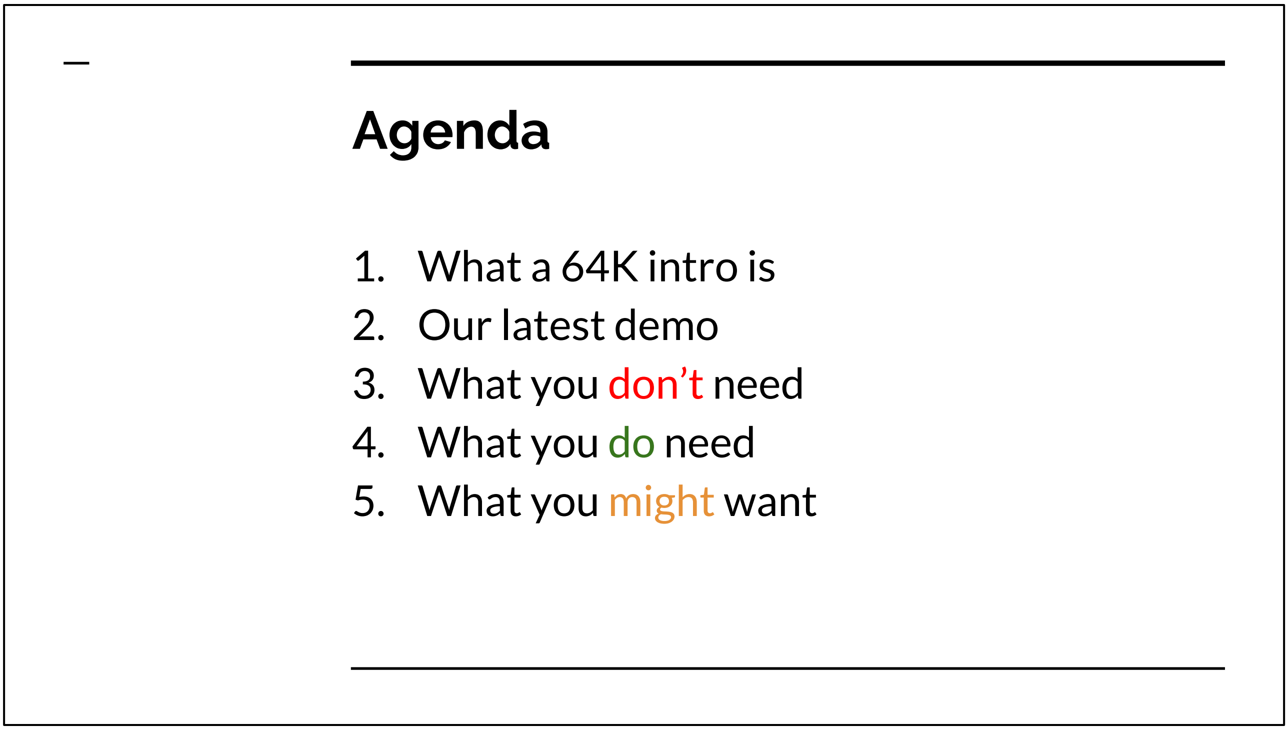
What you need to do a 64K intro

Pekka Väänänen

I’m Pekka Väänänen

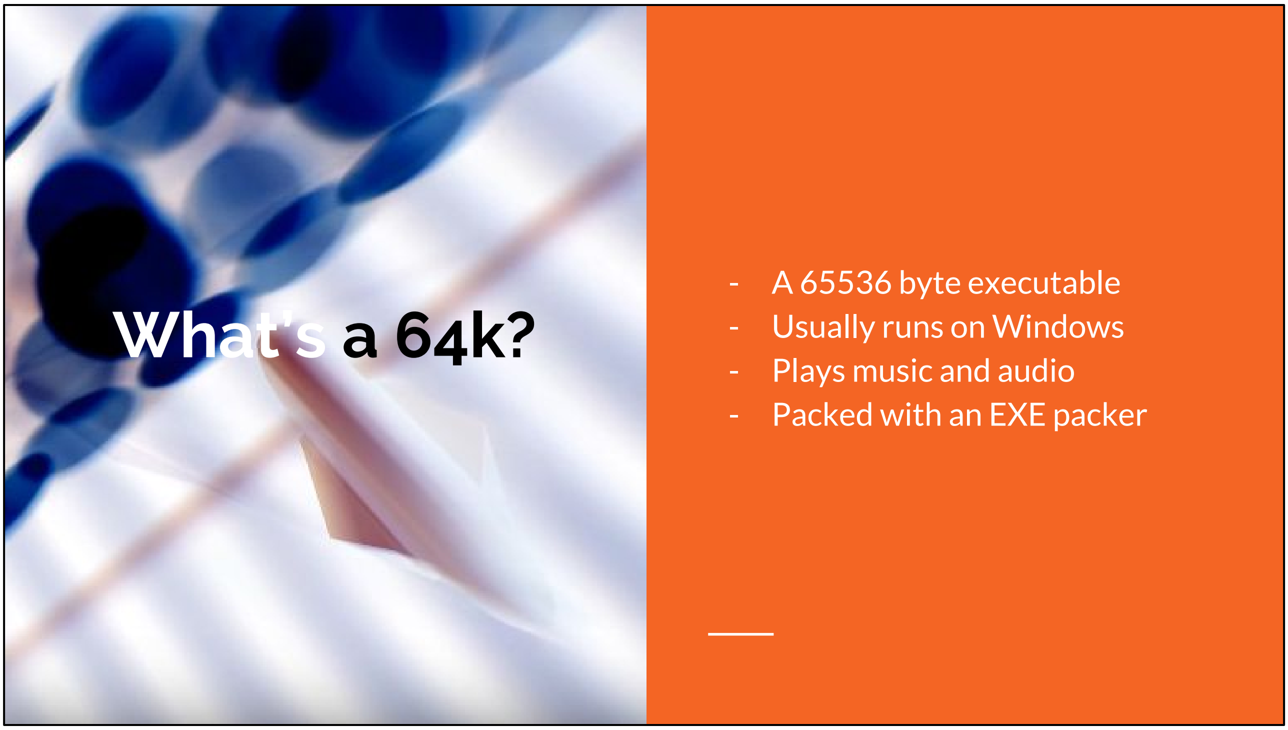
- Nickname: cce
- PC demoscener, member of groups **Peisik** and **Macau Exports**
- Master’s student at University of Helsinki
- Been part of > 20 demo projects



Agenda

1. What a 64K intro is
2. Our latest demo
3. What you don’t need
4. What you do need
5. What you might want

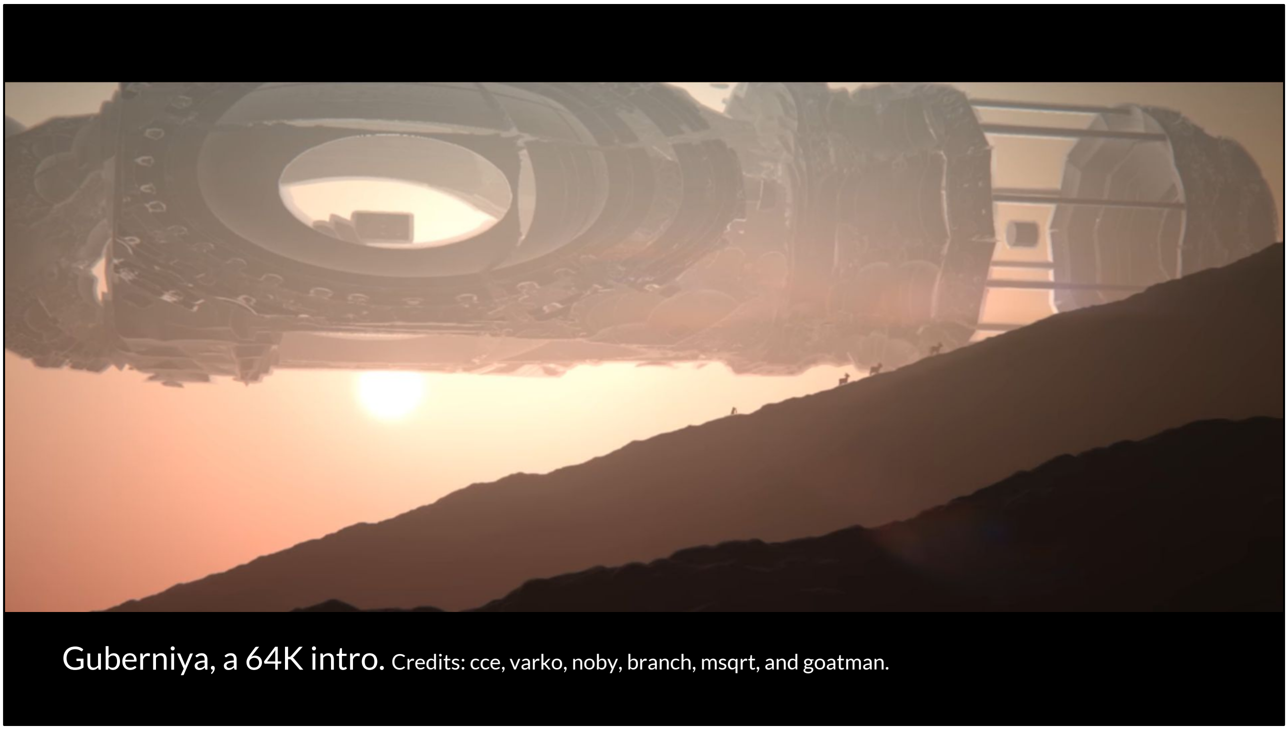
Divided into three parts. I’m trying to motivate you to make your own 64k intro by showing a way how we did ours. These rules are a bit cheeky so take them with a grain of salt :)



What's a 64k?

- A 65536 byte executable
- Usually runs on Windows
- Plays music and audio
- Packed with an EXE packer

64k intro = subset of demo
A bit of a dead art form

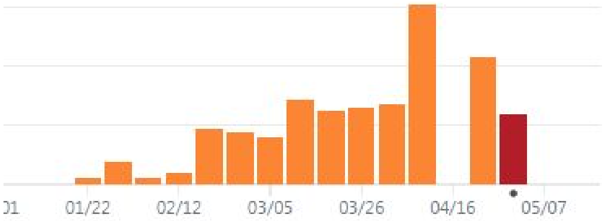


Guberniya, a 64K intro. Credits: cce, varko, noby, branch, msqrt, and goatman.

Released at Revision 2017 this Easter. Ranked sixth. This is the final version that’s a bit different from the one shown at the party.

Timeline

- Some synth experiments already past year
- Started working with an existing codebase in January
- Released at Revision 2017, April 15th
- ~350 commits, total of 5 contributors

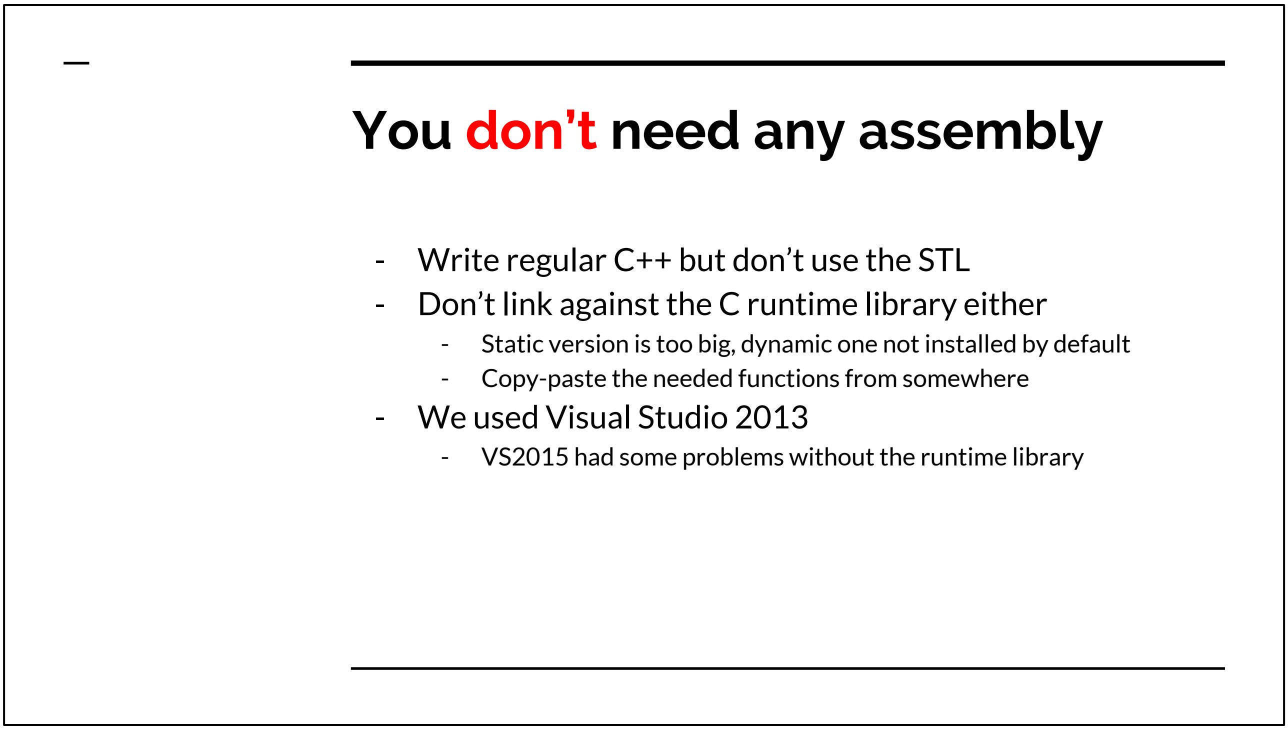


The codebase was Pheromone, another intro we made earlier so we had a basic skeleton to work with.
You can see a huge spike at easter. The bars after that are edits for the final version.

You don't need...

- ... to write any assembly
- ... a fancy tool
- ... a scene graph
- ... years of experience
- ... 3D models

You might already have ideas what it takes to write something like this. These ideas might be wrong!
Let's go over these point by point.

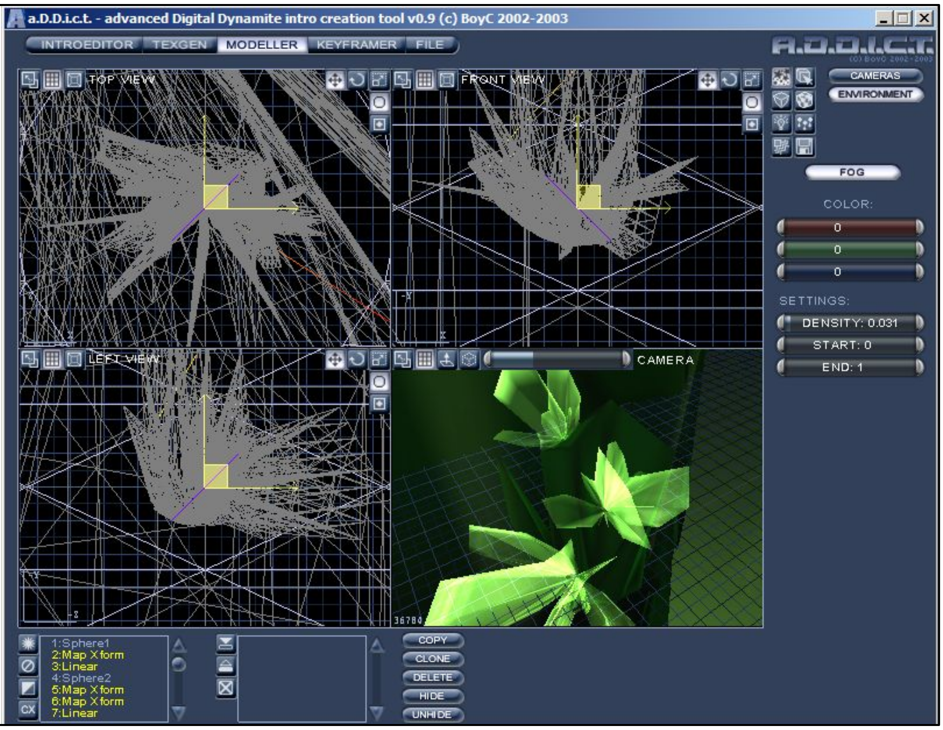


You don't need any assembly

- Write regular C++ but don't use the STL
- Don't link against the C runtime library either
  - Static version is too big, dynamic one not installed by default
  - Copy-paste the needed functions from somewhere
- We used Visual Studio 2013
  - VS2015 had some problems without the runtime library

Just write nice and clean C++, it'll get compressed anyway. If you depend on the standard library you pull in a lot of stuff with it.
I needed two lines of assembly to wrap one pow function, does that count?

You don’t need a fancy tool

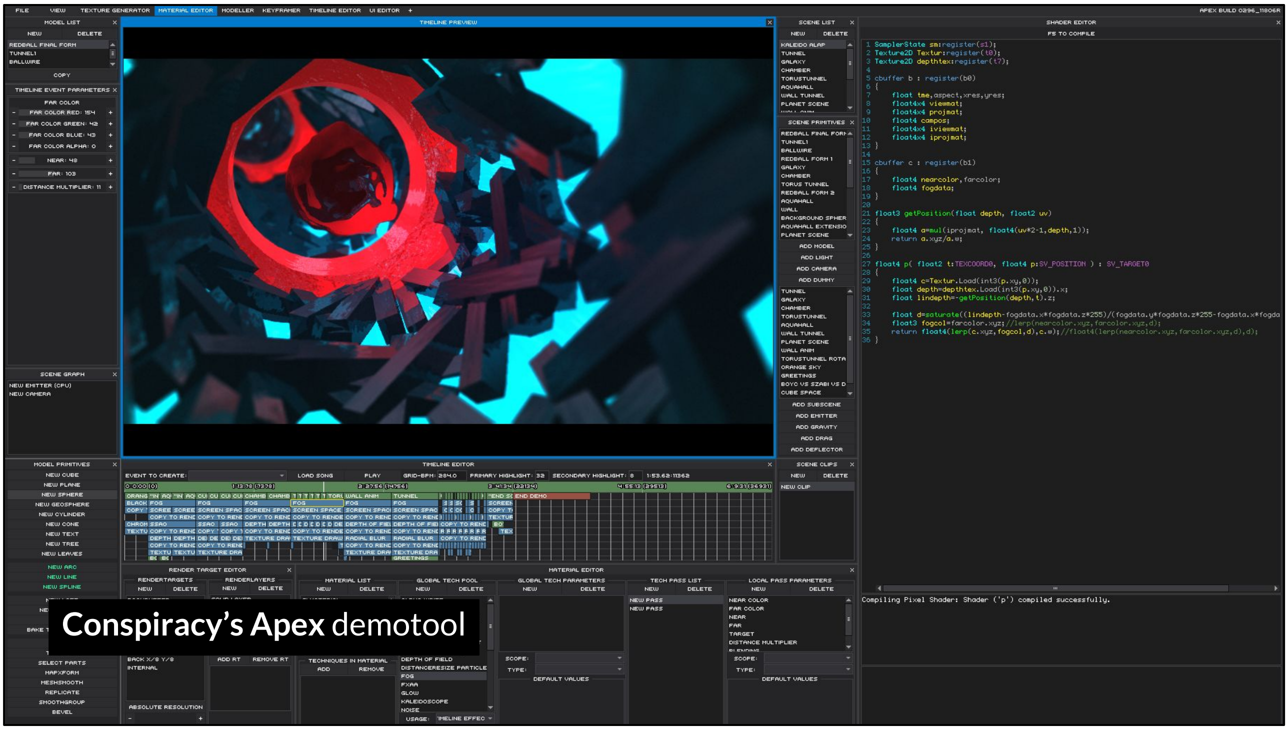


COLOR:
0
0
0

SETTINGS:
DENSITY: 0.031
START: 0
END: 1

Conspiracy a.D.D.i.c.t.

This is an oldskool tool from 2003. It’s actually pretty bare bones: you have simple modeller, a texture generator, some animation tool and then a timeline view. Don’t write anything like this until you’re sure what you’ll need.

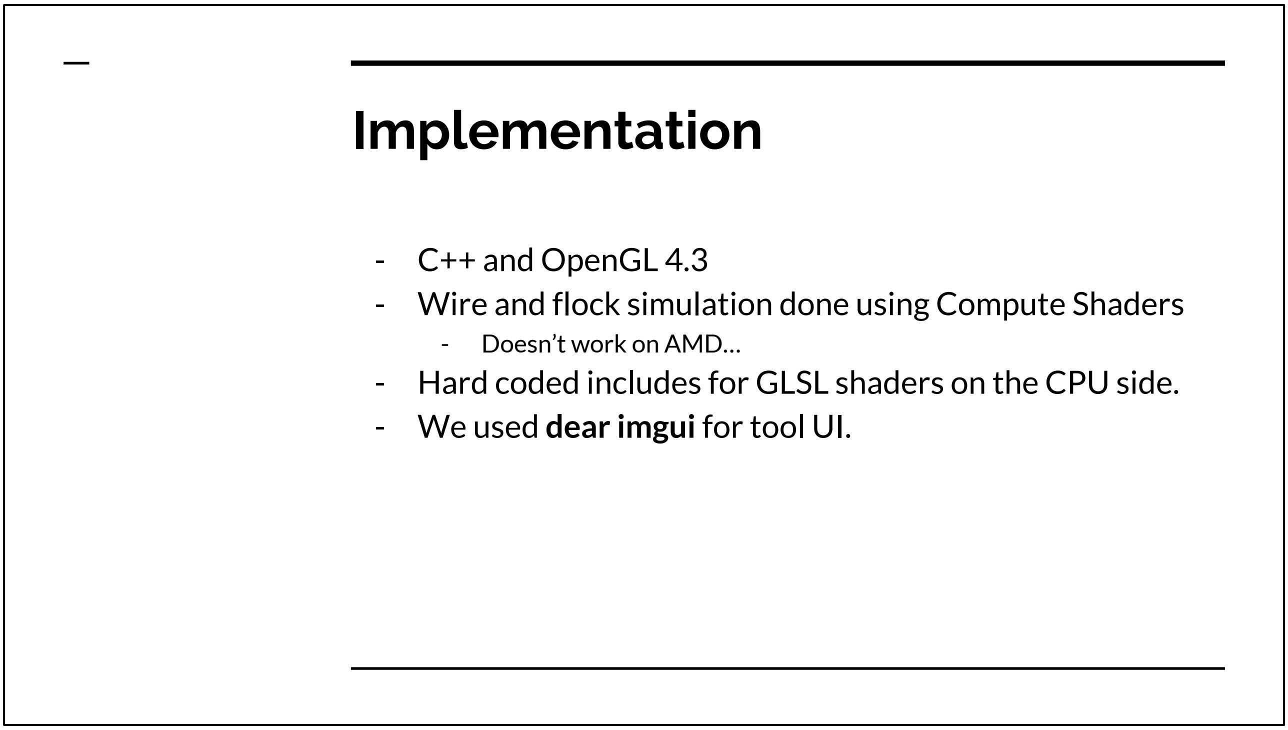


Conspiracy’s Apex demotool

```
SamplerState sm:register(s1);
Texture2D Textur:register(t0);
Texture2D depthtex:register(t7);

cbuffer b : register(b0)
{
    float tme,aspect,xres,yres;
    float4x4 viewmat;
    float4x4 projmat;
    float4 campos;
    float4x4 iviewmat;
    float4x4 iprojmat;
}

cbuffer c : register(b1)
{
    float4 nearcolor,farcolor;
    float4 fogdata;
}

float3 getPosition(float depth, float2 uv)
{
    float4 a=mul(iprojmat, float4(uv*2-1,depth,1));
    return a.xyz/a.w;
}

float4 p( float2 t:TEXCOORD0, float4 p:SV_POSITION ) : SV_TARGET0
{
    float4 c=Textur.Load(int3(p.xy,0));
    float depth=depthtex.Load(int3(p.xy,0)).x;
    float lindepth=-getPosition(depth,t).z;

    float d=saturate((lindepth-fogdata.x*fogdata.z*255)/(fogdata.y*fogdata.z*255-fogdata.x*fogda
    float3 fogcol=farcolor.xyz;//lerp(nearcolor.xyz,farcolor.xyz,d);
    return float4(lerp(c.xyz,fogcol,d),c.w);//float4(lerp(nearcolor.xyz,farcolor.xyz,d),d);
}
```

```
Compiling Pixel Shader: Shader ('p') compiled successfully.
```

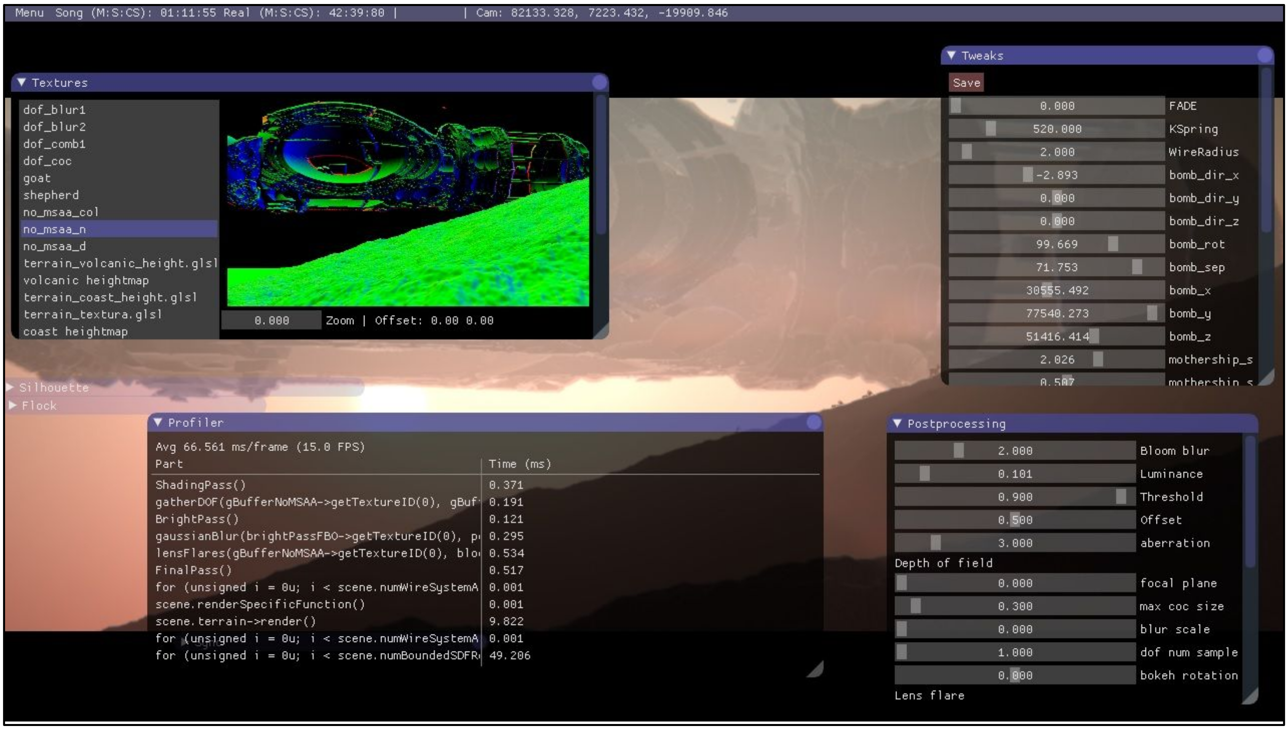
And this is the latest installment of their tool. Lots and lots of features but also tons of work.



Implementation

- C++ and OpenGL 4.3
- Wire and flock simulation done using Compute Shaders
    - Doesn’t work on AMD…
- Hard coded includes for GLSL shaders on the CPU side.
- We used **dear imgui** for tool UI.

Martin implemented the simulation stuff.



```
Menu  Song (M:S:CS): 01:11:55 Real (M:S:CS): 42:39:80 |          | Cam: 82133.328, 7223.432, -19909.846
```

```
dof_blur1
dof_blur2
dof_comb1
dof_coc
goat
shepherd
no_msaa_col
no_msaa_n
no_msaa_d
terrain_volcanic_height.glsl
volcanic heightmap
terrain_coast_height.glsl
terrain_textura.glsl
coast heightmap
```

```
0.000    Zoom | Offset: 0.00 0.00
```

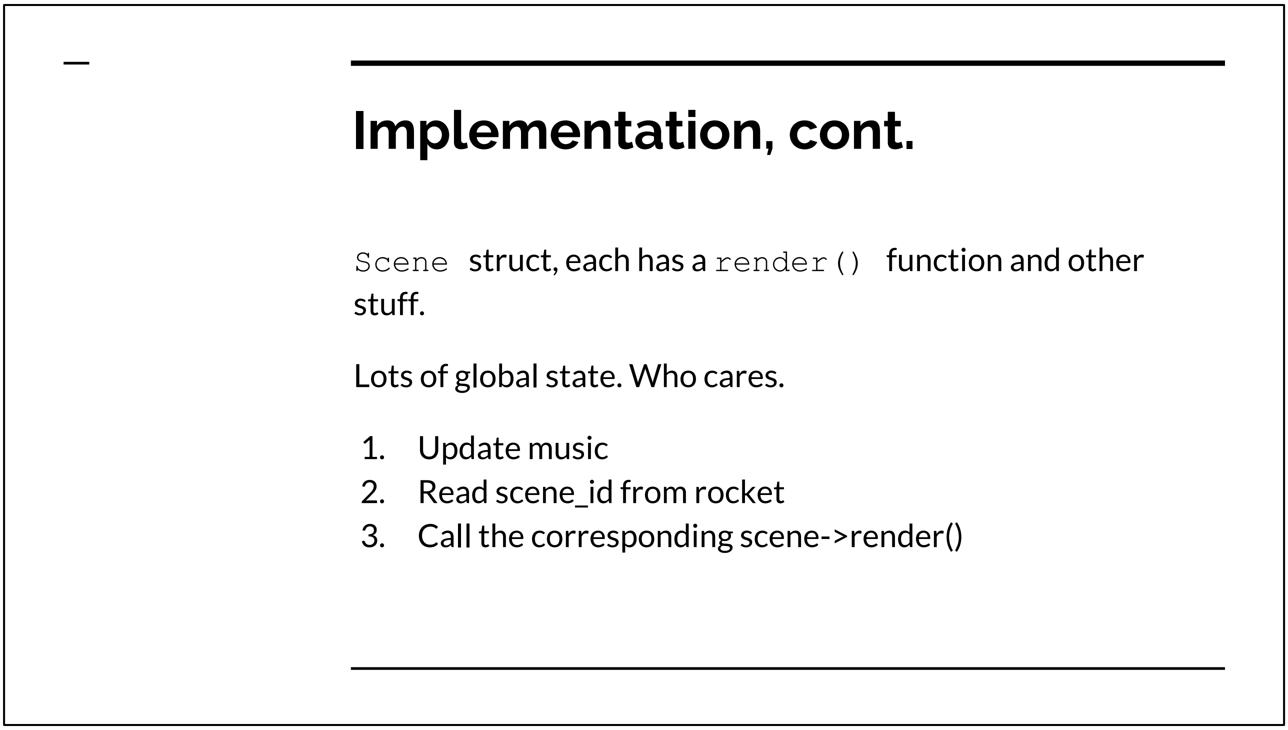
```
       0.000   FADE
     520.000   KSpring
       2.000   WireRadius
      -2.893   bomb_dir_x
       0.000   bomb_dir_y
       0.000   bomb_dir_z
      99.669   bomb_rot
      71.753   bomb_sep
   30555.492   bomb_x
   77540.273   bomb_y
   51416.414   bomb_z
       2.026   mothership_s
       0.507   mothership_s
```

```
Avg 66.561 ms/frame (15.0 FPS)
Part                                         Time (ms)
ShadingPass()                                0.371
gatherDOF(gBufferNoMSAA->getTextureID(0), gBuf 0.191
BrightPass()                                 0.121
gaussianBlur(brightPassFBO->getTextureID(0), p 0.295
lensFlares(gBufferNoMSAA->getTextureID(0), blo 0.534
FinalPass()                                  0.517
for (unsigned i = 0u; i < scene.numWireSystemA 0.001
scene.renderSpecificFunction()               0.001
scene.terrain->render()                      9.822
for (unsigned i = 0u; i < scene.numWireSystemA 0.001
for (unsigned i = 0u; i < scene.numBoundedSDFR 49.206
```

```
   2.000   Bloom blur
   0.101   Luminance
   0.900   Threshold
   0.500   Offset
   3.000   aberration
Depth of field
   0.000   focal plane
   0.300   max coc size
   0.000   blur scale
   1.000   dof num sample
   0.000   bokeh rotation
Lens flare
```

We use dear imgui for UI. The tweak values are saved into a C++ source file which is neat.

The texture viewer was mostly used for depth of field debugging, so I think we didn't actually need that either.



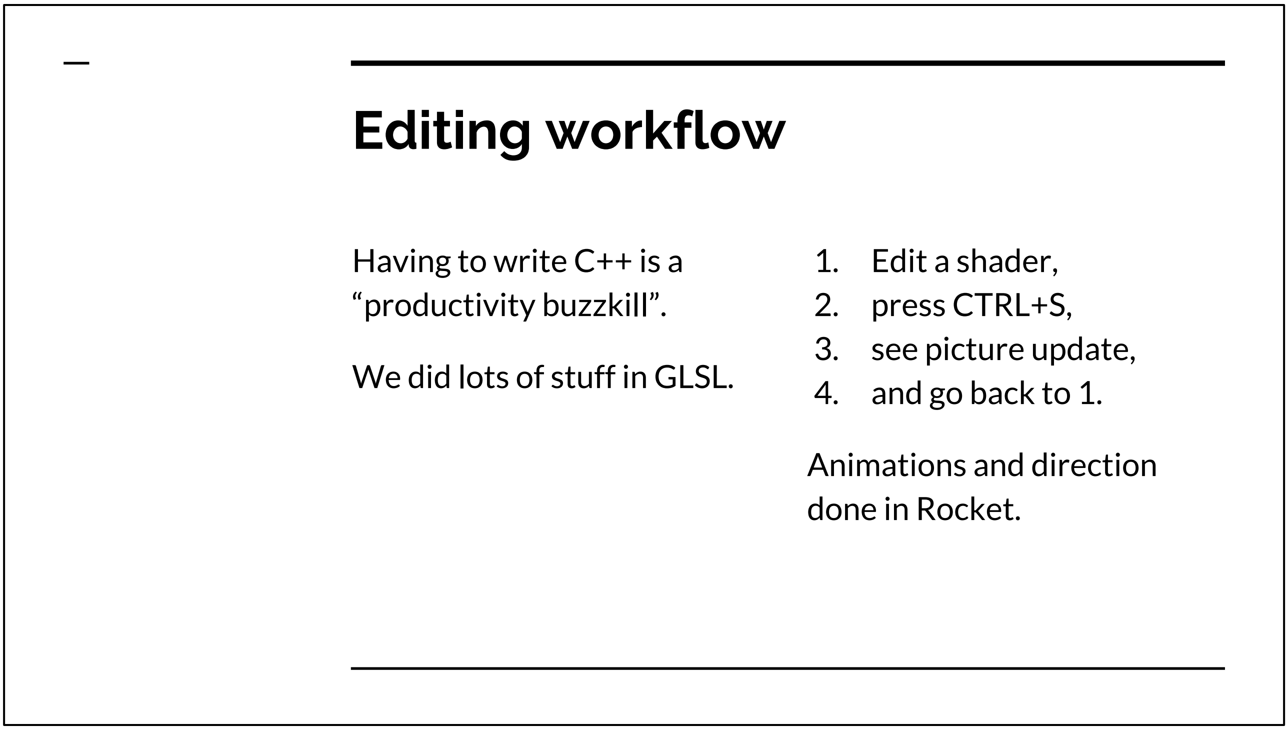
Implementation, cont.

`Scene` struct, each has a `render()` function and other stuff.

Lots of global state. Who cares.

1. Update music
2. Read scene_id from rocket
3. Call the corresponding scene->render()

Can’t do any crossfades or other advanced effects with this but was enough.
We tried to keep this simple. Sizewise it’s totally ok to use C++ inheritance and stuff but I just thought we might not need it.



Editing workflow

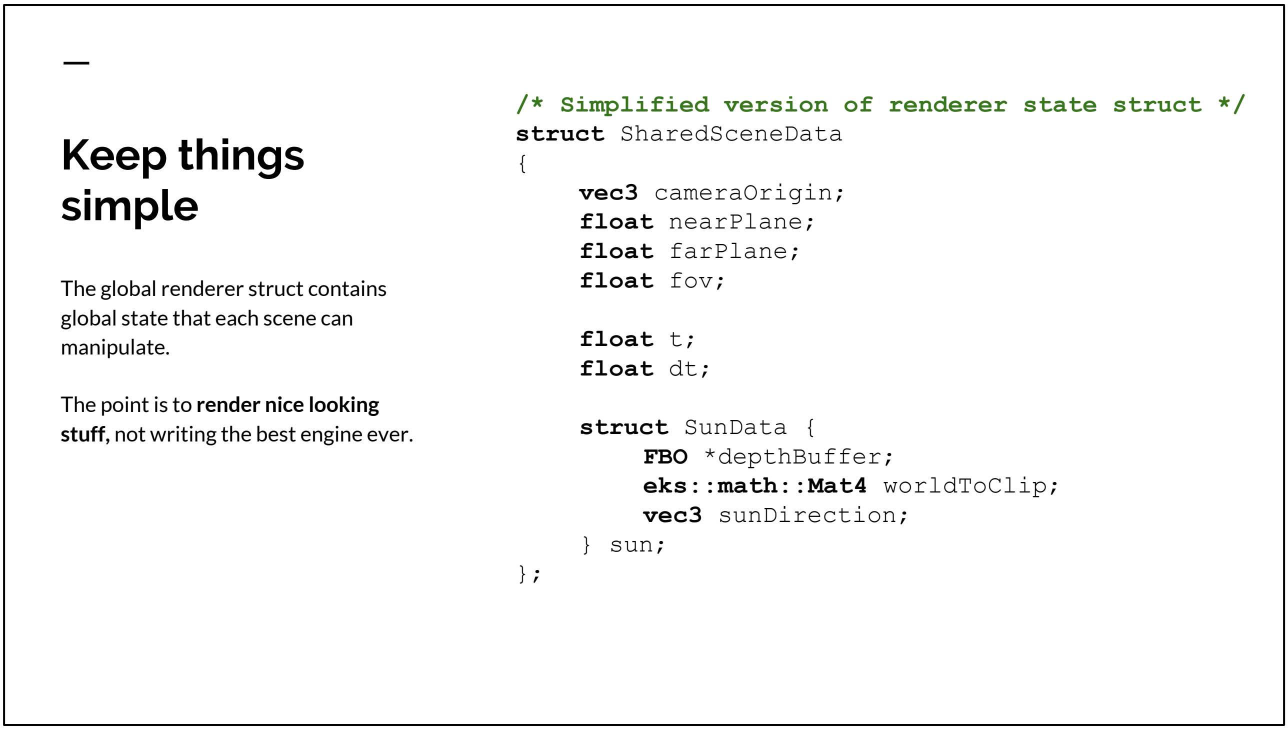
Having to write C++ is a "productivity buzzkill".

We did lots of stuff in GLSL.

1. Edit a shader,
2. press CTRL+S,
3. see picture update,
4. and go back to 1.

Animations and direction done in Rocket.

Ferris of Logicoma described C++ programming as "a major productivity buzzkill" when trying to do something creative.
He is definitely right and we tried to work around it by doing as much as possible in the shaders.
This didn't work out too well though and for example tweaking the wire positions and recompiling was very frustrating.



Keep things simple

The global renderer struct contains global state that each scene can manipulate.

The point is to **render nice looking stuff,** not writing the best engine ever.

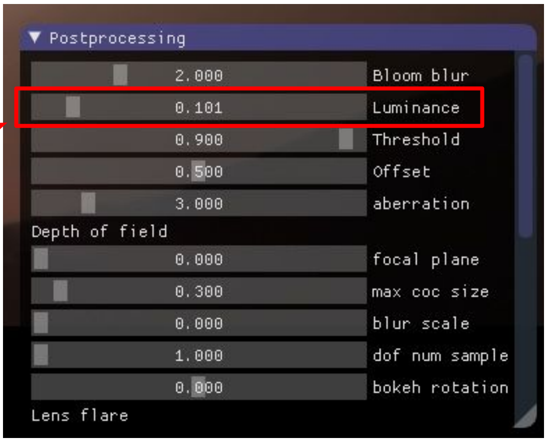
```
/* Simplified version of renderer state struct */
struct SharedSceneData
{
    vec3 cameraOrigin;
    float nearPlane;
    float farPlane;
    float fov;

    float t;
    float dt;

    struct SunData {
        FBO *depthBuffer;
        eks::math::Mat4 worldToClip;
        vec3 sunDirection;
    } sun;
};
```

We wrapped all rendering related globals in a single struct. This was useful in a situation where we wanted to make temporary modifications, the state can be easily saved to a variable and then restored.
We didn't have any abstractions over the rendering pipeline.

About GUIs

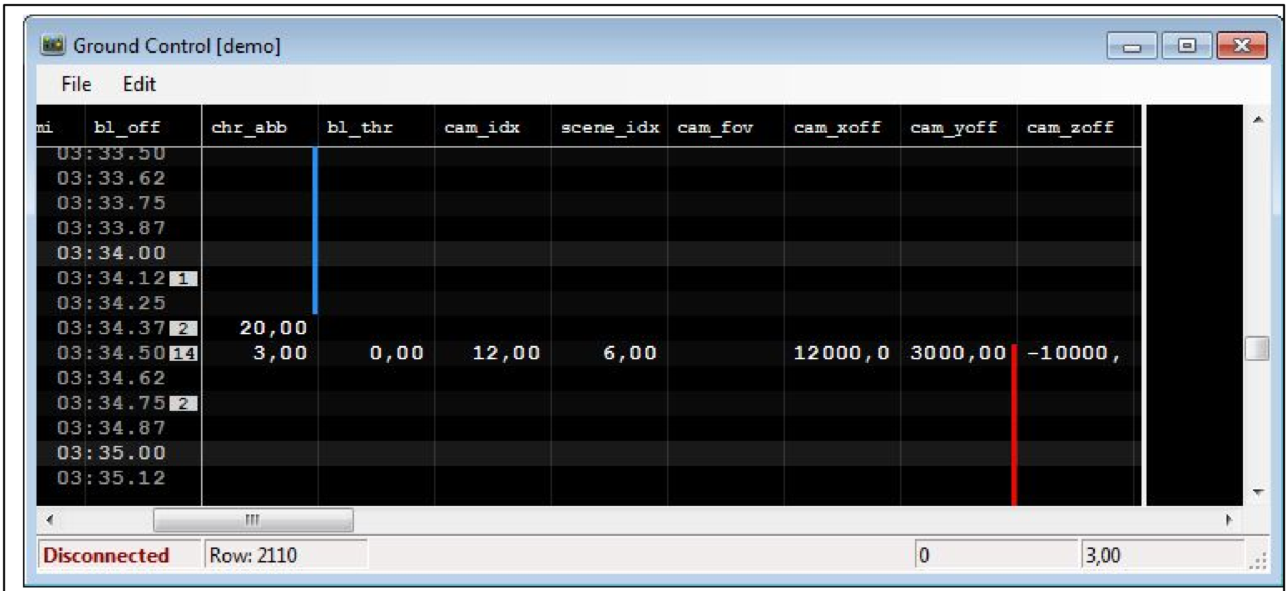
You don't need to use **Qt** or anything.
Immediate GUI with **dear imgui** is easy:

```
float postproc_luminance;

ImGui::SliderFloat("Luminance",
&postproc_luminance,
    0.0f, 1.0f, "%.3f", 1.0f);
```



▼ Postprocessing
2.000 Bloom blur
0.101 Luminance
0.900 Threshold
0.500 Offset
3.000 aberration
Depth of field
0.000 focal plane
0.300 max coc size
0.000 blur scale
1.000 dof num sample
0.000 bokeh rotation
Lens flare

You really want to have a way to tweak things. Imgui makes it super simple to add ad-hoc controls.



mi bl_off	chr_abb	bl_thr	cam_idx	scene_idx	cam_fov	cam_xoff	cam_yoff	cam_zoff
03:33.50								
03:33.62								
03:33.75								
03:33.87								
03:34.00								
03:34.12 1								
03:34.25								
03:34.37 2	20,00							
03:34.50 14	3,00	0,00	12,00	6,00		12000,0	3000,00	-10000,
03:34.62								
03:34.75 2								
03:34.87								
03:35.00								
03:35.12								

Disconnected

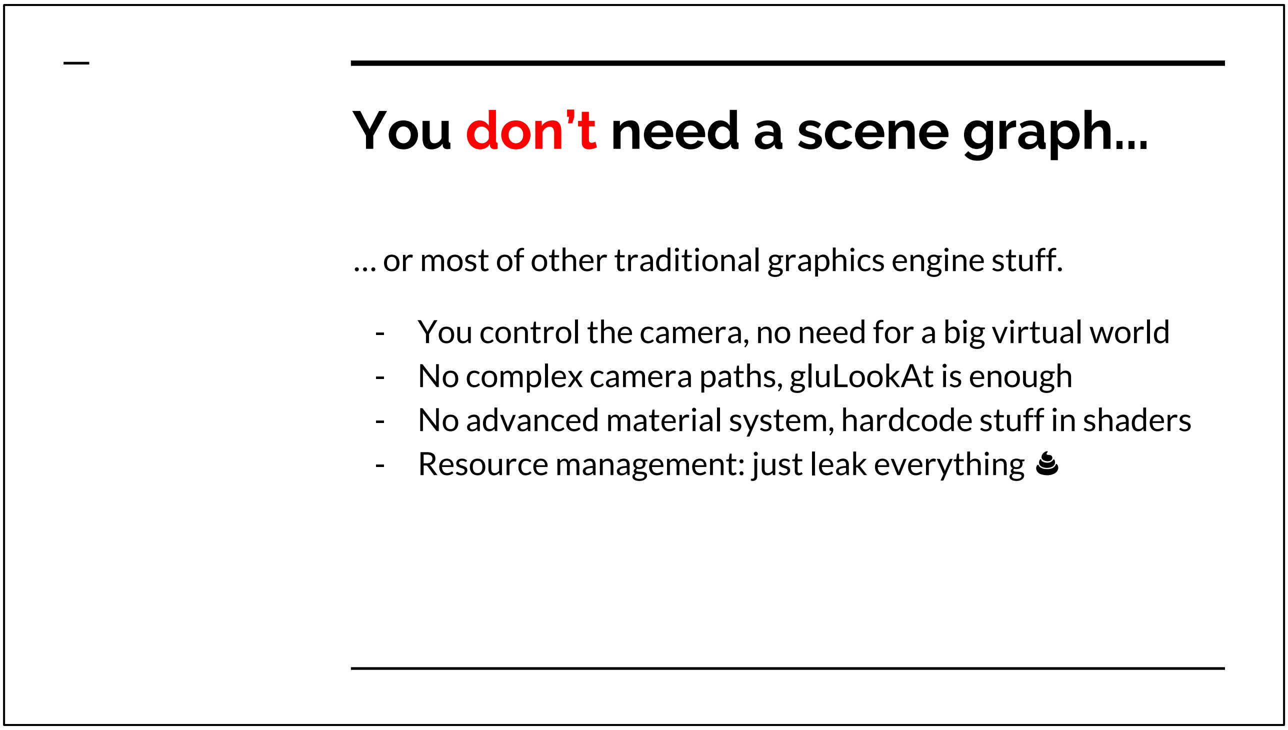
Row: 2110

0

3,00

GNU Rocket (Ground Control fork pictured). Also featured at Graffathon.

Graffathon veterans will recognize this beauty. It connects via TCP socket to your demo and allows you to animate everything without writing your custom editor. The keyframes are then saved and packed into the final executable.



You don’t need a scene graph...

… or most of other traditional graphics engine stuff.

- You control the camera, no need for a big virtual world
- No complex camera paths, gluLookAt is enough
- No advanced material system, hardcode stuff in shaders
- Resource management: just leak everything 💩

You need to carefully consider what actually helps you. The size limit is a good incentive to simplify your code.
You probably won’t reuse most of your code anyway. I think of it as part of the piece, just like ink on paper.

Sounds terrible?

So what I’m advocating here is not having advanced tools and just hard coding everything...

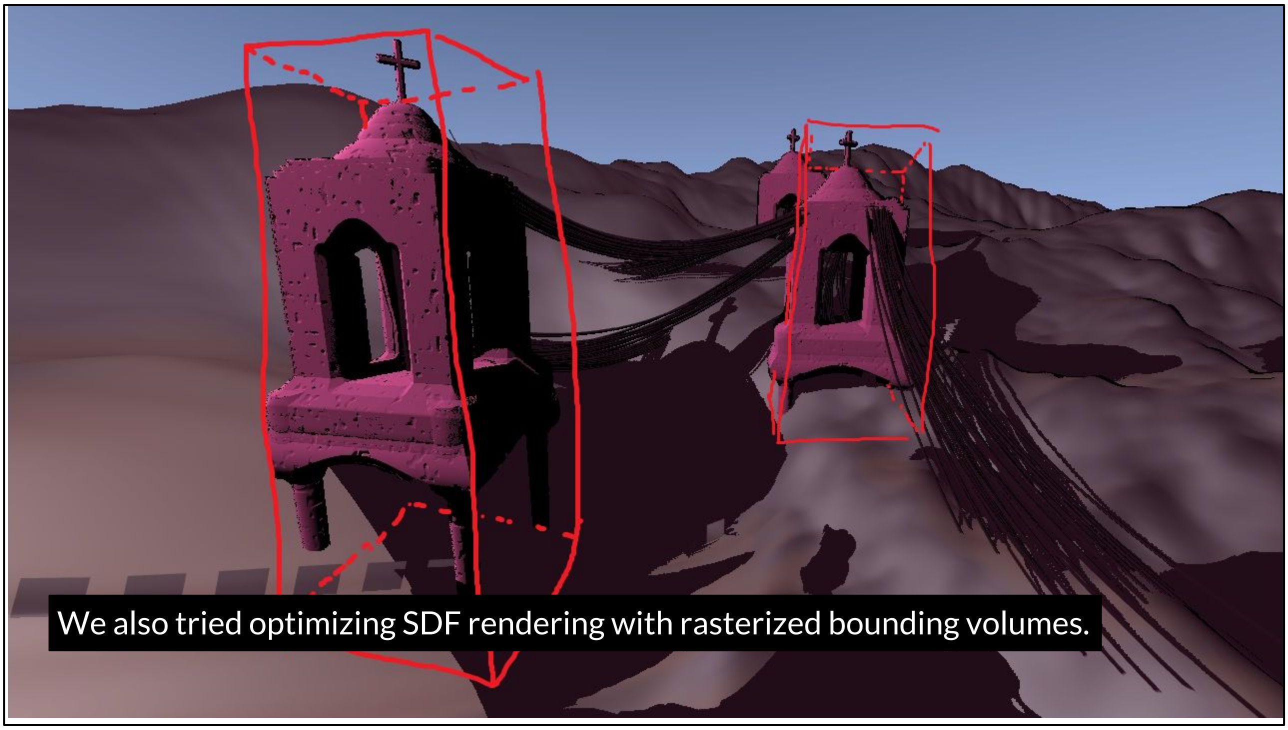
It is terrible!

...but it’s better than spending a year working on a tool.

Just deal with it. When you’ve made a couple of intros then you can start thinking of making a tool but really, don’t overthink it.
It would be easier with a tool but this isn’t about doing things the easy way, is it?

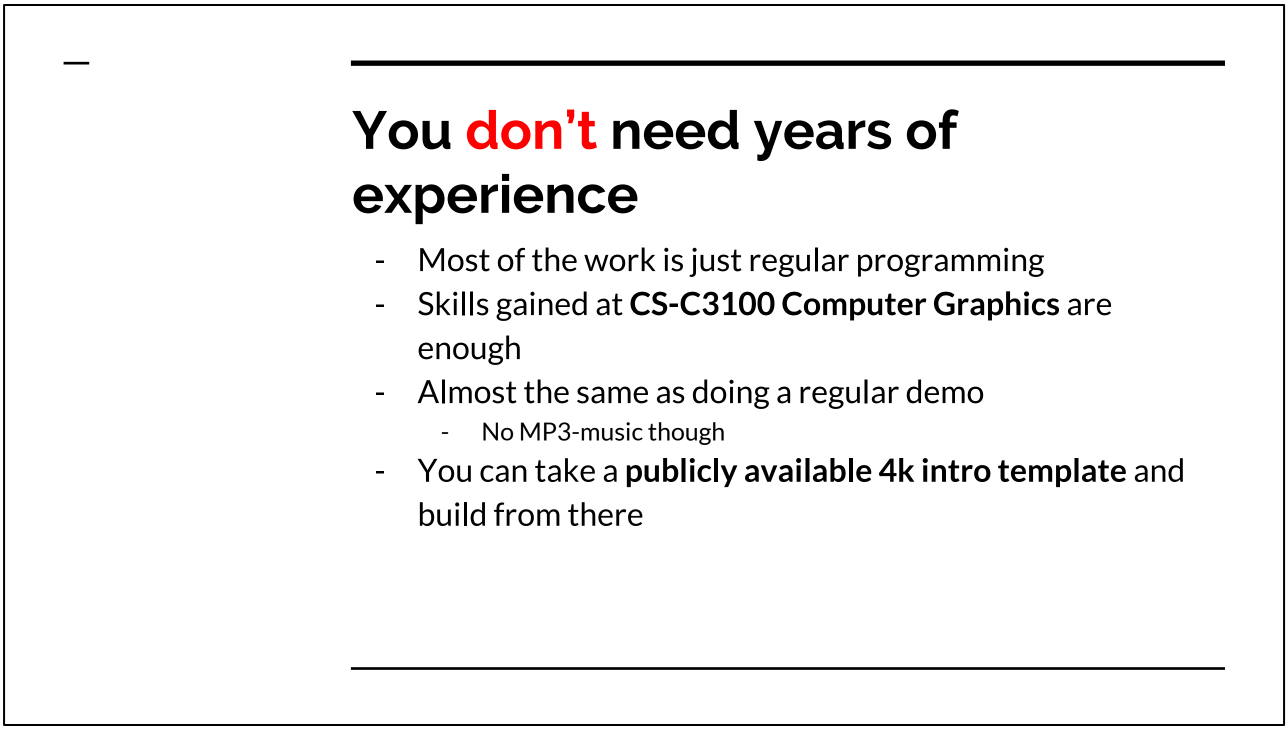
We experimented with shadow maps but that wasn’t the best idea.

As an example of a technique we didn’t end up needing: Shadow maps.
They were too slow. I wanted them to have unified global lighting but in the end we kind of did without by just hacking the lighting per scene basis.
By they way, we had only a single directional light.



We also tried optimizing SDF rendering with rasterized bounding volumes.

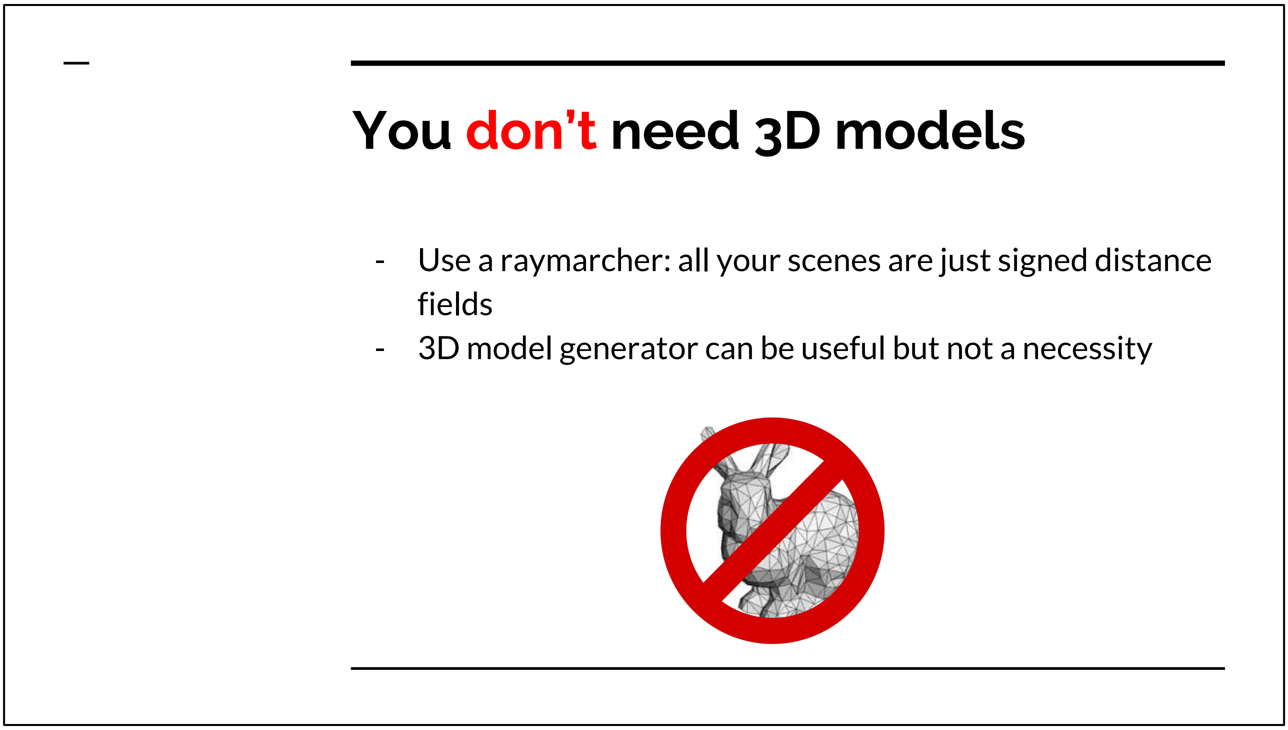
We tried a fancy SDF optimization by raymarching the shapes inside some rasterized bounding volumes but in the end this wasn’t really needed.
This is the reason I wanted shadow maps: to have unified lighting on stuff rendered with different techniques.
Martin did a lot work to figure this stuff out but we didn’t end up using it :(



You don’t need years of experience

- Most of the work is just regular programming
- Skills gained at **CS-C3100 Computer Graphics** are enough
- Almost the same as doing a regular demo
  - No MP3-music though
- You can take a **publicly available 4k intro template** and build from there

Once you have your OpenGL window on screen and shaders compiling then it’s pretty much like working on a regular demo project.
If you’ve taken the graphics course you are ready to implement pretty much all relevant techniques.

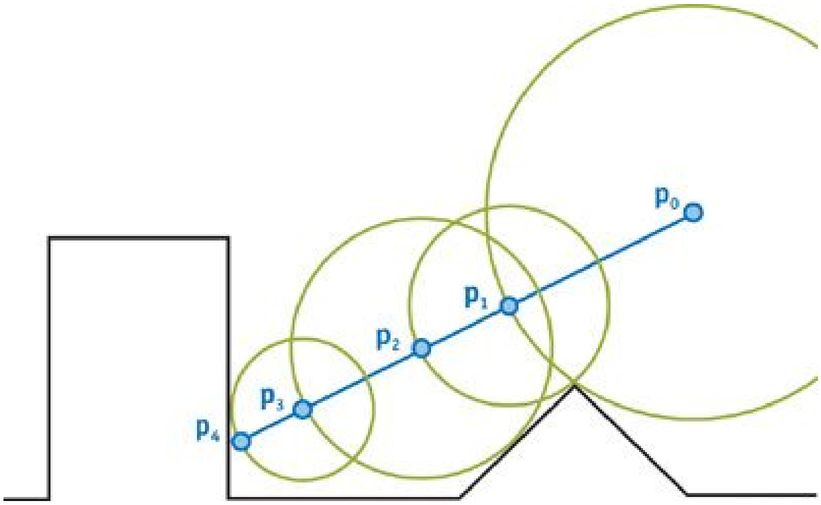


You don’t need 3D models

- Use a raymarcher: all your scenes are just signed distance fields
- 3D model generator can be useful but not a necessity

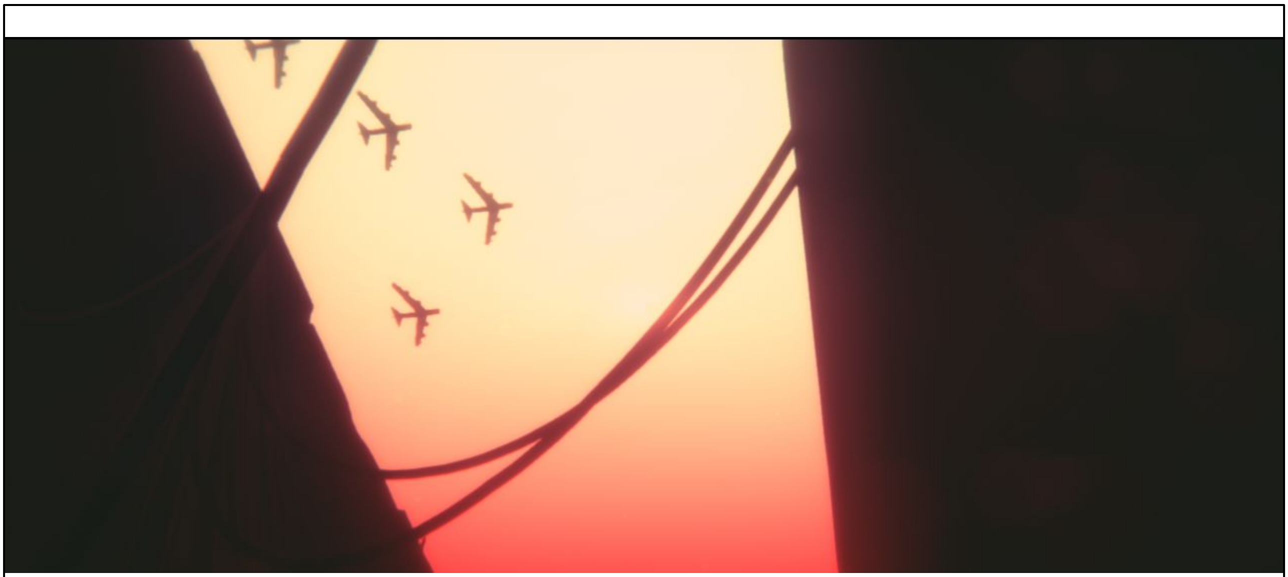
This is just something that worked very well for us: it allowed editing the scenes by shader code which was cumbersome but still better than doing it in C++.
It’s easy to generate fractals, landscapes and other interesting geometry.

Raymarching signed distance fields (SDFs)



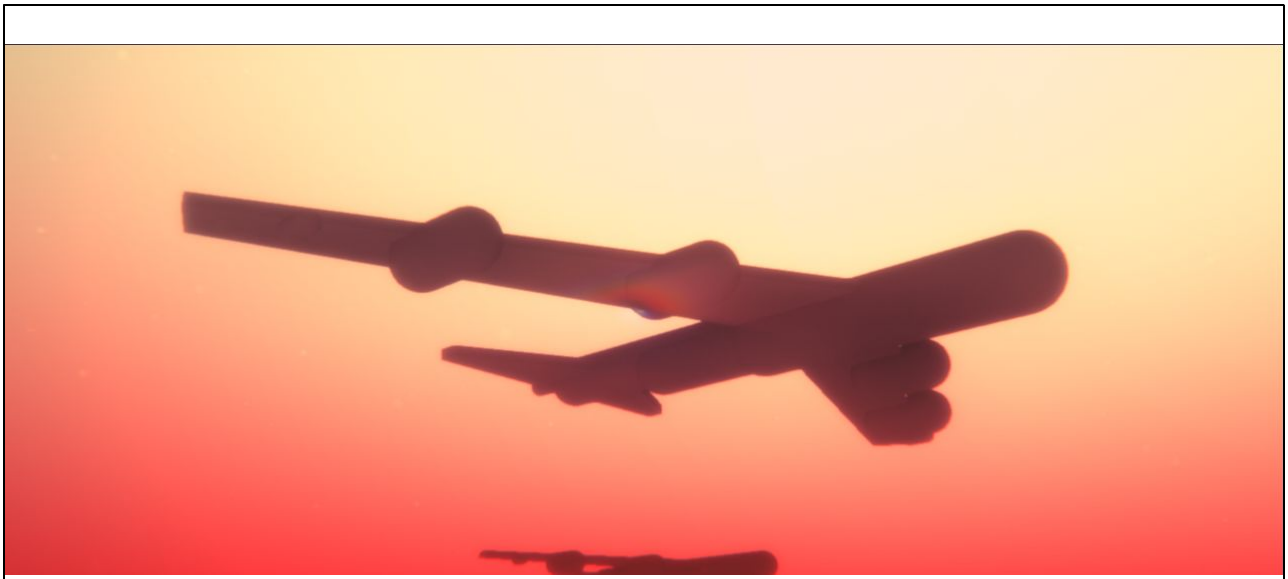
Picture from *From* GPU Gems 2: Chapter 8.

This is the same stuff everyone else is doing.
It can get slow but GPUs are also pretty fast nowadays.



Oooh! Photorealistic B-52 bomber planes!

About modeling: have a look at these bombers. They look pretty nice at a distance.



Actually just a bunch of capsules.

They are actually just a bunch of tubes. This took a long time to get right, though. Using a traditional 3D model and packing it in might have been a smarter choice.

You **definitely** need...

... native code (maybe?)

... to understand linkers

... an EXE packer

... some audio

... a way to edit values at runtime

... post-processing effects

You definitely(?) need native code

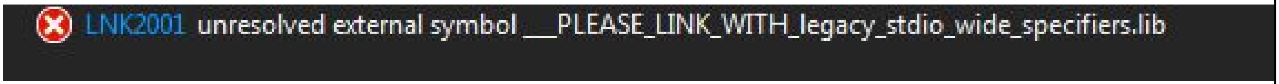
- We used VS2013 and C++
- **Web stuff is allowed too**, it just tends to break very fast
- **Logicoma**'s 64k intros are done using Rust:

Screenshots of **Engage** by **Logicoma**

Usually people develop on Windows but OS X and linux are OK too.
There was one JavaScript 64k intro at Revision this year.

You definitely need understanding of how linkers and compilers works

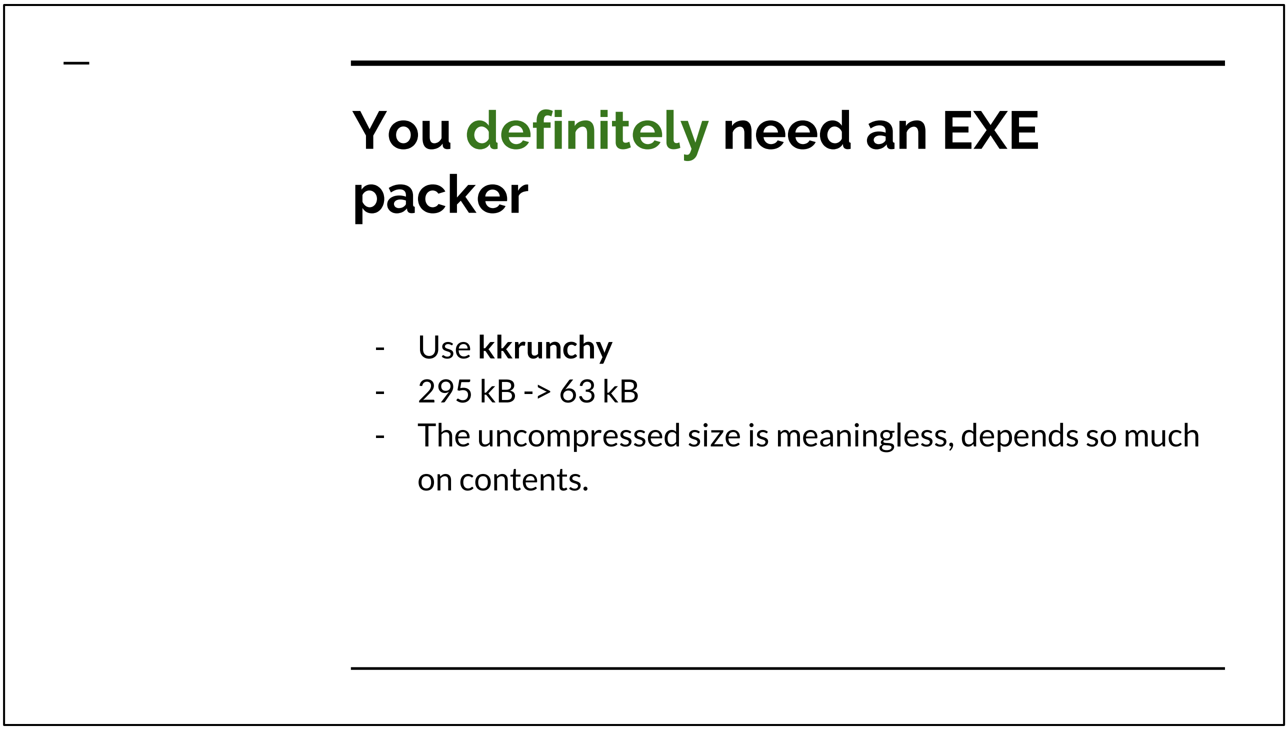
- Just to get rid of all the bloat that gets included by default!
- You *might* have to investigate weird linking problems.



```
LNK2001 unresolved external symbol ___PLEASE_LINK_WITH_legacy_stdio_wide_specifiers.lib
```

Pictured: VS2015 screaming for mercy.

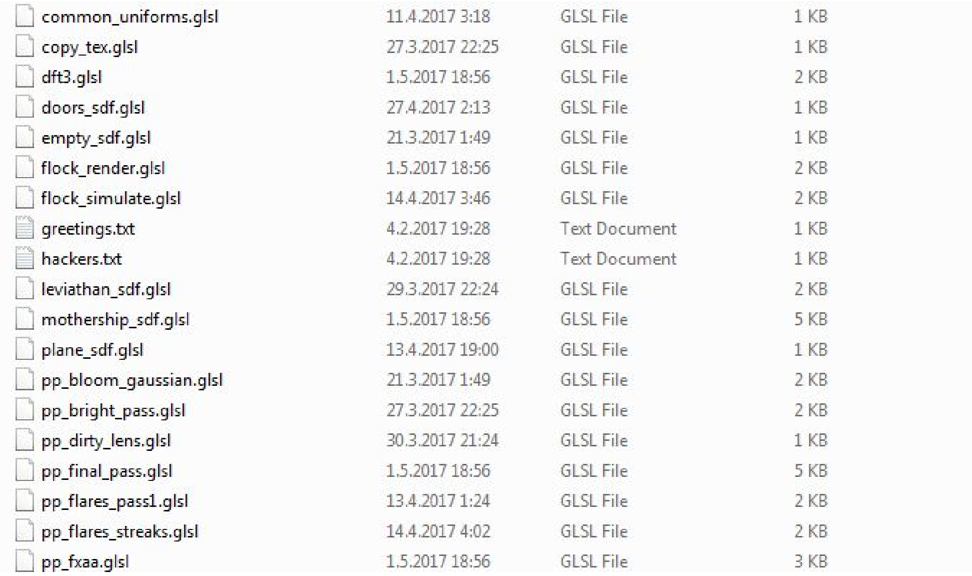
You know you're doing something right when the linker starts pleading you to stop.



You definitely need an EXE packer

- Use **kkrunchy**
- 295 kB -> 63 kB
- The uncompressed size is meaningless, depends so much on contents.

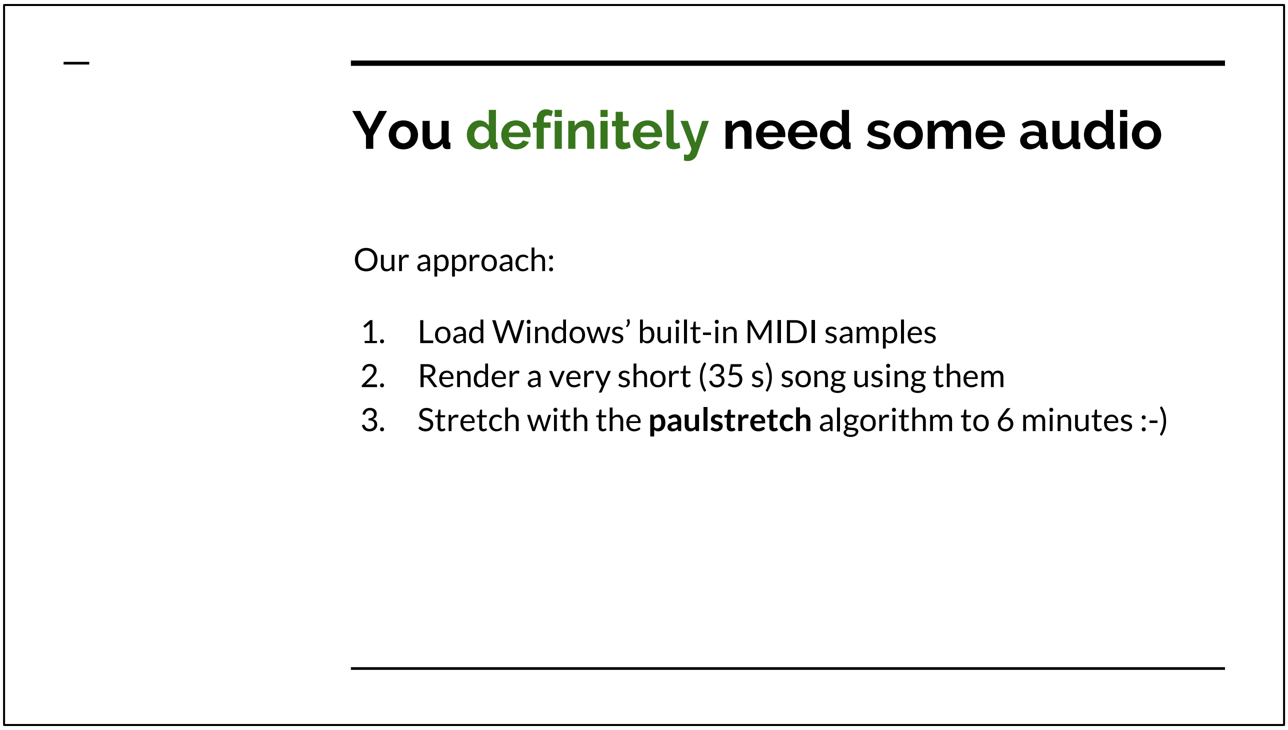
It just works. You can use it with C++, Rust, Object Pascal you name it. As long as it outputs Windows binaries.



common_uniforms.glsl	11.4.2017 3:18	GLSL File	1 KB
copy_tex.glsl	27.3.2017 22:25	GLSL File	1 KB
dft3.glsl	1.5.2017 18:56	GLSL File	2 KB
doors_sdf.glsl	27.4.2017 2:13	GLSL File	1 KB
empty_sdf.glsl	21.3.2017 1:49	GLSL File	1 KB
flock_render.glsl	1.5.2017 18:56	GLSL File	2 KB
flock_simulate.glsl	14.4.2017 3:46	GLSL File	2 KB
greetings.txt	4.2.2017 19:28	Text Document	1 KB
hackers.txt	4.2.2017 19:28	Text Document	1 KB
leviathan_sdf.glsl	29.3.2017 22:24	GLSL File	2 KB
mothership_sdf.glsl	1.5.2017 18:56	GLSL File	5 KB
plane_sdf.glsl	13.4.2017 19:00	GLSL File	1 KB
pp_bloom_gaussian.glsl	21.3.2017 1:49	GLSL File	2 KB
pp_bright_pass.glsl	27.3.2017 22:25	GLSL File	2 KB
pp_dirty_lens.glsl	30.3.2017 21:24	GLSL File	1 KB
pp_final_pass.glsl	1.5.2017 18:56	GLSL File	5 KB
pp_flares_pass1.glsl	13.4.2017 1:24	GLSL File	2 KB
pp_flares_streaks.glsl	14.4.2017 4:02	GLSL File	2 KB
pp_fxaa.glsl	1.5.2017 18:56	GLSL File	3 KB

We didn't run out of space even though we kept the filenames and other junk.

We have a python script that converts data files into C headers that can be loaded like files in Release mode.



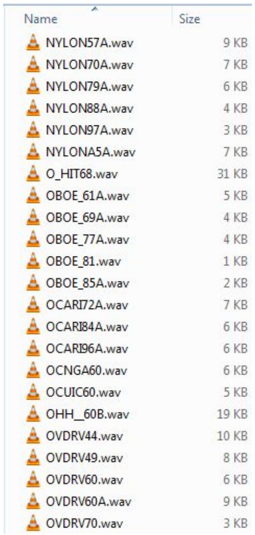
You definitely need some audio

Our approach:

1. Load Windows’ built-in MIDI samples
2. Render a very short (35 s) song using them
3. Stretch with the **paulstretch** algorithm to 6 minutes :-)

A 64k intro must have some kind of music or noise. I’ll tell about our approach first and a bit later about the alternatives.

GM.DLS

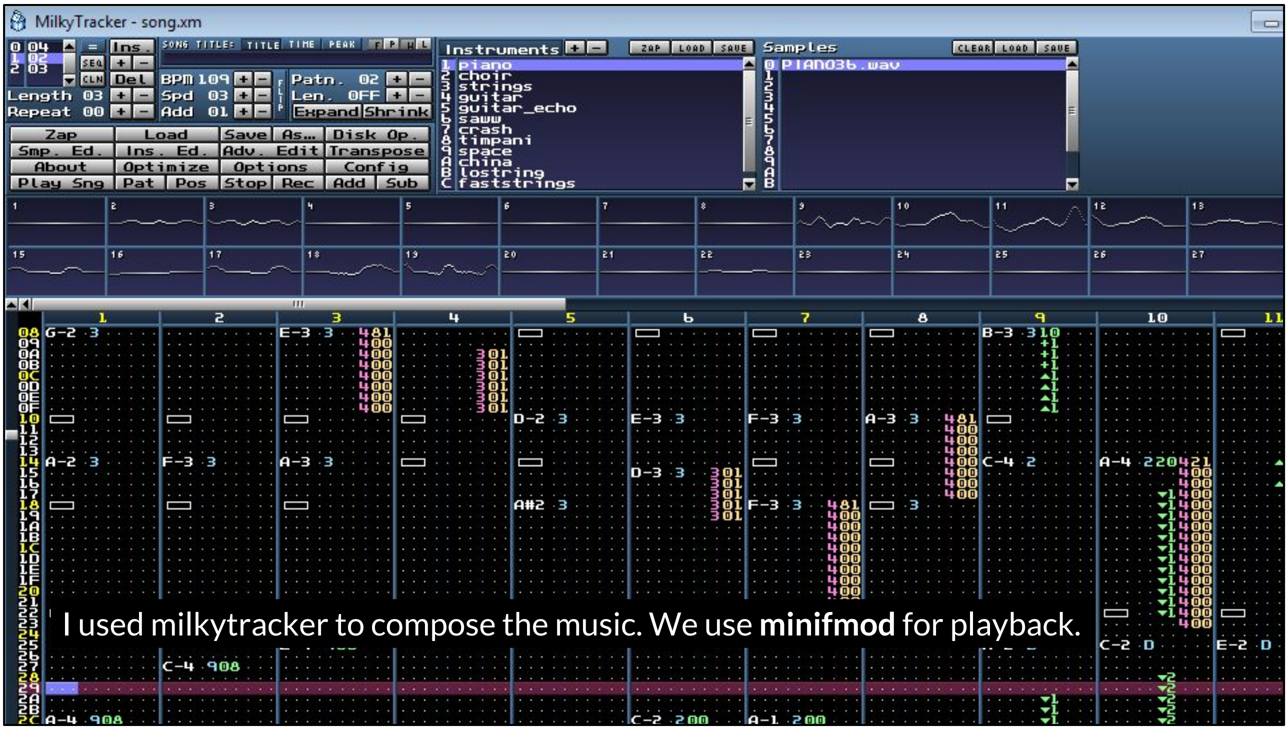


Name	Size
NYLON57A.wav	9 KB
NYLON70A.wav	7 KB
NYLON79A.wav	6 KB
NYLON88A.wav	4 KB
NYLON97A.wav	3 KB
NYLONA5A.wav	7 KB
O_HIT68.wav	31 KB
OBOE_61A.wav	5 KB
OBOE_69A.wav	4 KB
OBOE_77A.wav	4 KB
OBOE_81.wav	1 KB
OBOE_85A.wav	2 KB
OCARI72A.wav	7 KB
OCARI84A.wav	6 KB
OCARI96A.wav	6 KB
OCNGA60.wav	6 KB
OCUIC60.wav	5 KB
OHH__60B.wav	19 KB
OVDRV44.wav	10 KB
OVDRV49.wav	8 KB
OVDRV60.wav	6 KB
OVDRV60A.wav	9 KB
OVDRV70.wav	3 KB

- A cheap trick: load some Windows' built-in MIDI sounds from `C:\Windows\System32\drivers\gm.dls`
- Compose a song using those

Samples extracted with **dlsdumper**

Still present in Windows 10.



I used milkytracker to compose the music. We use **minifmod** for playback.

A classic demoscene music production tool. Time goes from top to bottom and audio channels flow from left to right.

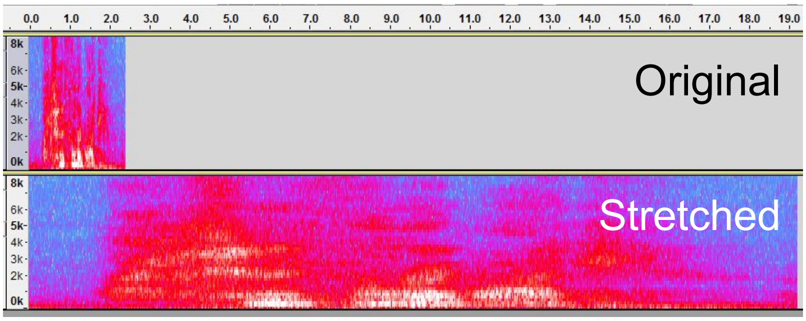
I didn’t want to write my own music editing tool so this was a good choice for me, because I was already familiar with it.

Paulstretch

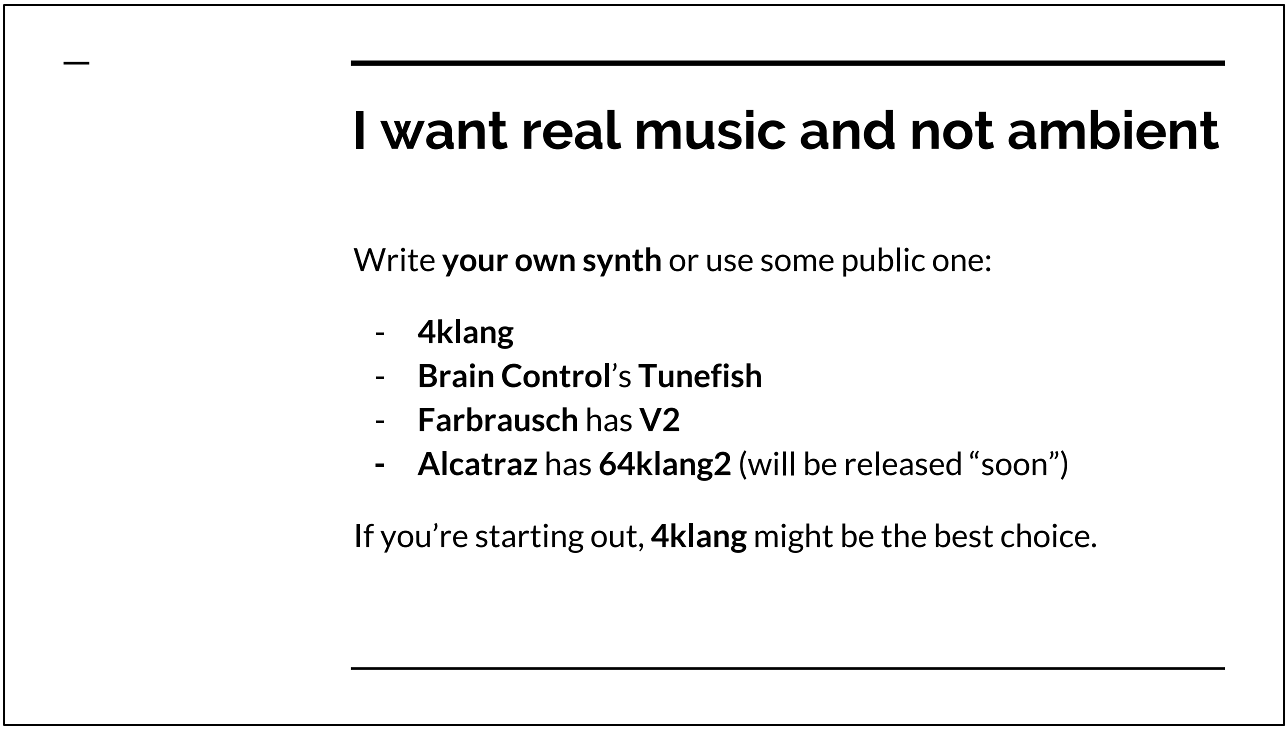
A pretty straightforward stretching algorithm. Requires Fourier transform: brute force on the GPU.

Input length: **30 s**

Output length: **5 m 40 s**



This took a long time to get right even though it's a simple algorithm.
It runs reasonably fast, the music precomputation takes something like 15 seconds on a GTX 1060.



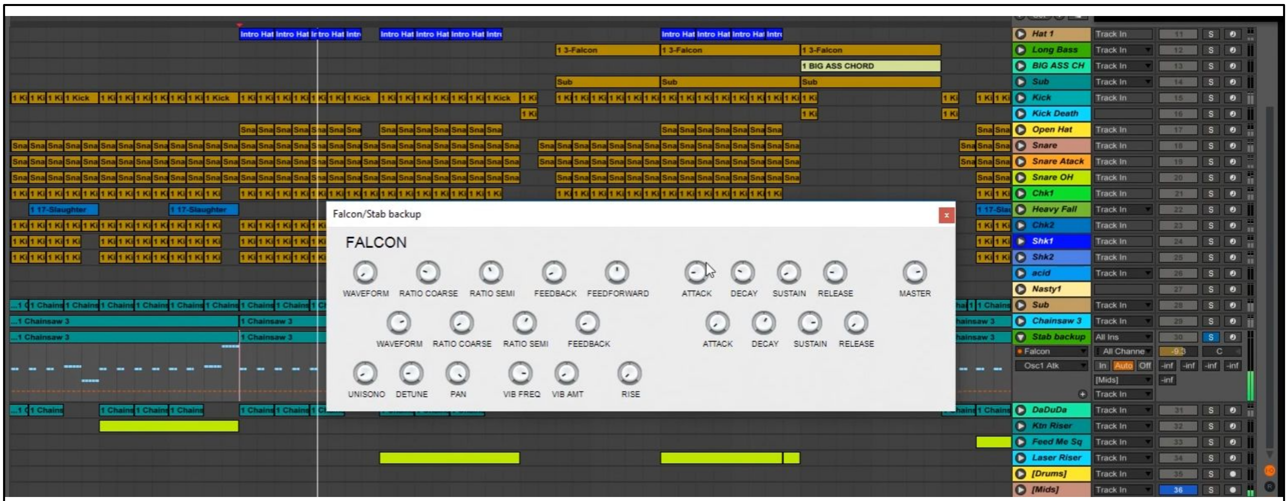
I want real music and not ambient

Write **your own synth** or use some public one:

- **4klang**
- **Brain Control**’s **Tunefish**
- **Farbrausch** has **V2**
- **Alcatraz** has **64klang2** (will be released “soon”)

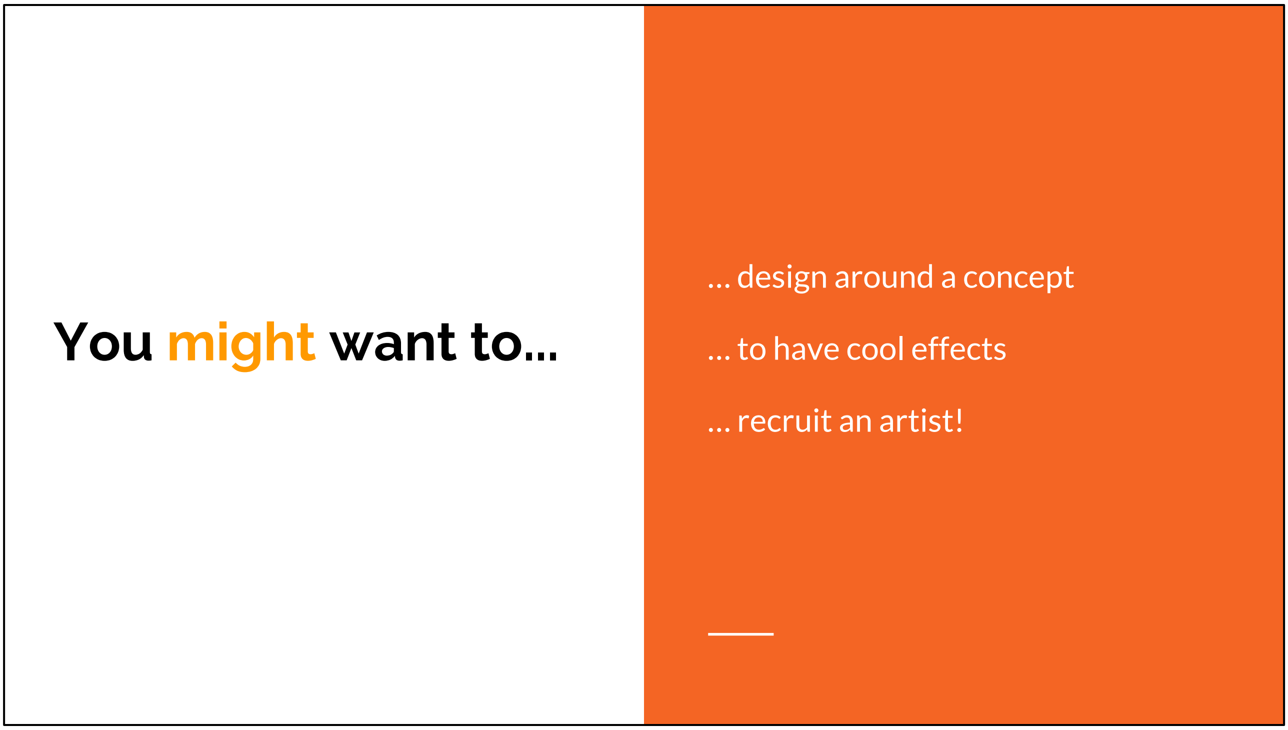
If you’re starting out, **4klang** might be the best choice.

4klang has some examples online but the VST instrument is difficult to use.
Writing your own can be fun but it’s hard to do something that’s nice to use.



For inspiration: **Logicoma’s Wavesabre** (unreleased) is a collection of VST instruments that you can use with Ableton live, and then convert your song to a custom format.

This is a pretty cool system. You can check out ferris’ streams if you want to see it in action. It’s actually pretty simple but making those VSTis must have been a lot of work.



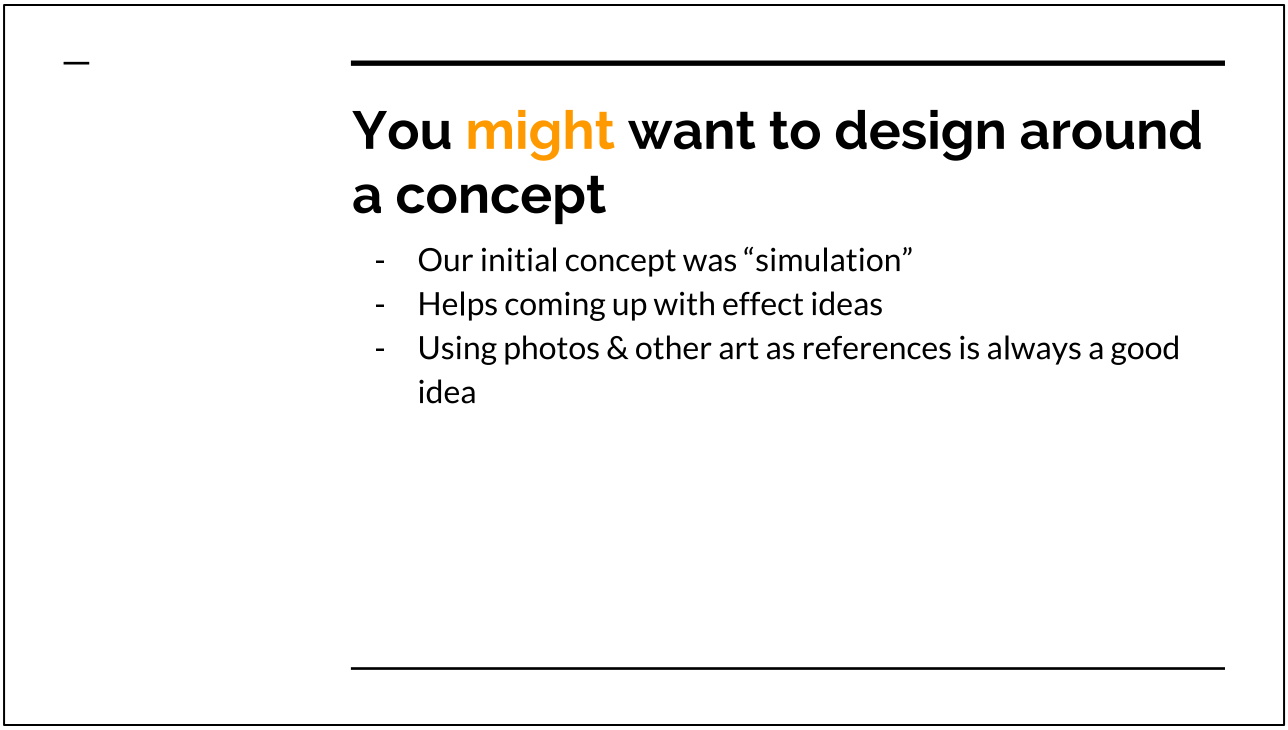
You might want to...

... design around a concept

... to have cool effects

... recruit an artist!

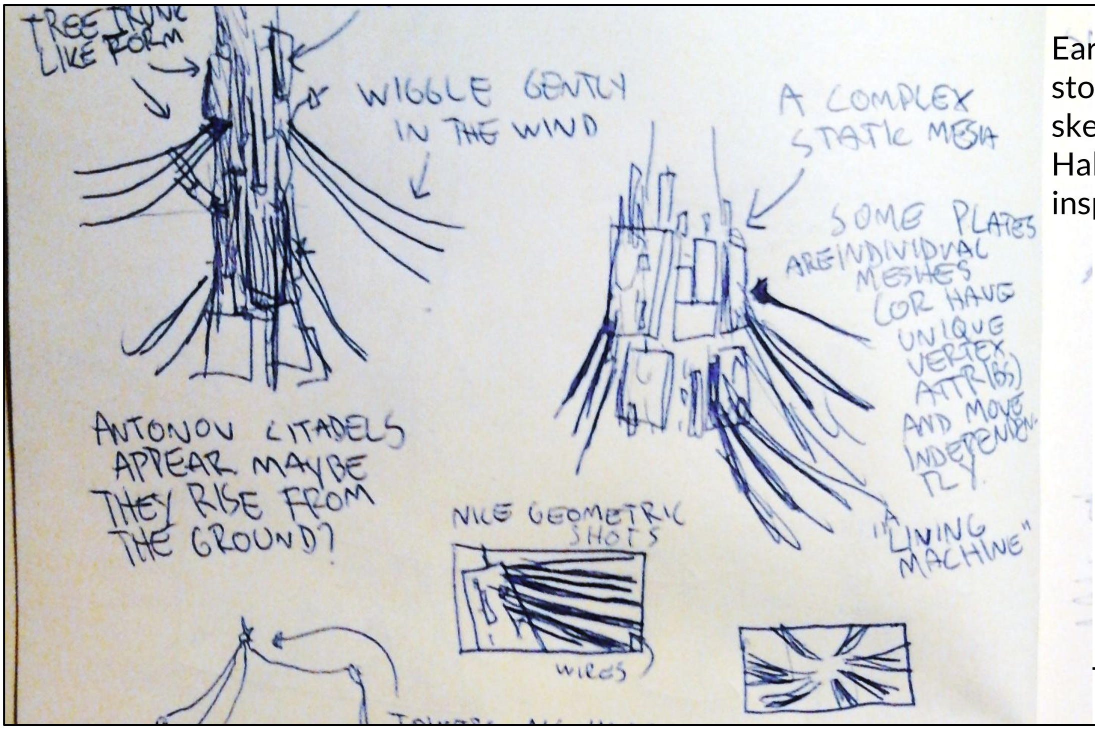
Ok so you have the basics right, what else do we need?



You might want to design around a concept

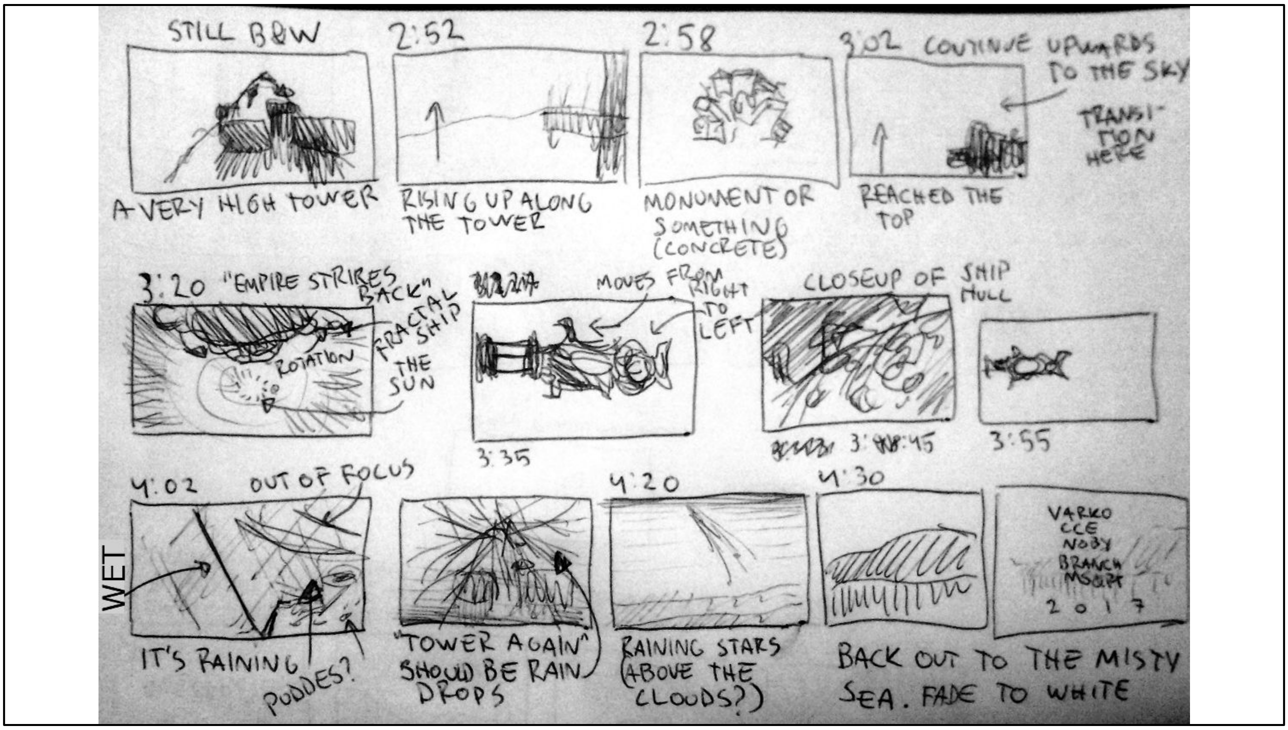
- Our initial concept was “simulation”
- Helps coming up with effect ideas
- Using photos & other art as references is always a good idea

It would’ve been nice to use those wires a bit more. So the concept wasn’t very clear in our intro but it still helped.


TREE TRUNK LIKE FORM
WIGGLE GENTLY IN THE WIND
A COMPLEX STATIC MESH
SOME PLATES ARE INDIVIDUAL MESHES (OR HAVE UNIQUE VERTEX ATTRIBS) AND MOVE INDEPENDEN-TLY.
"LIVING MACHINE"
ANTONOV CITADELS APPEAR MAYBE THEY RISE FROM THE GROUND?
NICE GEOMETRIC SHOTS
WIRES

Early storyboard sketches of a Half-Life 2 inspired tower.

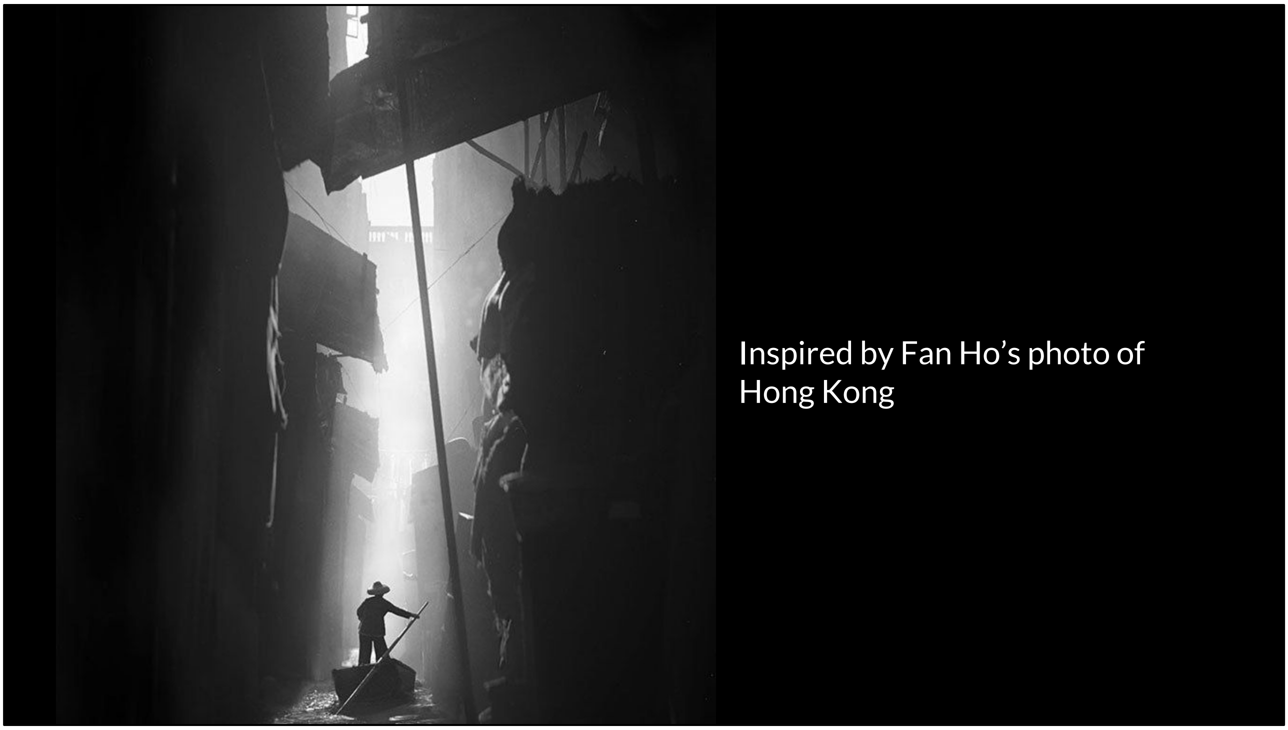
Viktor Antonov towers.


STILL B&W
A VERY HIGH TOWER
2:52
RISING UP ALONG THE TOWER
2:58
MONUMENT OR SOMETHING (CONCRETE)
3:02 CONTINUE UPWARDS TO THE SKY
REACHED THE TOP
TRANSITION HERE
3:20 "EMPIRE STRIKES BACK"
FRACTAL SHIP
ROTATION
THE SUN
MOVES FROM RIGHT TO LEFT
3:35
CLOSEUP OF SHIP HULL
3:45
3:55
4:02 OUT OF FOCUS
WET
IT'S RAINING
PUDDLES?
"TOWER AGAIN" SHOULD BE RAIN DROPS
4:20
RAINING STARS (ABOVE THE CLOUDS?)
4:30
BACK OUT TO THE MISTY SEA. FADE TO WHITE
VARKO
CCE
NOBY
BRANCH
MSGPT
2017

Having a storyboard was maybe a bit too ambitious here but it helped me personally to flesh out some ideas.

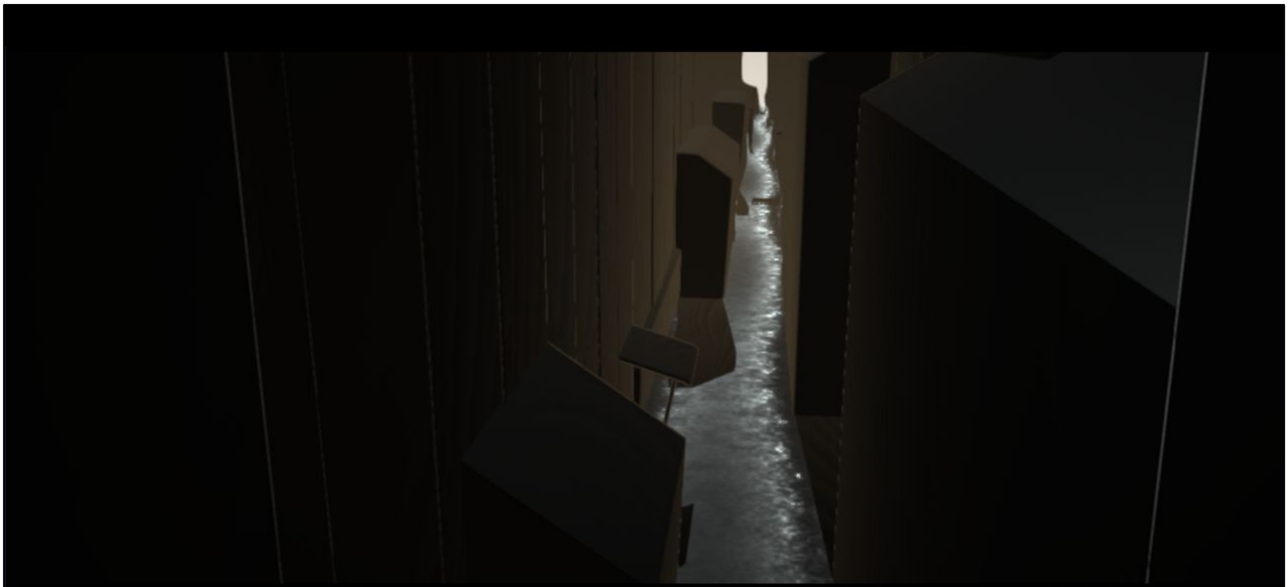
A cool effect: the Alley

So you might want some cool effects too. Here’s one: the alley scene.
It’s a good example why you need post processing effects. They hide a lot of mistakes in the scene.



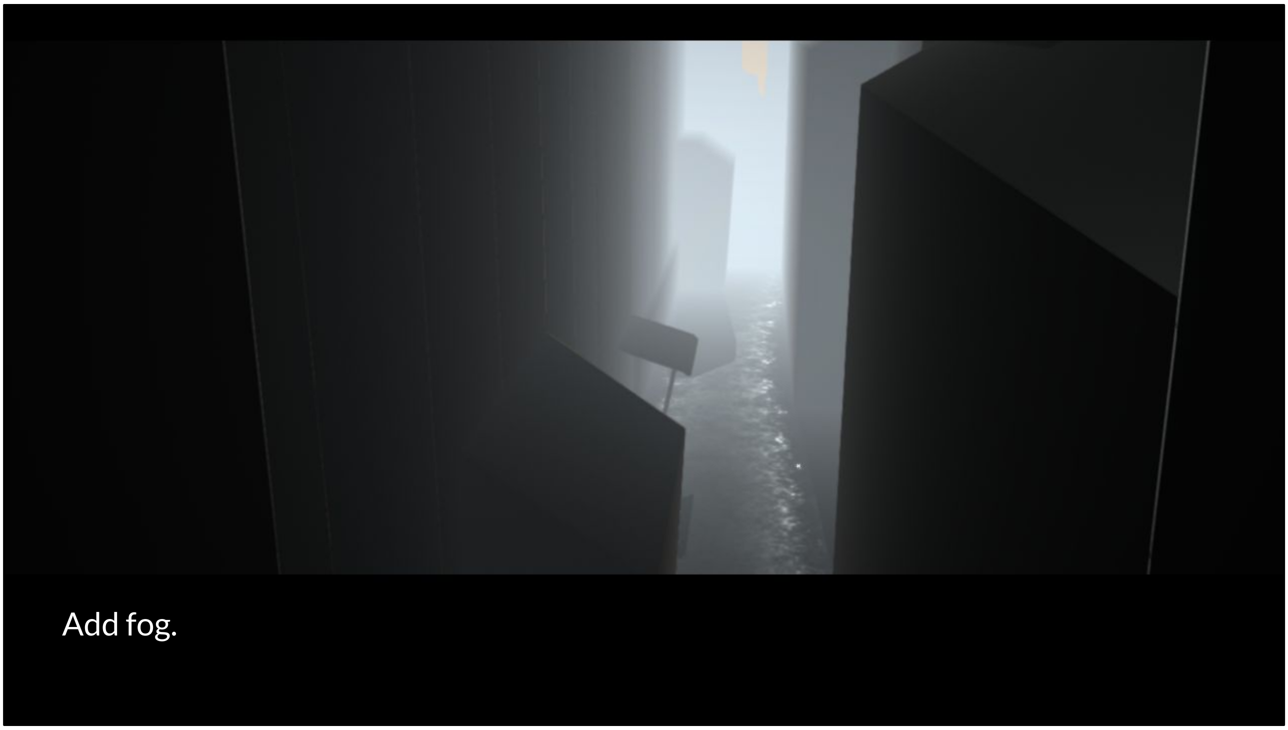
Inspired by Fan Ho’s photo of Hong Kong

Having a reference pic really helped us capture the mood.
When Roope helped us with the design for the final version it was great to point at this and say “I want this”



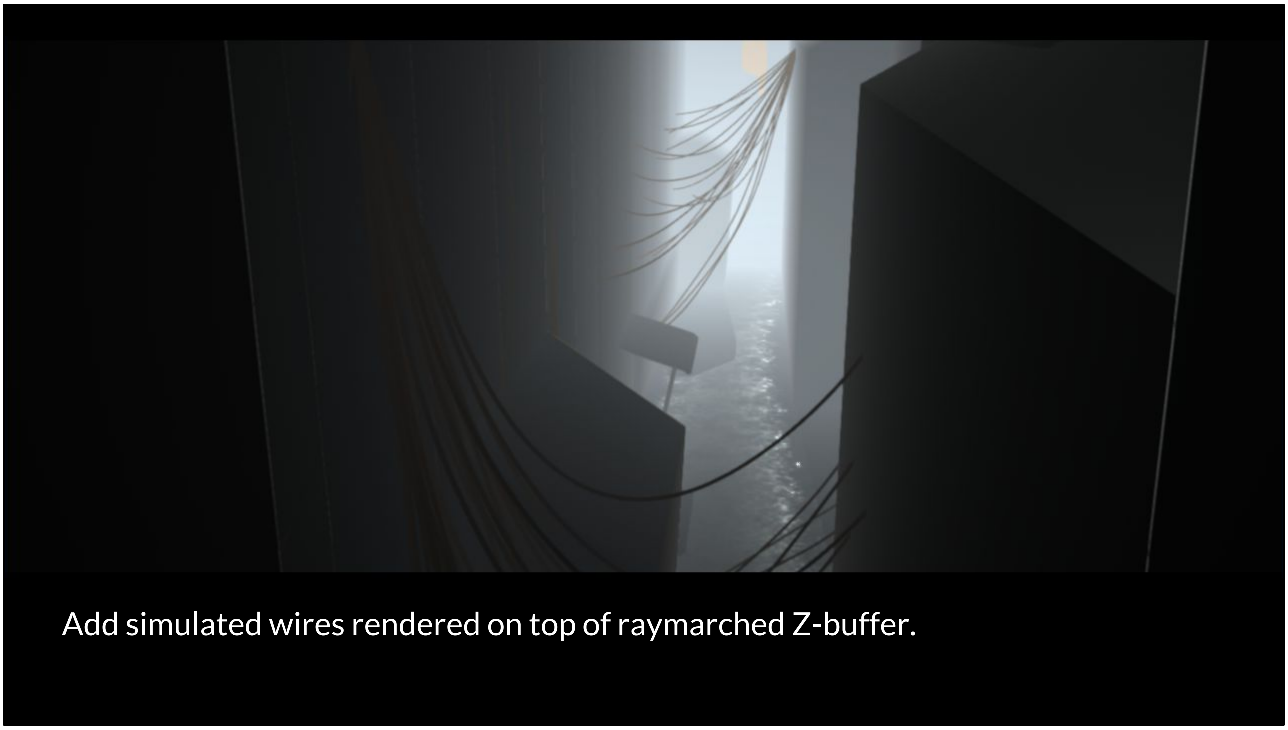
Initial alley scene SDF.

Modeled using GLSL



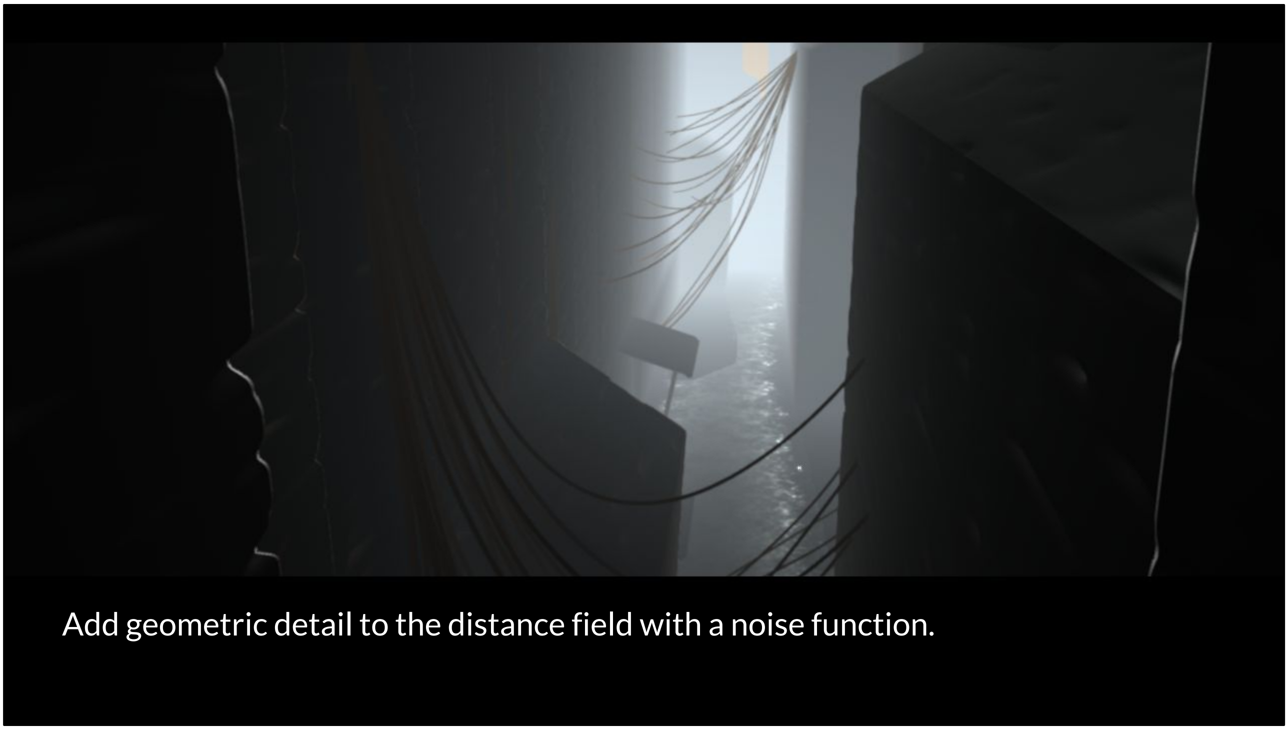
Add fog.

A super simple fog effect. Already makes it look larger.



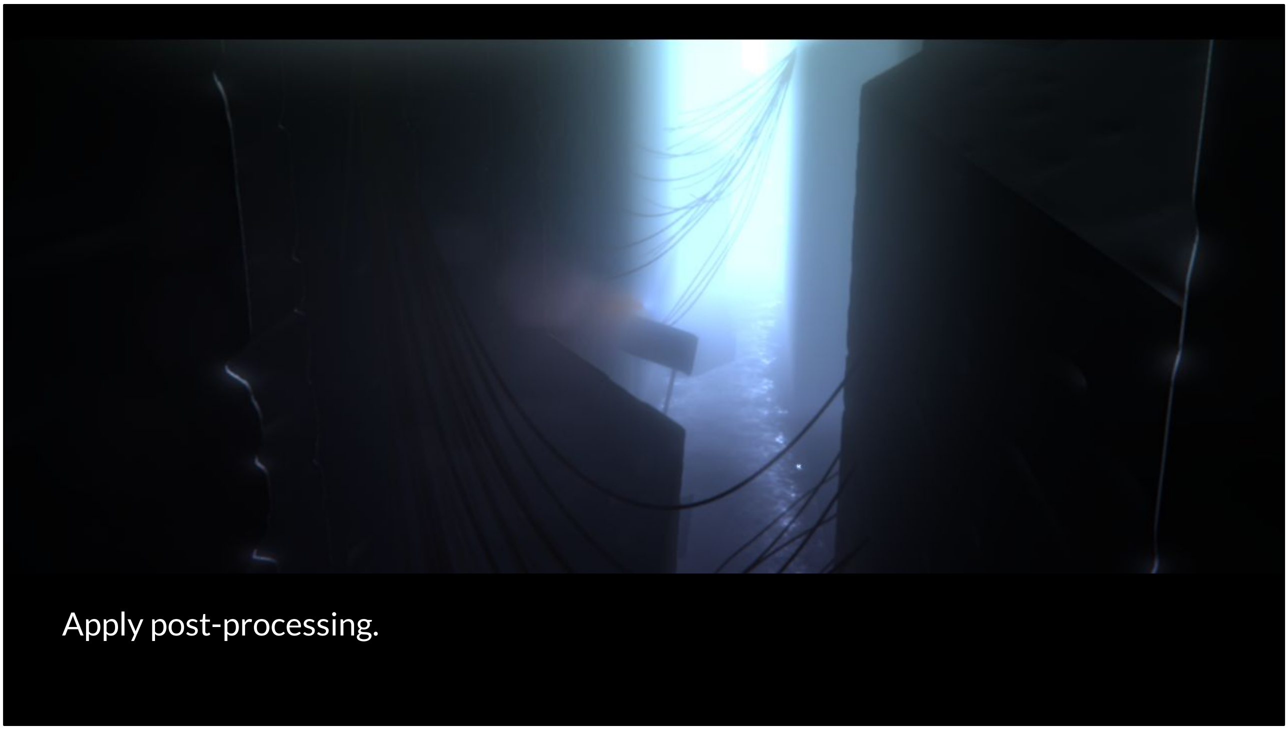
Add simulated wires rendered on top of raymarched Z-buffer.

The wires add some subtle animation and make it more visually interesting.



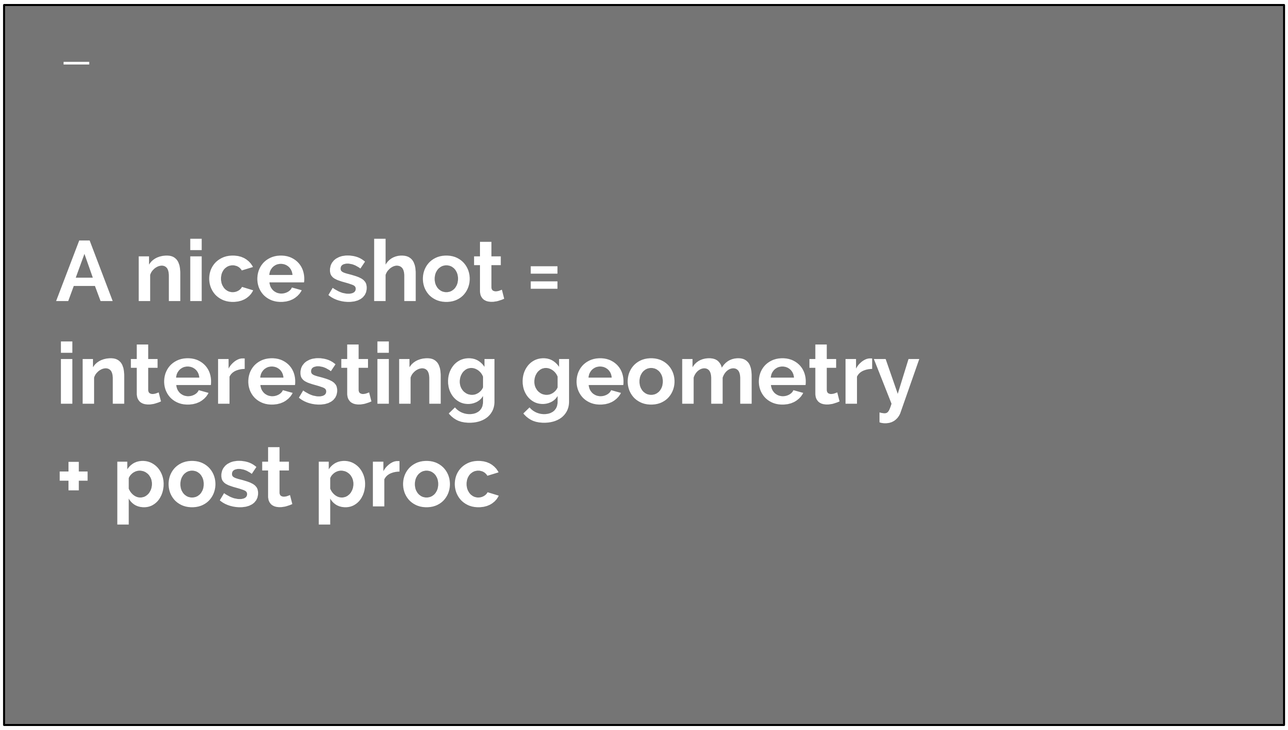
Add geometric detail to the distance field with a noise function.

This was added in the final version. Perturb the distance field so the walls look like brick walls.



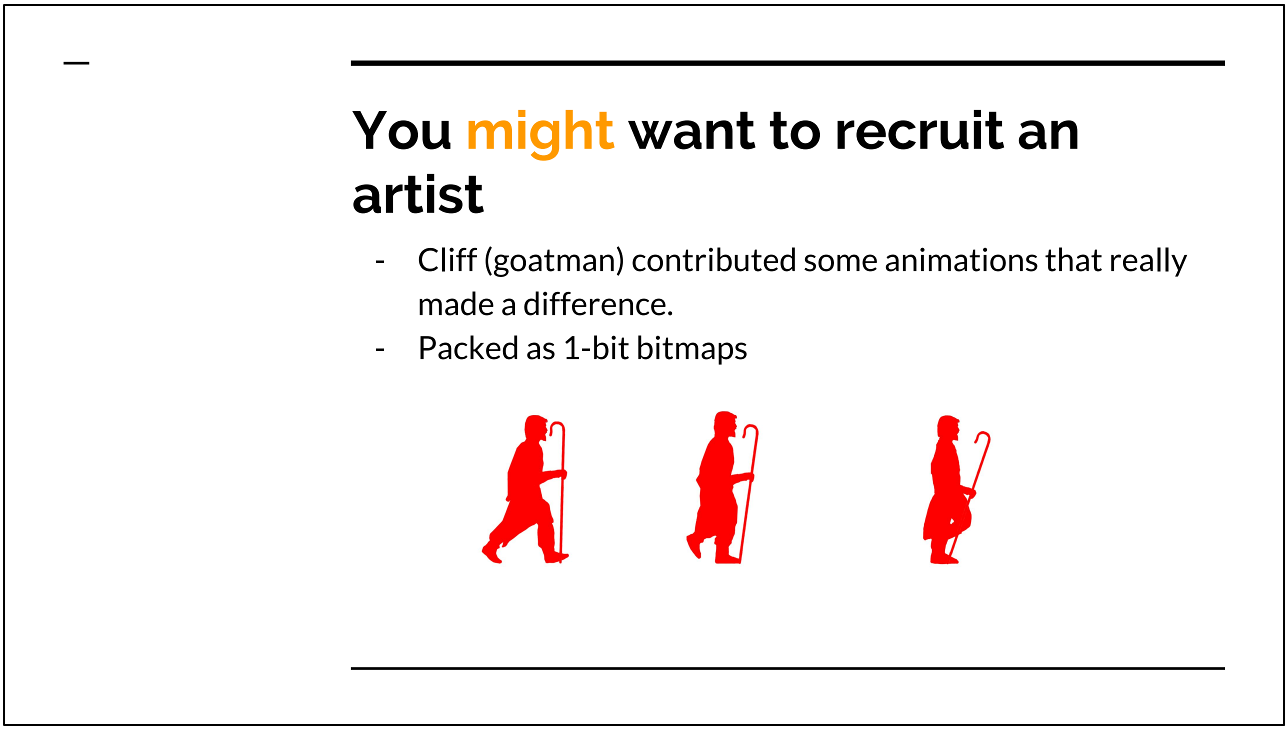
Apply post-processing.

This is why you need post processing effects. They hide a lot of mistakes in the scene.
There’s bloom, some lens flares, chromatic aberration and a screen space color gradient.



A nice shot = interesting geometry + post proc

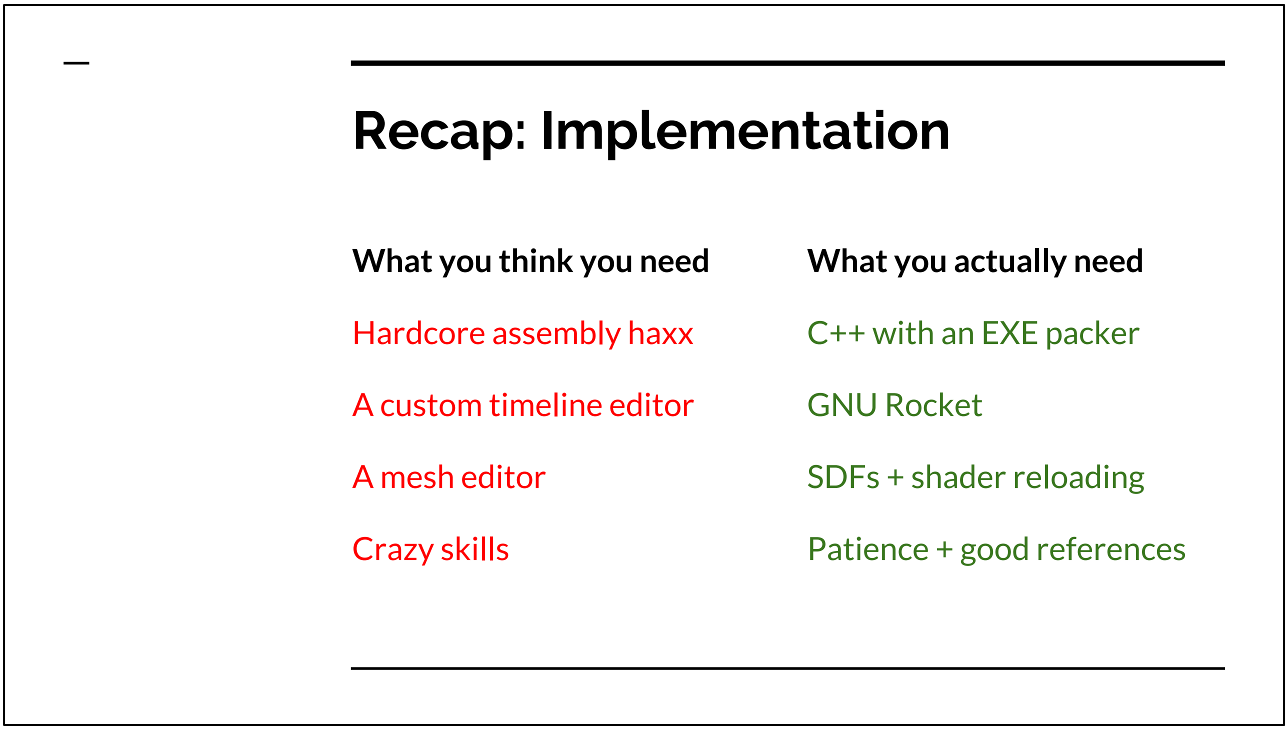
In this case we only needed some nice geometry and post processing.
We didn’t need ultra realistic materials or crazy camera angles.



You might want to recruit an artist

- Cliff (goatman) contributed some animations that really made a difference.
- Packed as 1-bit bitmaps

We fade inbetween frames. Martin implemented a packing scheme that utilizes temporal locality by packing all eight consecutive frames into a single byte.

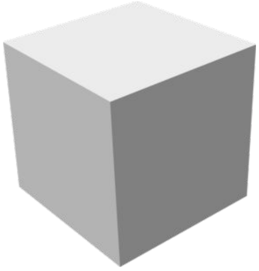


Recap: Implementation

What you think you need	What you actually need
Hardcore assembly haxx	C++ with an EXE packer
A custom timeline editor	GNU Rocket
A mesh editor	SDFs + shader reloading
Crazy skills	Patience + good references

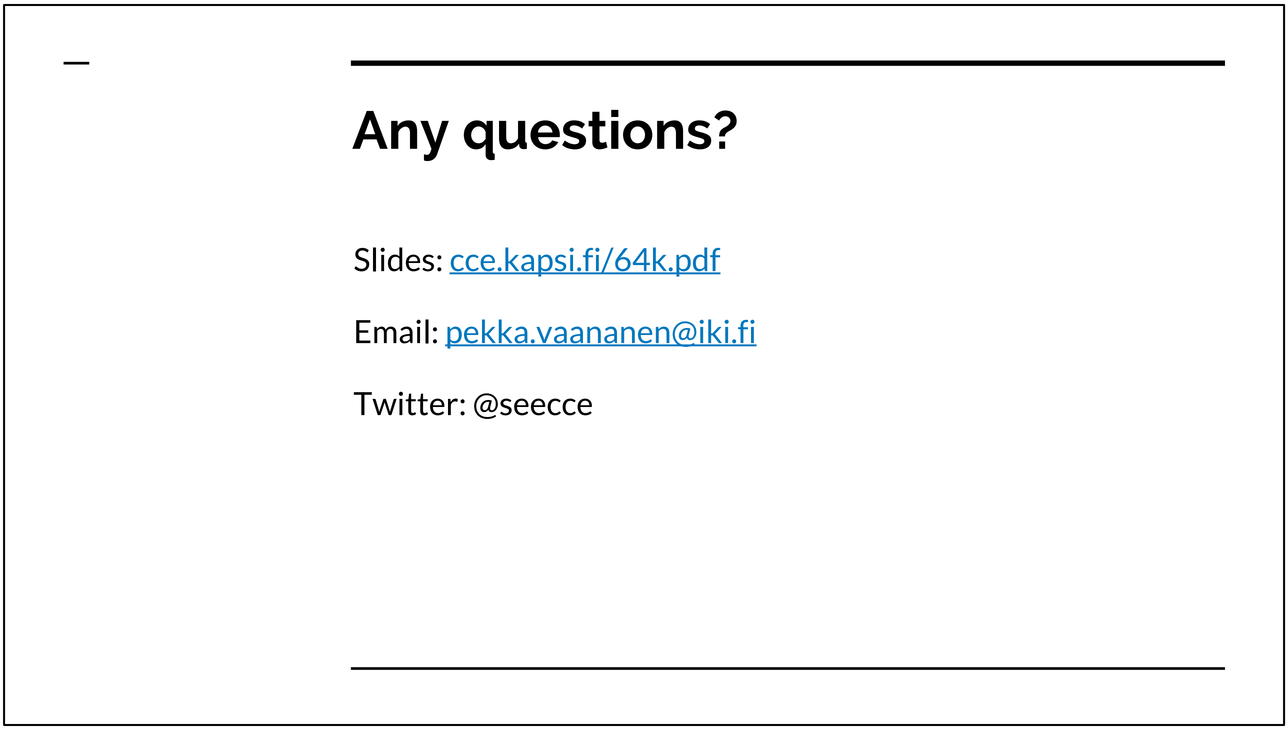
This isn’t really about being the best coder. It’s about thinking critically what exactly do you need and what’s the easiest way to implement it.
Cheating is allowed if you do it with style :)

A recipe for your first 64k



1. Download some 4k intro template project
2. Add some music with e.g. **4klang**
3. Write some huge shaders
4. Tweak them (during runtime) until they look nice
5. Pack with **kkrunchy**
6. Release at some party!

We had our own 64k intro template copy pasted from different demo sources.
Might be a good idea just to write a regular demo first with mp3 music etc.



Any questions?

Slides: cce.kapsi.fi/64k.pdf

Email: pekka.vaananen@iki.fi

Twitter: @seecce

The blog post at www.lofibucket.com