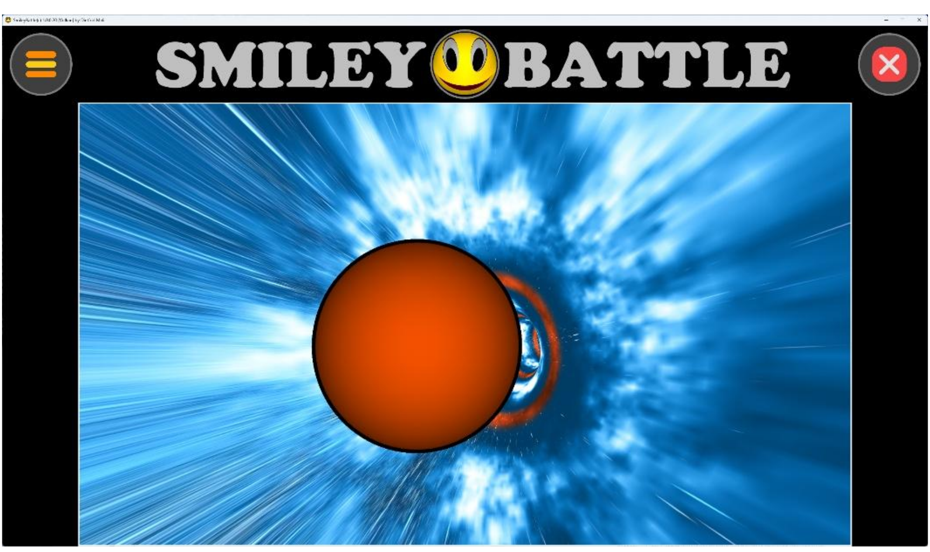
Smileys

My initial smiley just was a sphere with a face I had created with GIMP painted on it. It looked quite similar to the one from Midimaze, but when I had gotten that far I wanted a true 3D smiley with slightly recessed eyes and animations between its different facial animations. I tried to do that myself in Blender, but that was just too much of an effort to me. After several attempts on fiverr, I finally found a modeler who did an exceptional job and also was really pleasant to deal with (his fiverr name is Lathronelei and I can only warmly recommend him). What he had built matched my vision of how the smileys should look to 100%.



Creating An Animated Main Screen

A big question for me was what to fill the main UI screen with. It has only two buttons, one being the exit and the other being a hamburger button with a drop-down menu tied to it. I wondered whether to show a randomly selected level with some bots playing against each other there. However, I didn't have any bots at that time, and that would have also been technically quite challenging to do and probably also too much for the game's main screen. In the end I came up with a smiley travelling through a wormhole, waiting for a match it could enter. There was a twist to this though: I had seen a youtube tutorial video where a very fancy wormhole was created. I wanted exactly that look in my game. It took me a year to get there and required the volumetric cloud rendering as prerequisite. Only today did I finally manage to achieve that look.



Still, I hadn't reached the end of my journey here. There were more steps I had to take:

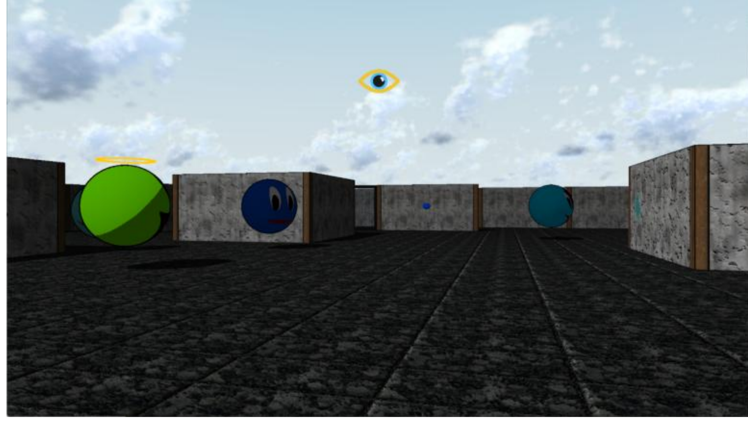
Integration With Steam

Implement a networking interface with Steam's network infrastructure and add bots.

The great thing about Steam is that it doesn't just take away a lot of your revenue when selling games on it, it also offers you a lot of value, particularly if you are a solo developer like me: You get full networking with all problems solved, something that would otherwise require a lot of servers world wide and a lot of networking problem solving. In my opinion, it is 100% worth the money.

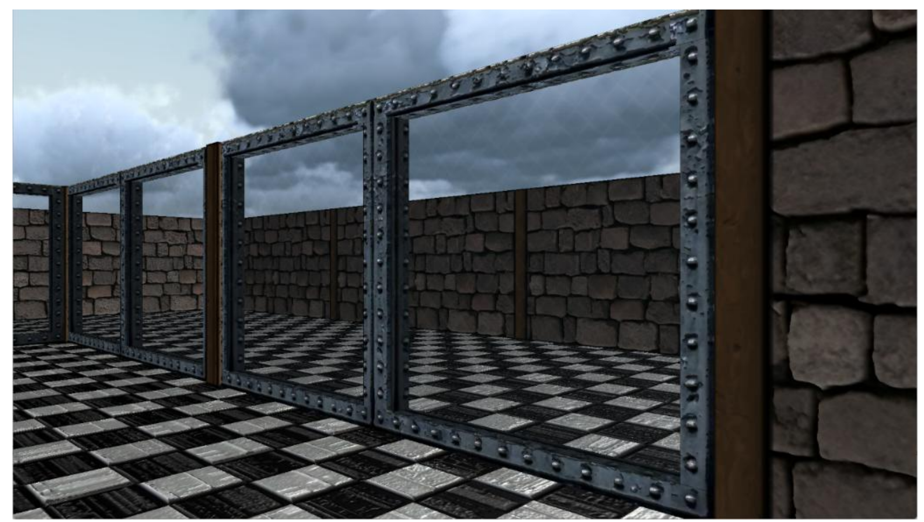
Bots!

The last big feature I added were bots. I shied away from that because creating good bots that are challenging and play well is not trivial, not even in an environment as simple as Smiley-Battle. Fortunately, I had implemented an almost complete bot controller in my C# prototype of the game (C# is awesome for quick prototyping), and the only thing I had to do apart from porting it to c++20 was to fix a few bugs and add a little more behavior. Bots now behave differently depending on the difficulty level chosen, and also have individual differences in their behavior, as they have various traits like intelligence, aggressivity and agility that make them use powerups more or less likely (yes, bots can and will use powerups!), attack you more or less often and dodge shots aimed at them better or worse. As I said, I had shied away from implementing bots, but given the fact that Smiley-Battle isn't exactly the next big block buster with a huge player base, I had to offer a good single player experience as well, and I hope I succeeded with that.



More Graphical Improvements

Guess what? I still wasn't quite done yet. A few days ago, Youtube started presenting videos about Crimson Desert's impressive graphical details to me. This game has just amazing details everywhere, becoming very obvious in stone and brick walls that look incredibly three dimensional. The video author didn't get the method the game's makers had been using right at the first attempt, but he managed to figure it after a few iterations, and it actually was a rather old way to produce a 3D effect on surfaces called screenspace displacement, producing much more convincing results than normal and parallax mapping, as it directly manipulates each pixel's distance from the viewer and doesn't just manipulate the brightness of each pixel depending on view, light source and a pixel's normal directions.



Porting To DirectX 12

Until around that time, Smiley-Battle was using OpenGL as graphics API. Since I also wanted to offer it on XBOX, DirectX 12 was mandatory. Porting it took me 12 days with AI support (which was awesome on one and a pain in the neck on the other side). Adding XBOX networking support took just a few hours, as its interface is almost identical to the Steam networking interface (all the same design principles).

Right now I am waiting for Microsoft to approve of my game and accept me as XBOX developer – only then will I be able to complete and test the XBOX port of the game.

Porting To Vulkan

After the DirectX port was done, I used it as a base to port the renderer to Vulkan. During the work on the DirectX port I had learned enough about how Vulkan works to already implement the required super structure in my renderer. Porting to Vulkan took only two days, as it is pretty similar to DirectX in its data structures and data flows.

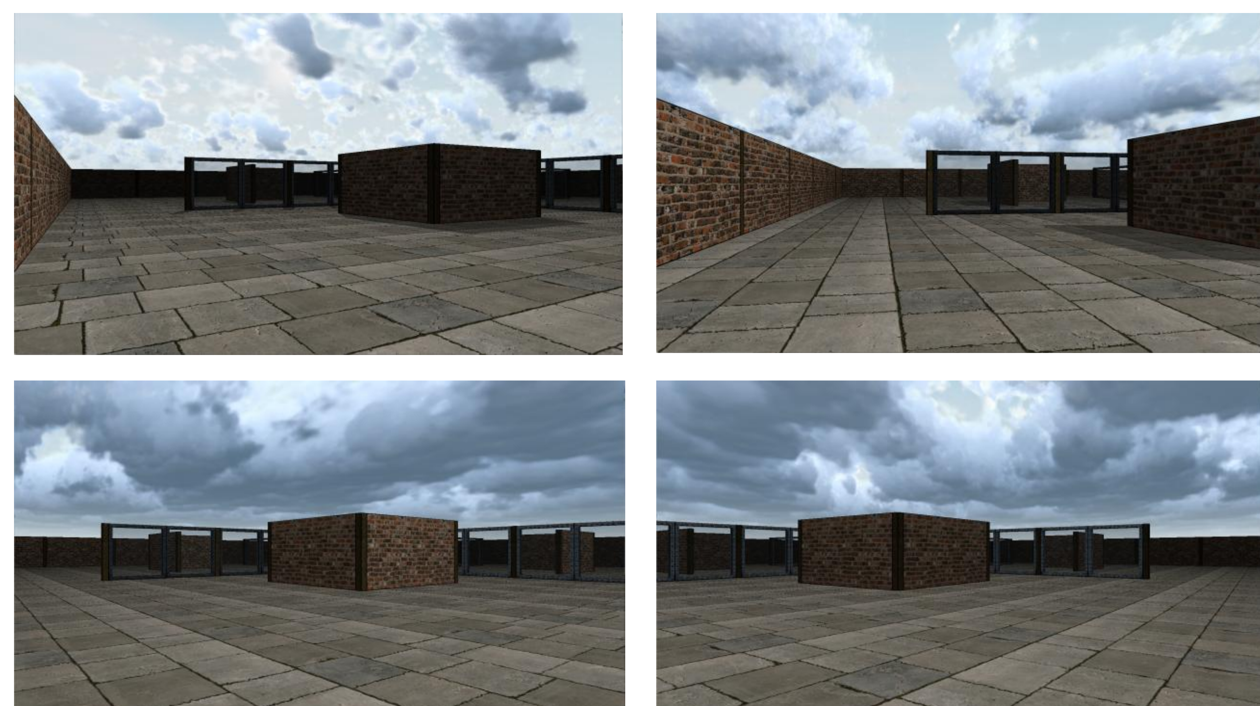
Renderer Optimizations

Once both DirectX and Vulkan renderer were working, I found that they were slower than my OpenGL renderer, which was a bit unexpected. I spent a few days for optimizations, and now the DirectX renderer beats the OpenGL one by about 10% and the Vulkan renderer by about 20%.

Volumetric Clouds - Again

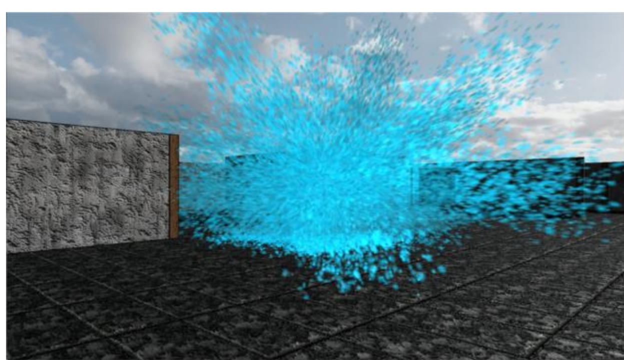
Yeah, again … my volumetric cloud renderer was just way to slow to be usable on any but the most powerful rigs, so I delved into this topic once more. I just didn't want all the work I had spent on it to go to waste. The solution was a TSP based cloud renderer. TSP stands for temporal space probes. What this simply means is that you do not update every cloud voxel every frame, but use an update stride between them. This results in slightly reduced visual quality when the clouds move, particularly when they move fast, but this gets compensated for by the increased rendering speed. This works reasonably well now for halfway fast computers. For slower ones there is still the also very good looking HDR image based skybox as fallback.

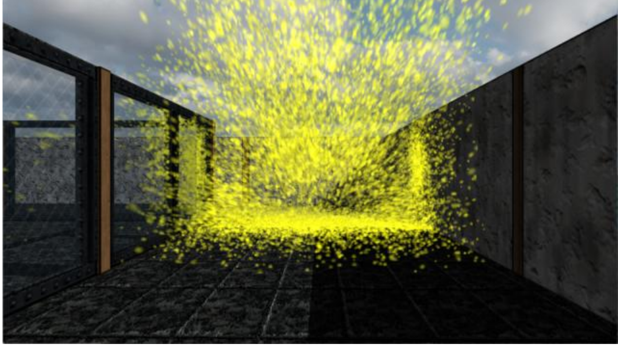
Even after all that work, I still wasn't happy. I felt there was more that I could do. So I went back to coding and tweaking the volumetric cloud renderer, changing settings here and there and everywhere, improving the physics simulation, squeezing every bit of performance and visual quality out of it I could. I went through different noise shapes, added various methods to gather light information inside a cloud, studied reference documents from the makers of Horizon Zero Dawn, who had pioneered high performance volumetric cloud rendering in computer games, until I finally reached the point where I had done all that could be done (at least by me). This was when I finally could settle with what I have achieved. My cloud renderer's output now looks better than ever and runs faster than ever before. I can now run volumetric clouds on lowered settings even ony my over a decade old gaming notebook with an ancient Intel Core i7 cpu and Nvidia GTX 970m graphics hardware on full hd.



Particle Systems

Since the beginning of my game, knocked out smileys started to wobble like a soap bubble until they finally burst. It was ok, but I was never really satisfied with this. Smiley-Battle is all about throwing paint bags and splattering everything with it. So you're basically full of color – where is it when you get knocked out and burst? This was screaming for a particle effect, and when everything was done, I finally found time and focus to create one. Now smileys that burst do so with a fountain of color going everywhere. I have to say that I find the effect quite gorgeous and well fitting.





Bot Improvements And Bug Fixing

After all this was done, I had some time to look into other areas of the program, seek flaws and improve the game there. The most obvious field for that were Bots with their rather complex autonomous decision processes and behavioral patterns. I used Claude code to analyze my code, and it did indeed find a few bugs and weaknesses I could iron out with its help.

While working on the bots, I also found a few crashes caused by the bot code -something I definitely wouldn't want to release the final version of my game with.

Another nasty bug was the app freezing when Windows opened the firewall dialog, asking whether Smiley-Battle was allowed to access the internet. The game completely froze even if you clicked "yes". That was a real show stopper. You wouldn't want to start your first multiplayer match only to have to shut down the game using the task manager ... Most people probably wouldn't bother starting and trying it again – which would have worked, as permissions were granted, but yet.

Code Optimization

The final steps in polishing Smiley-Battle were code optimizations and a bunch of bug fixes. A lot of the cloud renderer shader code was scattered over three different file versions: One for OpenGL, one for DirectX, and one for Vulkan. Fortunately, apart from the interface, HSL and GLSL are very similar, so I unified all common codes in shader snippets that can be used for every graphics api, making maintaining and updating all three versions very simple.

AI Supported Coding

I have been using ChatGPT and later on Claude code on its max plan to support my work with Smiley-Battle. To send that ahead: Claude code is worlds apart from ChatGPT.

Even so, Claude code will usually not produce perfect code, but it is very fast in gathering the knowledge required to complete a task and writing the code. What you got to do is look at what it did, understand it, and give it structure and quality. I have found that this worked pretty well for me and Claude followed my related instructions quite well. Bottom line: To have an LLM produce good (complex) code, it needs strong supervision and direction by a skilled SW dev, but if it gets that, it speeds up the SW dev's work significantly, and you don't sacrifice quality for that. I e.g. ported an OpenGL based app to Direct X without any knowledge of Direct X. It took me twelve days - not several weeks as it probably would have without Claude code.

In the process I gained a very clear understanding of how DirectX works and what infrastructure to implement on top of the vanilla DirectX data structures and calls to build a sleek system. Once that system was in place, it only took two days to port it to Vulkan, as I

already had that in mind when creating the DirectX infrastructure - and I had no clue of Vulkan either. I would never have been able to do all that in just two weeks on my own. It took a few more days to do api specific optimizations. Now I have blazing fast, high quality DirectX and Vulkan rendering code, and it took me three weeks to get there - from zero to hero. So if you understand the weaknesses of current LLM based coding agents, you can work very well with them and achieve great results in incredibly short amounts of time - slower than just accepting everything they throw at you, but that way you fuse together LLM speed with human expertise.

Color Your Friends

... And Your Life!