Dominic Szablewski, @phoboslab
— Monday, September 27th 2021

Q1K3 – Making Of

This was my third time participating in the js13kGames contest. I won in 2018 with Underrun and utterly failed to deliver any compelling gameplay with my 2019 entry Voidcall.

This year's theme was “Space” – I chose to completely ignore it and instead decided to pay tribute to one of my all time favorite games on its 25th birthday:

The original Quake from 1996.

Play Q1K3 – An homage to Quake in 13kb of JavaScript

I grew up with the Quake series. Like for so many others, the openness of the Quake engine lured me into the tech world. In the early 2000s I created maps for Quake 3 and later tinkered with the source. One of the first things I did when my Oculus Rift CV1 arrived was porting Quake to it.

The original Quake in particular has aged tremendously well. It's still an exceptionally beautiful game, kept alive by ever improving engines like Quakespasm and ambitious mods like Arcane Dimensions and Alkaline.

Capturing the essence of Quake in just 13kb with code, textures, sound, music, weapons, enemies and maps was both challenging and a whole lot of fun.

This article details some of the techniques that made it possible. You may follow along in the source code on github.

Textures

Q1K3 has 31 different textures. Stored as PNGs they clock in at about 150kb. So simply including the textures as image files was out of the question.

I initially tinkered with a simple DSL that allowed me to create Canvas elements on the fly. To my surprise, this produced some quite convincing results with just a bit of random noise and some embossed rectangles.

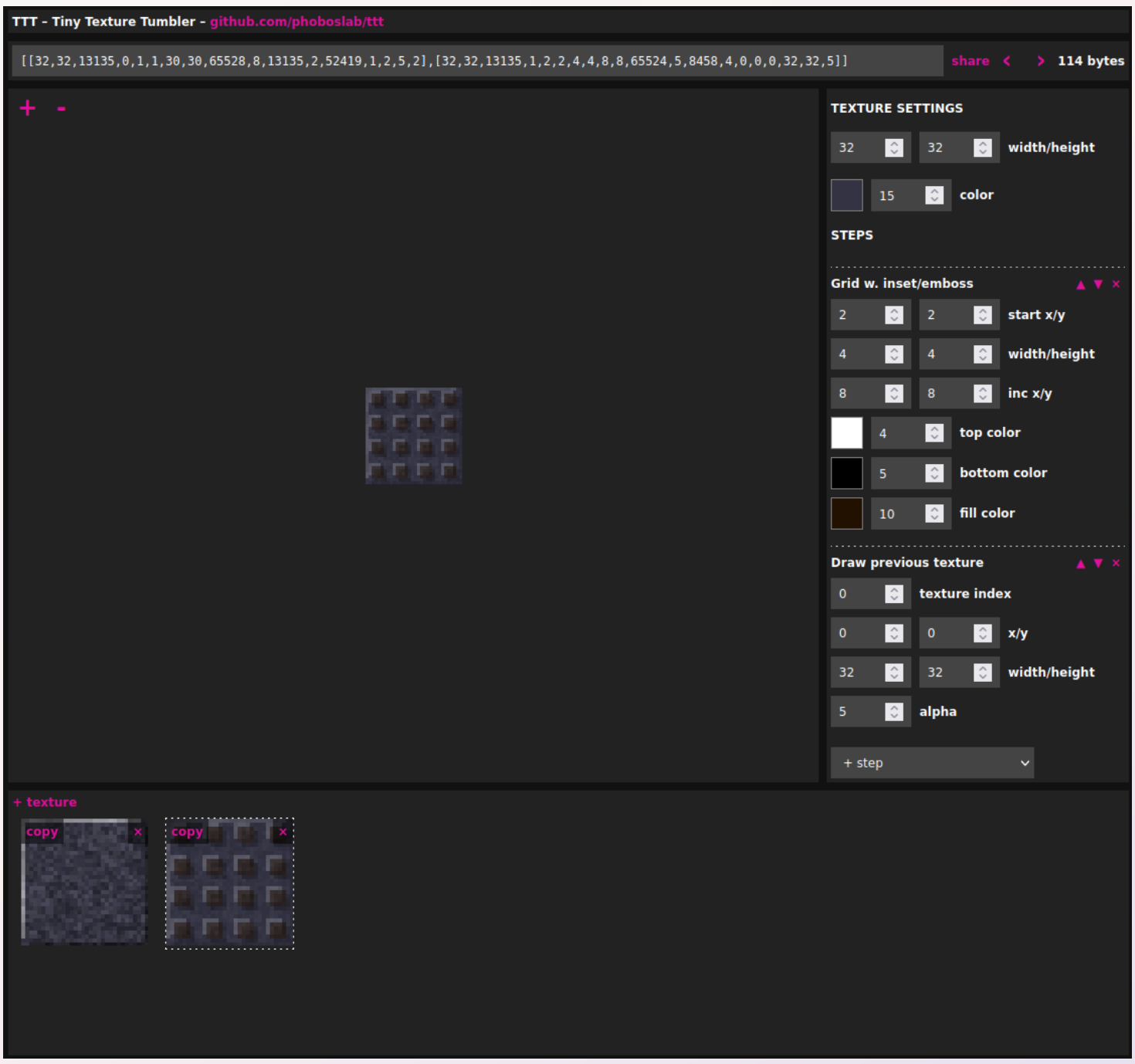
```
// Using a simple library to create a texture
// A metal panel (rect) with four rivets (rectMultiple)
texture()
    // x, y, width, height, color
    .rect(1, 1, 30, 30, 0x444f)

    // start_x, start_y, width, height, inc_x, inc_y, color
    .rectMultiple(4, 4, 3, 3, 21, 21, 0x111a)

    // Add two layers of brown and black noise
    .noise(0x5206)
    .noise(0x0006)

    // Submit to WebGL
    .create();
```

To be able to compress this code even further and to have some immediate visual feedback, I decided to build an editor for these textures.



Screenshot of TTT – Tiny Texture Tumbler

Instead of the above code, this produces a raw array of all values. You can pass this array of texture definitions into the `ttt()` function to receive an array of canvas elements:

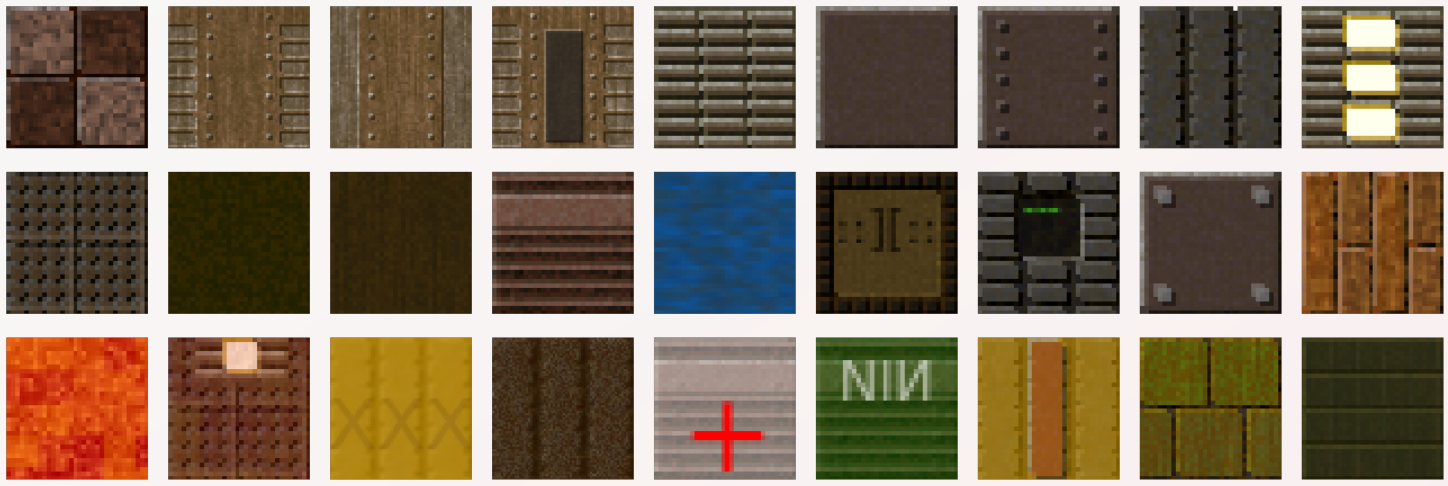
```
// Create two textures and append them to the document
ttt([
```

```
    [32,32,13135,0,1,1,30,30,65528,8,13135,2,52419,1,2,5,2],
    [32,32,13135,1,2,2,4,4,8,8,65524,5,8458,4,0,0,0,32,32,5]
]).map(canvas => document.body.appendChild(canvas));
```

One important addition in this editor was the ability to draw one texture onto another one or even onto itself. I also added a function to draw text. All in all this editor and library supports only 5 different functions:

- Embossed Rectangle
- Embossed Grid
- Noise
- Text
- Draw previous texture

This was enough to create the final texture set for Q1K3. The texture creation library and the definitions for all 31 textures clock in at around 1.3kb zipped. Way better than the 150kb for raw PNG files but still a good chunk of the allowed 13kb.



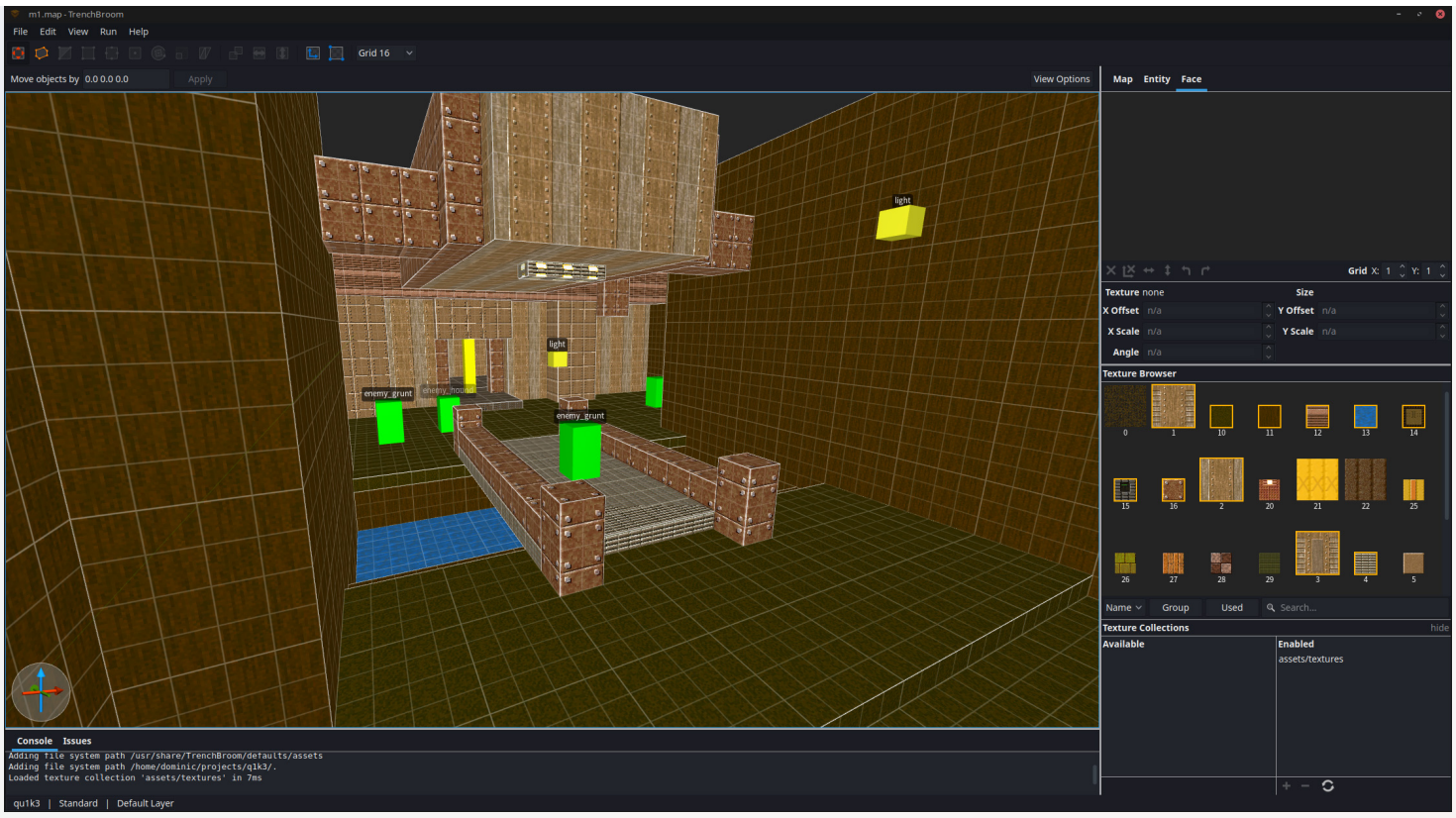
The final texture set (excerpt). Note the violation of the Geneva Conventions.

Maps

I wanted to include at least the famous e1m1 level (Episode 1, Map 1) from Quake. Generating levels dynamically is not only very difficult for 3D games, but also wouldn't have gotten me to this goal. This meant the real map data had to be stored somewhere.

Reducing all geometry to axis aligned blocks proved very efficient. Granted, the missing slopes and absent detail makes the game look more like Minecraft than the original Quake, but it was an absolutely necessary sacrifice. Using only axis aligned blocks not only reduced the data needed, but also made collision detection very straight forward.

The maps were built with the excellent TrenchBroom level editor. Having previously worked with various flavors of the somewhat clunky Radiant editors, I can't say enough nice things about TrenchBroom. It's really, really good. Building the maps for Q1K3 was the funnest part of this whole project.



Screenshot of Q1K3's first level in TrenchBroom

TrenchBroom saves the levels in the `.map` format; It's the de-facto standard for all Quake Engine games around the time. A `.map` file contains a list of “brushes”, where each brush is a convex object. However, the brush's geometry is not defined by the positions of each corner, but rather by a list of planes that stretch to infinity. Each plane subdivides a (conceptually) infinitely large space into an inside and outside half. With four or more of these subdivisions you end up with the remainder of all that's inside: a brush. This also explains why all brushes need to be convex by definition.

Historically this format of `.map` files was chosen to make various computations for Quake's Binary Space Partitioning (BSP) compiler easier. The Quake engine itself also uses these infinitely large planes for collision detection: testing if a point is inside of a brush is very cheap when you only have to test if this point is on the right side (i.e. the inside) of all planes.

For my purpose this `.map` format was quite difficult to work with. Luckily I already built a `.map` parser and loader in C for another side project. I quickly adapted the map loader to take the computed corner points of each brush and produce a list of “blocks”.

All blocks of a Q1K3 level are axis aligned. This means I only need to store the position and the size of each block. Initially my level format looked like this:

```
struct level {
    u16 num_blocks
    struct {
        struct { u8 x, y, z; } position;
        struct { u8 x, y, z; } size;
        u8 texture_index;
    } blocks[num_blocks];

    u16 num_entities;
    struct {
        u8 type_id;
        struct { u8 x, y, z; } position;
        u8 data1, data2;
    } entities[num_entities];
}
```

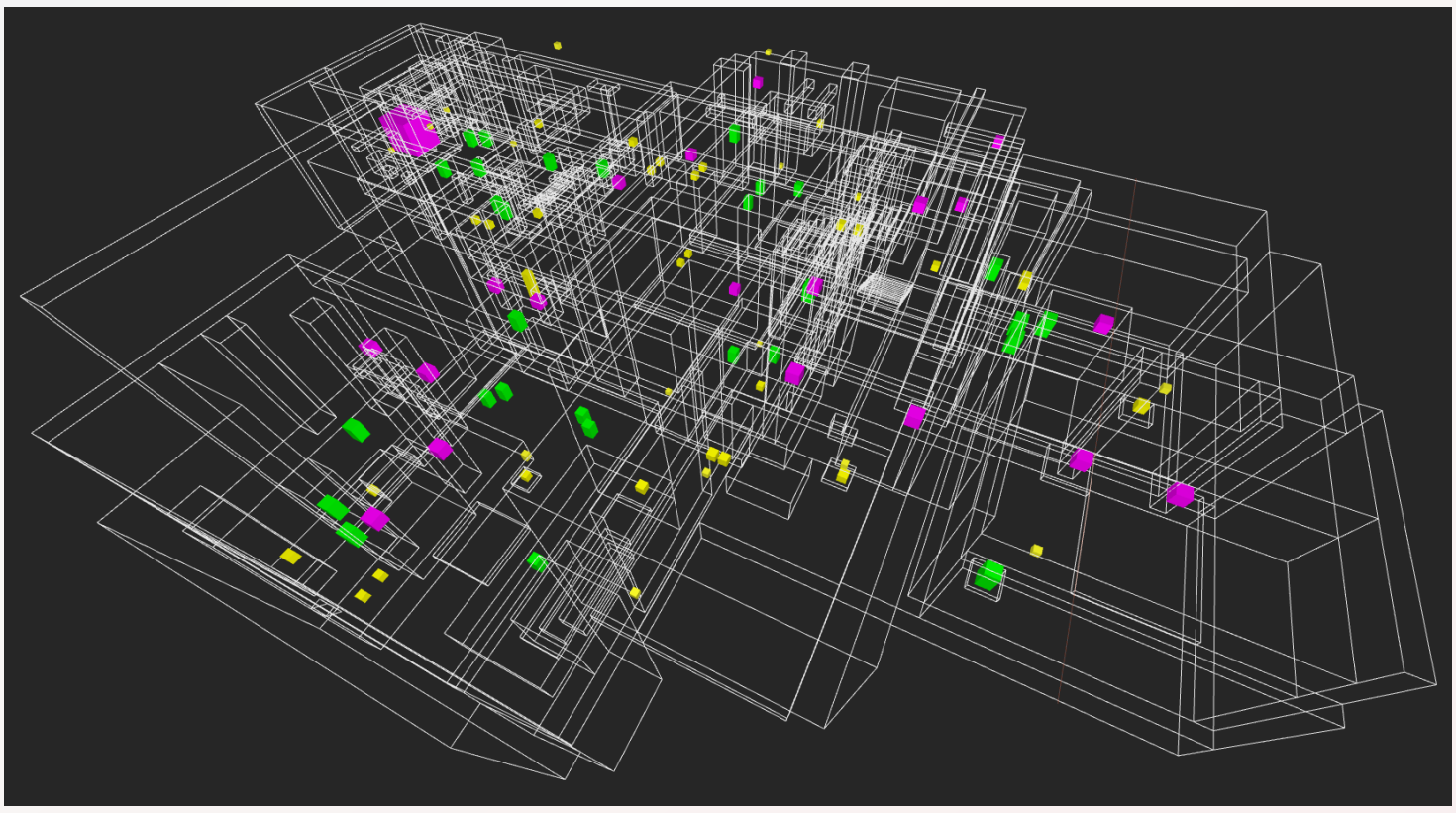
I.e. 7 bytes per block and 6 bytes per entity. This produced a 3kb file for the first level. The resulting block file ZIP-compresses to about 75%.

Towards the end of the project I had to revisit this block format to shave of some more bytes. In the final version of the game, each block takes only 6 bytes for the position and size. The list of blocks is sorted by `texture_index` and interleaved with a “texture change

sentinel" that denotes the texture index for all following blocks in the file. This saved around 300 bytes for each of the two levels.

I also experimented with just using 6 or 7 bits for each value, bringing done the block size to 5 bytes. As I painfully figured out, this is horrible for ZIP compression which examines the file on a byte by byte level. This however lead me to squeeze out another 50 bytes or so by just sorting the blocks by size, which aids the ZIP compression.

The entities stored in the file just have a `type_id`, a position and two extra bytes for data. This extra data is used by lights to store the light color and brightness and by enemies to indicate if they start idle or patrolling and which direction they're facing.



All 232 blocks and 92 entities of Q1K3's 2nd level

In the end I managed to cram most of e1m1 (sans secrets) and parts of e1m3 (with a lot of freestyling) into the game. In total the two levels consist of 563 blocks and 188 entities. A total of 4.5kb uncompressed or 3.2kb zipped.

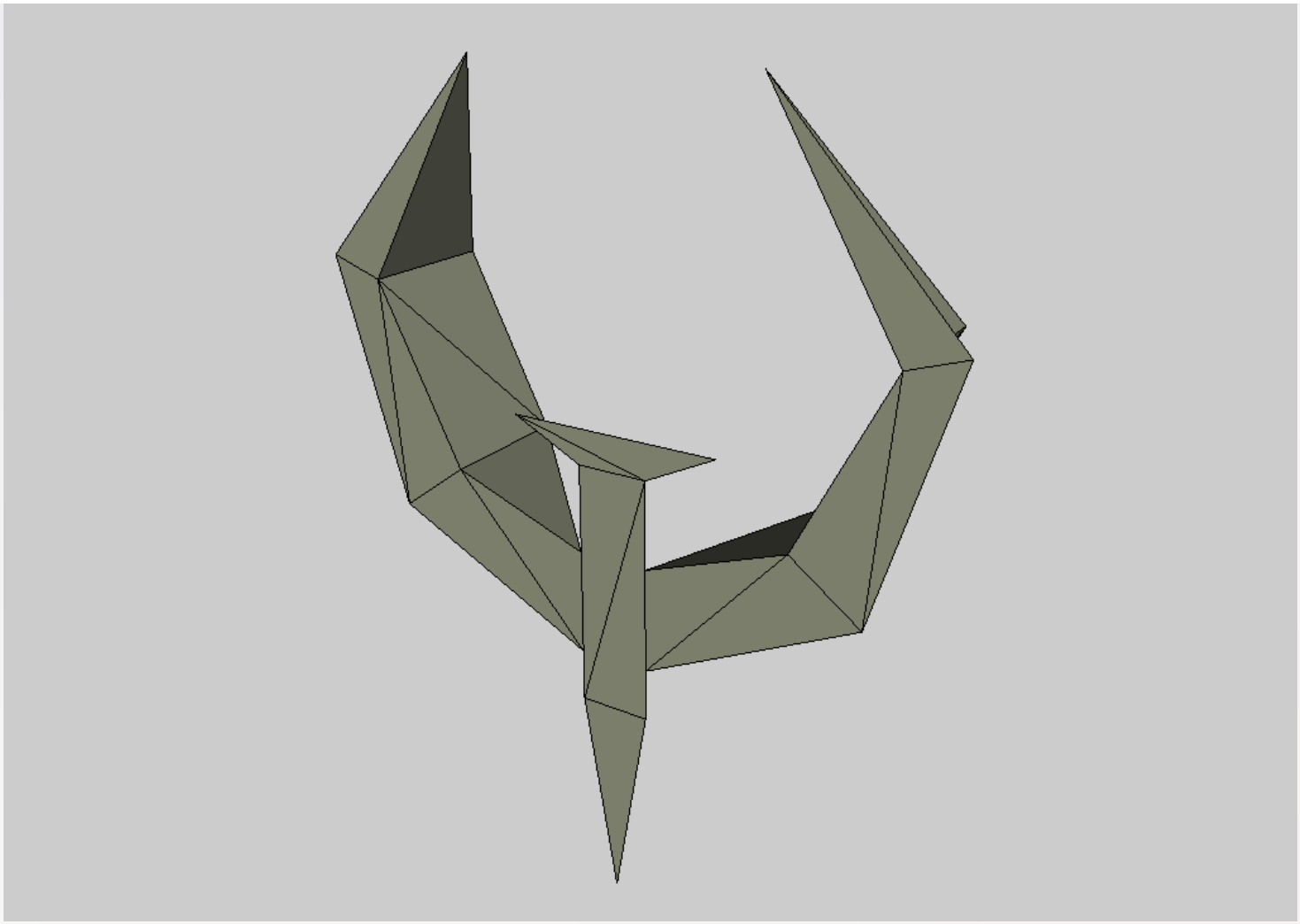
Models

The geometry for all game objects – weapons, enemies, various pickups and some map decorations – follows a similar approach as for my 2019 entry Voidcall: Just a list of vertex indices, followed by vertex positions. Animated models have the list of indices once and the vertex positions for each frame.

For Voidcall I tried very hard to minimize the byte size of the models. I only used 5 bits for each vertex position. This meant that each coordinate is interleaved in two bytes. As I noted earlier with the level files, this actually trips up ZIP compression, because each byte almost looks like random noise.

So for Q1K3, each index and vertex coordinate is stored in an individual byte. This allowed for better ZIP compression and saved a few hundred bytes in the final build.

I carefully removed all the faces from each model which will not be seen in the game to save a few bytes here and there. This means the box model has no underside and in the most extreme case, the Quake Logo on the title screen has most sides removed.



Quake Logo as used on the title screen

There's only 3 different animated models in the game: the basic humanoid with 6 animation frames, the dog with just 2 frames and a torch model (only used in the 2nd level) with 3 frames. The humanoid model is re-used for most enemy types by stretching or shrinking the x, y and z size and applying different textures. E.g. the Ogre is a transformed to be a bit shorter but wider than the Enforcer.

Most other models are re-used too:

- A cylinder model is used for exploding barrels, the projectiles fired by the Nailgun and Plasmagun (from the Enforcer), the Grenades and even the Grenade Launcher and Shotgun itself
- A simple box model is stretched to become the health and ammunition pickups and the doors(!)
- The humanoid model is divided up into individual pieces to become the gibs when killing enemies

All of Q1K3's Models: logo, generic blob (blood splat, debris), box (ammo, health, doors), cylinder (projectiles, shotgun, grenades, grenade launcher, barrel), nailgun, torch, humanoid, dog

Textures for the models are dynamically created with TTT and just projected onto the model from the front. This produced some ugly artifacts on the top of barrels and sides of the ammunition boxes because the texture is infinitely stretched or squashed. But it's hardly noticeable in the game.

All models combined take up 1.6kb uncompressed or 1.1kb zipped.

Sound and Music

I looked into the excellent ZzFX library for sounds and ZzFXM for music. The ZzFXM Tracker to create music is a particular nice piece of software. ZzFX works great for 8bit-ish effects, but I found it very hard to create anything that doesn't sound like an NES. Andy Lösch, who did the music for Q1K3, faced a similar challenge.

So I ultimately decided to go back to the trusty Sontant-X that I've been using in the previous years. The version used in Q1K3 has been heavily modified to make it smaller while retaining all the same features of the original.

In the previous years I modified the library to generate a lookup table at startup for all the different oscillators (Sin, Square, Sawtooth and Triangle), instead of computing those on the fly all the time.

```
// Generate the lookup tables
let s = 4096;
let tab = new Float32Array(s*4); // Space for 4 oscillators
for (let i = 0; i < s; i++) {
    tab[i         ] = Math.sin(i*6.283184/s);                 // Sin
    tab[i + s     ] = tab[i] < 0 ? -1 : 1;                    // Square
    tab[i + s * 2] = i / s - 0.5;                             // Sawtooth
    tab[i + s * 3] = i < s/2 ? (i/(s/4)) - 1 : 3 - (i/(s/4)); // Triangle
}
```

Getting the current value for an oscillator is then just a simple lookup. This speeds up the music generation a lot, to the point where there's only about a 100ms delay from clicking on the start screen to being the game with the music playing.

```
// Lookup current value. osc_type is 0..3
let mask = s - 1;
let v = tab[osc_type * s + ((pos * freq * s) & mask)];
```

This year I also *hacked* on the Sonant-X Tracker to store all the music in flat arrays instead of a JSON object. This further reduced the size needed for all sounds and music.

```
// Original format
let music_data = {
    "rowLen": 6615,
    "endPattern": 27,
    "songData": [
        {
            "osc1_oct": 7,
            "osc1_det": 0,
            "osc1_detune": 0,
            "osc1_xenv": 0,
            "osc1_vol": 180,
            "osc1_waveform": 0,
            "osc2_oct": 7,
            "osc2_det": 0,
            ...
}

// New format
let music_data = [6014,21,88,[[[7,0,0,1,255,0,7,0,0,1,255,0,0...
```

Like for all my previous games, the music for Q1K3 was composed by my good friend Andy Lösch. For Q1K3 space was even tighter than for the past JS13k games. He still ended up with a brilliant moody, minimalist ambient track that compresses nicely.

This minimalism fits the game perfectly. After all, the original Quake soundtrack from Nine Inch Nails (that we both are big fans of) has a similar theme going on. I'm really happy with the result.

In contrast, the sound effects were difficult to produce with the limited possibilities. In my opinion they don't live up to the game. With my very limited understanding of audio filters and synthesizers I have no idea how to improve the situation. Maybe it's just not possible or extremely complex to create good sound effects out of thin air.

What worked nicely though, is the very simplistic “spacial audio“ model used in Q1K3. Whenever an entity wants to play a sound effect, the distance and angle to the camera is computed and the volume and stereo panning is changed accordingly.

```
// Fade audio volume from 1..0 over the distance 64..1200
let distance = vec3_dist(pos, camera_pos);
let volume = scale(distance, 64, 1200, 1, 0);

// Compute panning (-1 = left, 0 = center, +1 = right)
let angle = vec3_horizontal_angle_to(pos, camera_pos) - camera_yaw;
let pan = Math.sin(angle) * -1;
audio_play(sound, volume, 0, pan);
```

It's only about 10 lines of code (mostly just connecting different WebAudio objects to each other), but greatly helps when you want to figure out who's shooting at you.

Stepping back a bit, it's quite funny to me how exhaustive the WebAudio API is, yet everyone generates samples directly in JavaScript instead of connecting the dots on a design-by-consortium audio graph. At this point it seems to be a tradition of the W3C to implement all the wrong abstractions.

The audio library, sound effects and music take up about 1.5kb in the final ZIP package.

Code

All the assets, with 3.2kb of level data, 1.1kb of model data, 1.3kb for the textures (including the texture lib) and 1.5kb for sounds and music (including the audio lib), sum up to a total of 7.1kb within the ZIP. This leaves me with a generous 5.9kb for all the remaining code.

In the remaining 5.9kb I have to handle:

- unpacking map files
- unpacking model files
- input (keyboard, mouse look)
- physics
- collision detection
- rendering
- model animations
- enemy ai
- all gameplay logic

The basic architecture for the game revolves around an array of “entities”. Each entity type (e.g. the player, ammo pickup and enemies) is a subclass of the main entity_t. They all share the same physics calculations, collision detection and a bit of game logic to receive damage and die.

For each frame the game iterates through this array, updates the physics for each entity and draws it. There's almost no game logic going on outside of the entity classes. Everything in Q1K3 is an entity, even the tiniest particle.

For all my previous games I tried hard to avoid 3D and vector math wherever possible. I told myself that I did this to avoid the added code, but the real reason was more that I suck at math. For this year it was evident that this would not be an option. I needed a basic vector math library.

Whenever I needed some 3D math I used Brandon Jones' excellent glMatrix library. It's the de-facto reference implementation for vector and matrix math in JavaScript.

glMatrix' goal is to be as fast as possible. This means it never allocates any objects if you don't explicitly tell it to do so. It also means that it's a bit cumbersome to work with. You have to pass an `out` object to almost all the functions, instead of receiving the result in the function's return value.

So I choose to ignore the efficiency gains of glMatrix' architecture and instead implemented a tiny vector math library that is nicely composable. All the `vec3_*` functions create and return a plain JS objects. This introduces a lot of work for the garbage collector but in turn allowed me to write code like this:

```
// Acceleration in movement direction
player.acceleration = vec3_mulf(
    vec3_rotate_y(
        vec3(
            keys[key_right] - keys[key_left],
            0,
            keys[key_up] - keys[key_down]
        ),
        player.yaw
    ),
    player.speed * (player.is_on_ground ? 1 : 0.3)
);
```

Having all these vector functions readily available instead of fiddling with `Math.sin()` and accessing the vector components directly was a good idea after all.

Collision Detection

Collision detection can be a hairy thing. Or more precisely: collision response. The detection of a collision is a tightly defined and exactly solvable problem. What happens when there is a collision can get all kinds of wonky.

In Q1K3 I made my life a lot easier by only dealing with axis aligned bounding boxes. Still, there's a bit of complexity involved by dealing with collisions against the static level geometry and also against other dynamic objects. In both cases, the same collision response is employed.

When the level is loaded, a `Uint8Array` for `128 * 128 * 128` grid elements is generated. Since I only needed to know which of these grid cells is occupied, I used a bitmap, where each bit corresponds to one cell. So instead of `2,097,152` bytes the `Uint8Array` only needs to have `262,144` bytes.

Looking up one bit in this array is then just:

```
let index = Math.floor((z * 128 * 128 + y * 128 + x) / 8);
let element = collision_map[index];
let bit = element & (1 << (x % 8));
```

Or more concisely:

```
let bit = collision_map[(z * 128 * 128 + y * 128 + x) >> 3] & (1 << (x & 7));
```

To check if an entity collides with one or more cells in this bitmap, the entity's whole box is checked.

```
map_block_at_box = (box_start, box_end) => {
    for (let z = box_start.z; z <= box_end.z; z++) {
```

```
        for (let y = box_start.y; y <= box_end.y; y++) {
            for (let x = box_start.x; x <= box_end.x; x++) {
                if (map_block_at(x, y, z)) {
                    return true;
                }
            }
        }
    }
    return false;
};
```

This `map_block_at_box()` check is done once for each x, y and z dimension and on top of that, these 3 checks might be done in a loop for fast moving entities.

Checking for each dimension individually ensures that we can slide along the floor and walls. We only need to stop movement in the direction in which we hit something.

```
// Divide movement into 16 unit steps
let move_dist = vec3_mulf(this.vel, game_tick);
let steps = Math.ceil(vec3_length(move_dist) / 16);
let move_step = vec3_divf(move_dist, steps);

for (let s = 0; s < steps; s++) {
    // Remember last position so we can roll back
    let lp = vec3_clone(this.pos);

    // Integrate velocity into pos
    this.pos = vec3_add(this.pos, move_step);

    // Collision with walls, horizonal
    if (this.collides(vec3(this.pos.x, lp.y, lp.z))) {
        // roll back x pos and stop x velocity
        this.pos.x = lp.x;
        this.vel.x = 0;
    }

    // Collision with walls, horizonal
    if (this.collides(vec3(this.pos.x, lp.y, this.pos.z))) {
        // roll back z pos and stop z velocity
        this.pos.z = lp.z;
        this.vel.z = 0;
    }

    // Collision with ground/Ceiling
    if (this.collides(this.pos)) {
        // roll back y pos and stop y velocity
        this.pos.y = lp.y;
        this.vel.y = 0;
    }
}
```

There's a bit more going on in the final code, including a check for stepping up stairs, bouncing on walls, floors and ceilings and setting a flag that indicates if the entity is on the ground.

This is all terribly inefficient, but it doesn't matter. CPUs are fast.

Rendering

The rendering for Q1K3 is quite simplistic. All entities are rendered for each frame. There's no check if an entity is outside the view frustum or if it's occluded by some level geometry. Likewise, all the level geometry is rendered for each frame. We get a lot of overdraw this way, but since the polygon count is quite low, it still works out nicely. GPUs are fast.

Light sources were the one thing I needed to address, though. The game has a hard limit of 32 light sources that can be rendered for a frame. This includes all the lights scattered around the level, as well as the dynamically created lights when shooting a weapon or when something explodes.

During one frame all entities submit their lights to the renderer. The renderer determines

the distance of this light to the camera and fades it out at a certain distance. If it's too far away, it's not rendered at all.

```
r_push_light = (pos, intensity, r, g, b) => {
    // Calculate the distance to the light, fade it out between 768--1024
    let fade = clamp(
        scale(
            vec3_dist(pos, camera_pos),
            768, 1024, 1, 0
        ),
        0, 1
    ) * intensity * 10;

    if (fade && r_num_lights < r_max_lights) {
        r_light_buffer.set([
            pos.x, pos.y, pos.z,
            r*fade, g*fade, b*fade
        ], r_num_lights * 6);
        r_num_lights++;
    }
};
```

There's only one shader program shared by the level geometry and models. The vertex shader takes care of the current camera position and view angles and also mixes the animation frames of the models.

The fragment shader calculates the lighting from all light sources, applies some cheesy gamma correction and reduces the number of colors for some extra dirty looks.

All geometry for the two levels and all the models and their animation frames are uploaded to a GPU buffer on load time. The renderer then just needs to issue draw calls with offsets into this buffer.

There's one draw call per model and one per level block. This sounds excessive, but is not a big problem. The level data is already sorted by texture index, so the renderer doesn't need to re-bind textures all too much. Similarly, for the static geometry, the vertex attribute pointers all stay the same. The renderer just needs to change those for the animated models.

Minification

The un-minified source of Q1K3 is about 71kb. The first step to get the size down is a PHP script that replaces WebGL constants with their actual numeric values. It also shortens WebGL function names, as explained in the Underrun Making Of.

Minifying the source with uglify-js brings this down to 26kb. Still a lot, but JS13k's size limit is only concerned with the size of a final ZIP file, containing all the assets and code for the game.

Zipping the minifyed source shrinks it to 10kb. But I needed some more space for the maps (4.4kb) and models (1.6kb). While ZIP compression helps a bit for these data files, the ZIP with the source and data still clocks in at 14.2kb.

Luckily, just a few days after the start of JS13k Kang Seonghoon released Roadroller - a JavaScript packer written in the spirit of the oldschool UPX. Roadroller compresses your source code and appends its own decompressing code. Loading the packed source in the browser unpacks and immediately executes it.

Roadroller essentially provided me with an extra 1.2kb - totally for free, without any need to change my source. Without it Q1K3 likely would have ended up with only level.

Final build sizes:

```
# bytes file
     48 boulder.rmf        # generic blob model
     57 box.rmf            # box model
     81 grenade.rmf        # cylinder model
    195 hound.rmf          # rottweiler model
    120 nailgun.rmf        # nailgun model
     96 torch.rmf          # torch model
    147 q.rmf              # quake logo model
    855 unit.rmf           # humanoid model
   2566 m1.plb             # e1m1 level
   1982 m2.plb             # e1m3 level

  73041 game.js            # un-minified source
  72267 game.packed.js     # source with replaced webgl constants
  26543 game.min.js        # uglify-js compressed source
  12003 game.roadrolled.js # roadroller compressed source

  12080 index.html         # final html file with the compressed source
   1599 m                  # all models combined
   4548 l                  # all levels combined

  13304 game.zip           # final zip file
  --------------
```

I truly loathe “modern JavaScript” with its ever changing “extremely fast” bundler of the week – be it Babel, Grunt, Gulp, Webpack or whatever else is in fashion. So my build process is just a simple shell script. Because of that, I can assure you that it still works tomorrow.

I also made sure my game works when you just load the uncompressed source code. This greatly simplifies debugging and makes seeing changes as easy as pressing F5. I still can't wrap my head around the idea of “compiling” JavaScript. Please get off my lawn!

Random Observations

- John Carmack, founder of Id-Software and lead programmer on Quake, liked my post on Q1K3. Yay!
- Neil Graham extended Tiny Texture Tumbler (source) to include his own Stackie library, SVG paths and more.
- I found a possible bug in Chrome, where TypedArrays are neutered by the GC, all while(!) the game is running. I worked around this problem in the final release. Still need to build a test case for it.
- Q1K3 made it to the front page of Hacker News. (So did my 2018 entry Underrun, albeit shamelessly self-submitted)
- Q1K3 is apparently good enough to compete with other (non size restricted) JS games. It's listed on many gaming sites.
- In general, riding on the nostalgia wave seems to be a good way to get media attention. My 2015 Reverse Engineering of WipEout produced a lot of buzz too, presumably for the same reason.

Final Thoughts

With each game I built for JS13k it becomes more and more evident that the way to go is generating data in code, instead of directly authoring it. For Underrun all textures and levels were directly authored in Photoshop, resulting in a tiny low-res game. For Voidcall I generated the game world with lots of perlin noise and made it more organic. For Q1K3 I generated the textures.

Naturally I want to explore generating 3D models and animations.

In this regard another game for this year's JS13k, The Adventures of Captain Callisto, particularly impressed me with its clean and smooth 3D models. It's a very cool aesthetic that nicely works around the size limitations.

(I still have to go through the full list of games for this year, but some of my favorites so far include The Maze of Space Goblins, Two Ships Passing in the Night, Shadow of the Keening Star and Galaxies)

For the next time I also want to go into sound generation a bit more. I wasn't really happy with how the sound effects for Q1K3 turned out, so that's a field I need to learn more about.

All in all Q1K3 was one of the most fun projects I ever worked on. The game is neither original nor very creative, it doesn't match the contest's theme and I don't expect to win anything. But having a very clear goal and just figuring out how to get there is extremely rewarding.

I can't wait for next year's JS13k Contest, where I will completely ignore the theme again and try fit to Super Mario 64 or another classic into 13kb.