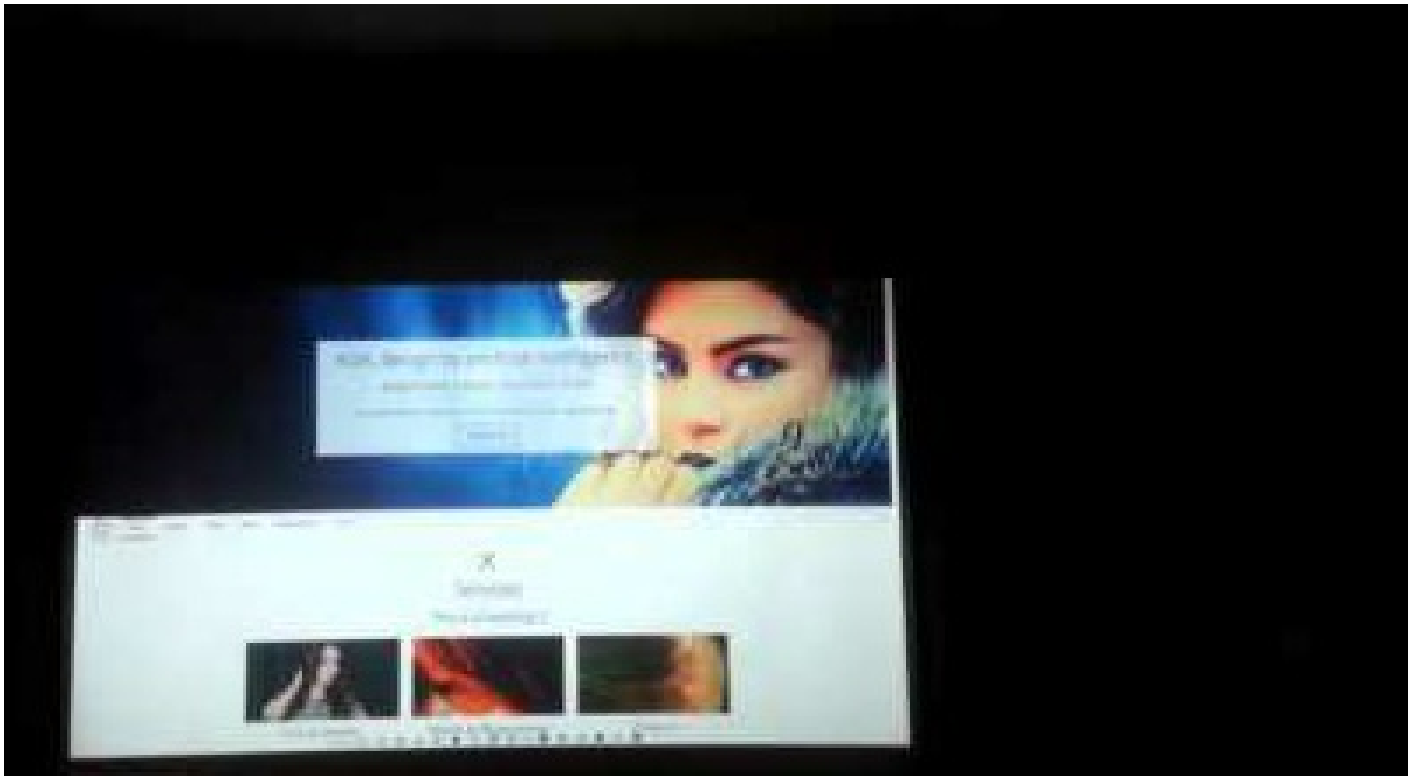
developer blog

developer blog

Blog posts by the 1stbase team.

a conversation with ada



a conversation with ada

We wondered what would happen if we would give ADA ears and a voice. It was... awkward, fun, extraordinary.

the c64 app: the making of

the c64 app: the making of

The venerable 1982 Commodore 64.

Is our Commodore 64 app the most awesome demonstration of the C64's capabilities ever?

Not by a long shot. For the truly great examples of what the C64 is capable of with some very hard work and imagination, have a look at these amazing pieces of audio-visual art. We got nothing on Fairlight, Noice, Offence, Crest, Booze Design, or Onslaught (and many more).

Now, this is going to get technical, so if that's not your thing, feel free to skip this post.

Of course, making the greatest app ever was not the point of this exercise. And to be fair, the above amazing examples invariably borrow, cheat, steal (for example by doing a lot of pre-processing off-line) to bring you those amazing effects and tend to use a lot of multi-loading to constantly load new code and data into the C64's paltry (by our modern standards) 64KB of memory.

We on the other hand, had some limitations and requirements of our own;

- We needed to be able to be compile in the cloud
- We needed to be able to compile app and resources quick (ideally < 10 seconds), precluding any brute-force number crunching.
- The program needed to be a self-contained app that fits in the 64Kb; no multi-load, so that we could push out (e.g. play audio) directly from the Internet.
- The app needed to play nice with emulators; no interlacing graphics modes.
- In the spirit of 1stbase, we wanted to maximise coverage and thus distribution formats (hence audio, see above)
- We wanted it to still capitalise on the C64's unique abilities
- Showcase at least some "modern" (post-1989) programming techniques.
- We needed to be able to mix graphics and text.
- In a team of just 2 people, for obvious reasons I couldn't spend too much time on this.

development tools

First order of business was procuring the development tools. I quickly settled on CC65, a C-compiler tool-chain. To expedite the process, I decided to use some C for the non-performance critical parts.

Availability of the source code let me compile and use the tool-chain on the 1stbase servers.

I cut my teeth on an MSX 1 (Z80-based) computer as a young lad, so coding in 6502 assembly was a bit different (as well as refreshing!). Fortunately there is heaps of documentation available on-line, and getting to grips with the Commodore 64 intricacies didn't take a lot of time. Really, this fantastic talk by Michael Steil is all you need to know.

creating a graphics converter

By far the most important piece is the graphics converter, which is responsible for crunching down 1stbase content into their C64 equivalent representations.

The requirement that we be able to mix graphics and text meant that I had to pick the C64's "hi-res" mode.

This mode offers 320x200 pixel with one foreground color and one background color for every 8x8 blocks.

The full Commodore 64 character set.

The screen buffer in this mode requires (pixels on/off 320x200 / 8) + (colors 320x200 / (8*8)) = 9000 bytes.

Memory is at a premium and this choice means that we have blown 9Kb of our precious memory straight off the bat. However, there are perks to this mode as well. This mode would provide enough resolution to render text in a legible way and had another benefit; I would be able to use the Character ROM's font without too much trouble - we'll get back to that important bit later.

This also meant that if I wanted to show some images from the home page's image slider (cropped and scaled neatly to 320x200 by ADA's routines), they would come in at 9000 bytes each - without compression. About 4 of these would blow all the memory I had to work with. And I wanted a lot more graphics than that!

Some quick compression tests on some 1stbase data sets only got me so far. I wanted something that would allow me to (if necessary) trade off size and quality. This way I could - if absolutely necessary - drop the quality to fit more in if I was about to run out of memory.

I then realised that if I could use one of the C64's ROMs as a codebook for a compression algorithm, I could temporarily dispense with such a codebook if I needed more memory for some task. That 4Kb could definitely come in handy.

The nice thing is that, if we take the C64 character ROM (which always takes up space anyway as we require text rendering and thus character bitmaps), then if we temporarily need an extra 4Kb RAM, we can use the character RAM equivalent (the copy of the ROM that resides in RAM) and reconstitute it later by temporarily swapping in the character ROM.

The character ROM is a very suitable codebook, as there are many shapes and correlated patterns available.

So, now that we have a "free" code book we can start matching (collections of) pixels in the image against the codebook to find the best match in the codebook. All we have to do then is record the offset of the position of our match and “replay” that position on the commodore 64; does a 8x8 pixel block roughly look like an "A"? Then record an "A"; that's another 8x8 pixels taken care of. And so on, and so forth.

The first version of the converter effectively was a PETSCII (Commodore's version of ASCII) converter. It would convert a 320x200 into coloured letters and numbers, roughly resembling the input picture.

An early version pure-PETSCII converter.

However, there was nothing stopping me from reducing the block size to, say, match and record every 8x4 pixel blocks instead of 8x8 pixel blocks. This would give me a better image that was more recognisable, though at the cost of more storage space. I had now found my means of trading off quality vs size.

With that out of the way, I should mention that matching to the color palette of the C64 is equally tricky. Unlike, say, the ZX Spectrum from the same era, the C64's colors are somewhat arbitrary and don't map perfectly to any one red/green/blue combination; much depends on the television set and color standard that the C64 is connected to. This is the reason why there are many different colour palettes settings floating around for C64 emulators, created by enthusiasts that sought to mimic and relive "their" colour rendition as they remembered it.

I myself was impressed with the very thorough analysis by Philip 'Pepto' Timmermann, which leads to a particular palette that should be close to "calibrated".

speed

As mentioned above, most “modern” C64 demos use copious amounts of off-line number crunching to come up with amazing effects. Unfortunately, given that we build our C64 in the cloud in a Just-In-Time manner, we don't have that luxury.

Matching 8000 pixels with 15*16 different colour combinations for each 8x8 took upwards of 3 minutes on my 3rd gen i5 desktop. This is fine if you're doing this once-off, but not feasible if this is supposed to run in the cloud on a (much lower specced) Amazon Web Services instance.

A fair bit (too much :) time went into optimising the converter to be more clever with which blocks to test against.

Blocky? Sure. But at just over 1KB I'll take it!

dithering

There is a neat trick that gave me "free" dithering - allowing to represent (if you squint) more colours than are really available by toggling adjacent pixels.

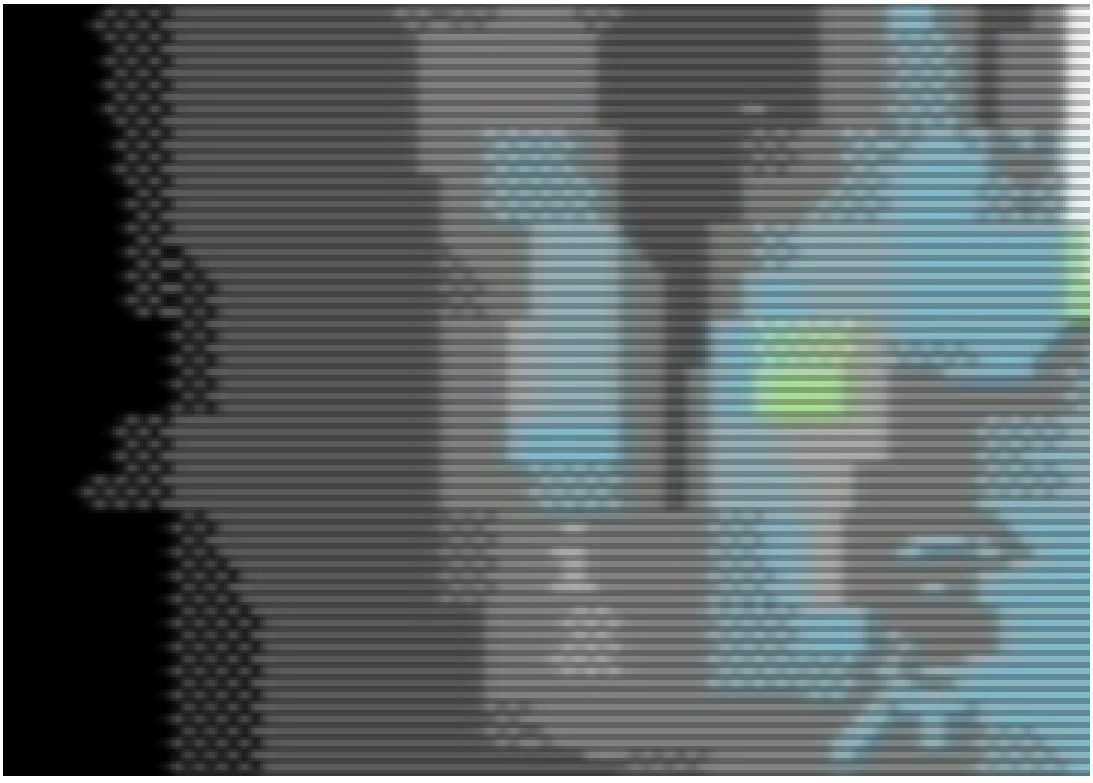
On the converter side, I blurred the converter's version of the codebook a little – just enough to make adjacent pixels blend into a slightly fuzzy grayscale instead of being perfectly on/off-black/white.

This effectively made the converter a bit far-sighted, not being able to distinguish perfectly which pixels were on and off in the codebook. Given that we've intentionally introduced dithering patterned versions of the first 128 characters into the codebook, this little trick causes the converter to pick the dithered patterns when coming across areas in the image that are nicely in between the on (foreground colour under evaluation) and off (background colour under evaluation). The result is that we get 'free dithering' while doing the conversion– we got some more pseudo colours to work with, which helps somewhat with the 2 color-per-8x8-block limitation of the hi-res mode.

On the C64 side, whenever I make a copy of the Character ROM, I modify the copy in such a way that the upper 128 characters (of 256 total) are exact copies of the lower 127 characters, except that they have a dithering pattern (on/off) over them. The code to do this is pretty quick, so not a problem whenever I need to get a new copy of the character ROM (for example when I temporarily needed the 4Kb that its copy occupies in RAM).

Final compression of the codebook conversion data stream is compressed with LZ4; a super light-weight, extremely fast compression and decompression algorithm (the 6502 decompression one by Peter Ferrie that I used was only 143 bytes).

qr codes



Dithering, combined with the slight blurring effect of a composite connection to a CRT can help create the illusion of more colours.

A separate converter was written for the QR codes. Given that these are just 2 tone images (on/off), they are easy to encode into their equivalent bits which render neatly on the "hi-res" screen of the C64 as-is.

raster interrupts; hacking the vic ii

The Commodore 64's screen is actually wider and higher than 320x200. At the top, bottom, left and right are coloured borders.

By synchronising with the raster beam which builds up the picture, it is possible to give the borders different colours for some horizontal lines by changing them before the scan line starts.

When styling your content, ADA draws heavily on your presence's colours, logo and background graphic. So for the C64 we're using the background graphic to pick the horizontal bar colours. The result is colours in the borders that are reminiscent of your background graphic, even though strictly speaking we can't put any graphics (except sprites - with yet another hack) there.

True hacking of the VIC-II video chip involves advanced techniques like "Flexible Line Distance". By tricking the VIC-II through some register twiddling into thinking it is not time to start rendering, the start of the screen can be "moved down" at will. Timing is critical – you only have a few CPU cycles per raster line and cycle counting is a must to avoid jitter and glitches.

The FLD hack, for example, allows for smooth scrolling of full-screen images. This allowed us to scroll the images to scroll off-screen, stage the image (decompress, add text) and scroll the screen back in without apparent glitches and without the need for a double buffer.

A bit of additional trickery allowed us to scroll the border “graphic” along with the main screen, so that it appears as if the borders are "attached" to the image.

In addition to tel: and mailto: scheme QR codes, A SID - based DTMF dialer is also included.

using sid to generate dtmf tones

Finally, I wanted to see if I could create a working DTMF tone dialer for the C64 to dial the phone number associated with a 1stbase presence. This would work by holding up a PSTN landline phone receiver to the TV speaker while the C64 sends out the tones. This should in turn dial the number.

Generating a DTMF tone is really quite simple; all you have to do is mix two tones of the right frequency.

The problem was, however, that as amazing as the SID sound chip was for its time, its 4 oscillator modes (triangle, saw, pulse and noise) do not provide for a sine wave - the required pulse type for generating a DTMF tone.

A triangle wave is the closest the SID's oscillators can generate to a pure sine wave. Fortunately, the SID also comes equipped with 3 types of filters. By using a low-pass filter at its highest cut-off frequency, we were able to closely approximate a pure sine wave.

All we have to do then is mix two of those filtered triangle waves at the right frequency to generate DTMF tones based on the digits of the phone number.

distribution

1stbase being 1stbase, we obviously take the distribution quite seriously - we want as many eyeballs on your content as possible.

So we use WAV-PRG and UberCassette (which needed a little fix to work on our 64-bit Amazon Linux instances) to generate .TAP files and .WAV (44Khz 8-bit) audio from the .PRG file that the compiler spits out.

We wrap all that in an installer page which is, of course, also styled by ADA.

the result

The result is an application that shows a fair bit of content, styling elements, images and colours from your 1stbase presence.

And the great thing is, by optimising hard from the start at 4x full-screen images from the home item, we have roughly 20Kb to spare.

Functionally it's not entirely complete, but with 20Kb to spare, we're not finished yet with our little breadbin...