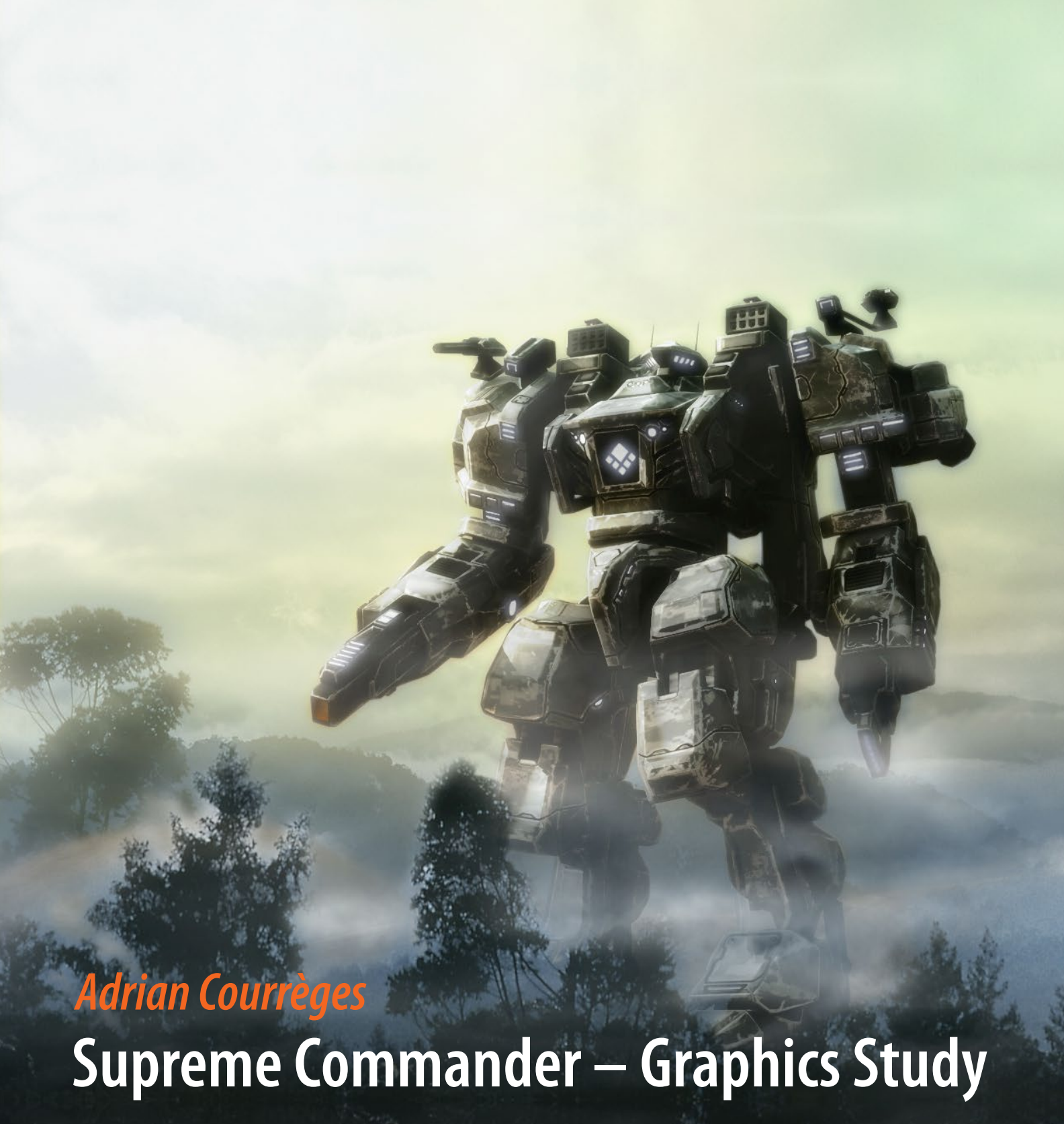


Adrian Courrèges

Supreme Commander – Graphics Study

HACKERMONTHLY Issue 64 September 2015

Curator
Lim Cheng Soon

Contributors
Adrian Courrèges
James Somers
Joe Savage
Rudis Muiznieks
Yan Zhu
Kyle Kingsbury

Proofreader
Emily Griffin

Illustrators
Daniel Rutherford
James H.

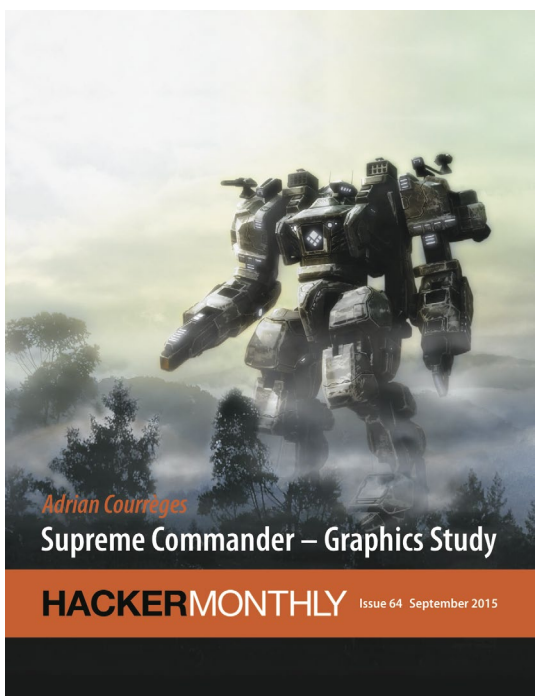
Printer
Blurb

HACKER MONTHLY is the print magazine version of Hacker News — *news.ycombinator.com*, a social news website wildly popular among programmers and startup founders. The submission guidelines state that content can be "anything that gratifies one's intellectual curiosity." Every month, we select from the top voted articles on Hacker News and print them in magazine format.
For more, visit *hackermonthly.com*

Advertising
ads@hackermonthly.com

Contact
contact@hackermonthly.com

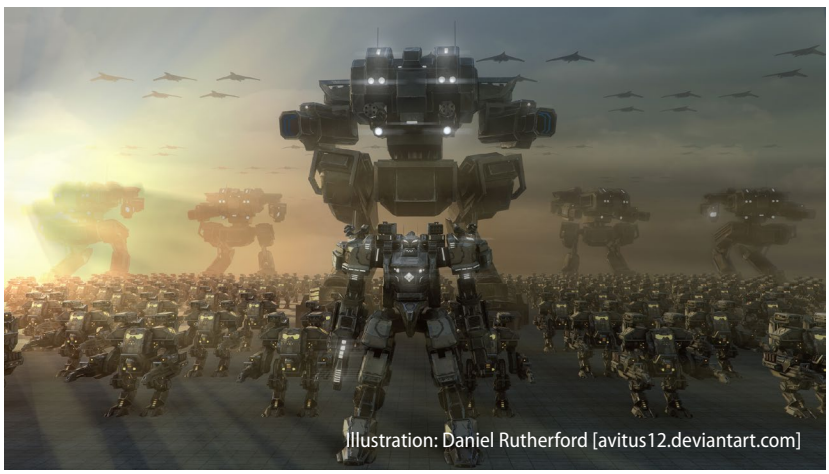
Published by
Netizens Media
46, Taylor Road,
11600 Penang,
Malaysia.



Cover Illustration: Daniel Rutherford [avitus12.deviantart.com]

Contents

FEATURES



SPECIAL

PROGRAMMING

For links to Hacker News dicussions, visit *hackermonthly.com/issue-64*

Illustration: Daniel Rutherford [avitus12.deviantart.com]

Supreme Commander - Graphics Study

By ADRIAN COURRÈGES

Total Annihilation has a special place in my heart since it was the very first RTS I played. It was with Command & Conquer and Starcraft one of the best RTS's released in the late '90s.

Ten years later, in 2007, its successor was released: Supreme Commander.

With Chris Taylor as the designer, Jonathan Mavor in charge of the engine programming, and Jeremy Soule as the music composer (some of the main figures behind the original Total Annihilation), the fans' expectations were very high.

Supreme Commander turned out to be highly praised by critics and players, with nice features like the "strategic zoom" or physically realistic ballistics.

So let's see how Moho, the engine powering SupCom, renders a frame of the game!

Since RenderDoc [hn.my/rdoc]doesn't support DirectX 9 games, reverse-engineering was done with the good old PIX. [hn.my/pix]

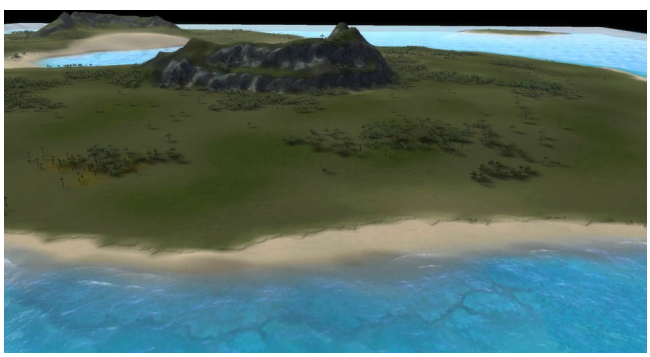
Terrain Structure

Before we dig into the frame rendering, it's important to first talk about how terrains are built in SupCom and which technique is used.

Here is an overview of "Finn's Revenge," a 1 versus 1 map.

On the left is a top-view of the entire map like it appears in-game on the mini-map.

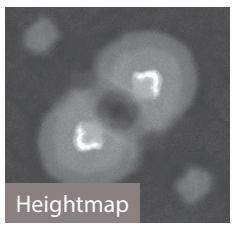
Below is the same map viewed from another angle:

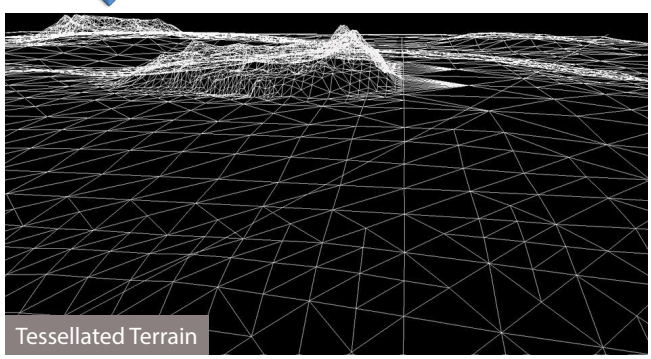


First, the geometry of the terrain is calculated from a heightmap. [hn.my/heightmap]

The heightmap describes the elevation of the terrain. A white color represents a high altitude and a dark one a low altitude.

For our map, a 513x513 single-channel image is used, it represents a terrain of 10x10 km in-game. SupCom supports much larger maps, up to 81x81 km.

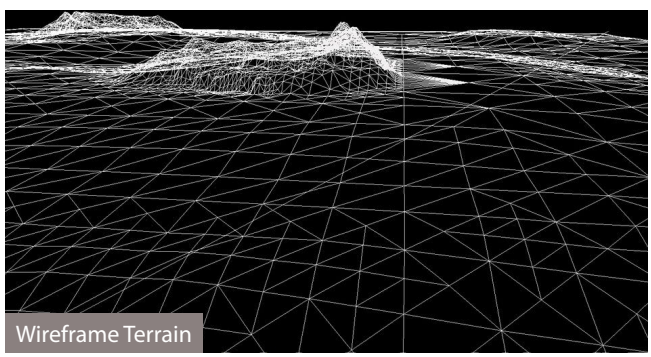


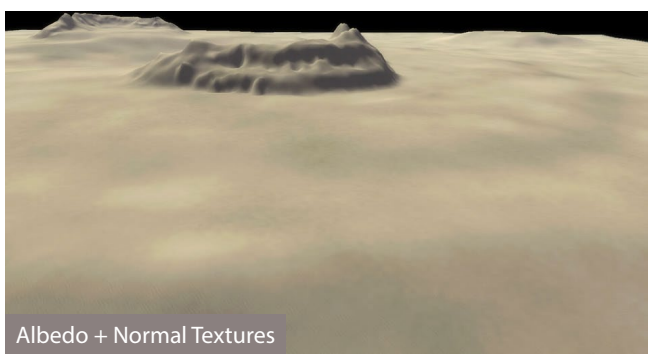


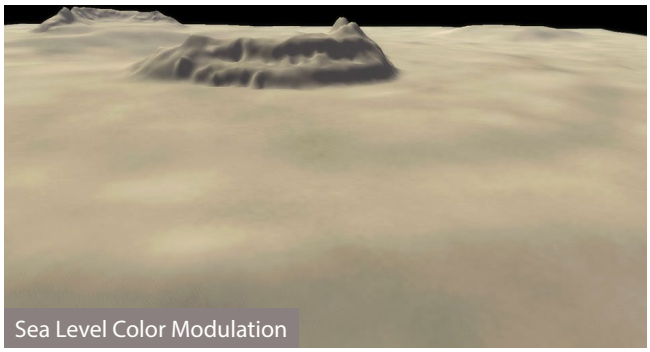
So we have a mesh which represents our terrain.

Then the game applies an albedo texture combined with a normal texture to cover all these polygons.

For each map, the sea level is also specified, so the game modulates the albedo color of the pixels under the sea surface to give them a blue tint.







Sea Level Color Modulation

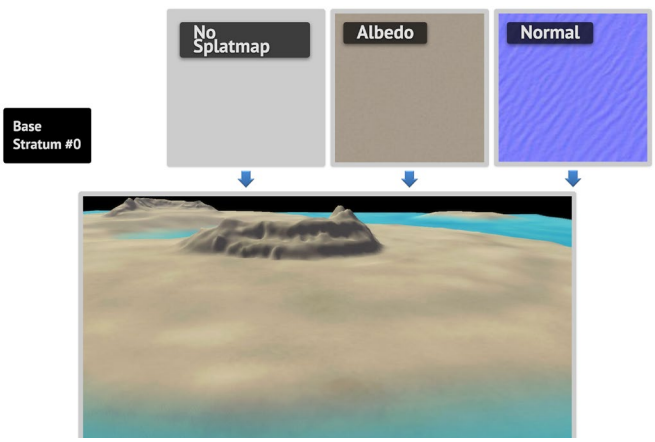
Okay, so having altitude-based texturing is nice, but it gets limiting quite quickly.

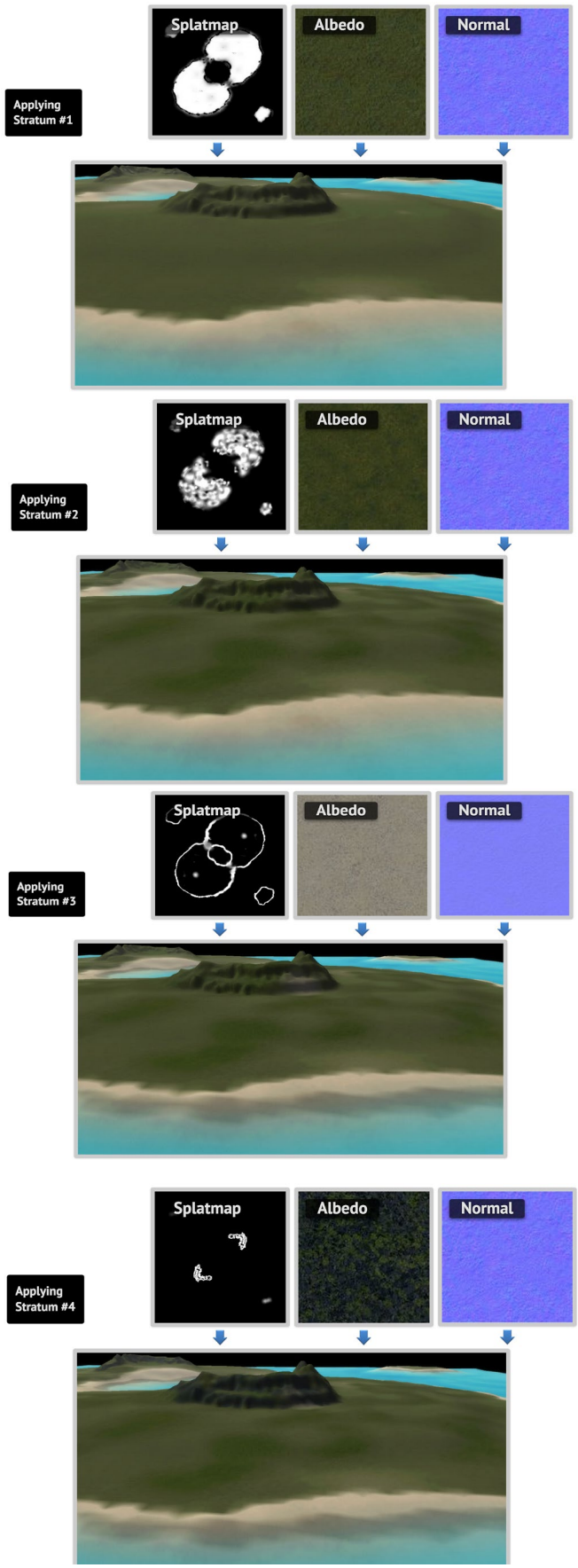
How do we add more details and variations to our map?

The technique used is called "Texture splatting": the game draws a series of additional albedo+normal textures. Each step adds what's called a "stratum" to the terrain.

We already have stratum 0: the terrain with its original albedo+color textures.

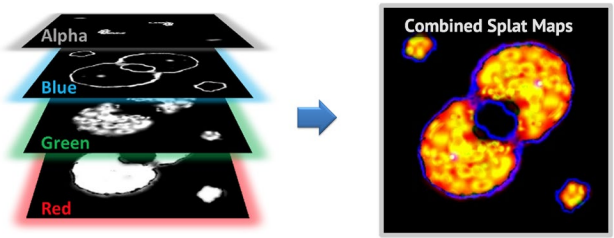
To apply the next stratum, we need some extra information: a "splat map," to tell us where to draw the new albedo+normal and more importantly where not to draw! Without such a "splat map" also called alpha-map, applying a new stratum would completely cover the previous stratum. The albedo and normal textures both have their own scaling factor when they are applied to the mesh.





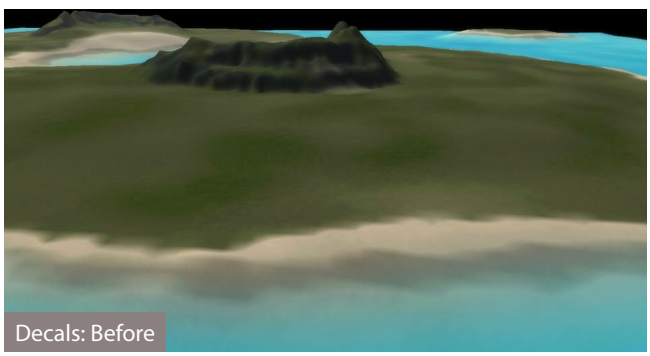
So we applied strata 1, 2, 3 and 4, each one relying on 3 separate textures. The albedo and normal textures use 3 channels (RGB) each, but the splat map uses only one channel.

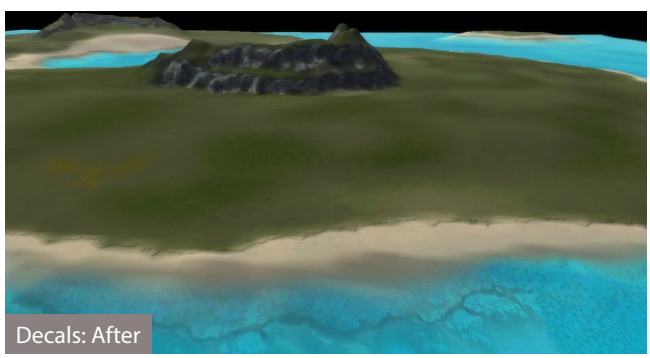
So as an optimization the 4 splat maps are combined into a single RGBA texture.

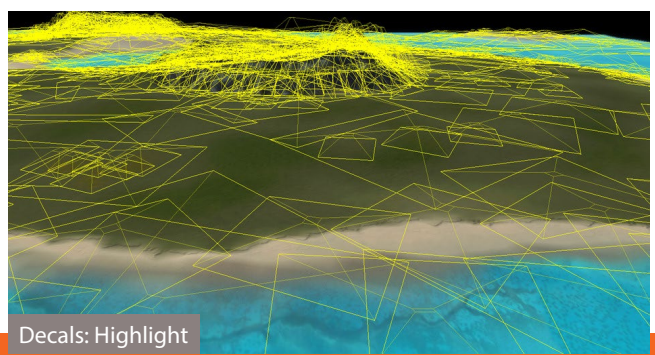


Now we've got more texture variations for our terrain. It looks nice from far away, but if you zoom in, you quickly notice the lack of high-frequency details.

This is when decals are applied: these are like small sprites which locally modify the albedo color and the normal of a pixel. This terrain has 861 instances of 21 unique decals.

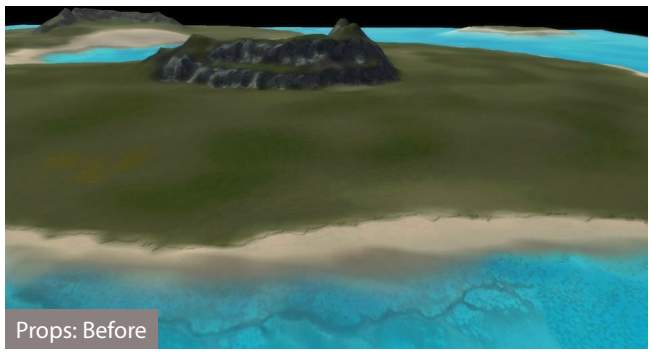

Decals: Before

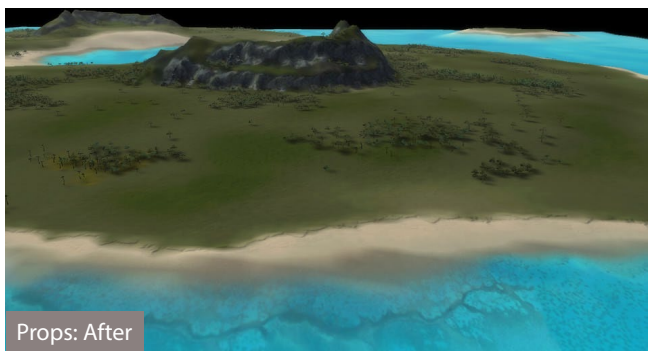

Decals: After

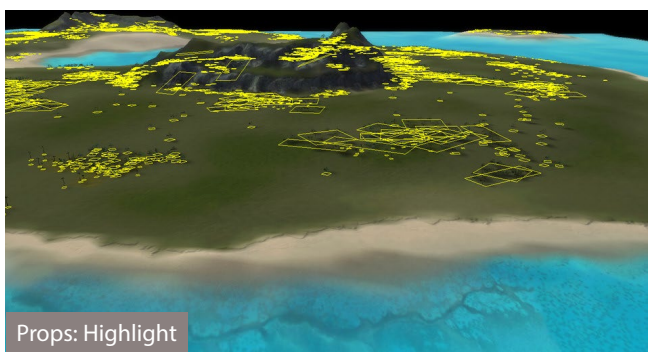

Decals: Highlight

It's much better, but what about some vegetation?

The next step is to add to the terrain what the engine calls "Props": tree or rock models. For this map there are 6026 instances of 23 unique models.

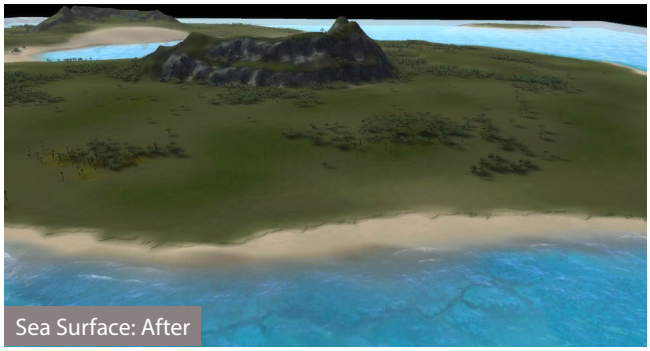

Props: Before


Props: After


Props: Highlight

Now the final touch: the sea surface. It is a combination of several normal maps with UV scrolling along different directions, an environment map for reflections and sprites for the waves near the shores.


Sea Surface: Before


Sea Surface: After

Our terrain is now ready.

Creating good heightmaps and splat maps can be challenging for map designers, but fortunately there are several tools to help with the task: there is the official "Supcom Map Editor" or World Machine [world-machine.com] with even more advanced features.

So now that we know the theory behind the SupCom terrains, let's move on to an actual frame of the game.

Frame Breakdown

This is the game frame we'll dissect:

Frustum Culling

The game has in RAM the terrain mesh, created from the heightmap. It is tessellated by the CPU, and the position of each vertex is known. When the zoom level changes, the CPU re-calculates the tessellation of the terrain.

Our camera focuses on a scene near the shore. Rendering the whole terrain would be a waste of calculation, so instead the engine extracts a submesh of the whole terrain, only the portion visible to the player, and feeds this small subset to the GPU for rendering.


Frustum Culling: Before

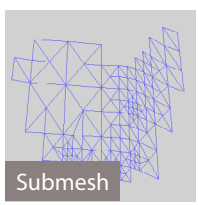

Frustum Culling: After

Normal Map

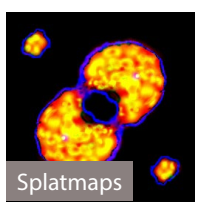
First, only the normals are calculated.

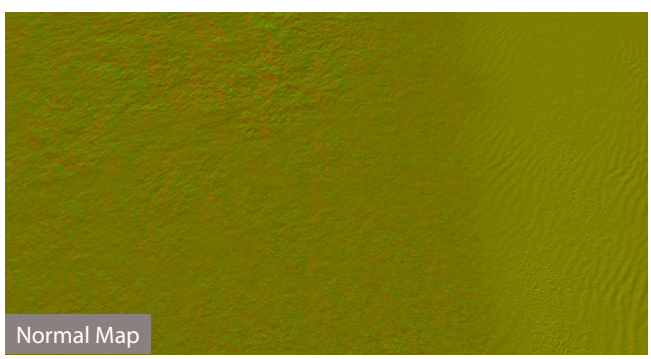
A first pass computes the normals resulting from the combination of the 5 strata (5 normal maps and 4 splat maps).

The different normal maps are blended together, all the operations are done in tangent space.


Submesh


Normals x5


Splatmaps

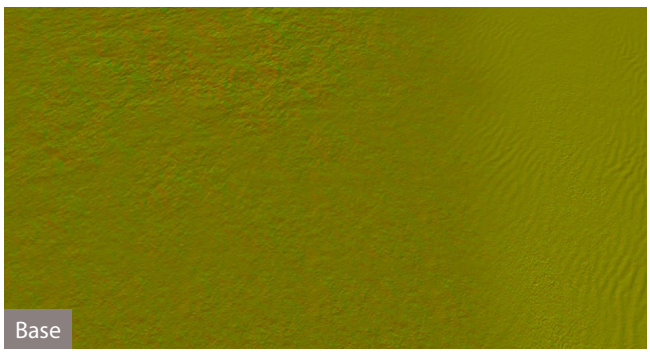

Normal Map

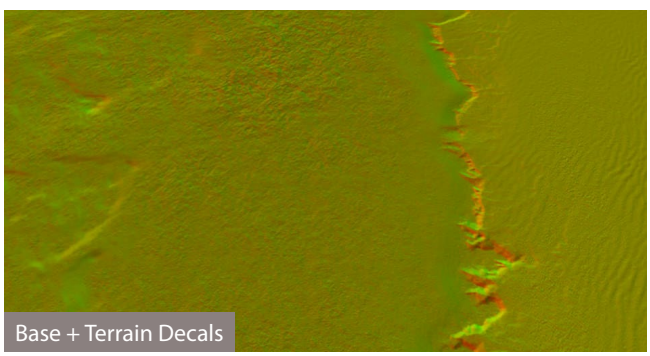
This is done in a single draw call with 6 texture fetches. You'll notice the result is quite yellowish. It contrasts with the other normal maps which tend to be blue. And indeed: here the Blue channel is not used at all, only the Red and Green.

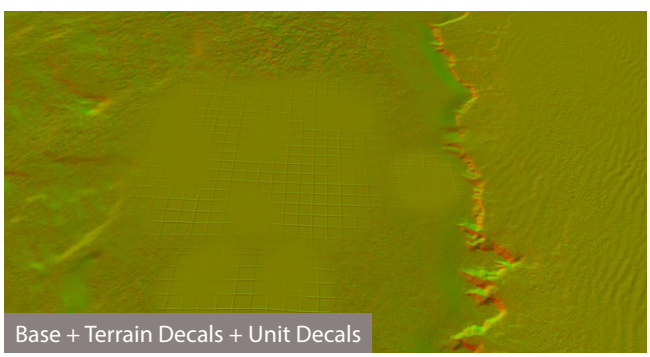
But wait, a normal is a 3-component vector. How can it be stored only with 2 components? It's actually a compression technique: only the X and Y components are stored, Z can be derived from them.

For now let's just assume the Red and Green channels contain all the information we need about the normals.

Strata are done, now it's the turn of the decals: terrain decals and building decals are added to modulate the stratum normals.

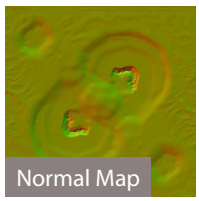

Base


Base + Terrain Decals

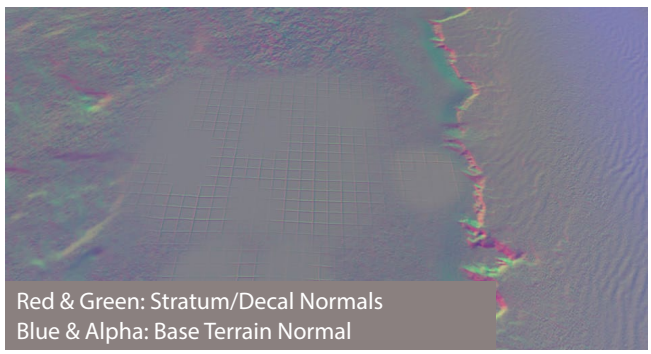

Base + Terrain Decals + Unit Decals

We still haven't used the Blue and Alpha channels of our render target.

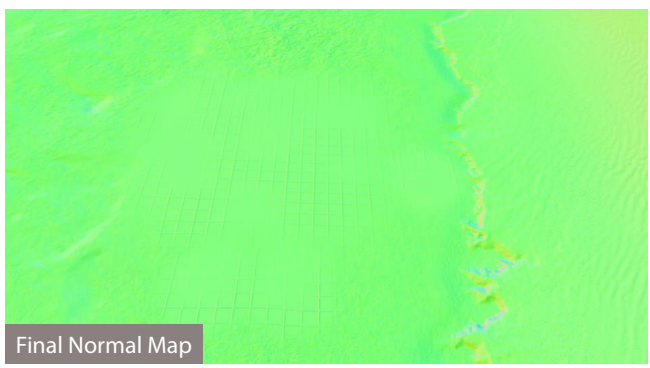
So the game reads from a 512x512 texture representing the whole normals of the terrain (baked from the original heightmap), and calculates for each pixel its normal using a bicubic interpolation. The results are stored in the Blue and Alpha channels.


Normal Map


Bicubic interpolation stored in Blue and Alpha.


Red & Green: Stratum/Decal Normals
Blue & Alpha: Base Terrain Normal

Then the game combines these two sets of normals (stratum/decal normals and terrain normals) into the final normals used to calculate the lighting.


Final Normal Map

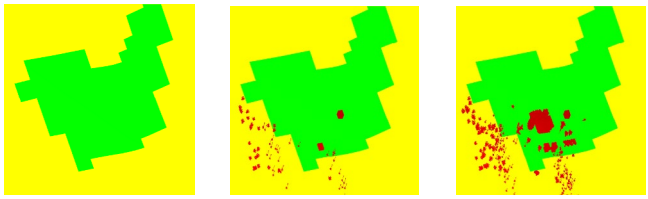
This time there's no compression: the normals use the 3 RGB channels, one for each component.

It might look very green, but this is because the scene is quite flat. The result is correct: you can take any pixel and calculate its normal vector by doing `colorRGB * 2.0 - 1.0`. You can also check that the norm of the vector is 1.

Shadow Map

The technique used to render the shadows is the "Light Space Perspective Shadow Maps" or LiSPSM technique. [hn.my/lspsm]

Here we just have the sun as a directional light. Each mesh of the scene is rendered, and its distance from the sun is stored into the Red channel of a 1024x1024 texture. The LiSPSM technique calculates the best projection space to maximize the precision of the shadow map.



If we stop here, we would just be able to draw hard shadows. When the units are rendered, the game actually tries to smooth out the edges of the shadows by using some PCF sampling.

But even with PCF, there would be no way to obtain the beautiful, smooth shadows we see on the screenshot, especially the smooth silhouettes of the buildings on the ground…. So how was this achieved?

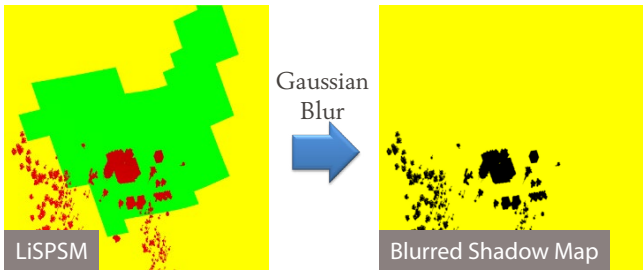
Even during the final parts of the development process of the game, it seems the implementation of shadows was still an on-going effort. This is what Jonathan Mavor was saying 11 months before the game public release:

> *The shadows in those shots are not finished and we do have a little bit of work to do on them yet. […]*
> *We are not finished with the game graphically by any stretch at this point.*
> *– **Jonathan Mavor**, February 24th 2006*

Just one month after this declaration, a new groundbreaking shadow map technique was emerging: Variance Shadow Maps or VSM. [hn.my/vsm] It was able to render gorgeous soft shadows very efficiently.

It seems the SupCom team tried to experiment with this new technique: decompiling the D3D bytecode reveals a reference to a `DepthToVariancePS()` function which computes a blur version of the shadow map. Before VSM was invented, shadow maps could not be blurred.

Here SupCom performs a 5x5 Gaussian blur (horizontal and vertical pass) of the shadow map.



However in the D3D bytecode, there is no instruction about storing the depth and the squared-depth (information required by the VSM technique). It seems to be only a partial implementation. Maybe there was no time to perfect the technique during the final stages of the development, but anyway the code as-is can already produce nice results.

Note though that the pseudo-VSM map is used only to produce soft-shadows on the ground.

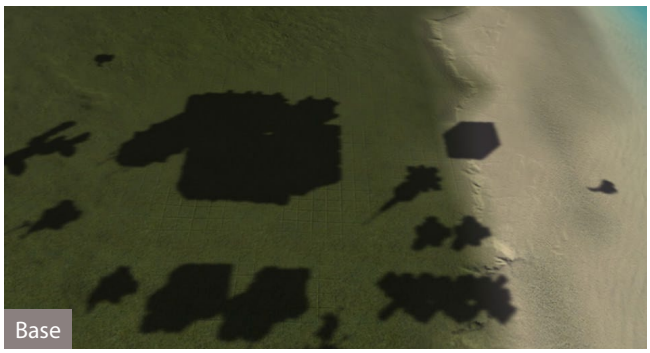
When a shadow must be drawn onto a unit, it is done through the LiSPSM map with PCF sampling. You can see the difference in the screenshot below (PCF has blocky artifacts at the shadow border):

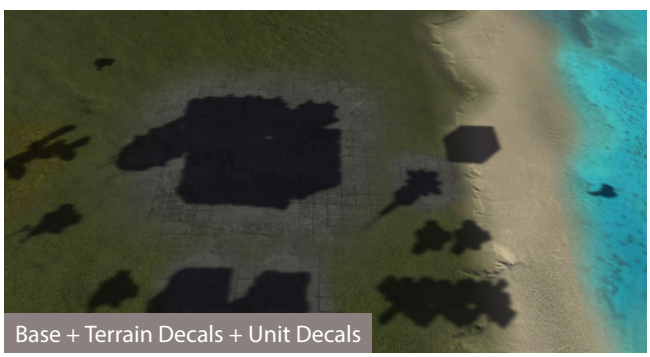
Shadowed Terrain

Thanks to the normal map and the shadow map that were generated, it is possible to finally start rendering the terrain: a textured mesh with lighting and shadows.

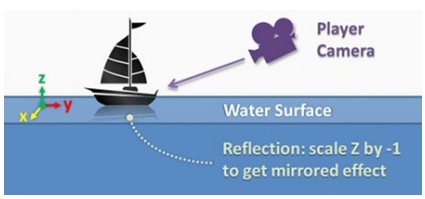
Decals

The albedo components of the decals are drawn, using the normal information to calculate the lighting equation.





Water Reflection

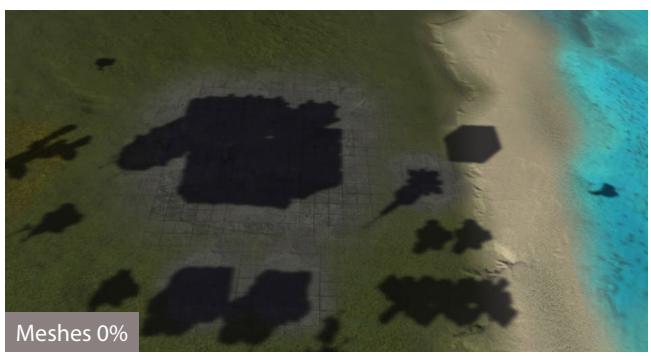


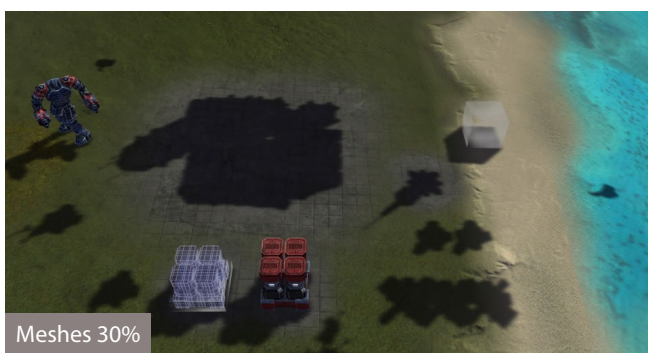
We have the sea on the right of our scene, so if a robot is sitting in the middle of the water we should be able to see its reflection on the sea surface.

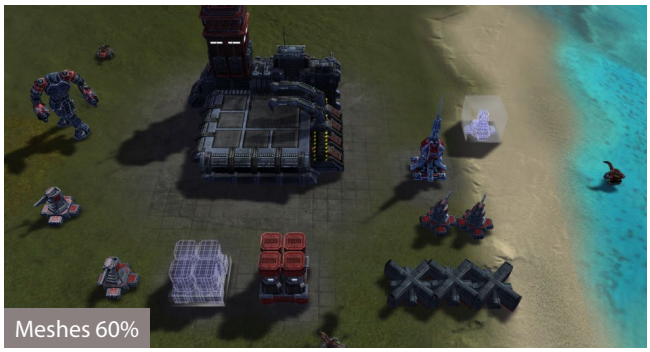
A classic trick exists to render the reflection of a surface: an additional pass is performed, and just before applying the camera transformation, the vertical axis is scaled by -1 so the entire scene becomes symmetric with regards to the water surface (just like a mirror) which is exactly the transformation needed to render the reflection. SupCom uses this technique and renders all the mirrored unit meshes into a reflection map.

Mesh Rendering

All the units are then rendered one by one. For the vegetation, geometry instancing is used to render multiple trees in one draw call. The sea is rendered using a single quad, with a pixel shader fetching several normal maps, a refraction map (the scene rendered up until now), a reflection map (just generated above), and a skybox for additional reflection.







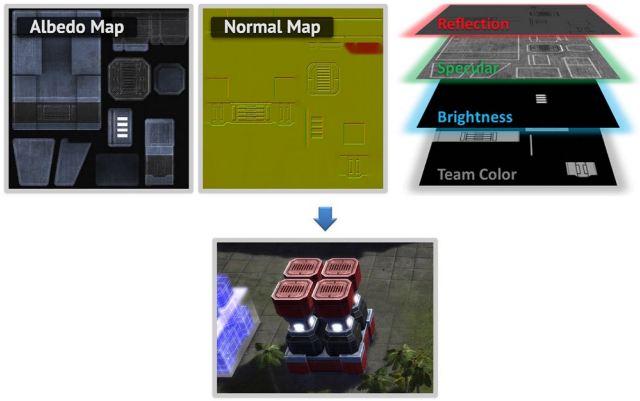
Notice in the last image the small black artifacts of the sea near the screen border: it's because the sampling of the surface of the water is disrupted to create an illusion of movement. Sometimes the disruption brings texels from outside the viewport within the viewport. But such information does not exist, hence the black areas.

During the game, the UI actually hides these artifacts under a thin border covering the edges of the viewport.

Mesh Structure

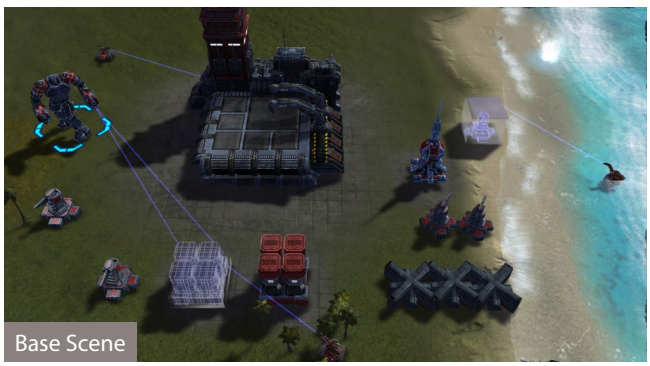
Each unit in SupCom is rendered in a single draw call. A model is defined by a set of textures:

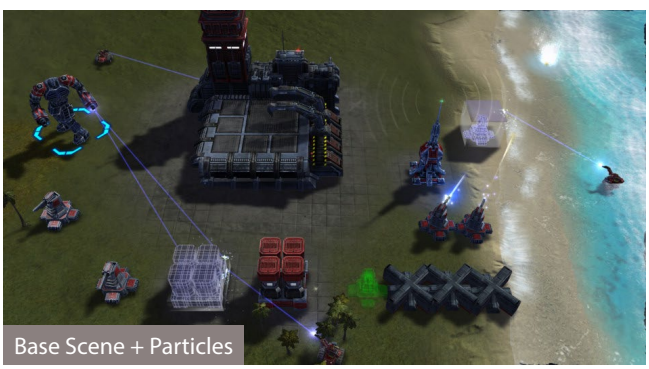
- An albedo map
- A normal map
- A "specular map" which actually contains much more information than the specular. It's an RGBA texture with:
  - Red: Reflection. How much the environment map is reflected.
  - Green: Specular. In regards to the sun light.
  - Blue: Brightness. Used later to control the bloom.
  - Alpha: Team Color. It modulates the unit albedo depending on the team color.

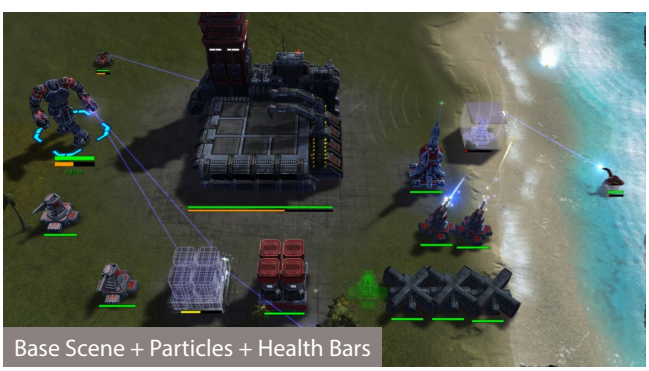


Particles

All the particles are then rendered and the health bars of each unit are also added.





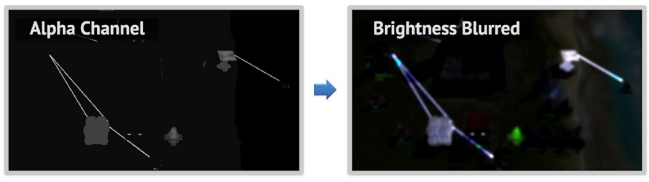


Bloom

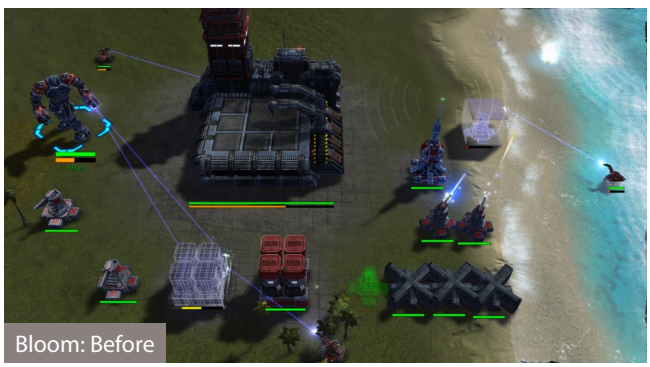
Time to make things shine! But how do we get the "brightness information" since we're working with LDR buffers?

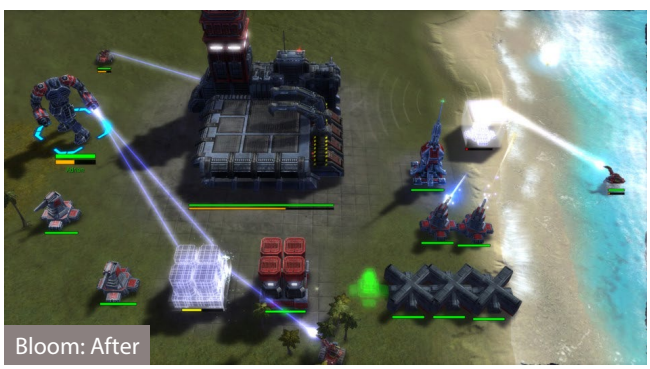
The brightness map is actually contained within the alpha channel, it was being built at the same time the previous meshes were drawn.

A downscaled copy of the frame is created, the alpha channel is used to make only the bright areas stand out and two successive Gaussian blurs are performed.



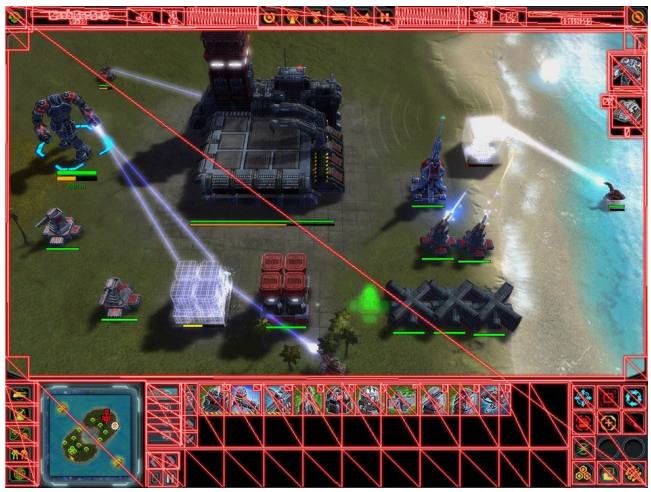
The blurred buffer is then drawn on top of the original scene with additive blending.


Bloom: Before


Bloom: After

User Interface

We're done concerning the main scene. The UI is finally rendered, and it is beautifully optimized: single draw call to render the entire interface. 1158 triangles are pushed at once to the GPU.



The pixel shader reads from a single 1024x1024 texture which acts like a texture atlas. When another unit is selected, the UI is modified, the texture atlas is regenerated on-the-fly to pack a new set of sprites.

And we're done for the frame! ■

Adrian Courrèges is a software engineer working in Tokyo on several game-related fields, ranging from network to real-time graphics and console code. He founded his own indie studio [breakingbyte.com] and has a passion for development and reverse-engineering. Follow him on Twitter at *@ado_tan*

Reprinted with permission of the original author.
First appeared in *hn.my/supcom* (adriancourreges.com)

Speed Matters

Why Working Quickly Is More Important Than It Seems

By JAMES SOMERS

The obvious benefit to working quickly is that you'll finish more stuff per unit time. But there's more to it than that. If you work quickly, the cost of doing something new will seem lower in your mind. So you'll be inclined to do more.

The converse is true, too. If every time you write a blog post it takes you six months, and you're sitting around your apartment on a Sunday afternoon thinking of stuff to do, you're probably not going to think of starting a blog post, because it'll feel too expensive.

What's worse, because you blog slowly, you're liable to continue blogging slowly — simply because the only way to learn to do something fast is by doing it lots of times.

This is true of any to-do list that gets worked off too slowly. A malaise creeps into it. You keep adding items that you never cross off. If that happens enough, you might one day stop putting stuff onto the list.

* * *

I've noticed that if I respond to people's emails quickly, they send me more emails. The sender learns to expect a response, and that expectation spurs them to write. That is, speed itself draws emails out of them, because the projected cost of the exchange in their mind is low. They know they'll get something for their effort. It'll happen so fast they can already taste it.

It's now well known on the web that slow server response times drive users away. A slow website feels broken. It frustrates the visitor's desire. Probably it deprives them of some dopaminergic reward.

Google famously prioritized speed as a feature. They realized that if search is fast, you're more likely to search. The reason is that it encourages you to try stuff, get feedback, and try again. When a thought occurs to you, you know Google is already there. There is no delay between thought and action, no opportunity to lose the impulse to find something out. The projected cost of googling is nil. It comes to feel like an extension of your own mind.

It is a truism, too, in workplaces, that faster employees get assigned more work. Of course they do. Humans are lazy. They want to preserve calories. And it's exhausting merely thinking about giving work to someone slow. When you're thinking about giving work to someone slow, you run through the likely quagmire in your head; you visualize days of halting progress. You imagine a resource — this slow person — tied up for a while. It's wearisome, even in the thinking. Whereas the fast teammate — well, their time feels cheap, in the sense that you can give them something and know they'll be available again soon. You aren't "using them up" by giving them work. So you route as much as you can through the fast people. It's ironic: your company's most valuable resources — because they finish things quickly — are the easiest to consume.

The general rule seems to be: systems which eat items quickly are fed more items. Slow systems starve.

Two more quick examples. What's true of individual people turns out also to be true of whole organizations. If customers find out that you take two months to frame photos, they'll go to another frame shop. If contributors discover that you're slow to merge pull requests, they'll stop contributing. Unresponsive systems are sad. They're like buildings grown over with moss. They're a kind of memento mori. People would rather be reminded of life. They'll leave for places that get back to them quickly.

Even now, I'm working in a text editor whose undo feature, for whatever reason, has suddenly become slow. It's killing me. It disinclines me, for one thing, from undoing stuff. But it's also probably subtly changing the way I work. I feel like I can't rely on undo. So if I want to delete something but think I might want it later, I'm copying it to the bottom of the file, like it's the 1980s. All this because undo is so slow that it might as well not exist. Undo, when it's fast, is an incredible feature; at any moment, you can dip into the past, borrow something, and zip back. But now it feels like a dead end.

Part of the activation energy required [hn.my/activation] to start any task comes from the picture you get in your head when you imagine doing it. It may not be that going for a run is actually costly; but if it feels costly, if the picture in your head looks like a slog, then you will need a bigger expenditure of will to lace up.

Slowness seems to make a special contribution to this picture in our heads. Time is especially valuable. So as we learn that a task is slow, an especial cost accrues to it. Whenever we think of doing the task again, we see how expensive it is and bail.

That's why speed matters.

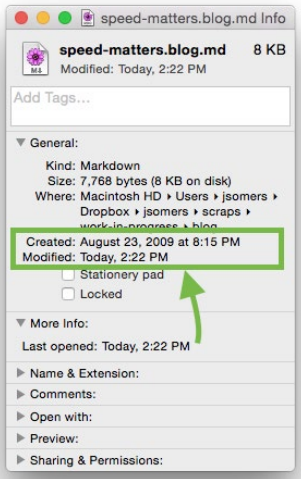
* * *

The prescription must be that if there's something you want to do a lot of and get good at — like write, or fix bugs — you should try to do it faster.

That doesn't mean be sloppy. But it does mean, push yourself to go faster than you think is healthy. That's because the task will come to cost less in your mind; it'll have a lower activation energy. So you'll do it more. And as you do it more (as long as you're doing it deliberately), you'll get better. Eventually you'll be both fast and good.

Being fast is fun. If you're a fast writer, you'll constantly be playing with new ideas. You won't be bogged down in a single dread effort. And because your to-do list gets worked off, you'll always be thinking of more stuff to add to it. With more drafts in the works, more of the world will pop alive. You will feel flexible and capable and practiced so that when something demanding and long arrives on your desk, you won't back down afraid.

Now, as a disclaimer, I should remind you of the rule that anyone writing a blog post advising against X is himself the worst Xer there is. At work, I have a history of painful languished projects, and I usually have the most overdue assignments of anyone on the team. As for writing, well, I have been working on this little blog post, on and off, no joke, for six years. ■



James Somers is a writer and programmer based in New York. He works at Genius.

Reprinted with permission of the original author.
First appeared in *hn.my/speed* (jsomers.net)

Writing a Game Boy Advance Game

By JOE SAVAGE

Illustration: James H. [blueamnesiac.deviantart.com]

I SPENT A LOT of time as a kid playing (generally pretty terrible) games on my Game Boy. Having never written code for anything other than "regular" general purpose computers before, I've been wondering recently: how easy is it to write a Game Boy (Advance) game?

For those unfamiliar, the Game Boy Advance (GBA) was a popular handheld games console produced by Nintendo (pictured below). This thing is kitted out with a 240x160 (3:2) 15-bit color LCD display, along with six face buttons and a directional pad for input.

On the inside, the GBA's CPU contains a 32-bit ARM7tdmi RISC core (operating at 16.78 MHz). Along with regular 32-bit ARM instructions, this chip can also execute 16-bit Thumb [hn.my/thumb] instructions. The Thumb instruction set is essentially just a 16-bit encoding for some of the most common 32-bit ARM instructions, which we can use to save space.

In terms of memory, the device has 130 KB of embedded memory within the CPU (96 KB of which is used for VRAM, 32 KB of which is for general usage, and 2 KB of which are used elsewhere), and 256 KB of RAM external to the CPU. The system also has 16 KB of System ROM, which is used to store the BIOS. There are also some additional details in all this regarding backwards compatibility of the Game Boy Advance with the Game Boy Color, but we're not going to discuss them here.

Along with all this internal memory, the GBA is typically loaded with some form of game cartridge. These typically consist of some ROM (to store instructions, read-only data, etc.), and some form of mutable storage (typically SRAM, Flash Memory, or EEPROM). As the Game Pak ROM is connected via a 16-bit wide bus, it makes sense to use 16-bit Thumb instructions rather than 32-bit ARM instructions most of the time in game code.

A GBA Game Pak [reinerziegler.de]

All of the memory sections we've discussed, along with I/O hardware registers (to control graphics, sound, DMA, etc.), are mapped into memory, giving a memory layout something like the following:

- `0x00000000 - 0x00003FFF`: 16 KB System ROM (executable, but not readable)
- `0x02000000 - 0x02030000`: 256 KB EWRAM (general purpose RAM external to the CPU)
- `0x03000000 - 0x03007FFF`: 32 KB IWRAM (general purpose RAM internal to the CPU)

- `0x04000000 - 0x040003FF`: I/O Registers
- `0x05000000 - 0x050003FF`: 1 KB Color Palette RAM
- `0x06000000 - 0x06017FFF`: 96 KB VRAM (Video RAM)
- `0x07000000 - 0x070003FF`: 1 KB OAM RAM (Object Attribute Memory – discussed later)
- `0x08000000 - 0x????????`: Game Pak ROM (0 to 32 MB)
- `0x0E000000 - 0x????????`: Game Pak RAM

These sections have varying bus widths and read/write widths (e.g., you can't write individual bytes into VRAM!), and some sections are mirrored in memory at multiple points. There is also some extra complexity to this in reality, but this is the main structure that we'll need to build a basic GBA game.

With this knowledge of the memory structure of the device, the plan to make a "Hello, World!" GBA ROM is as follows: write some Thumb code for our Game Pak ROM which sets display parameters in I/O registers as appropriate for some particular display mode, and then write some graphical data into VRAM that we want to display. With some of the theory about the device out of the way, let's actually try to build something.

Setting Up A Development Environment

To begin executing our plan to build a GBA ROM, we need to know a bit about the ROM format. Without going too far into the details, GBA ROMs begin with a standard header. This should start with a four byte ARM instruction to branch to the start address of our program, followed by some "magic" bytes representing the Nintendo logo.

Additionally, in this header there is some data about the game (its title, etc.), and a "check" value for this data. We will need to ensure that the header is perfectly correct if we want our ROM to execute properly (particularly if we're aiming to execute on an actual device rather than an emulator).

Thankfully, most of the details of ROM creation can be handled by a good toolchain. I use the devkitARM toolchain myself (one of the devkitPro toolchains, based on the GCC toolchain), which makes the process extremely easy. Essentially, once the toolchain is set up, we can turn some C code into a GBA ROM in four steps:

1. Cross-compile our C code to Thumb instructions for the GBA's CPU, creating a Thumb object file with our ROM code.
2. Link our object file into an executable using a specific "specs" file to control the behavior of the linking. Typically the specs file includes a link script (to specify segment locations [most mutable data will get stored in IWRAM, "const" data in ROM, etc.] and alignments for correct compilation for the GBA), and some other object files (usually, a standard ROM header, startup routines, program initialization and termination code, etc.).
3. Strip our executable file of information we don't need (executable header, symbol and relocation information, etc.), to get a near-complete ROM file.
4. Run a utility on the ROM file from the previous step to fix its header (ensure that the Nintendo logo data in the header is correct, set any "check" values as appropriate, etc.)

With the version of the toolchain I have on my machine running OS X, I can run the following commands (providing I have `/opt/devkitpro/devkitARM/bin` in my PATH environment variable) to compile a C file into a GBA ROM (as we described above):

1. `arm-none-eabi-gcc -c main.c -mthumb-interwork -mthumb -O2 -o main.o`
2. `arm-none-eabi-gcc main.o -mthumb-interwork -mthumb -specs=gba.specs -o main.elf`
3. `arm-none-eabi-objcopy -v -O binary main.elf main.gba`
4. `gbafix main.gba`

There are also some additional flags you might want to pass in for the first step (the compilation). I'd recommend `-fno-strict-aliasing`, for example, as we'll be dealing with raw memory and pointers a lot and don't really want C's strict aliasing rule to bite us. You might also find it beneficial to write a Makefile or shell script with these commands to make ROM compilation easier — these details seem a little unnecessary to include in this post though.

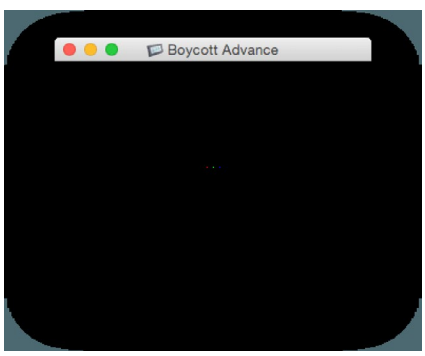
With a process for ROM compilation from C established, let's test it out. To help out any readers that are getting bored from the theory, I'll present the code for our "Hello, World" GBA ROM first and then discuss is afterwards.

```
int main(void) {
	// Write into the I/O registers,
	// setting video display parameters.
	volatile unsigned char *ioram = (unsigned
char *)0x04000000;
	ioram[0] = 0x03;
	// Set the 'video mode' to 3 (in which
	// BG2 is a 16 bpp bitmap in VRAM)
	ioram[1] = 0x04; // Enable BG2 (BG0 = 1,
BG1 = 2, BG2 = 4, ...)

	// Write pixel colours into VRAM
	volatile unsigned short *vram = (unsigned
short *)0x06000000;
	vram[80*240 + 115] = 0x001F; // X = 115,
Y = 80, C = 000000000011111 = R
	vram[80*240 + 120] = 0x03E0; // X = 120,
Y = 80, C = 000001111100000 = G
	vram[80*240 + 125] = 0x7C00; // X = 125,
Y = 80, C = 111110000000000 = B

	// Wait forever
	while(1);
	return 0;
}
```

The code above is relatively simple, and should result in a horizontal set of three pixels being drawn in the middle of the GBA screen — one red, one green, and one blue:



Now, time to explain the code. Firstly, we write some display parameters to the memory mapped I/O registers. In particular, the first 16 bits of this memory is a display control register (often called DISPCNT). The first three bits of this register indicate the video mode, and the 11th bit indicates whether background #2 (BG2) is enabled or not. Thus by writing the values we do, we set the video mode to mode 3 and enable BG2.

Why do we need to do this? Well, first because of the video mode. It turns out that video mode 3 is a mode in which we can write bitmap data into VRAM, and BG2 will display this bitmap (hence, why we also want to enable BG2). You might also be wondering why I've chosen to use video mode 3 rather than another video mode. The reason for this is that video modes 0 to 2 are much more difficult to explain.

As I mentioned earlier, the LCD on the GBA can display 15-bit colors. Therefore, we can express GBA colors using a 15-bit long color format. For data alignment reasons, though, the GBA uses a 16-bit color format. Specifically, the format is as follows: `?BBBBB-GGGGGRRRRR`. So that's an unused bit, followed by five bits of blue, five bits of green, and then five bits of red.

Using this format, and with knowledge of how video mode 3 treats VRAM as a 240x160 bitmap, our "Hello, World" ROM simply writes some color values at specific pixel offsets. For example, as we're assuming that `unsigned short` is 16-bits in size, `vram[80*240 + 120]` skips 80 horizontal lines of 240 pixels, and then accesses the middle pixel on that horizontal line. Note, by the way, that all the memory accesses for interfacing with hardware in the code occur through `volatile` pointers. This prevents the compiler optimizing out what it might think are useless memory operations.

Writing A Pong-esque Game

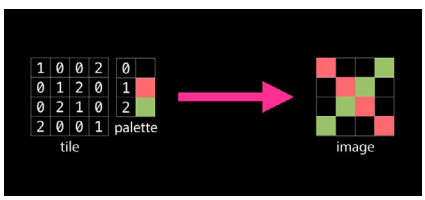
With the basics out of the way, let's build something a little more interesting. We're still going to hack the solution together rather than building a bunch of project infrastructure and helper functions, but we'll make use of some more advanced features of the GBA's graphics rendering. In particular, we'll depart from drawing using the bitmap video modes.

While drawing in the GBA's bitmap video modes (modes 3, 4, and 5) is very easy, for many games it's not really practical. Our 240x160 bitmap itself takes up the majority of VRAM just to fill the screen once, and pushing around so many pixels every frame can be computationally expensive, too (we might not be able to afford this if we're aiming to render our game at a reasonable framerate). Thus, we have video modes 0, 1, and 2.

There is a fair amount of complexity wrapped up in these modes, so we're only going to attempt to run through the most important pieces. Instead of operating on individual pixels, the GBA's first three video modes operate on tiles. A tile is an 8x8 bitmap. These

exist in 4 and 8 bits per pixel (bpp) variants, but here we'll be using the 4bpp type. Thus, the tiles we'll be using have a size of 32 bytes (8 * 8 * 4 = 256 bits).

If you're wondering how we're supposed to fit 15-bit color values for each pixel in 4 (or 8) bits, we don't. Instead of referring directly to colors, the pixel values in tiles refer to colors within a particular color palette. We can define color palettes by writing color values into the color palette memory we mentioned earlier (`0x05000000`). This can store 512 sets of 16 bits (hence, 512 colors), which essentially means that we can store 2 palettes of 256 colors, or 32 palettes of 16 colors.



A visual example of a paletted tile

In the case of our 4bpp 8x8 tile bitmaps, we'll treat the color palette memory as 32 palettes of 16 colors. This way, we can use our four bits for each pixel to specify the color index (within some palette of 16 colors) for this pixel. When using tile-based video modes, tiles are sectioned in VRAM into "tile blocks" or "charblocks." Each tile block is 16 KB in size, so we can fit 512 4bpp tiles in a tile block, and 6 tile blocks in VRAM.

The theoretical set of 6 tile blocks in VRAM are split into two groups. The first four (0 - 3) can be used for backgrounds, and the last two (4 and 5) can be used for sprites. Similarly, the 32 palettes of 16 colors in palette memory are split into 16 palettes for backgrounds and 16 palettes for sprites. Since we're not going to deal with backgrounds in our game, we're only interested in tile blocks 4 and 5 in VRAM (i.e., those starting at addresses `0x6010000` and `0x6014000`), and color palette block 1 (address `0x5000200`).

So, say that we've loaded some tiles into tile block 4. What can we do with this?

Well, the whole point of us dealing with tiles in this case is to create sprites which use them.

A sprite, in Computer Graphics, is a 2D image that fits within a larger scene. It turns out that the GBA has hardware that can render "objects" (i.e., sprites) for you, and these objects get rendered such that the object can move around without leaving a trail of modified pixels. Providing that objects are enabled (bit 13 in the display control I/O register is set), an object can be created from a particular set of tiles by writing the object's attributes into the GBA's Object Attribute Memory (OAM). In this case, as we're looking to make a "pong"-esque game, we'll probably want at least two sprites: a paddle and a ball. Any particular "object" has three sets of 16-bit attributes:

- **Attribute 0**: includes, among other things, the y coordinate of the object, the shape of the object, and the color mode of the object's tiles (4bpp or 8bpp).
- **Attribute 1**: includes, among other things, the x coordinate of the object, and the size of the object.
- **Attribute 2**: includes, among other things, the base tile index of the object, and the color palette the object should use (when in 4bpp mode).

The specifics of these values can be viewed elsewhere, but essentially, the y coordinates are the lowest 8 bits of attribute 0, the x coordinates are the lowest 9 bits of attribute 1, and the color mode defaults to 4bpp (i.e. zero = 4bpp).

The "shape" and "size" bits of an object define its form, and different combinations of these four bits result in different final shapes (entities more complex than this system are made up of multiple smaller objects). If an object should be larger than one tile in size, it will use different tiles for its appearance depending on the mapping mode that is set (the 7th bit of the display control I/O register). It's easiest for us to use the 1D mapping mode, so if an object is bigger than one tile, it will fill itself using the tiles that follow its "base tile" in memory.

With sprites explained, we're almost ready to start building. We want to use video mode 0 for this program, in which BG0 - BG3 operate in "regular" mode (we can't perform affine transformations on them). Now we just need to feed the input from the GBA's directional pad into some primitive physics code, put that all inside some sort of game loop, and we have ourselves a game!

The last pieces of this puzzle are both in I/O registers. The input state of the device can simply be read from the KEYINPUT I/O register (`0x04000130`), and we can use the particulars of how this is laid out to create masks on this state to determine whether particular keys have been pressed. As for the game loop... unfortunately, this requires one last piece of theory.

A typical game loop consists of a draw period, and an update period; in this case, we can't just choose

when these occur ourselves, though. If we decide to change what we want to display when the Game Boy is halfway through drawing an object, we might get screen tearing (as half of the object was drawn with one set of data, and the other half with another). As a result, we need some way to synchronize our drawing and updating with the GBA's display refresh cycle.

The device gives us a little time to update after every horizontal line (or "scanline") that it draws, but gives us even more time (around 5ms) after it's finished drawing to the whole screen. In this case, we'll just use the time available after drawing to the entire screen to do our updates. This period is called a "V-Blank" (as opposed to a "V-Draw", when the screen is still being drawn vertically).

To check how far the device has drawn vertically at current, we can check the 8 bit value in the VCOUNT I/O register (at `0x04000006`), which continues increasing during the V-Blank as if scanlines were still being drawn (thus, has a range from 0 to 227). If the count is greater than or equal to 160, we're in a V-Blank. Thus, if we wait for a V-Draw to end before we begin the "update" stage of our game loop, we have a primitive form of synchronization.

With this synchronization, we finally have enough information to build our game. In this case, I've chosen to build a single-player pong-esque game (with extremely primitive physics), the commented source code of which follows.

```
typedef unsigned char uint8;
typedef unsigned short uint16;
typedef unsigned int uint32;
typedef uint16 rgb15;
typedef struct object_attributes {
	uint16 attribute_zero;
	uint16 attribute_one;
	uint16 attribute_two;
	uint16 pad;
} __attribute__((aligned(4))) object_attributes;
typedef uint32 tile4bpp[8];
typedef tile4bpp tile_block[512];

#define SCREEN_WIDTH  240
#define SCREEN_HEIGHT 160

#define MEM_IO   0x04000000
#define MEM_PAL  0x05000000
#define MEM_VRAM 0x06000000
#define MEM_OAM  0x07000000

#define REG_DISPLAY        (*((volatile uint32 *)(MEM_IO)))
#define REG_DISPLAY_VCOUNT (*((volatile uint32 *)(MEM_IO +
0x0006)))
#define REG_KEY_INPUT      (*((volatile uint32 *)(MEM_IO +
0x0130)))

#define KEY_UP     0x0040
#define KEY_DOWN   0x0080
#define KEY_ANY    0x03FF

#define OBJECT_ATTRIBUTE_ZERO_Y_MASK  0xFF
#define OBJECT_ATTRIBUTE_ONE_X_MASK  0x1FF

#define oam_memory ((volatile object_attributes *)MEM_OAM)
#define tile_memory ((volatile tile_block *)MEM_VRAM)
#define object_palette_memory ((volatile rgb15 *)(MEM_PAL +
0x200))

// Form a 16-bit BGR GBA colour from three component values
// (hopefully, in range).
static inline rgb15 RGB15(int r, int g, int b) { return r | (g <<
5) | (b << 10); }

// Set the position of an object to specified x and y coordinates
// (hopefully, in range).
static inline void set_object_position(volatile object_attributes
*object, int x, int y) {
	object->attribute_zero = (object->attribute_zero &
```

```
~OBJECT_ATTRIBUTE_ZERO_Y_MASK) | (y & OBJECT_ATTRIBUTE_ZERO_Y_MASK);
	object->attribute_one = (object->attribute_one & ~OBJECT_ATTRIBUTE_ONE_X_MASK) | (x &
OBJECT_ATTRIBUTE_ONE_X_MASK);
}

// Clamp 'value' in the range 'min' to 'max' (inclusive).
static inline int clamp(int value, int min, int max) { return (value < min ? min : (value > max ?
max : value)); }

int main(void) {
	// Write the tiles for our sprites into the 4th tile block in VRAM.
	// Particularly, four tiles for an 8x32 paddle sprite, and 1 tile for an 8x8 ball sprite.
	// 0x1111 = 0001000100010001 [4bpp = colour index 1, colour index 1, colour index 1, colour
index 1]
	// 0x2222 = 0002000200020002 [4bpp = colour index 2, colour index 2, colour index 2, colour
index 2]
	// NOTE: We're using our own memory writing code here to avoid the byte-granular writes that
	// something like 'memset' might make (GBA VRAM doesn't support byte-granular writes).
	volatile uint16 *paddle_tile_memory = (uint16 *)tile_memory[4][1];
	for (int i = 0; i < 4 * (sizeof(tile4bpp) / 2); ++i) { paddle_tile_memory[i] = 0x1111; }
	volatile uint16 *ball_tile_memory = (uint16 *)tile_memory[4][5];
	for (int i = 0; i < (sizeof(tile4bpp) / 2); ++i) { ball_tile_memory[i] = 0x2222; }

	// Write the colour palette for our sprites into the first palette of
	// 16 colours in colour palette memory (this palette has index 0).
	object_palette_memory[1] = RGB15(0x1F, 0x1F, 0x1F); // White
	object_palette_memory[2] = RGB15(0x1F, 0x00, 0x1F); // Magenta

	// Create our sprites by writing their object attributes into OAM memory.
	volatile object_attributes *paddle_attributes = &oam_memory[0];
	paddle_attributes->attribute_zero = 0x8000; // This sprite is made up of 4bpp tiles and has
                                                // the TALL shape.
	paddle_attributes->attribute_one = 0x4000; // This sprite has a size of 8x32 when the TALL
                                               // shape is set.
	paddle_attributes->attribute_two = 1; // This sprite's base tile is the first tile in tile
                                          // block 4, and this sprite should use colour palette 0.
	volatile object_attributes *ball_attributes = &oam_memory[1];
	ball_attributes->attribute_zero = 0; // This sprite is made up of 4bpp tiles and has the
                                         // SQUARE shape.
	ball_attributes->attribute_one = 0; // This sprite has a size of 8x8 when the SQUARE shape
                                        // is set.
	ball_attributes->attribute_two = 5; // This sprite's base tile is the fifth tile in tile
                                        // block 4, and this sprite should use colour palette 0.

	// Initialize our variables to keep track of the state of the paddle and ball,
	// and set their initial positions (by modifying their attributes in OAM).
	const int player_width = 8, player_height = 32, ball_width = 8, ball_height = 8;
	int player_velocity = 2, ball_velocity_x = 2, ball_velocity_y = 1;
	int player_x = 5, player_y = 96;
```

```
	int ball_x = 22, ball_y = 96;
	set_object_position(paddle_attributes, player_x, player_y);
	set_object_position(ball_attributes, ball_x, ball_y);

	// Set the display parameters to enable objects, and use a 1D object->tile mapping.
	REG_DISPLAY = 0x1000 | 0x0040;

	// Our main game loop
	uint32 key_states = 0;
	while (1) {
		// Skip past the rest of any current V-Blank, then skip past the V-Draw
		while(REG_DISPLAY_VCOUNT >= 160);
		while(REG_DISPLAY_VCOUNT < 160);

		// Get current key states (REG_KEY_INPUT stores the states inverted)
		key_states = ~REG_KEY_INPUT & KEY_ANY;

		// Note that our physics update is tied to the framerate rather than a fixed timestep.
		int player_max_clamp_y = SCREEN_HEIGHT - player_height;
		if (key_states & KEY_UP) { player_y = clamp(player_y - player_velocity, 0, player_
max_clamp_y); }
		if (key_states & KEY_DOWN) { player_y = clamp(player_y + player_velocity, 0, player_
max_clamp_y); }
		if (key_states & KEY_UP || key_states & KEY_DOWN) { set_object_position(paddle_attri-
butes, player_x, player_y); }

		int ball_max_clamp_x = SCREEN_WIDTH - ball_width, ball_max_clamp_y = SCREEN_HEIGHT -
ball_height;
		if ((ball_x >= player_x && ball_x <= player_x + player_width) && (ball_y >= player_y
&& ball_y <= player_y + player_height)) {
			// This is not good physics / collision handling code.
			ball_x = player_x + player_width;
			ball_velocity_x = -ball_velocity_x;
		} else {
			if (ball_x == 0 || ball_x == ball_max_clamp_x) { ball_velocity_x = -ball_
velocity_x; }
			if (ball_y == 0 || ball_y == ball_max_clamp_y) { ball_velocity_y = -ball_
velocity_y; }
		}
		ball_x = clamp(ball_x + ball_velocity_x, 0, ball_max_clamp_x);
		ball_y = clamp(ball_y + ball_velocity_y, 0, ball_max_clamp_y);
		set_object_position(ball_attributes, ball_x, ball_y);
	}

	return 0;
}
```

And there we have it, our basic game is complete! Would it work on a real Game Boy Advance? Uh, maybe. If I've made no mistakes, it should work properly, but it's entirely possible that I've messed up somewhere along the line.

Our Game Boy Advance game running in an emulator.

Conclusion

This article turned out to be a lot longer than I expected. There's a lot more to GBA development than is detailed in this post, too. Like any platform, it has its interesting features and its quirks. If you'd like to know more about GBA development, or about any of the device specifics in this article, I found the following resources invaluable:

- Nintendo's AGB Programming Manual [hn.my/gbaman]
- GBATEK [hn.my/gbatek]
- CowBiteSpec [hn.my/cowbitespec]
- Tonc [hn.my/tonc] ■

Joe Savage is a computer science student and software developer from England, interested in a wide variety of areas ranging from reverse engineering to game development. He writes about technical topics on his blog at *reinterpretcast.com*

Reprinted with permission of the original author.
First appeared in *hn.my/gba* (reinterpretcast.com)

Exploiting Android Users

By RUDIS MUIZNIEKS

A Dark Past

I'm going to tell you about some stuff I've done that I'm not particularly proud of. This happened during a period of my life when I was working for a company in the advertising industry. The company already had a pretty strong handle on the email and display advertising markets, but the team I was hired into was a newer group whose job was to break into the desktop advertising game.

It may not be immediately apparent to you what I mean by "desktop advertising," but I can guarantee that you've run into it at some point before. Every time your Grandma calls you up on the weekend complaining that her computer is running slow, and you fire up her copy of Internet Explorer 7 to find that she's got twenty different toolbars installed, you've encountered the kind of thing my team was working on. Every time you've tried to install some open source software through a Google link, didn't pay attention to checkboxes in the installer, and ended up with half a dozen useless registry scanners, disk cleaners, and so-called "anti-malware" programs unintentionally installed on your computer, you may have me to thank for that.

By way of apology, if you ever meet me in real life, I'll buy you a beer. I promise. Just please try to resist the urge to punch me. I am very sorry for my involvement in everything that you are about to read.

As morbidly interesting as the desktop side of things might be, I'm going to tell you a little bit about what we eventually branched into after the desktop business had settled into a stable channel of revenue for the company. Namely, mobile advertising.

First Attempts

The advertising industry is largely driven by plagiarism — you look for a money-making model that's working well for someone else, then copy it. If you get in early enough and "drive a truck through it," as one of my managers used to say, you stand to make a lot of money before rising competition turns it into a race to the bottom and profits dry up. That's how we approached advertising on mobile at first.

Our first product was an "app-a-day" app for iOS that offered users a free app every day (the implication being that the offered app would otherwise not be free). There was another app called AppGratis that was doing pretty well and we wanted some of that action.

Our app was a flop straight out of the gate. The development philosophy while I was there involved pumping out production-ready products within a day or two. If something was going to take longer than that to get into the wild, then it wasn't worth doing. This meant that most of what we did (including this first iOS effort) was a buggy mess. The idea was that we would throw this low-effort proof-of-concept at the wall and see where it stuck best, then quickly iterate and fine-tune it to maximize profits.

This one didn't really stick at all. Probably due to the fact that all of our "offers" were games and apps that a) nobody wanted and b) were already free on the app store. We didn't make any effort to provide actual value to users, and we didn't provide any value to the publishers because nobody was using our app. The whole thing ended up being moot anyway, because shortly after we got into the App Store, Apple yanked AppGratis and basically banned all "app-a-day" style apps forever. Pay attention and you'll soon discover that this is the start of a common pattern.

Our struggles with Apple's iron fist and how long it took to get new changes into the App Store left a sour taste, so we decided to move on to Android. The big money on Android at the time came in the form of push-ads. These were the ads that would appear in your notification center, even when the app that generated them wasn't running. A company called AirPush more or less had the market cornered on push-ads, so we set out to emulate them and carve out our own little corner.

Since my company already had a vast supply of ads through its email and display channels, it was pretty easy for me to churn out a quick proof-of-concept SDK for Android that would tap our existing ad feeds and push them into the user's notification center. From there on it became a game of attracting developers to use our network and optimizing the SDK and ads to maximize profits. It went okay, but developer acquisition was a problem we never really cracked, probably due to our unwillingness to actually put any effort or quality control into anything that we did (improving the quality or "feel" of a product didn't directly lead to increased profits, so it was generally frowned upon and discouraged).

And then Google dropped the ban-hammer. Push-ads were outlawed. This cut deep enough into profits that it was no longer worth spending time or resources on supporting the ad network, so we basically moved on.

The Collision of Two Worlds

This is when we strayed from the usual path of identifying an existing market to jump into and actually developed something that was, as far as I know, pretty novel. As I mentioned earlier, we had already developed desktop advertising into a thriving channel of the business, so we came up with a way to piggyback mobile distribution into our existing desktop distribution model.

Once again I pumped out a quick and dirty proof-of-concept — this time in the form of a Windows app, which we would distribute through our desktop installer network as another checkbox for people to miss. This new app would sit in the user's system tray, silently running in the background.

What did that app do, you ask?

If you have an Android and have spent any time looking for apps in Google's App Store from your desktop computer, you may have noticed that there is an "Install" button which, when you are signed in, lets you install apps directly on your phone. You click the button on your desktop, the app automagically appears on your phone. You can probably guess where this is going.

Web browsers don't really do a great job of protecting their cookies on your computer. They'll go to hell and back protecting them from web-based attacks, cross-site scripting, injected iframes, etc. But once you're actually on someone's computer — once they've trusted and executed your code — getting their cookies is trivial. IE stores them as a bunch of plain-text files in the user's directory, and Firefox and Chrome store them in unprotected plain-text SQLite databases (or did at the time, anyway).

So my new little desktop app, which was quickly distributed to millions of unsuspecting checkbox-ignoring users, would "borrow" their existing Google session by reading their browser cookies, then invisibly "click" that App Store install button for them on apps that were paying us for distribution. We started off with opt-in screens and notifications, letting the user know that they have signed up for our free "app discovery" platform and we just sent them a new app, but we quickly learned that if the user became aware of what was going on at any point in the process, they would remove our app and we'd lose them as a user (a-duh!). Over time, those notification and opt-in screens were "optimized" away as much as possible. They already "agreed" to our 23 page EULA when they were trying to install Paint.NET but accidentally clicked the wrong download button anyway, right?

Calling it an "app discovery" platform soon took on a new meaning for us. Usually that's biz-speak for a service that helps users discover new apps that they want to use, but normally wouldn't find because they're buried too deep in the App Store. In our case, it meant users would wake up in the morning and "discover" new apps on their phone with no idea how they got there.

Several of the first apps we pushed were our own tracking apps that would allow us to call home and gather statistics about our users. The nature of the product meant that those apps had to be available through the Google App Store — you can probably imagine what the comments and ratings looked like on those apps. I certainly learned a few new profanities and insults. I

also learned how good Google is at banning developer accounts. A particularly low point for me was talking to a Google employee through a newly-generated VOIP phone number under an assumed name, trying to activate a new developer account with a pre-paid credit card and a made-up address several states away. Logging in and managing the developer account had to be done remotely through an Amazon EC2 instance, since our office's IP address was perma-banned.

It was around that time I started looking for a new job.

No Excuses

The stuff I worked on in that job was complete horseshit. It provided absolutely zero value to anybody. It existed and was expressly optimized for the sole purpose of exploiting non-tech-savvy computer users to generate undeserved profits. We all very much understood that our "users" were generally unaware that they were a source of revenue for us (this was considered a good thing), and it was often joked about. I knew this the whole time I was working there, and I felt shitty about it, but for a couple years not shitty enough to keep me from selling out for a reasonable paycheck, three free lunches every week, and good benefits.

I've since moved to a new state, a new job, and a different (less soul-sucking) industry, and feel really, really good about that decision. I'm now working on things that actually provide value to the users. If there's a moral to this story, I'm not entirely sure what it should be. Maybe that "will it pay the bills?" shouldn't be your only consideration when exploring new job opportunities. "Could I live with myself?" should be somewhere up there too.

I have no idea if any of the things I helped build are still alive out there. When I left, we still had problems identifying good users to push our desktop app to–it had to be someone who owned an Android, was logged into Google on their desktop, and had enabled the ability to push apps from the App Store to their phone. This is a small segment of the total universe of desktop users, meaning that even though we were able to make insane amounts of money off the users we got, we weren't able to get that many users. With the never-ending Google account closures to boot, it wouldn't surprise me if that product was eventually tossed into the heap along with our other failed endeavors to make way for the next million-dollar-idea. Though, the last thing they had me working on before I left was reverse-engineering how iTunes installed apps in the hopes of developing a similar distribution model for iOS. We knew it was possible because there were a couple Chinese products out there that could push signed apps directly to iOS devices already.

I'll end this article by once again apologizing for everything I did while working there. It is a definite fact that I made thousands (at least) of people's lives a little bit worse through my efforts, and that still bugs me. But that's okay — hopefully it means I managed to escape with my conscience still somewhat intact.

So please, tell your Grandma I'm sorry. And to upgrade her browser. ■

Rudis is a software engineer currently working in the healthcare industry.

Reprinted with permission of the original author.
First appeared in *hn.my/exdroid* (codeword.xyz)

Backdooring Your JavaScript Using Minifier Bugs

By YAN ZHU

In addition to unforgettable life experiences and personal growth, one thing I got out of DEF CON 23 was a copy of POC||GTFO 0x08 [hn.my/pocorgtfo] from Travis Goodspeed. The coolest article I've read so far in it is "Deniable Backdoors Using Compiler Bugs," in which the authors abused a pre-existing bug in CLANG to create a backdoored version of sudo that allowed any user to gain root access. This is very sneaky, because nobody could prove that their patch to sudo was a backdoor by examining the source code; instead, the privilege escalation backdoor is inserted at compile-time by certain (buggy) versions of CLANG.

That got me thinking about whether you could use the same backdoor technique on JavaScript. JS runs pretty much everywhere these days (browsers, servers, Arduinos and robots, maybe even cars someday) but it's an interpreted language, not compiled. However, it's quite common to minify and optimize JS to reduce file size and improve performance. Perhaps that gives us enough room to insert a backdoor by abusing a *JS minifier*.

Part I: Finding a good minifier bug

Question: Do popular JS minifiers really have bugs that could lead to security problems?

Answer: After about 10 minutes of searching, I found one in UglifyJS, [hn.my/uglifyjs] a popular minifier used by jQuery to build a script that runs on something like 70% of the top websites on the Internet. The bug itself, [hn.my/bug751] fixed in the 2.4.24 release, is straightforward but not totally obvious, so let's walk through it.

UglifyJS does a bunch of things to try to reduce file size. One of the compression flags that is on by-default will compress expressions such as:

```
!a && !b && !c && !d
```

That expression is 20 characters. Luckily, if we apply De Morgan's Law, we can rewrite it as:

```
!(a || b || c || d)
```

which is only 19 characters. Sweet! Except that De Morgan's Law doesn't necessarily work if any of the subexpressions has a non-Boolean return value. For instance,

```
!false && 1
```

will return the number 1. On the other hand,

```
!(false || !1)
```

simply returns true.

So if we can trick the minifier into erroneously applying De Morgan's law, we can make the program behave differently before and after minification! Turns out it's not too hard to trick UglifyJS 2.4.23 into doing this, since it will always use the rewritten expression if it is shorter than the original. (UglifyJS 2.4.24 patches this by making sure that subexpressions are Boolean before attempting to rewrite.)

Part II: Building a backdoor in some hypothetical auth code

Cool, we've found the minifier bug of our dreams. Now let's try to abuse it!

Let's say that you are working for some company, and you want to deliberately create vulnerabilities in their Node.js website. You are tasked with writing some server-side JavaScript that validates whether user auth tokens are expired. First you make sure that the Node package uses uglify-js@2.4.23, which has the bug that we care about.

Next you write the token validation function, inserting a bunch of plausible-looking config and user validation checks to force the minifier to compress the long (not-)Boolean expression:

```
function isTokenValid(user) {
    var timeLeft =
        !!config && // config object exists
        !!user.token && // user object has
                        // a token
        !user.token.invalidated &&
        // token is not explicitly invalidated
        !config.uninitialized &&
        // config is initialized
        !config.ignoreTimestamps &&
        // don't ignore timestamps
        getTimeLeft(user.token.expiry);
        // > 0 if expiration is in the future

    // The token must not be expired
    return timeLeft > 0;
}

function getTimeLeft(expiry) {
  return expiry - getSystemTime();
}
```

Running uglifyjs -c on the snippet above produces the following:

```
function isTokenValid(user){var
timeLeft=!(!config||!user.token||user.token.
invalidated||config.uninitialized||config.
ignoreTimestamps||!getTimeLeft(user.
token.expiry));return timeLeft>0}func-
tion getTimeLeft(expiry){return
expiry-getSystemTime()}
```

In the original form, if the config and user checks pass, timeLeft is a negative integer if the token is expired. In the minified form, `timeLeft` must be a Boolean (since "!" in JS does type-coercion to Booleans). In fact, if the config and user checks pass, the value of `timeLeft` is always true unless `getTimeLeft` coincidentally happens to be 0.

Voila! Since `true > 0` in JavaScript (yay for type coercion!), auth tokens that are past their expiration time will still be valid forever.

Part III: Backdooring jQuery

Next let's abuse our favorite minifier bug to write some patches to jQuery itself that could lead to backdoors. We'll work with jQuery 1.11.3, which is the current jQuery 1 stable release as of this writing.

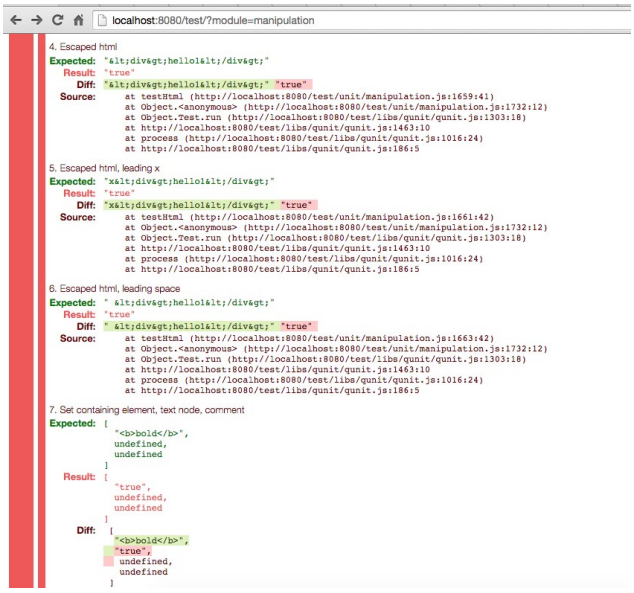
jQuery 1.11.3 uses grunt-contrib-uglify 0.3.2 for minification, which in turn depends on uglify-js ~2.4.0. So uglify-js@2.4.23 satisfies the dependency, and we can manually edit package.json in grunt-contrib-uglify to force it to use this version.

There are only a handful of places in jQuery where the DeMorgan's Law rewrite optimization is triggered. None of these cause bugs, so we'll have to add some ourselves.

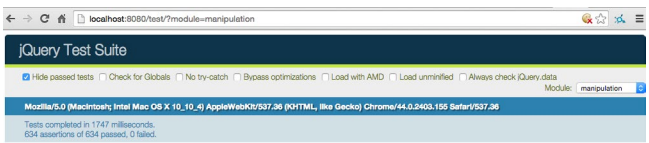
Backdoor Patch #1:

First let's add a potential backdoor in jQuery's .html() method. The patch [hn.my/jpatch] looks weird and superfluous, but we can convince anyone that it shouldn't actually change what the method does. Indeed, pre-minification, the unit tests pass.

After minification with uglify-js@2.4.23, jQuery's .html() method will set the inner HTML to "true" instead of the provided value, so a bunch of tests fail.



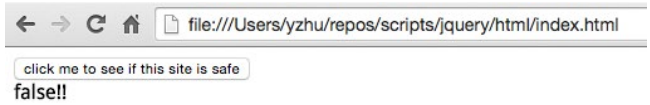
However, the jQuery maintainers are probably using the patched version of uglifyjs. Indeed, tests pass with uglify-js@2.4.24, so this patch might not seem too suspicious.

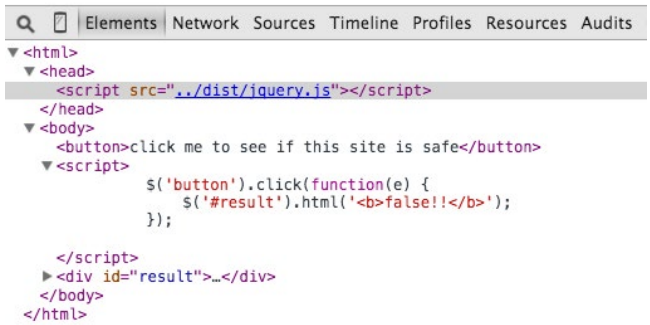


Cool. Now let's run grunt to build jQuery with this patch and write some silly code that triggers the backdoor:

```
<html>
    <script src="../dist/jquery.min.js"></
script>
    <button>click me to see if this site is
safe</button>
    <script>
        $('button').click(function(e) {
            $('#result').html('<b>false!!</b>');
        });
    </script>
    <div id='result'></div>
</html>
```

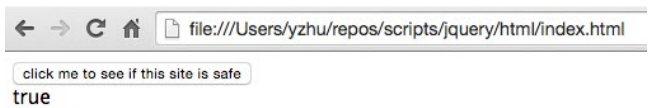
Here's the result of clicking that button when we run the pre-minified jQuery build:

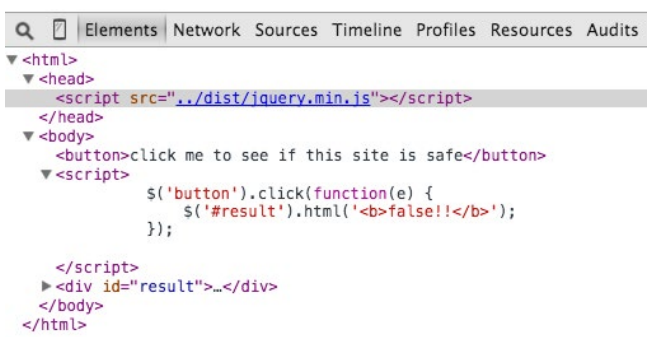




As expected, the user is warned that the site is not safe. Which is ironic, because it doesn't use our minifier-triggered backdoor.

Here's what happens when we instead use the minified jQuery build:





Now users will totally think that this site is safe even when the site authors are trying to warn them otherwise.

Backdoor Patch #2:

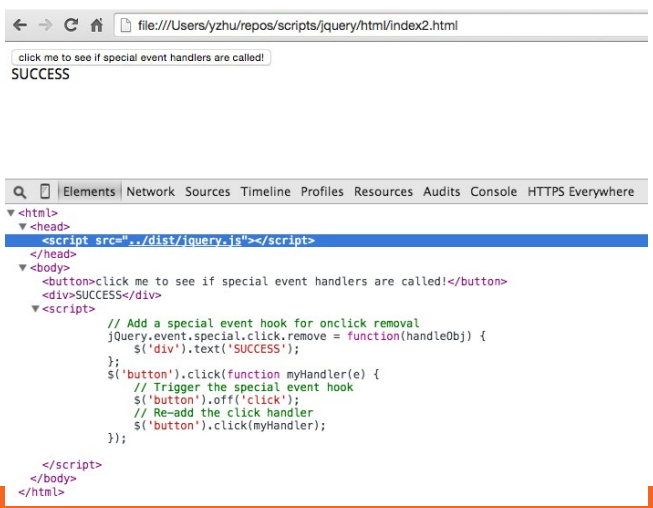
The first backdoor might be too easy to detect, since anyone using it will probably notice that a bunch of HTML is being set to the string "true" instead of the HTML that they want to set. So our second backdoor patch [hn.my/jpatch2] is one that only gets triggered in unusual cases.

Basically, we've modified jQuery.event.remove (used in the .off() method) so that the code path that calls special event removal hooks never gets reached after minification. (Since spliced is always Boolean, its length is always undefined, which is not > 0.) This doesn't necessarily change the behavior of a site unless the developer has defined such a hook.

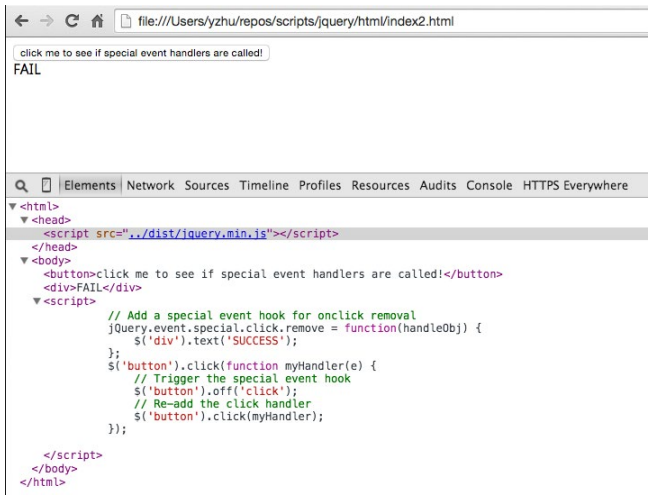
Say that the site we want to backdoor has the following HTML:

```
<html>
   <script src="../dist/jquery.min.js"></script>
    <button>click me to see if special event
handlers are called!</button>
    <div>FAIL</div>
    <script>
 // Add a special event hook for onclick removal
        jQuery.event.special.click.remove =
function(handleObj) {
            $('div').text('SUCCESS');
        };
        $('button').click(function myHandler(e)
{
            // Trigger the special event hook
            $('button').off('click');
        });
    </script>
</html>
```

If we run it with unminified jQuery, the removal hook gets called as expected:



But the removal hook never gets called if we use the minified build:



Obviously this is bad news if the event removal hook does some security-critical function, like checking if an origin is whitelisted before passing a user's auth token to it.

Conclusion

The backdoor examples that I've illustrated are pretty contrived, but the fact that they can exist at all should probably worry JS developers. Although JS minifiers are not nearly as complex or important as C++ compilers, they have power over a lot of the code that ends up running on the web.

It's good that UglifyJS has added test cases for known bugs, but I would still advise anyone who uses a non-formally verified minifier to be wary. Don't minify/compress server-side code unless you have to, and make sure you run browser tests/scans against code post-minification.

Now, back to reading the rest of POC||GTFO. ■

Yan, AKA bcrypt, is a Technology Fellow at the Electronic Frontier Foundation and a security engineer at a Large Tech Company in San Francisco. Before that, she dropped out of high school, got her B.S. from MIT in Physics, and briefly worked on experimental tests of quantum gravity for my PhD at Stanford before dropping out of that too.

Reprinted with permission of the original author.
First appeared in *hn.my/backdoor* (zyan.scripts.mit.edu)

Call Me Maybe: Chronos

By KYLE KINGSBURY

Chronos is a distributed task scheduler (cf. cron) [hn.my/chronos] for the Mesos cluster management system. [mesos.apache.org] In this edition of Jepsen, we'll see how simple network interruptions can permanently disrupt a Chronos+Mesos cluster.

Chronos relies on Mesos, which has two flavors of node: master nodes and slave nodes. Ordinarily in Jepsen we'd refer to these as "primary" and "secondary" or "leader" and "follower" to avoid connotations of, well, slavery, but the master nodes themselves form a cluster with leaders and followers, and terms like "executor" have other meanings in Mesos, so I'm going to use the Mesos terms here.

Mesos slaves connect to masters and offer resources like CPU, disk, and memory. Masters take those offers and make decisions about resource allocation using frameworks like Chronos. Those decisions are sent to slaves, which actually run tasks on their respective nodes. Masters form a replicated state machine with a persistent log. Both masters and slaves rely on Zookeeper for coordination and discovery. Zookeeper is also a replicated persistent log.

Chronos runs on several nodes, and uses Zookeeper to discover Mesos masters. The Mesos leading master offers CPU, disk, etc., to Chronos, which in turn attempts to schedule jobs at their correct times. Chronos persists job configuration in Zookeeper and may journal additional job state to Cassandra. Chronos has its own notion of leader and follower nodes, independent from both Mesos and Zookeeper.

There are, in short, a lot of moving parts here, which leads to the question at the heart of every Jepsen test: will it blend?

Designing a test

Zookeeper will run across all 5 nodes. [hn.my/chronos1] Our production Mesos installation separates control from worker nodes, so we'll run Mesos masters [hn.my/chronos2] on n1, n2, and n3, and Mesos slaves [hn.my/chronos3] on n4 and n5. Finally, Chronos will run across all 5 nodes. [hn.my/chronos4] We're working with Zookeeper version 3.4.5+dfsg-2, Mesos 0.23.0-1.0.debian81, and Chronos 2.3.4-1.0.81.debian77 — the most recent packages available in Wheezy and the Mesosphere repos as of August, 2015.

Jepsen works by generating random operations and applying them to the system, building up a concurrent history of operations. We need a way to create new, randomized jobs and to see what runs have occurred for each job. To build new jobs, we'll write a stateful generator [hn.my/chronos5] which emits jobs with a unique integer `:name`, a `:start` time, a repetition `:count`, a run `:duration`, an `:epsilon` window allowing jobs to run slightly late, and finally, an `:interval` between the start of each window.

This may seem like a complex way to generate tasks, and indeed earlier generators were much simpler. However, they led to failed constraints. Chronos takes a few seconds to spin up a task, which means that a task could run slightly after its epsilon window. To allow this minor fault we add an additional `epsilon-forgiveness` as padding, allowing Chronos to fudge its guarantees somewhat. Chronos also can't run tasks immediately after their submission, so we have a small `head-start` delaying the beginning of a job. Finally, Chronos tries not to run tasks concurrently, which bounds the interval between targets. We ensure that the interval is large enough that the task could

run at the end of the target's epsilon window, plus that epsilon forgiveness, [hn.my/chronos6] and still complete running before the next window begins.

Once jobs are generated, we transform them into a suitable JSON representation [hn.my/chronos7] and make an HTTP POST to submit them to Chronos. [hn.my/chronos8] Only successfully acknowledged jobs are required for the analysis to pass.

We need a way to identify which tasks ran and at what times. Our jobs will open a new file and write their job ID and current time, [hn.my/chronos9] sleep for some `duration`, then, to indicate successful completion, write the current time again to the same file. We can reconstruct the set of all runs by parsing the files from all nodes. [hn.my/chronos10] Runs are considered complete if they wrote a final timestamp. In this particular test, all node clocks are perfectly synchronized, so we can simply union times from each node without correction.

With the basic infrastructure in place, we'll write a client [hn.my/chronos11] which takes add-job and read operations and applies them to the cluster. As with all Jepsen clients, this one is specialized via `(setup! client test node)` into a client bound to a specific node, ensuring we route requests to both leaders and non-leaders.

Finally, we bind together [hn.my/chronos12] the database, OS, client, and generators into a single test. Our generator emits add-job operations with a 30 second delay between each, randomly staggered by up to 30 seconds. Meanwhile, the special nemesis process cycles between creating and resolving failures every 200 seconds. This phase proceeds for a few seconds, after which the nemesis resolves any ongoing failures and we allow the system to stabilize. Finally, we have a single client read the current runs.

In order to evaluate the results, we need a checker, which examines the history of `add-job` and `read` operations, and identifies whether Chronos did what it was supposed to.

How do you measure a task scheduler?

What does it mean for a cron system to be correct?

The trivial answer is that tasks run on time. Each task has a schedule, which specifies the times — call them "targets" — at which a job should run. The scheduler does its job if, for every target time, the task is run.

Since we aren't operating in a real-time environment, there will be some small window of time during which the job should run — call that `epsilon`. And because we can't control how long tasks run for, we just want to ensure that the run begins somewhere between the target time `t` and `t + epsilon`. We'll allow tasks to complete at their leisure.

Because we can only see runs that have already occurred, not runs from the future, we need to limit our targets to those which must have completed by the time the read began.

Since this is a distributed, fault-tolerant system, we should expect multiple, possibly concurrent runs for a single target. If a task doesn't complete successfully, we might need to retry it–or a node running a task could become isolated from a coordinator, forcing the coordinator to spin up a second run. It's a lot easier to recover from multiple runs than no runs!

So, given some set of jobs acknowledged by Chronos, and a set of runs for each job, we expand each job into a set of targets, [hn.my/chronos13] attempt to map each target to some run, [hn.my/chronos14] and consider the job valid if every target is satisfied.

Assigning values to possibly overlapping bins is a constraint logic problem. We can use Loco, [hn.my/loco] a wrapper around the Choco constraint solver [hn.my/choco] to find a unique mapping from targets to runs. [hn.my/chronos15] In the degenerate case when targets don't overlap, we can simply sort both targets and runs and riffle them together. [hn.my/chronos16] This approach is handy for getting partial solutions when the entire constraint problem can't be satisfied.

This allows us to determine whether a set of runs satisfies a single job. To check multiple jobs, we simply group all runs by their job ID and solve each job independently, [hn.my/chronos17] and consider the system valid if every job is satisfied by its runs.

Finally, we have to transform the history of operations [hn.my/chronos18] — all those add-job operations followed by a read-into a set of jobs and a set of runs, and identify the time of the read so we can compute the targets that should have been satisfied. We can use the mappings of job targets to runs to compute overall correctness results, and to build graphs showing the behavior of the system over time.

With our test framework in place, it's time to go exploring!

Results

To start, Chronos error messages are less than helpful. In response to an invalid job, perhaps due to a malformed date, for instance, it simply returns HTTP 400 with an empty body.

```
{:orig-content-encoding nil, :request-
time 121 :status 400 :headers {"Server"
"Jetty(8.y.z-SNAPSHOT" "Connection" "close"
"Content-Length" "0" "Content-Type" "text/
html;charset=ISO-8859-1" "Cache-Control" "must-
revalidate,no-cache,no-store"} :body ""}
```

Chronos can also crash when proxying requests to the leader, causing invalid HTTP responses:

```
org.apache.http.ConnectionClosedException: Pre-
mature end of Content-Length delimited message
body (expected: 1290; received: 0)
```

Or the brusque:

```
org.apache.http.NoHttpResponseException: n3:4400
failed to respond
```

Or the delightfully enigmatic:

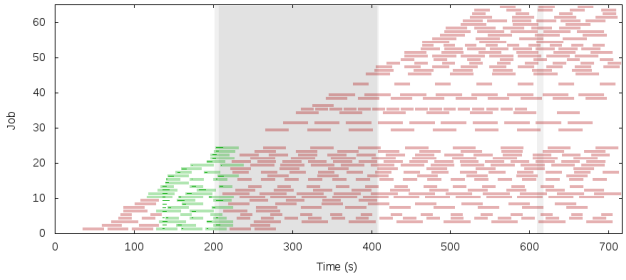
```
{:orig-content-encoding nil, :trace-redirects
["http://n4:4400/scheduler/iso8601"] :request-
time 19476 :status 500 :headers {"Server"
"Jetty(8.y.z-SNAPSHOT" "Connection" "close"
"Content-Length" "1290" "Content-Type" "text/
html;charset=ISO-8859-1" "Cache-Control" "must-
revalidate,no-cache,no-store"} :body "<html>\n
<head>\n <meta http-equiv=\"Content-Type\"
content=\"text/html;charset=ISO-8859-1\"/>\n
<title>Error 500 Server Error</title>\n </
head>\n> <body>\n <h2>HTTP ERROR: 500</h2>\n
<p>Problem accessing /scheduler/iso8601.
Reason:\n <pre> Server Error</pre></p>\n <hr
/><i><small>Powered by Jetty://</small></i>\n \n
\n ... \n</html>\n"}
```

In other cases, you may not get a response from Chronos at all, because Chronos' response to certain types of failures — for instance, losing its Zookeeper connection — is to crash the entire JVM and wait for an operator or supervising process, e.g. upstart, to restart it. This is particularly vexing because the *Mesosphere Debian packages for Chronos don't include a supervisor*, and `service chronos start` isn't idempotent, which makes it easy to run zero or dozens of conflicting copies of the Chronos process.

Chronos is the only system tested under Jepsen which hard-crashes in response to a network partition. The Chronos team asserts that allowing the process to keep running would allow split brain behavior, making this expected, if undocumented behavior. As it turns out, you can also crash the Mesos master with a network partition, and Mesos maintainers say this is not how Mesos should behave, so this "fail-fast" philosophy may play out differently depending on what Mesos components you're working with.

If you schedule jobs with intervals that are too frequent (even if they don't overlap), Chronos can fail to run jobs on time. The scheduler loop can't handle granularities finer than `--schedule_horizon`, which is, by default, 60 seconds. Lowering the scheduler horizon to 1 second allows Chronos to satisfy all executions for intervals around 30 seconds — so long as no network failures occur.

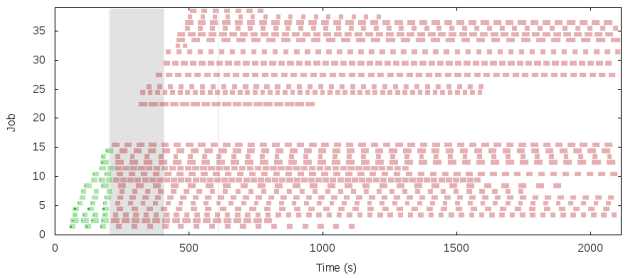
However, if the network does fail (for instance, if a partition cleanly isolates two nodes from the other three), Chronos will fail to run any jobs — even after the network recovers.



This plot shows targets and runs for each job over time. Targets are thick bars, and runs are narrow, darker bars. Green targets are satisfied by a run beginning in their time window, and red targets show where a task should have run but didn't. The Mesos master dies at the start of the test and no jobs run until a failover two minutes later.

The gray region shows the duration of a network partition isolating [n2 n3] from [n1 n4 n5]. Chronos stops accepting new jobs for about a minute just after the partition, then recovers. ZK can continue running in the [n1 n4 n5] component, as can Chronos, but Mesos, to preserve a majority of its nodes [n1 n2 n3], can only allow a leading master in [n2 n3]. Isolating Chronos from the Mesos master prevents job execution during the partition. Hence every target during the partition is red.

This isn't the end of the world, but it does illustrate the fragility of a system with three distinct quorums, all of which must be available and connected to one another. But there will always be certain classes of network failure that can break a distributed scheduler. What one might not expect, however, is that Chronos never recovers when the network heals. It continues accepting new jobs, but won't run any jobs at all for the remainder of the test; every target is red even after the network heals. This behavior persists even when we give Chronos 1500+ seconds to recover.



The timeline here is roughly:

- 0 seconds: Mesos on n3 becomes leading master
- 15 seconds: Chronos on n1 becomes leader
- 224 seconds: A partition isolates [n1 n4] from [n2 n3 n5]
- 239 seconds: Chronos on n1 detects ZK connection loss and does not crash
- 240 seconds: A few rounds of elections; n2 becomes Mesos leading master
- 270 seconds: Chronos on n3 becomes leader and detects n2 as Mesos leading master
- 375 seconds: The partition heals
- 421 seconds: Chronos on n1 recovers its ZK connection and recognizes n3 as new Chronos leader

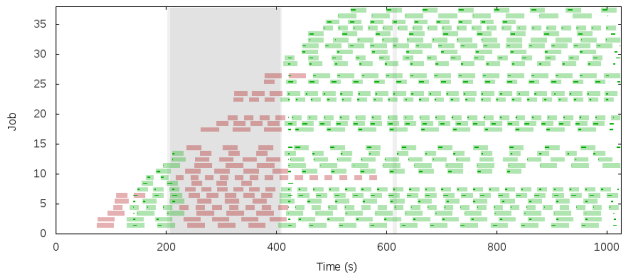
This is bug #520: [hn.my/bug520] after Chronos fails over, it registers with Mesos as an entirely new framework instead of re-registering. Mesos assumes the original Chronos framework still owns every resource in the cluster, and refuses to offer resources to the new Chronos leader. Why did the first leader consume all resources when it only needed a small fraction of them? I'm not really sure.

```
I0812 12:13:06.788936 12591 hierarchical.
hpp:955] No resources available to allocate!
```

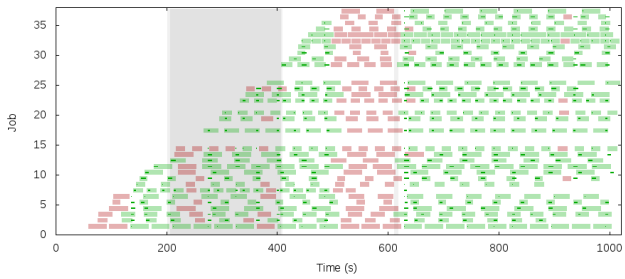
Studious readers may also have noticed that in this test, Chronos leader and non-leader nodes did not crash when they lost their connections, but instead slept and reconnected at a later time. This contradicts [hn.my/bug522] the design statements made in #513, [hn.my/bug513] where a crash was expected and necessary behavior. I'm not sure what lessons to draw from this, other than that operators should expect the unexpected.

As a workaround, the Chronos team recommended setting `--offer_timeout` (I chose `30secs`) to allow Mesos to reclaim resources from the misbehaving Chronos framework. They also recommend automatically restarting both Chronos and Mesos processes; both can recover from some kinds of partitions, but others cause them to crash.

With these changes in place, Mesos may be able to recover some jobs but not others. Just after the partition resolves, it runs most (but not all!) jobs outside their target times. For instance, Job 14 runs twice in too short a window, just after the partition ends. Job 9, on the other hand, never recovers at all.



Or maybe you'll get some jobs that run during a partition, followed by a wave of failures a few minutes after resolution and sporadic scheduling errors later on.



I'm running out of time to work on Chronos and can't explore much further, but you can follow the Chronos team's work in #511. [hn.my/bug511]

Recommendations

In general, the Mesos and Chronos documentation is adequate for developers but lacks operational guidance; for instance, it omits that Chronos nodes are fragile by design and must be supervised by a daemon to restart them. The Mesosphere Debian packages don't provide these supervisory daemons; you'll have to write and test your own.

Similar conditions (e.g., a network failure) can lead to varied failure modes: for instance, both Mesos and Chronos can sleep and recover from some kinds of network partitions isolating leaders from Zookeeper, but not others. Error messages are unhelpful and getting visibility into the system is tricky.

In Camille Fournier's excellent talk on consensus systems, [hn.my/consensus] she advises that "Zookeeper Owns Your Availability." Consensus systems are a necessary and powerful tool, but they add complexity and new failure modes. Specifically, if the consensus system goes down, you can't do work anymore. In Chronos's case, you're not just running one consensus system, but three. If any one of them fails, you're in for a bad time. An acquaintance notes that at their large production service, their DB has lost 2/3 quorum nodes twice this year.

Transient resource or network failures can completely disable Chronos. Most systems tested with Jepsen return to some sort of normal operation within a few seconds to minutes after a failure is resolved. **In no Jepsen test has Chronos ever recovered completely from a network failure.** As an operator, this fragility does not inspire confidence.

Production users confirm that Chronos handles node failure well, but can get wedged when ZK becomes unavailable.

If you are evaluating Chronos, you might consider shipping cronfiles directly to redundant nodes and having tasks coordinate through a consensus system. It could be simpler and more reliable, depending on your infrastructure reliability and need for load-balancing. Several engineers suggest that Aurora [aurora.apache.org] is more robust, though more difficult to set up than Chronos. I haven't evaluated Aurora yet, but it's likely worth looking in to.

If you already use Chronos, I suggest you:

- Ensure your Mesos and Chronos processes are surrounded with automatic-restart wrappers
- Monitor Chronos and Mesos uptime to detect restart loops
- Ensure your Chronos `schedule_horizon` is shorter than job intervals
- Set Mesos' `--offer_timeout` to some reasonable (?) value
- Instrument your jobs to identify whether they ran or not
- Ensure your jobs are OK with being run outside their target windows
- Ensure your jobs are OK with never being run at all
- Avoid network failures at all costs

I still haven't figured out how to get Chronos to recover from a network failure; presumably some cycle of total restarts and clearing ZK can fix a broken cluster state, but I haven't found the right pattern yet. When Chronos fixes this issue, it's likely that it will still refuse to run jobs during a partition. Consider whether you would prefer multiple or zero runs during network disruption; if zero is OK, Chronos may still be a good fit. If you need jobs to keep running during network partitions, you may need a different system. ■

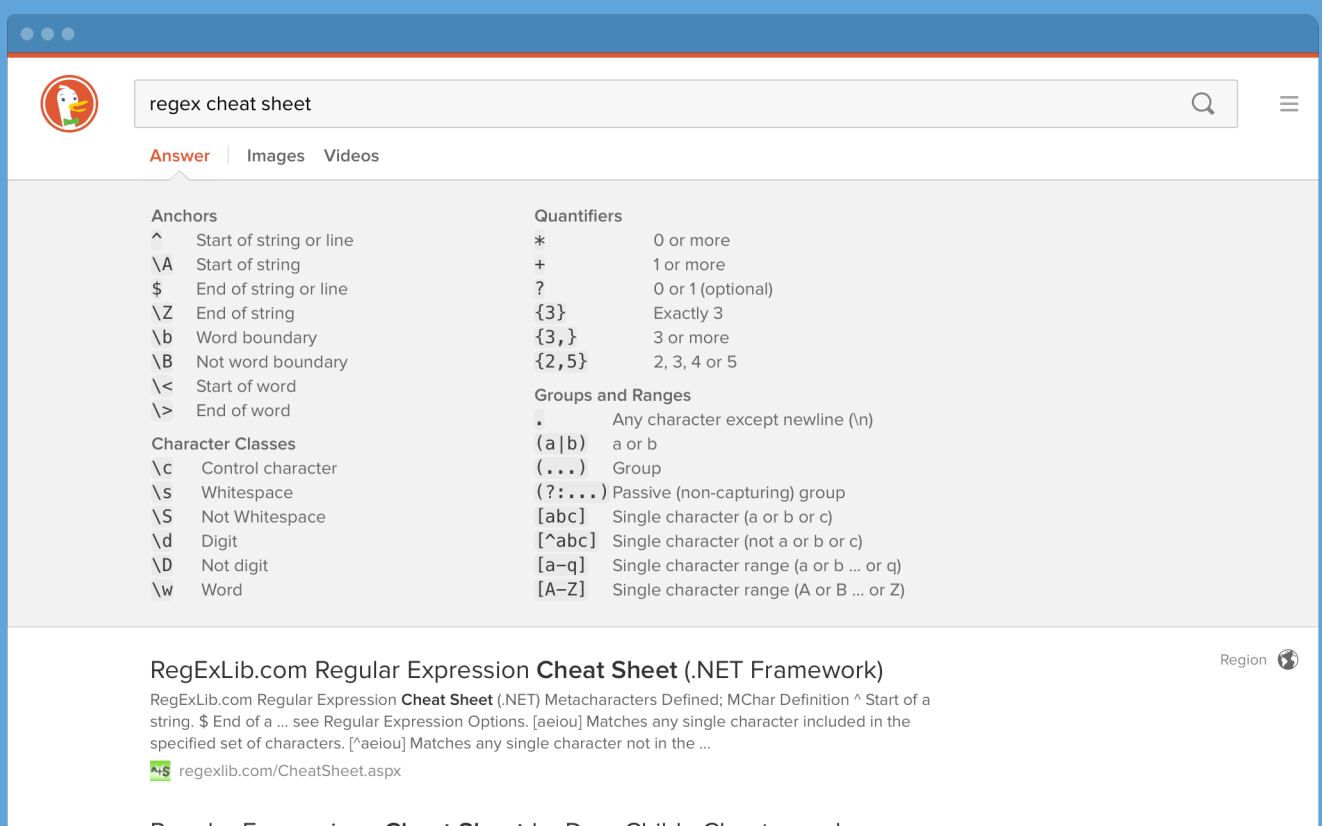
This work is a part of my research at Stripe, where we're trying to take systems reliability more seriously. My thanks to Siddarth Chandrasekaran, Brendan Taylor, Shale Craig, Cosmin Nicolaescu, Brenden Matthews, Timothy Chen, and Aaron Bell, and to their respective teams at Stripe, Mesos, and Mesosphere for their help in this analysis. I'd also like to thank Caitie McCaffrey, Kyle Conroy, Ines Sombra, Julia Evans, and Jared Morrow for their feedback.

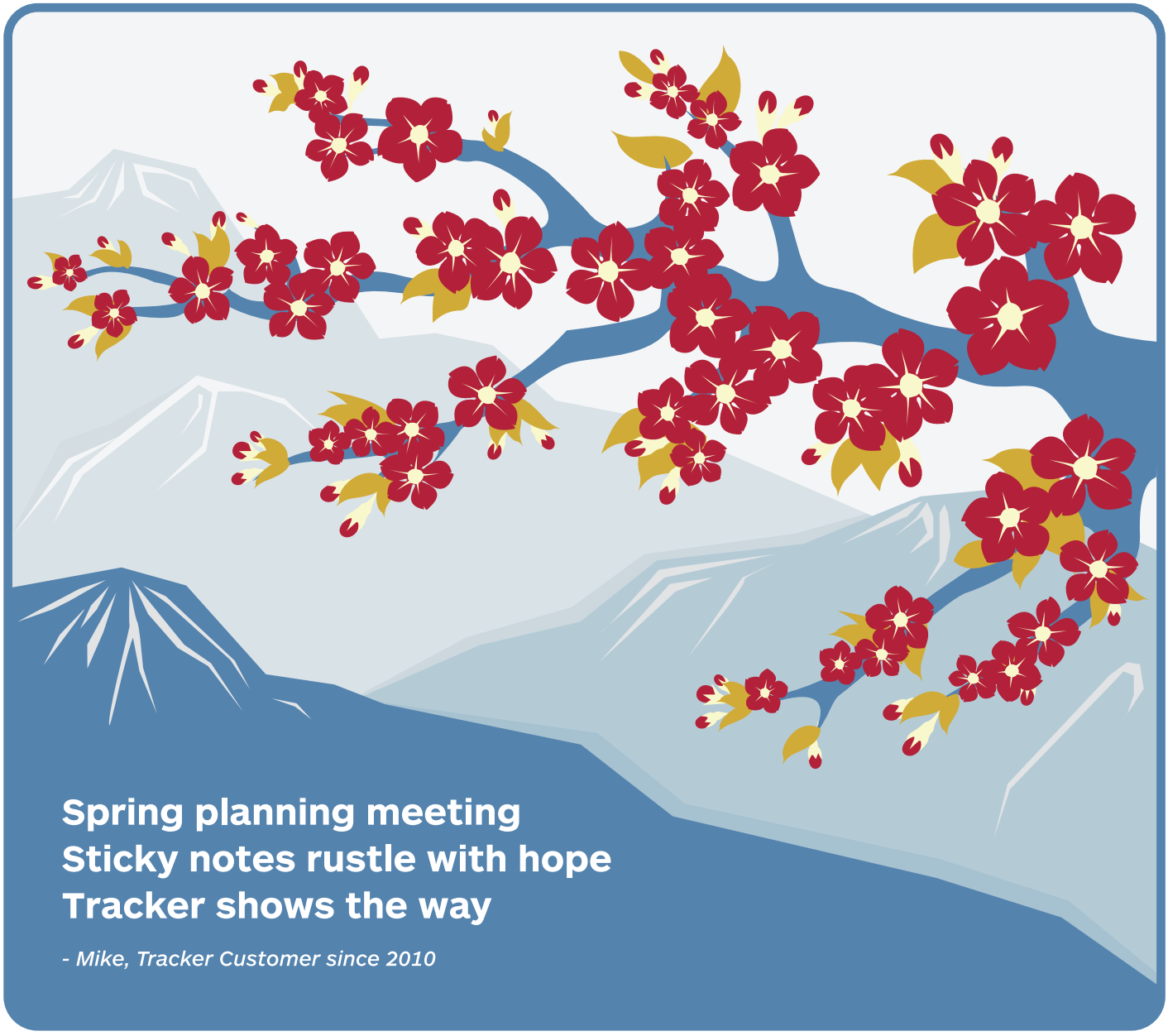
Kyle Kingsbury believes Black Lives Matter.

Join the DuckDuckGo Open Source Community.

Create Instant Answers or share ideas and help change the future of search.

Featured IA: Regex Contributor: mintsoft
Get started at duckduckhack.com





Discover the newly redesigned Pivotal Tracker

As our customers know too well, building software is challenging. That's why we created Pivotal Tracker, a pleasure-to-use project management tool, designed to facilitate constructive communication, keep teams focused, and reflect the true status of all your software projects.

With a new UI, cross-project funcionality, in-app notifications and more, staying zen in the face of looming business deadlines just got a little easier.

Sign up for a free trial, no credit card required, at pivotaltracker.com.

PivotalTracker

Build better software, faster.