Red Eyes

- gory technical details -

Introduction

The ‘Red Eyes’ demo / video is part of Remute’s music album ‘Technoptimistic’ which got released on a Sega Mega Drive cartridge for ‘authentic’ listening pleasure (https://remute.bandcamp.com/album/technoptimistic) . On a 4 Megabytes cartridge (hardware provided by Everdrive producer Krikzz) it features 16 tracks along with the video - embedded into a neat GUI. It allows the user to select a track, play it with button A, stop the playing track with button C and pause / ‘freeze’ it with button B which often generates an interesting ‘hanging note’ effect, inviting the user to toy around with it. It’s a rock-solid product, but working on it took months and was a very interesting experience for Kabuto, Exocet and Remute.

Releasing the music video as a demo (downloadable here) was an afterthought because it felt really demo-like. To support the demo-like feel and acceptance there wasn’t any kinds of advertisements added to the demo itself.

Music

I, Remute, composed ‘Red Eyes’ with the tracker program Deflemask. Tracker programs like Deflemask do not record any music - instead they ‘tell’ the sound chip which notes to play and which kind of sound to generate in realtime - that saves a lot of data storage space which comes in handy when you only have 4 megabytes to work with. And so the music makes very sparse use of pre-recorded samples and instead almost fully focuses on the sonic capabilities of the Mega Drive’s YM2612 FM-based soundchip to generate music in realtime.

The only pre-recorded PCM sample in ‘Red Eyes’ is the vocoder vocal sample which I recorded previously with the help of an old-school hardware vocoder unit by Roland and then imported it into the Deflemask project. Everything else gets generated in realtime by the YM2612 - even the snappy drum sounds!

The limitation to 6 voices polyphony was quite challenging, but limitations like these can trigger great ideas though - I was forced to dive deep into editing of FM instrument parameters.

The FM-generated pad sounds make use of a slight vibrato effect to give them a ‘floating’ and slightly detuned feel. The pretty analog sounding lead bassline of ‘Red Eyes’ has a quite

uncommon sound for FM and proves that with some tweaking the YM2612 is even capable of classic ‘warm sounding’ synth sounds.

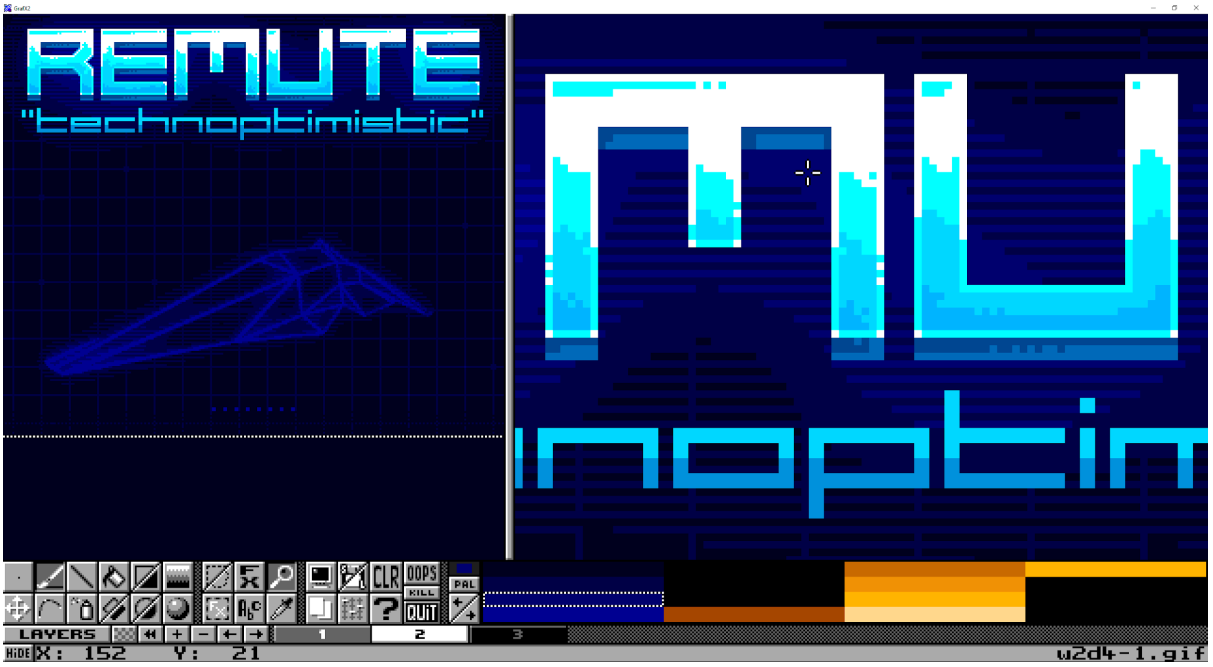
The tempo is 125 bpm as typical for Techno music.

Art

Exocet created both 2D and 3D graphics - videos and photos were taken by Remute.

2D graphics

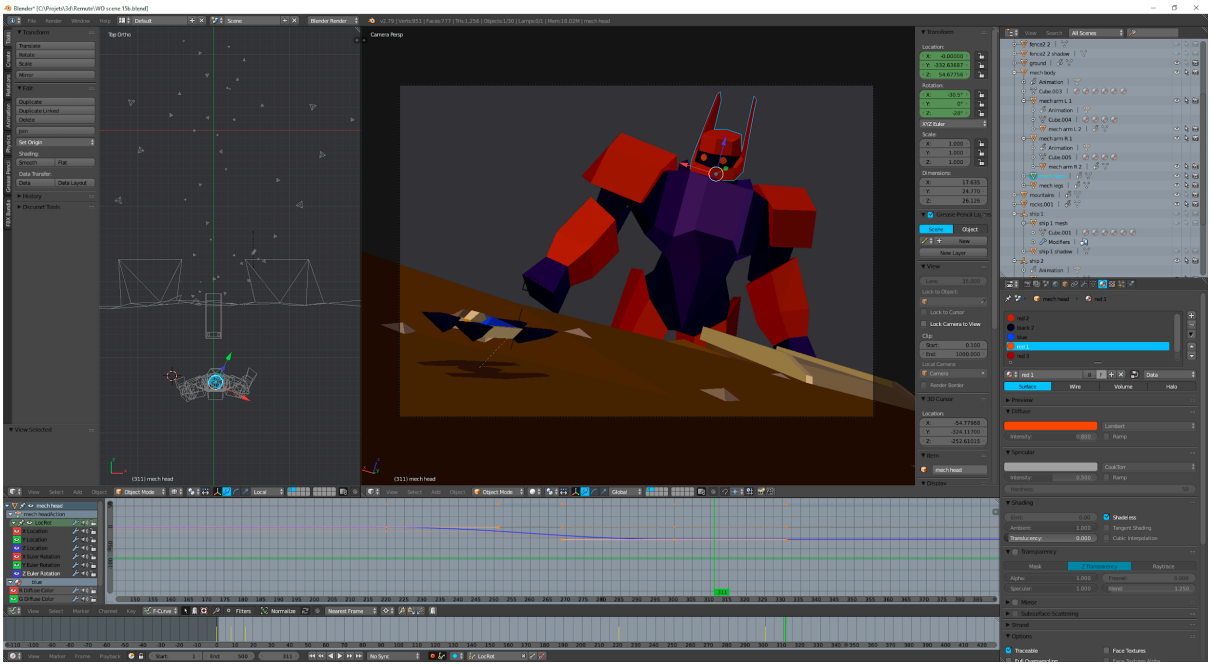
The creation of those were pretty straightforward, it was mostly a matter of keeping the console specs in mind: 320×224 pixel, 16 colors maximum per element, picked from a palette of 512 colors (8 levels per each RGB component). I (Exocet) used a paint program called Grafx2, which has a history of over 20 years but it still being updated nowadays and packs some great features targeted at older computers and consoles. The logo was pixeled by hand and uses only 8 colors. The track titles underneath use a free font from [04.jp.org](04.jp.org).



To generate the glow effect behind some of the graphics, like behind the logo on the main menu, I used Photoshop and then reduced the color count. Due to the limited palette of the Mega Drive, dithering was necessary but I didn't want to use the traditional algorithms offered by Photoshop (pattern, noise...) as they look either messy or too artificial. Instead I took advantage of one of the many features from the image manipulation software Imagemagick that lets you define custom dithering patterns.

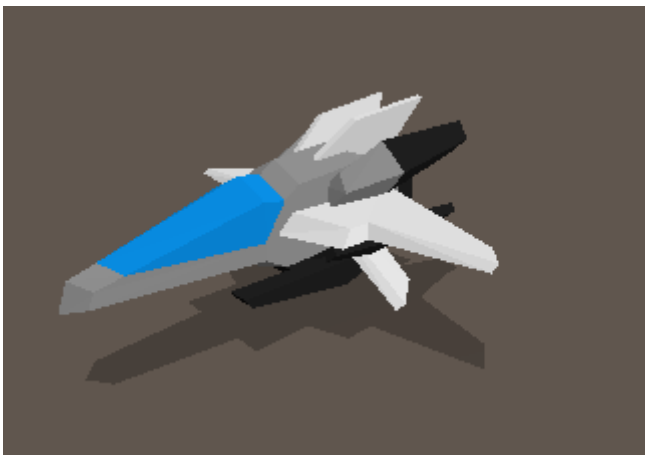
3D graphics

I used Blender to model, animate and render everything with flat shading, no lighting (self-illuminated materials only) and no anti-aliasing. The color values for the materials were picked in Grafx2 to make sure they matched the Mega Drive palette perfectly. The goal was to output a sequence of PNG at 50 FPS that matched exactly the final look it was supposed to have on the Mega Drive. Kabuto's renderer was then to decide on the achievable framerate depending on the complexity of the scenes.



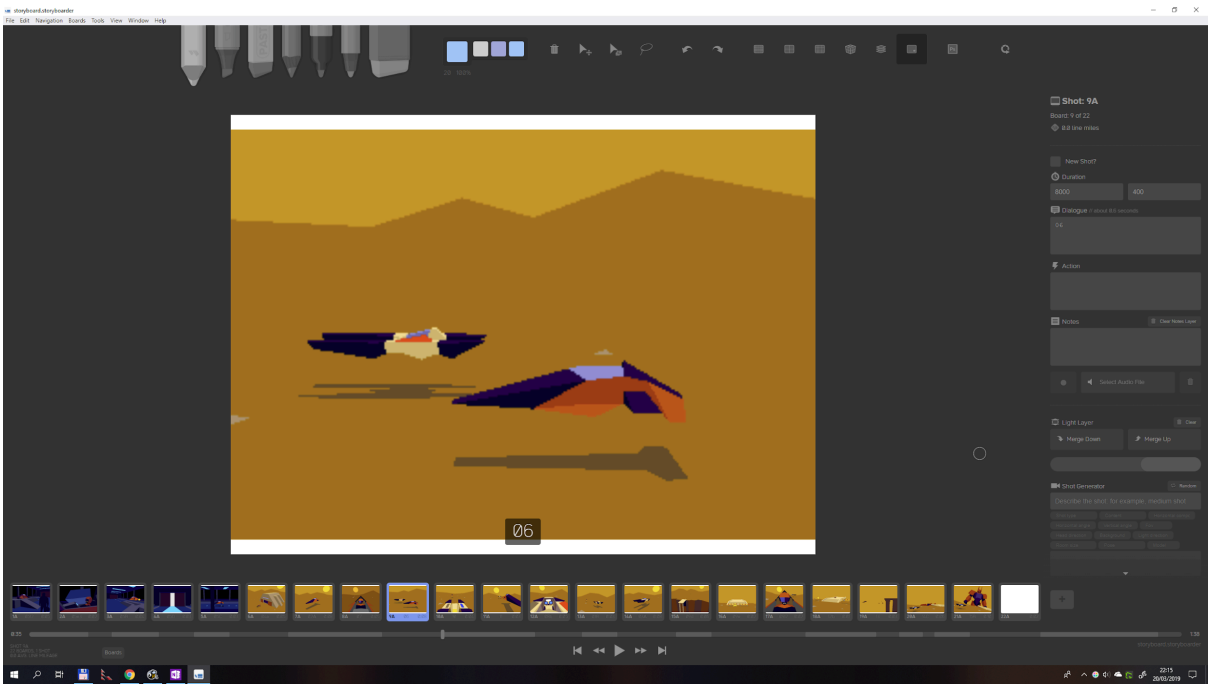
Setting a polycount budget was not so simple because of the way the renderer works. I decided to aim for scenes using less than 500 polygons in general, which seemed like a good compromise - pretty lightweight, but still enough to create relatively interesting environments. If the framerate happened to take too much of a hit, I could always have simplified them a bit further. We also experimented with both dynamic and static lighting but I found that due to the limited palette of the Mega Drive, static was a better option to ensure a consistent color scheme.

To validate the workflow we did an initial test with a fairly complex spaceship design. Everything looked good so we were all set to start working on the animation proper. I knew more or less what I wanted to achieve but many more ideas came up as I worked on the first few scenes, which was the part with the pursuit in the desert.



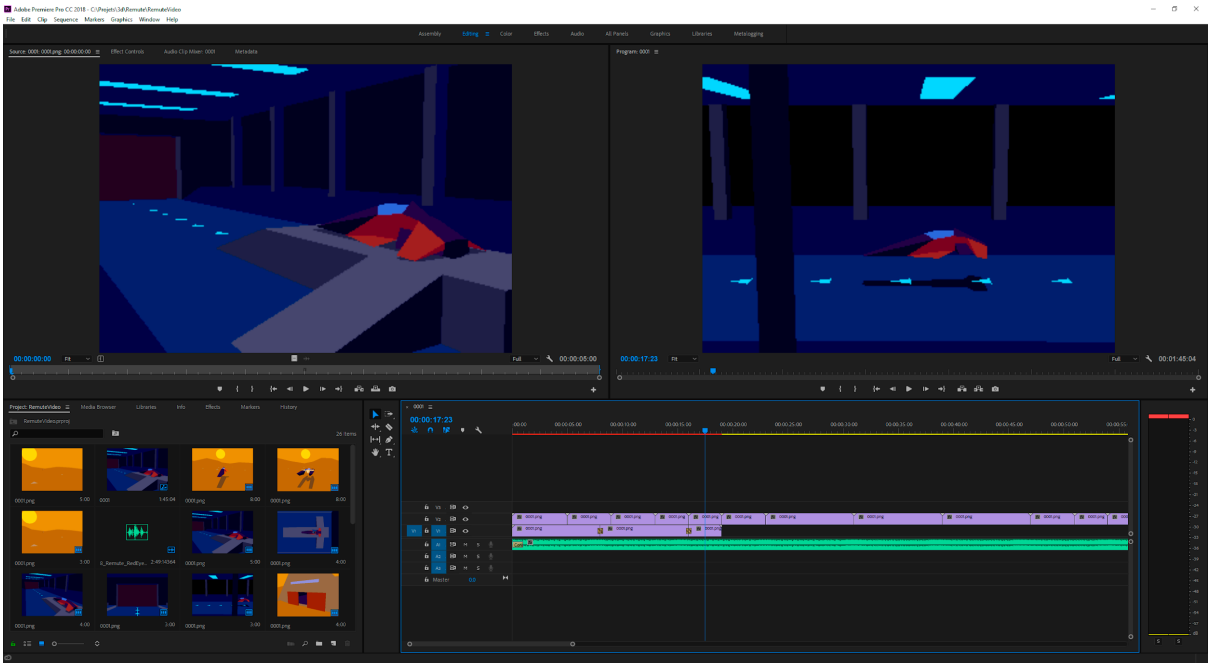
The good thing about using a fixed camera, as opposed to a game where the player is free to look around, is that it was easy to cheat to achieve the desired effect, while saving precious polygons or even more precious time. The most obvious cheat is the use of billboards in many places, simple 2D shapes aligned with the camera that gives the impression of something much more complex: the silhouette in the hangar, the mountains in the background when outside, etc. On a few of the scenes where the vehicles appear to be moving, it's actually just the environment which is scrolling underneath like a treadmill and it makes the whole scene setup a lot easier.

As the animation grew more complex, it dawned on me I'll need some sort of storyboard tool to keep track of the different scenes (initially 21 of them, almost each a separate Blender file) and the overall flow and duration. I went with a software called Storyboarder which is free and did the job perfectly.



The 3D part had to last exactly 85 seconds. After animating all my scenes I was more than 10 seconds above, therefore I had to cut or shorten the less interesting scenes. Something

that saved me a lot of time to do that was to realize you could import and export PNG sequences in Adobe Premiere Pro, where cutting and re-arranging sequences can be done in a very visual way. That made the last few adjustments a breeze.



Code

Music conversion

To prepare songs for playback on the Mega Drive, Deflemask must be instructed to convert the songs to an intermediate format (VGM); this format can then be fed into Kabuto's converter which generates binary files for playback on hardware using his own player.

Songs as composed in Deflemask use a very high-level format that's optimised to be readable by humans. Musicians place notes on a chart, assign instruments to channels, define instruments and add predefined effects (such as note slides).

The generated VGM files are however optimised for playback - they contain raw register writes necessary to make the YM2612 sound chip output the desired sound so one only needs to emulate the sound hardware and not Deflemask's behaviour - all sound effects are already contained in those register writes. This also means that simple Deflemask instructions (e.g. 2-byte note slides) are often expanded to a large number of register writes (for that note slide e.g. 10 frequency updates of 4 bytes each = 40 bytes total). Deflemask also truncates PCM samples to 8 bits of resolution and then (!) applies volume and speed adjustments using rather crude algorithms for VGM output.

Kabuto's converter converts most VGM register writes to the corresponding instructions in his player. Since that player can only output samples at a fixed rate of roughly 26 kHz, PCM samples from the VGM file (using a bunch of common sample rates such as 16 or 32 kHz) are resampled. Samples are - if available - read from the original Deflemask file since resampling an 8-bit sample and then truncating it to 8 bits again introduces additional noise and this way volume and pitch changes can also be re-done using better quality algorithms and before truncating to 8 bits. The 8-bit conversion is only done as the last step.

However, doing pitch changes properly had an unintended side effect. Remute chose to speed up a hi-hat sample by a factor of 4 which was turned into a snappy white-noise drum by Deflemask's algorithm whereas Kabuto's converter shifted it up into ultrasonics, making it hardly audible. So in the end a compromise was needed to stay closer to Deflemask's sound.

Audio player

The player is nearly the same as the one used in the Overdrive 2 demo with just a few additions and bugfixes. I (Kabuto) originally wrote this player because I was unhappy with existing players which used low sample rates, had lots of sample jitter and/or distorted samples.

The solution was a novel player that solved all these issues at once, (like many other players) running solely on the secondary CPU (Z80) so the main CPU doesn't need to care about audio playback.

As mentioned above the player outputs samples at a rate of roughly 26 kHz. Why 26 kHz? The sound chip has a fixed clock rate of (roughly) 53 kHz at which it can output new samples and its timer can only be made to run at integer fractions of that so it only really makes sense to output samples at such an integer fraction, too. Possible sample rates are e.g. 53, 26, 18, 13, 11, 9, ... kHz. Going for a higher sample rate gives better sound quality but implementing a player gets more difficult, samples need more ROM space and loading samples from ROM steals more main CPU time - even though it's a Z80-only player, every ROM access will briefly pause the main CPU. I went for 26 kHz because it provides a well-noticeable quality gain compared to lower rates and implementation complexity was still manageable - and it costs about 4% of main CPU performance which felt alright.

Still, getting 26 kHz sample rate with near-zero sample jitter was very difficult (and that's probably also why to the best of my knowledge no one implemented such a player before). There are neither interrupts nor sample hardware acceleration - the sound code needs to actively wait for the right moment in time for outputting a fresh sample, typically by repeatedly polling a timer. Doing this the usual way (wait for timer, acknowledge timer, write sample, loop) leaves hardly any CPU time for other tasks. The solution was to only check the timer every now and then but this means having to count CPU cycles. Since that's very cumbersome, especially with code full of branches, I wrote a special assembler that prints

warnings with a detailed log when there's a path of execution that could violate timing constraints. This simplified the task a lot and made it manageable.

The root cause of sample distortion in most players is DMA - while the video chip is loading fresh graphics data into its RAM the Z80 cannot load fresh data from ROM - if it tries to it's forcibly paused until the DMA is over. (Even worse, there's a very small chance of corrupting the main CPU's RAM when a Z80 ROM access coincides with a DMA.) To address this, the player buffers samples and instructions in Z80 RAM (8 KB, shared between buffers and player code) - and it requires the main CPU (68k) to use a handshake protocol to tell the secondary CPU (Z80) whether or not it's safe to fetch fresh sample and instrument data from ROM.

Unfortunately the sound code can't be told to "stop all ROM accesses now" - while it's prefetching data from ROM it cannot be interrupted so it needs to be told way in advance which in turn is difficult to do on the Mega Drive. In most cases DMAs are only done right after the start of vertical blank, for this case there's a 3rd state of the ROM access flag, telling the Z80 to "do ROM accesses as long as they don't run into the start of the next VBlank". To ensure this the Z80 reads the raster line counter. Unfortunately reading the raster line counter also conflicts with DMAs, so soon after vertical blank starts and before doing DMAs the 68k must still tell the Z80 to stop ROM accesses, but since the Z80 won't be in the middle of a prefetch it will cease to access ROM and the raster line counter nearly instantly - not exactly instantly but faster than the 68k could set up and start a DMA.

When the Z80 reads data from ROM the bus arbiter needs to reserve the 16-bit bus first before the ROM access can happen. This imposes a variable delay which on average takes 3 to 3.3 Z80 cycles depending on the particular Mega Drive model. The music code expects those delays to take that many cycles on average - however, many emulators don't emulate that delay at all; this causes the player to play music too fast and distort samples. Other emulators have trouble emulating the timer frequency of 26 kHz and play music too slowly (and distort samples as well). Both issues are checked for at startup by comparing music playback speed with the video signal and a warning screen is shown if a difference is detected (I added a huge error margin that's still small enough to detect all affected emulators tested.)

One nasty bug (also present in Overdrive 2) caused music player crashes on some Mega Drive 2 models (including Nomads). The root cause was reading a status register and not masking out undefined bits which turned out to sometimes not be 0 on those models (though still on most affected models eventually settling down to 0 after the chip has warmed up). Once caught this was easy to fix, but catching it took a while since of course no emulator emulated it. One more reason to always test on real hardware ;-)

Vector graphics

The vector graphics renderer is based on the renderer from Overdrive 2 which was written by Jix which itself was a port of Jix' renderer from the Wonderswan demo "Finally". I

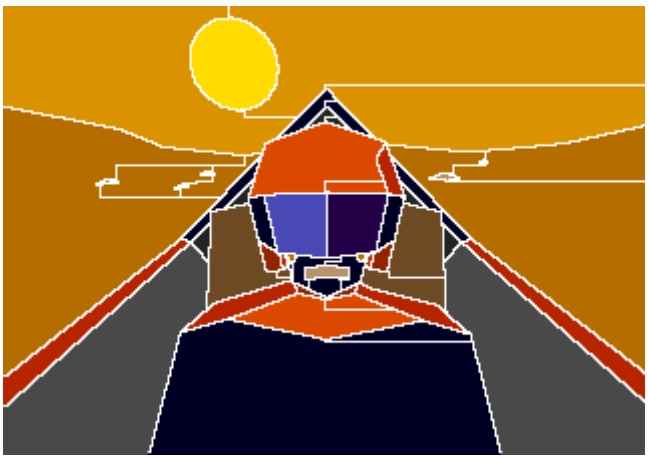
(Kabuto) did a complete rewrite which just shares the same principle - splitting pre-rendered frames into row-convex polygons and encoding those - just way more efficiently both ROM-space-wise and speed-wise. On top of that a few scenes use additional tricks to meet my goal of at least 16.5 frames per second - where render speed allowed for it I went for 25 FPS. Playback looks smoother on PAL because both frame rates evenly divide 50 FPS and thus each frame is shown for the same duration for any given frame rate whereas that's not the case for NTSC and duration each frame is shown jitters between 3 and 4 (16.5 FPS) resp. 2 and 3 (25 FPS). There's no easy way around that without re-rendering the video again for 60 FPS and effectively storing it twice, consuming twice as much space.

Have a look at this screen from the video:

The encoder dissects it into solid-coloured row-convex shapes - row-convex means that no horizontal line may intersect the shape more than once.

Also, shapes must have a strict left-to-right order, it must be possible to order them in such a way that the left edge of any shape is only adjacent to the right edge of shapes earlier in the list. This is important because it could be violated in theory as shapes are allowed to be disconnected - e.g. thin polygons tend to generate loose groups of pixels but still technically belong to the same shape. Imagine a 2x2 checkerboard - each colour could be encoded as a single shape (only a single intersection with any given horizontal line) but if both colours are encoded as a single shape each then there wouldn't be a strict order anymore.

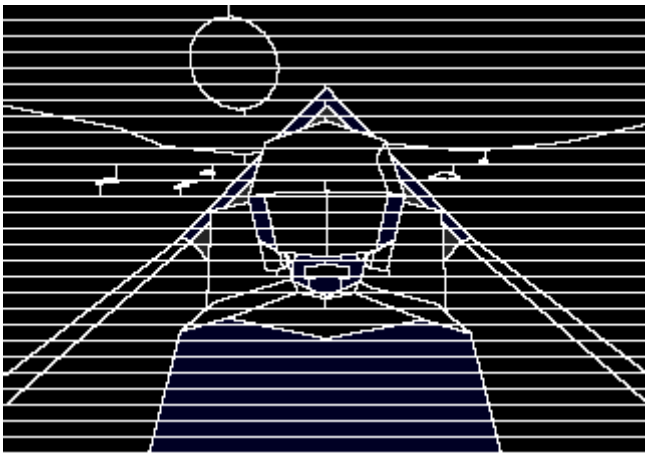
The outcome of splitting the image above into shapes looks like this:



Shapes that would intersect a horizontal line more than once such as the sky around the sun have been split up into multiple shapes to fulfil those conditions - the renderer prefers to split along tile boundaries since that allows for faster rendering.

These shapes are then encoded. Since the left border of each shape is equal to the combined rightmost border of all shapes that were already encoded earlier, it doesn't need to be encoded and we only encode the right edge - and of course the colour. The encoding scheme has a large set of encodings, e.g. for long slopes, and also short common sequences.

The decoder doesn't directly draw those shapes - it's a deferred renderer that first splits up all shapes across bins with a height of 8 pixels each. Each such part of a shape is stored in the corresponding bin - all shape segments in all bins look like this:



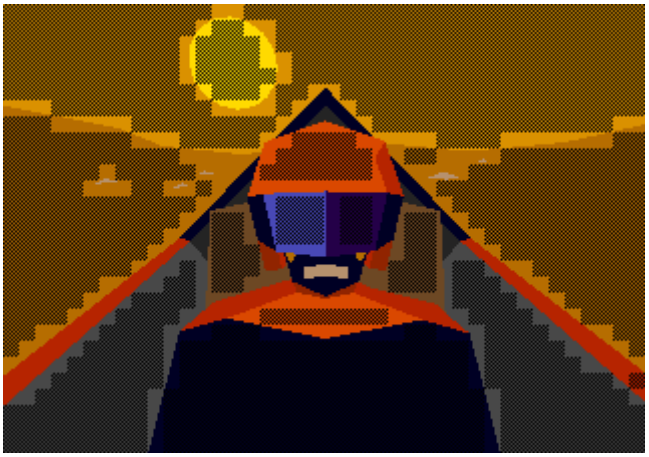
When done decoding all shapes, the renderer renders one such bin after another, generating tiles. Tiles that are completely contained within a shape are solid and the renderer will use a reference to one of 16 predefined solid tiles (one for each possible colour) instead - that's the main reason why the renderer can render so fast.

All tiles that are covered by multiple shapes are allocated in a tile buffer and drawn onto, even if all those shapes have the same colour - that's why it's beneficial to align boundaries

between same-colour shapes to tile boundaries to avoid unnecessary allocation of such solid multi-shape tiles.

Using an individual tile for each tile slot would take 35 KB and take almost 5 NTSC frames' vertical blanks for transfer and would also need too much VRAM space for double buffering. By using solid tiles we can get along with less than 14 KB of tiles per frame.

The final image with solid tiles marked looks like this:

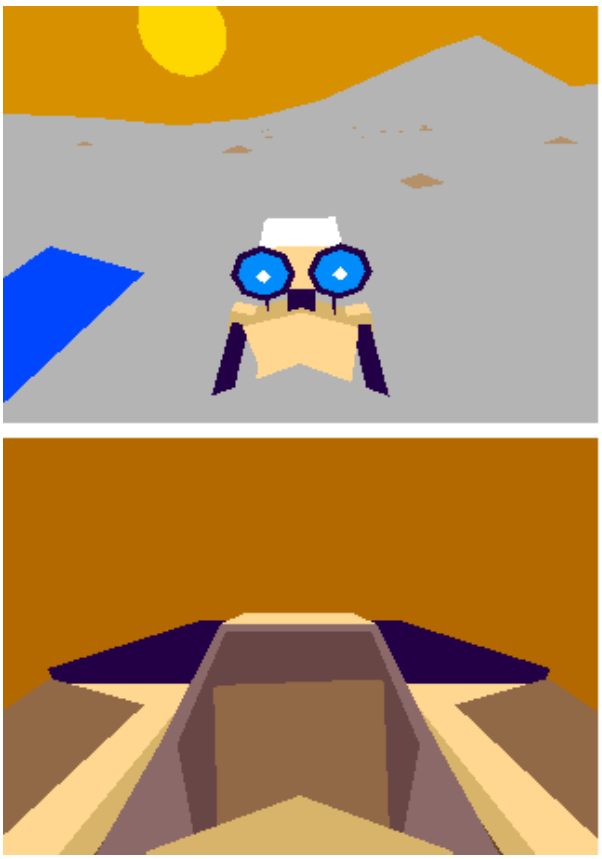


When done, we have a tile map and a tile buffer, both of which need to be transferred to graphics memory.

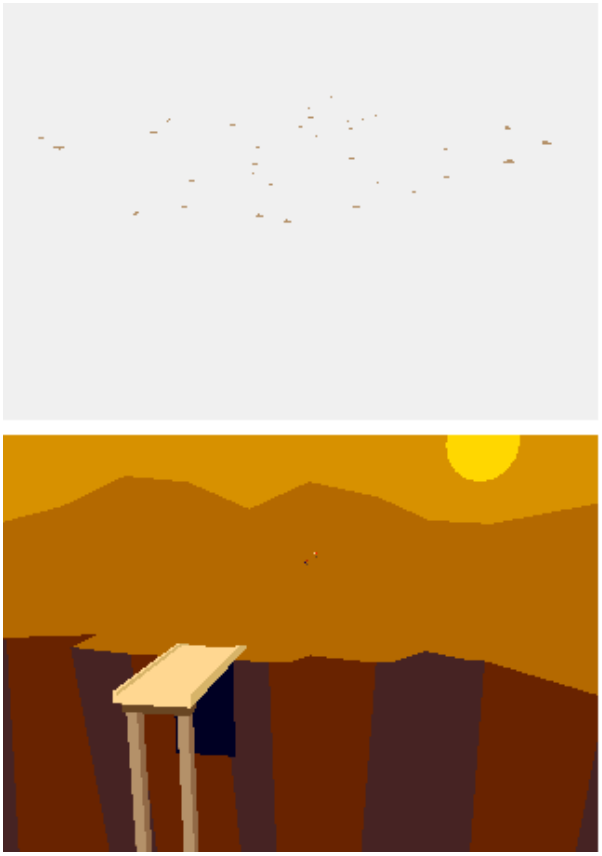
Since on NTSC such a transfer can take multiple frames, we use 2 buffers both in RAM and VRAM (Video RAM). Not using 2 VRAM buffers would lead to glitches while a frame is only partially transferred; not using 2 RAM buffers would stall the decoder since it would have to wait with rendering the next frame until the current frame is fully transferred.

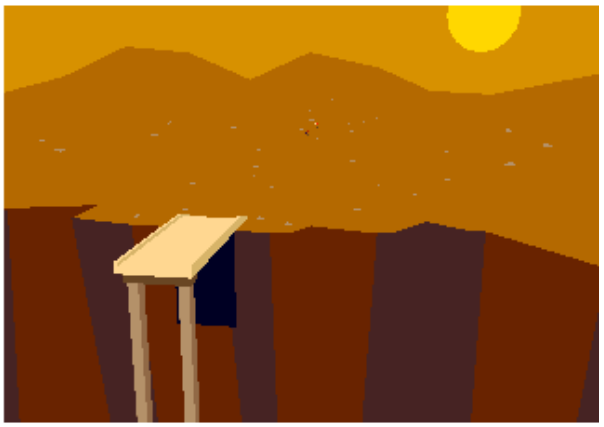
Having 2 buffers in both RAM and VRAM has the nice side effect that the renderer is typically running a bit ahead of what's currently on screen - and when a few frames need more CPU time than available it won't instantly lag.

A few scenes were still too slow after all (or used more than 14 KB for custom tiles). Most of these scenes had some parts that didn't move at all - e.g. the cockpit pilot view had the camera fixed to the ship so only the pilot and the environment moved while the ship was static. For those scenes I created an underlay image containing all static elements and removed those static elements from the frames to be rendered:

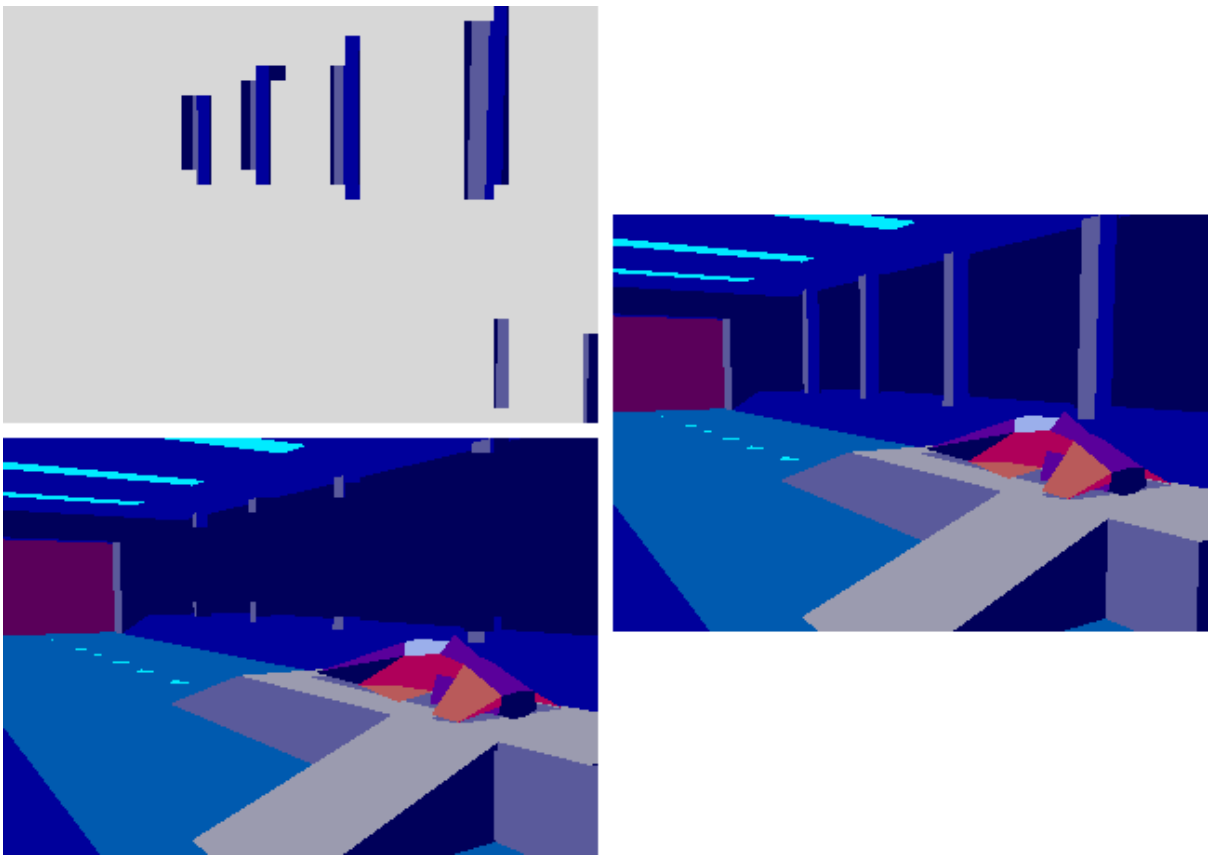


2 scenes had no static elements but a large number of small rocks. For those scenes I created a dictionary of 256 rock tiles which I removed from those images and re-added as sprites during playback:





And some of the hangar scenes were still too complex - I added a special renderer for vertical pillars which used a set of static predefined tiles and replaced pillars with solid tiles for the vector renderer. Even though my pillar detection algorithm wasn't very accurate it still shaved off enough CPU time for meeting the goal:



Still, I needed to load those auxiliary graphics dynamically into VRAM. Pillar and rock tiles are preloaded since they fit but all the backgrounds need to be loaded dynamically. I split each of those into 4 transfers (to avoid exceeding available VBlank time), and to help not exceeding NTSC VBlank time I added a register that counts how many tiles can still be transferred without exceeding it. Some tests and maths were needed to adjust those counters whenever an auxiliary operation is done (transferring background graphics or decoding a sprite list).

FMV

I (Kabuto) added FMV because there were about 1.2 MB of ROM space left after including all music tracks and the vector graphics animation - up to this point the first half of the music video only consisted of static images (with distortion effects similar to those in the final video).

We had some video footage which Remute originally planned to use for the first half of the music video. After deshaking this video it was pretty steady - and the low amount of motion left helped a lot with encoding.

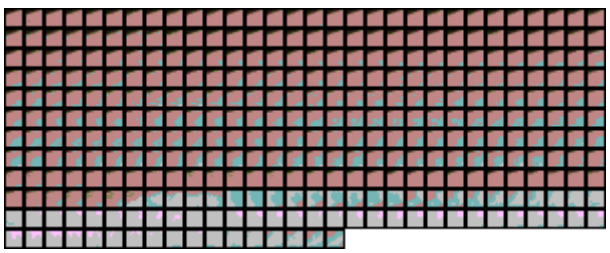
I planned to use more advanced compression but to get started I just implemented a very simple vector quantisation compression scheme - in the end it was good enough for this task and I was running out of time finishing everything so I didn't bother implementing a better scheme.

The compressor first converts the video to use a palette that respects the Mega Drive's colour limitations (16 colours total, only 8 available brightness levels per colour component (red, green, blue)):

Of course it tries to choose the palette so that converted images are as close to the original as possible. I chose to rate luma (brightness) much higher than chroma (hue / saturation) for a better and distinct overall look. I could have added dithering but time constraints forced me to focus on other things.

The palettized video is then sliced into tiles (8x8 pixel squares) and each resulting tile-sized video is compressed individually. I marked an example tile for which I'll explain the next steps:

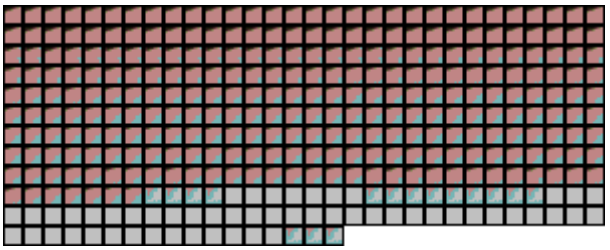
The marked tile's content across all frames (contrast enhanced to make differences easier to spot):



My algorithm then chooses a small set of tile instances across those that are similar enough to all tiles:

In this particular case it decided that 8 tile instances are enough to represent all tile instances. Depending on the variety of tile instances it can build a set consisting of 1 to 64 instances. The required similarity is controlled by a global parameter - I adjusted it until videos fit well into the remaining 1.2 MB of ROM space.

Once a set is chosen, each original tile instance is replaced with one from that set:



As you can probably see it's similar to the original tile instance list but differences do exist.

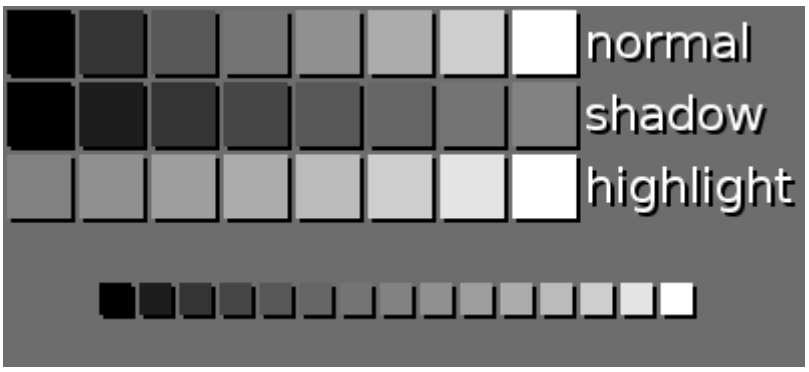
This procedure is done for each tile. When done, the encoder encodes the reduced sets of tile instances and for each frame it encodes which tiles changed compared to the previous frame. Tiles tend to not change from one frame to the next in the majority of cases, thus omitting them saves ROM space (and speeds up updating as well). The image right above shows how little changes there are (at least for tiles that end up having a low number of instances).

The video is decoded at 30 frames per second. Since only tiles that changed are copied to video RAM this frame rate was easy to hold.

Greyscale picture

This picture uses a little trick for showing greyscale pictures of high quality. This trick has already been used for a hidden part in Overdrive 2 but hardly anyone has seen it so Kabuto chose to re-use it.

Normally a Mega Drive greyscale palette would only have 8 luminance levels. However, the hardware offers the unique S/H feature which (roughly speaking) adds 2 special sprite colours - shadow (50% opacity black) and highlight (50% opacity white). Putting these on top of the 8-step greyscale palette gives us 15 luminance levels total (since black with 50% opacity white and white with 50% opacity black yield the same level of grey).



To go even further I chose to deviate from achromatic colours. E.g. by mixing 2 adjacent colours by taking one colour's green component and the other colour's red and blue components we get 2 colours that are about in the middle regarding their brightness, they're just slightly green-ish / purple-ish. To get rid of the unwanted chroma I alternate between both hues on both alternate rows and frames. This adds another luminance level between each pair of 2, giving us a total of 29 levels.

Since reducing an image to 29 levels of grey would still result in visible banding I added dithering (though in this case you really need to look closely and zoom the image to spot it):

On my CRT TV set there was no residual chroma left and no flickering either - the image looks totally stable. On emulators (and when playing the video on YouTube) you might or might not see a small amount of flickering depending on how well the emulator / video player blends frames to match the display's frame rate.