INSIDE shipping on iOS

Mikkel Gjøl
Graphics Programmer, PLAYDEAD
@pixelmager

version 1.0

Digital Dragons
21-22 May 2018
Krakow, Poland

KRAKOW TECHNOLOGY PARK

github.com/playdeadgames/publications

www.playdead.com

Released a game called **LIMBO** back in 2012
And more recently **INSIDE** in 2016

Not a talk on how we **MADE** those games – but rather how we **SHIPPED** them

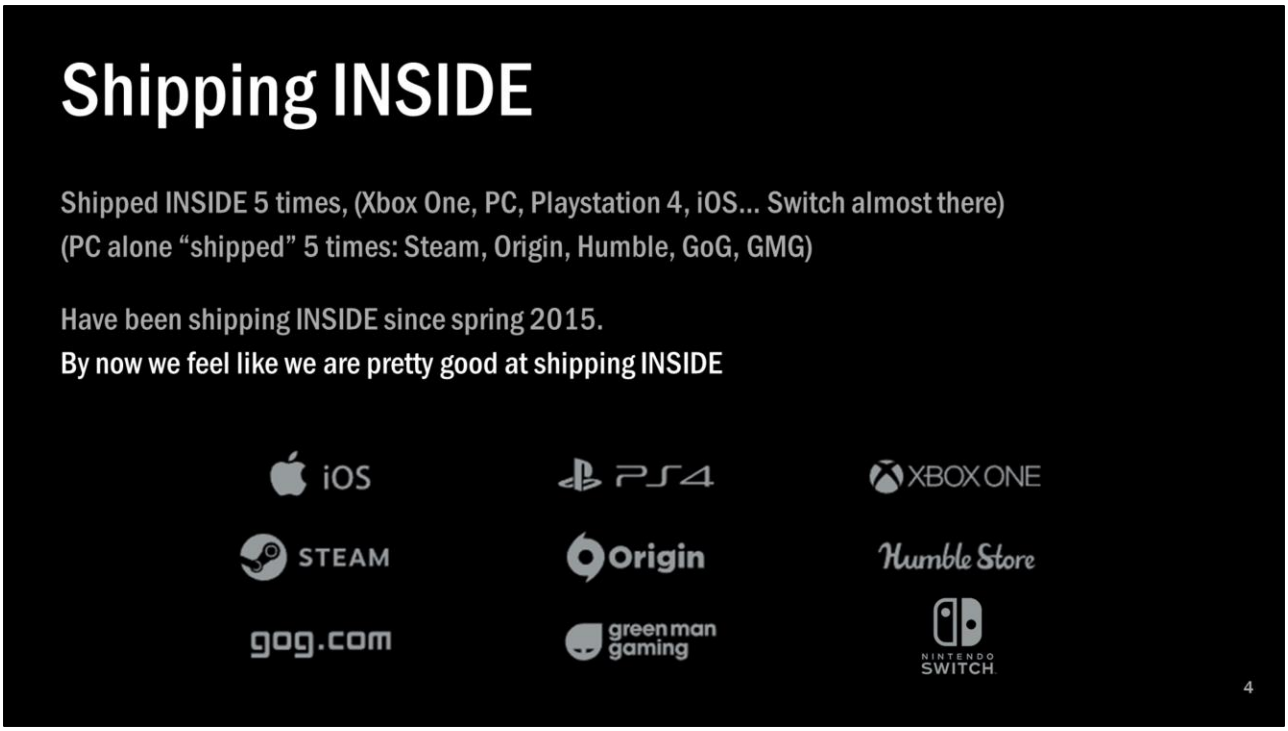
Presenting the work of...

INSIDE by PLAYDEAD

iOS port

- Kristian Kjems
- Mikael Garde Nielsen
- Erik Rodrigues Pedersen
- Lasse Fuglsang
- Maria Angelova
- Tobias Biehl
- Martin Fasterholdt
- Søren Trautner
- Nicholas Brancaccio

(Didn’t do this alone..)



Shipping INSIDE is something we have been doing for a while - a couple of years now

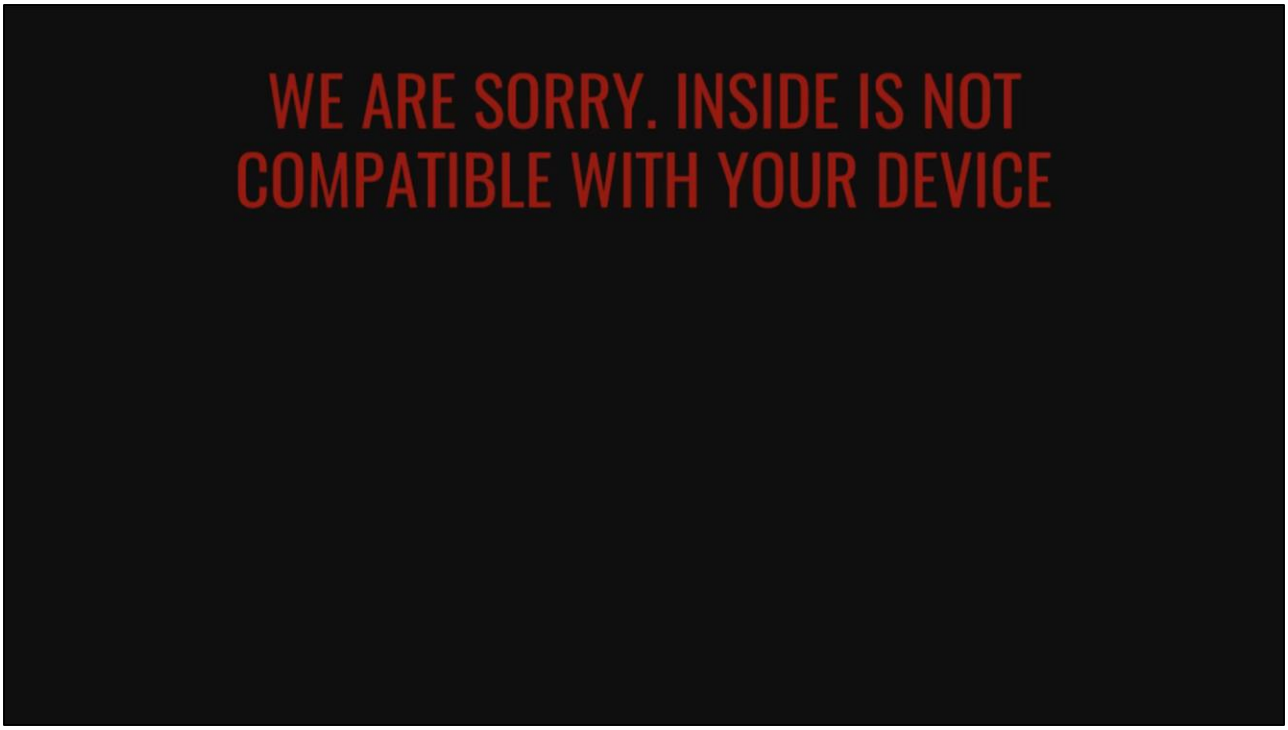
We don’t do mobile-work often - roughly every 5 years, LIMBO released on IOS in 2013

Strictly talking **about inside** - other games probably have other issues.
INSIDE is not a very common mobile-game...

Same gameplay, levels, rendering-pipeline (LightPrepass), geometry, textures, shaders, particle-effects, animations, everything

Slightly tweaked gameplay to make touch-controls work better

High-end mobile CPUs, e.g. ipad pro 2 10”, roughly on par with laptops

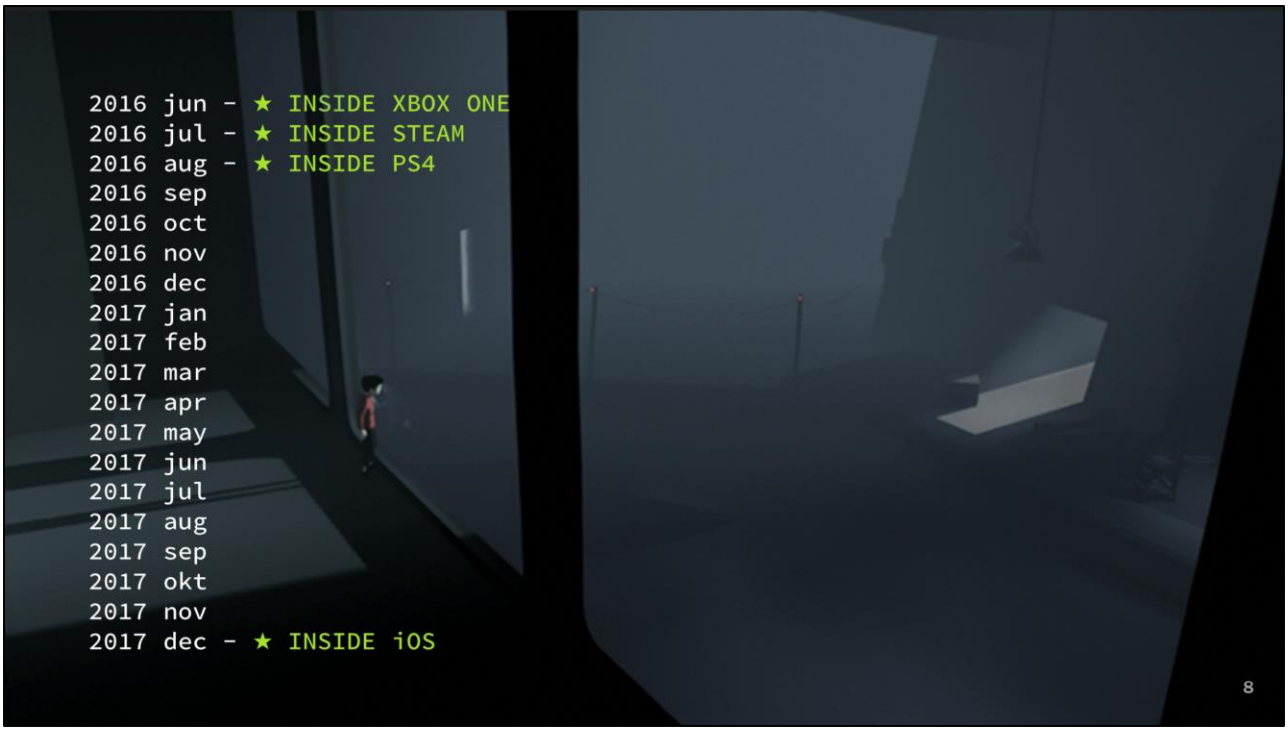


Not wanting to cut content…
….and INSIDE being a **console** game…
We were unable to fit within the roughly 600MB of memory apps get on 1GB-RAM devices.

1GB device => ~600MB **MEMORY** app
2GB device => ~1200MB app
INSIDE ~900MB (worst case)

No way to filter for memory in appstore, no way to blacklist devices (unlike on android)
– so our only option to make sure people would not be able to pay for the game without being able to play it
was to make the game free with an in-app-purchase to unlock the full game.

⇒ made initial part of game a **free trial**
⇒ **Trial unlock: iap**
⇒ providing fallback “sorry”-scene
- can download on incompatible device
- ...but not buy it
=> hopefully less disappointed customers: **>:(** => **:´/**



Had been finalizing/shipping xbone/ps4/pc since start 2015

Maintained functional ps4/xbox/pc releases during production (developed xbox one, as it was the slower device, tested on all platforms for at least the last couple of years before release)

Updated from Unity 5.0 -> Unity 5.6 (to get 64bit required for ios, and metal2 useful for ios)
- removed our custom renderer multi-threading used on consoles
- kept other changes like
    - static shadowmaps
    - local directional lights
    - Lightpass (for ss decals)

Initial ios-port - proof-of-concept, getting an idea about the remaining work

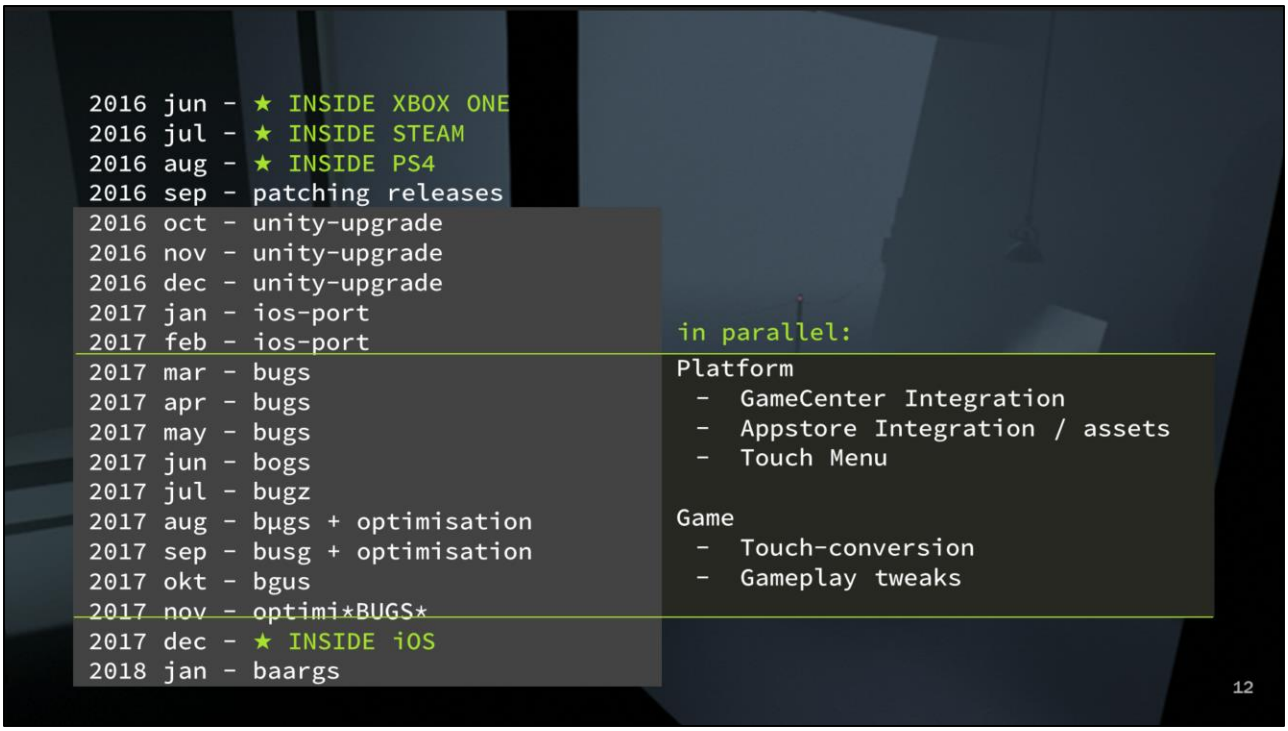
Tons of bugs, tons of missing stuff, but the game was playable and ran at around 30fps

Bug sources mainly
- Unity upgrade (tons of stuff breaking on upgrade)
- IOS specific choices (everything from NaNs in shaders, not using FMAs for vertex-transform, tiled issues, using floats for for-loop-counters in shaders)
- re-implementing unity-src-changes (very real downside to src-changes - always need to keep changes minimal, easily trackable and mergable)

Fix bugs in engine-code suddenly exercised (timing etc)
- new compilers, new data-formats, new features/improvements/optimisations...

Optimizing for specific platform
- different hardware restrictions - CPU/GPU, disk, memory tradeoffs
- new OS, tweak thread-priorities/scheduling
- new hardware platform layer (unity)

…very, very few script- / game-bugs

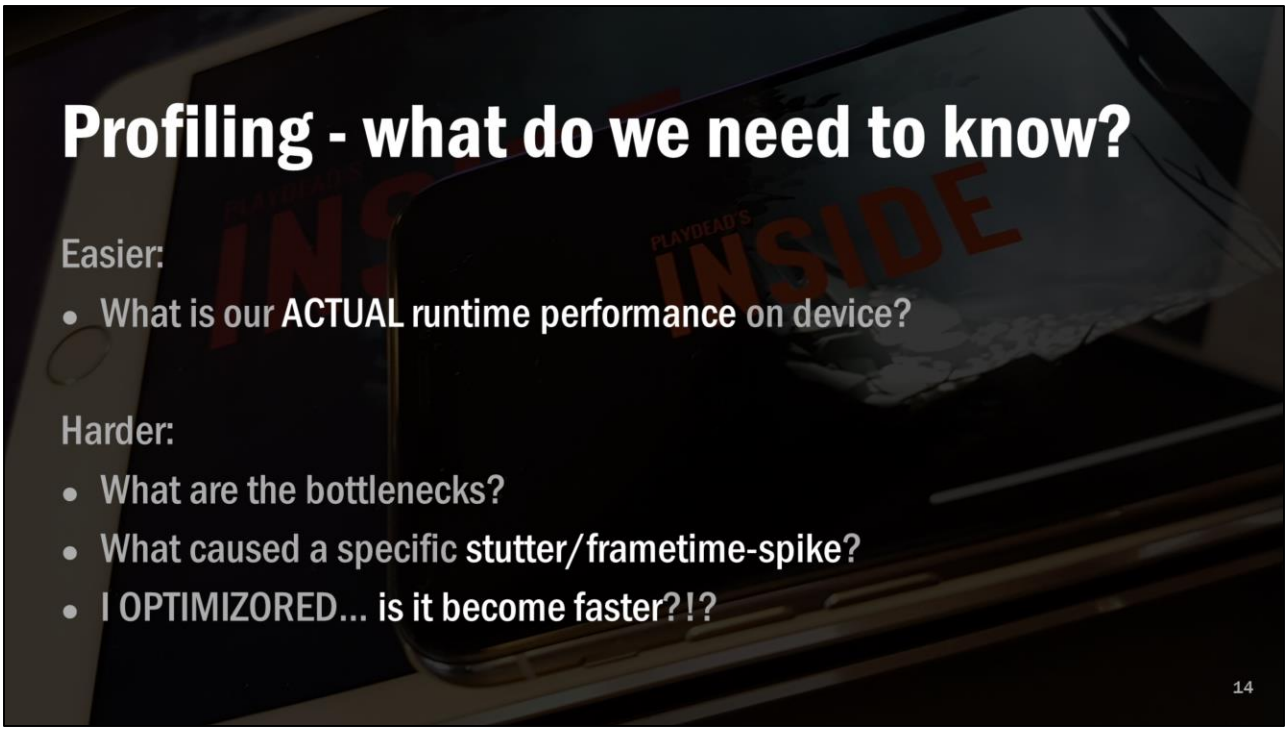


Updating game for platform
- Achievements, gamecenter-integration, store-integration, ui-adjustments
- tweaking for new input/gameplay for controls (specific analog sticks, touch-controls etc)
- tweaking for new output (smaller screensize, aspect ratios, brighter screens/brighter surroundings)

Playdead did a presentation at Unite on our profiling
- mentioning here again, in part because too few have seen that presentation
- but mostly because profiling-tools are **one of the most important parts of optimisation.**

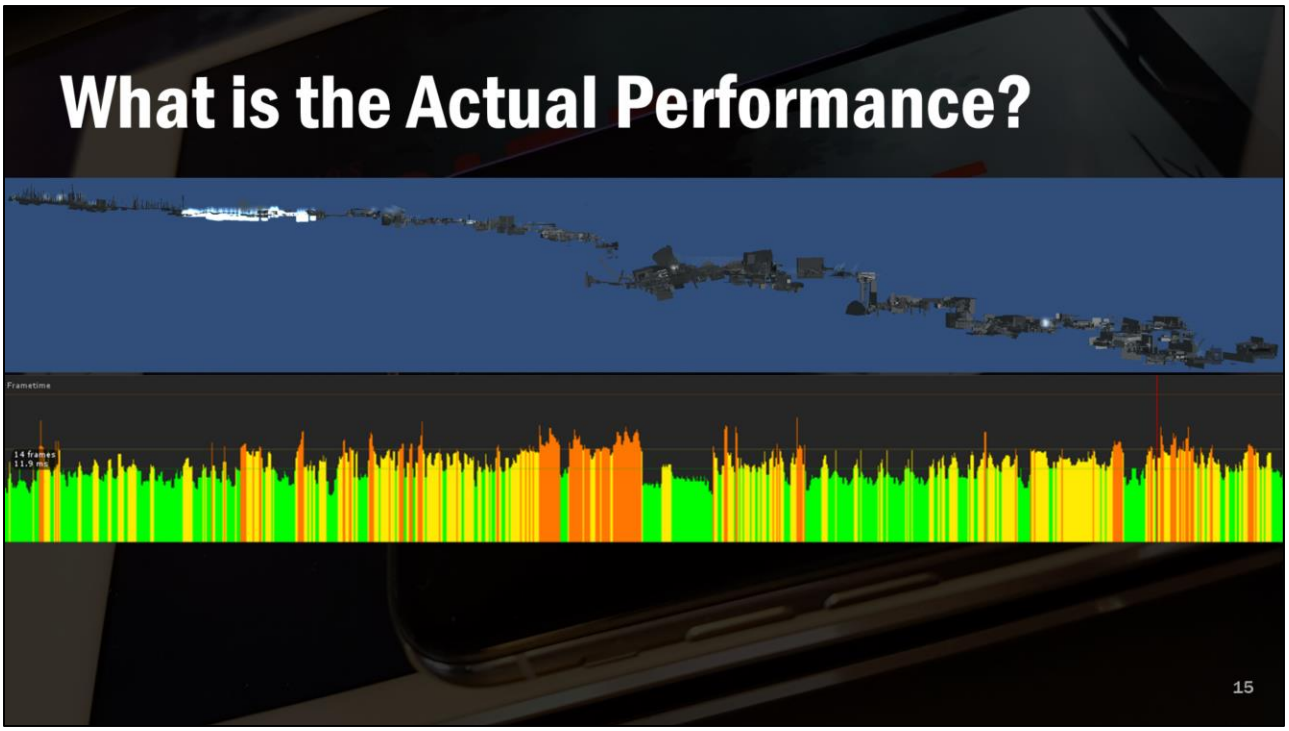
- No time to dive too much into hardware of platforms
- Often not enough information on hardware-details anyway to really make smart optimisations
    - ...most performance issues are not due to lack of smart optimisation but rather dumb solutions...

See this presentation for all details
https://docs.google.com/presentation/d/1dew0TynVmtQf8OMLEz_YtRxK32a_0SAZU9-vgyMRPlA/edit#slide=id.g18da8570e0_0_280
https://www.youtube.com/watch?v=mQ2KTRn4BMI



Important: This is what is shipping. “Just” profile, preferably using system-wide traces (Apples Instruments/metal-system trace is nice)

so when it turns out the ACTUAL performance is terrible…
We obviously do not need to optimize everything at this point - we are in shipping mode… “good enough”
=> which codepaths do we need to optimize?

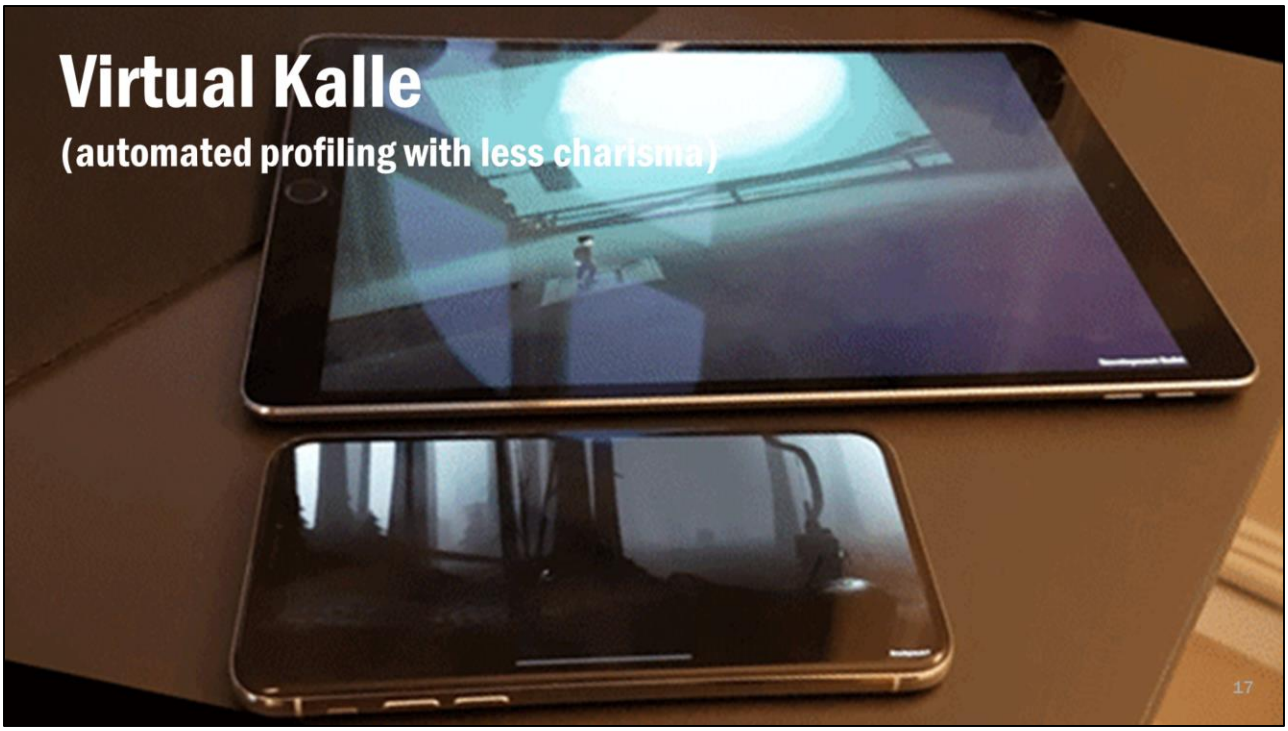


Ideally, for every position in the game, we need to know the performance
(we are lucky that INSIDE is essentially a linear game – but really we are just **tracking performance in a playthrough** – could apply to any game)
(you get it, you're smart people)

So we use Kalle

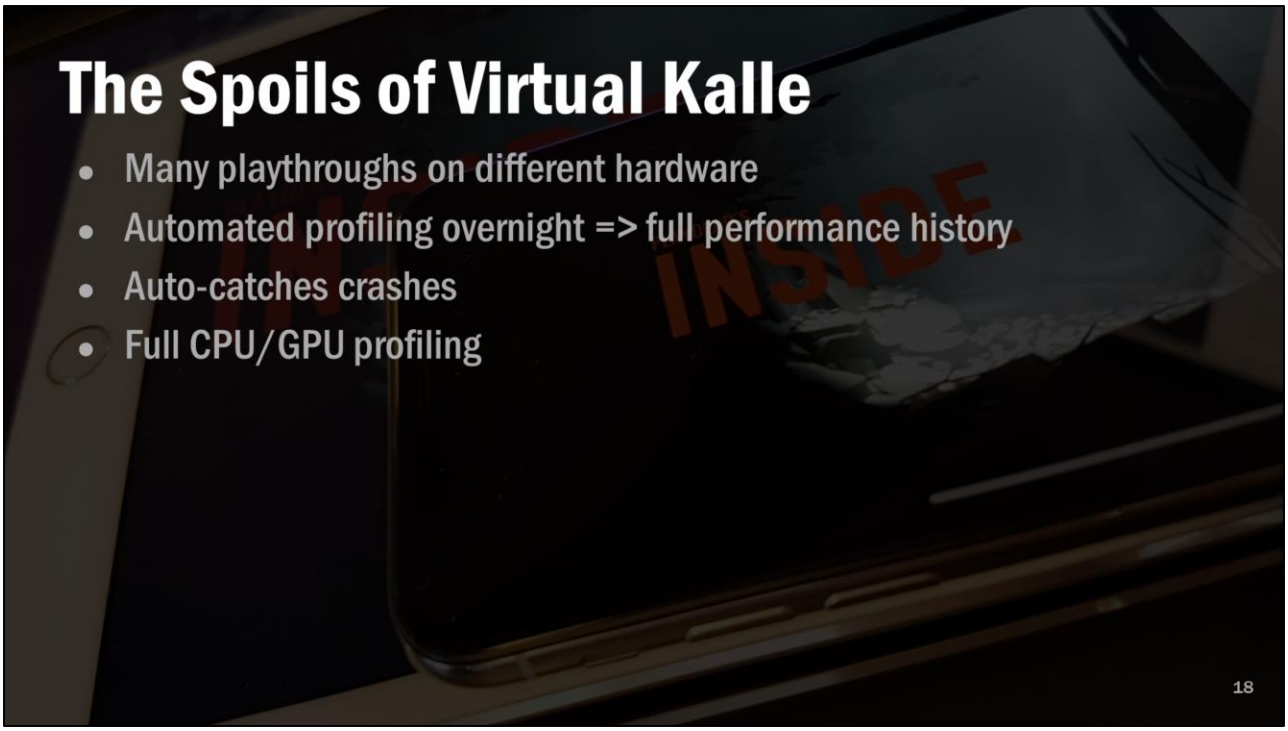
So this is Kalle

Ultimate surveillance: We record everything Kalle does (yes, Kalle, everything… no, don’t eat that, it’s bad for you)



Load a specific point of the Playthrough-recording, and profile the next few meters
- reloads entire scene every time => very close to deterministic between runs

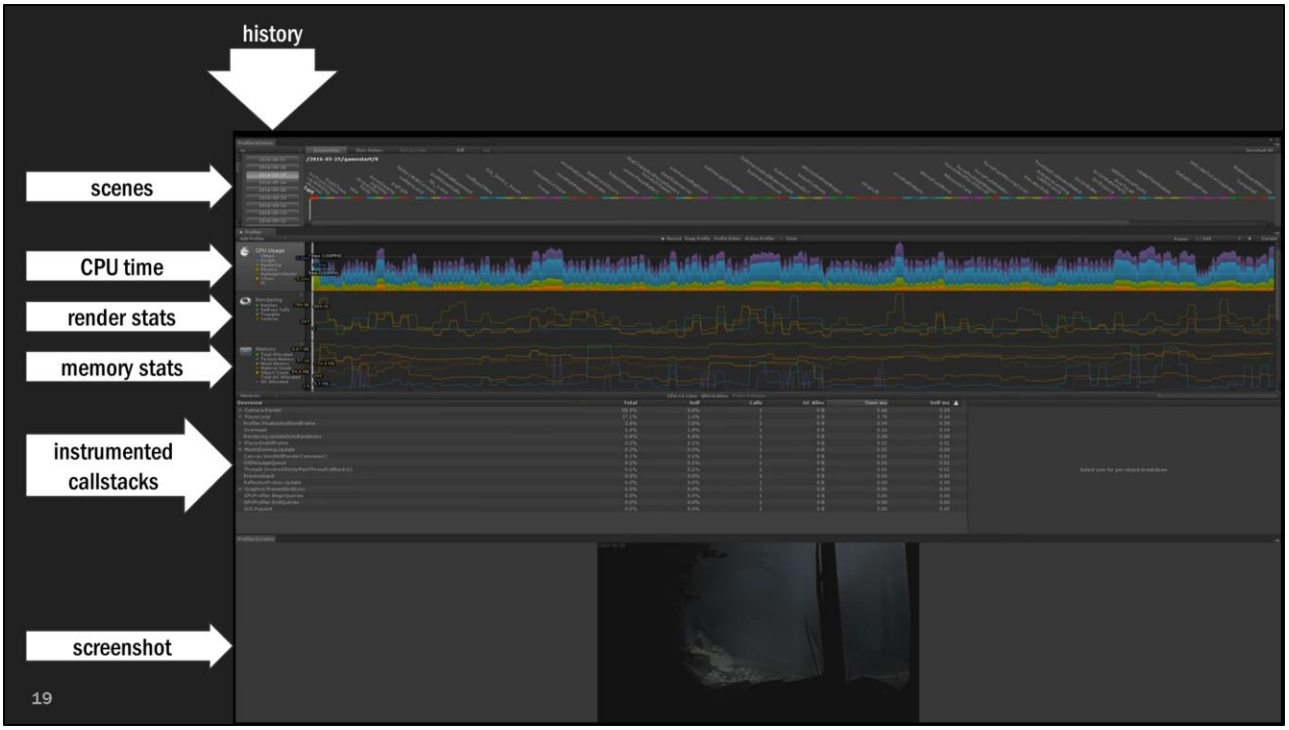
But can’t be used for mem-profiling- fortunately that is one of the things we recorded when Kalle played the first time... but means he has to replay if we optimise mem :(



Easy to run playthroughs on **lots of different hardware**
Can automatically **run it overnight** (we didn't set this up for ios though… only consoles)

While profiling:
- Low variance, reproducible performance – great while profiling
- Variance in the game can be measured by doing multiple runs
- Possible downside is too low variance. Exact same playthrough…

apart from Kalle - we also use an **external QA** company, VMC, to test all the weird parts...
(to test ordering a beer, ordering NaN beers, ordering -1 beer)

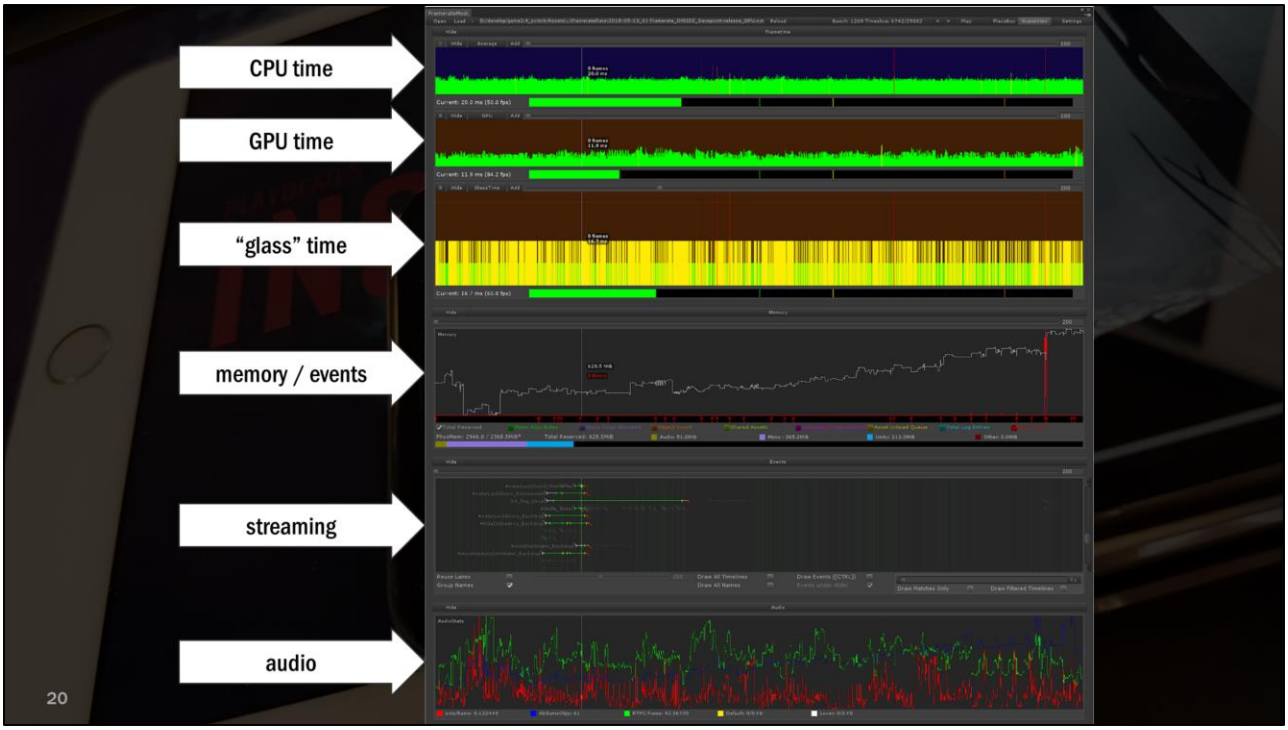


Performance: Full Instrumented Profiling (with callstacks)

- We can locate slow areas afterwards, and inspect every part of a playthrough afterwards

- everyday, can track regressions (even after the fact)

We integrated this with the Version-Control system, showing a performance-graph in-editor when selecting a scene to work on.



Looking at this, you can quickly get a **very good** idea where you should start looking - and a **pretty good** idea of what to look at

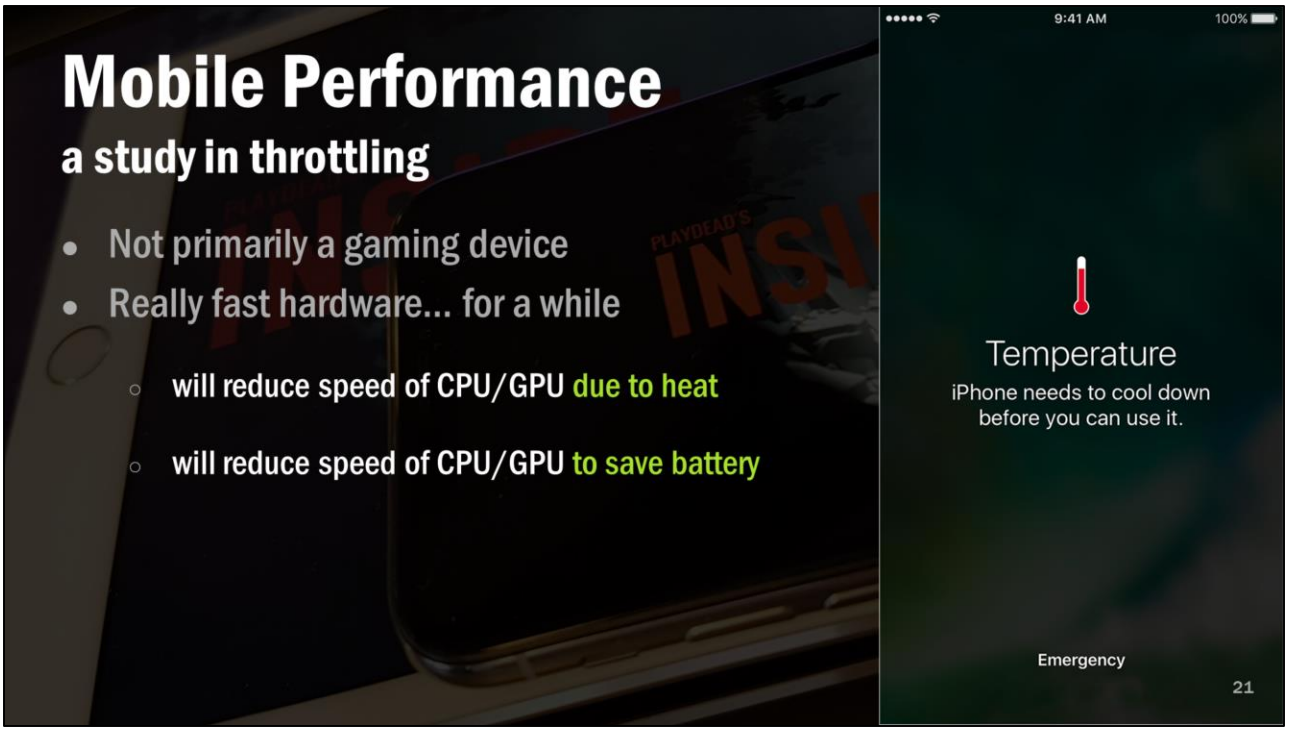
SOME insight as well: streaming, memory-details (GC)
- early screenshot, final is much less interesting: All green and no GC during playthrough \o/

Framerate, memory-use, streaming, audio-events

Arrows in memory-view are GC - purple graph

- Auto-catches memory-use (with mem-profile and history)

(again, see unite-presentation for all details)



Moving on to the "**harder**" part of profiling: Figuring out what part of our code is **actually slow**

Not a gaming device means:
- will interrupt process for a long time, 10s of ms, for more important background services (say, prefetching gifs of kittens)
  this ADDS NOISE to profiling-data, as we can't be sure if the cause of stutters is OUR code.
- Not much we can do there, **WHILE PROFILING reduce noise** DISABLING ALL DATA (wifi/mobile-data), removing all irrelevant apps

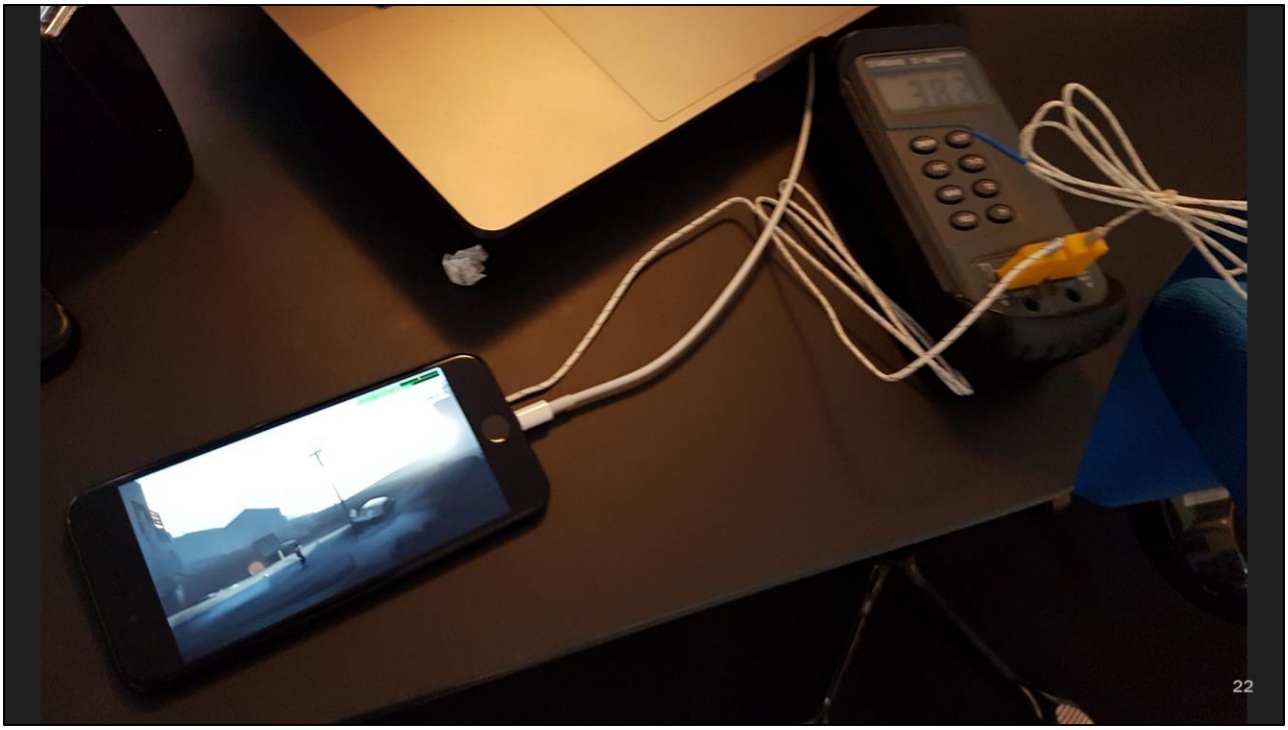
While fast, it is still a mobile device - ipad pro2 10" comparable to desktops at peak speeds

Second
- Thermal throttling - CPU/GPU due to heat
- Frequency scaling - CPU/GPU to save battery
- CPU<=>GPU power - CPU and GPU share power/thermals, workloads influence each other
- **screen** too! Affects thermal budget and reduces brightness if device overheats

Eventually will temporarily halt operation

https://support.apple.com/en-us/HT201678

All of this makes sense for a mobile device, and for most apps (scrolling a page, then reading, requires bursts of performance)
- but it does complicate profiling

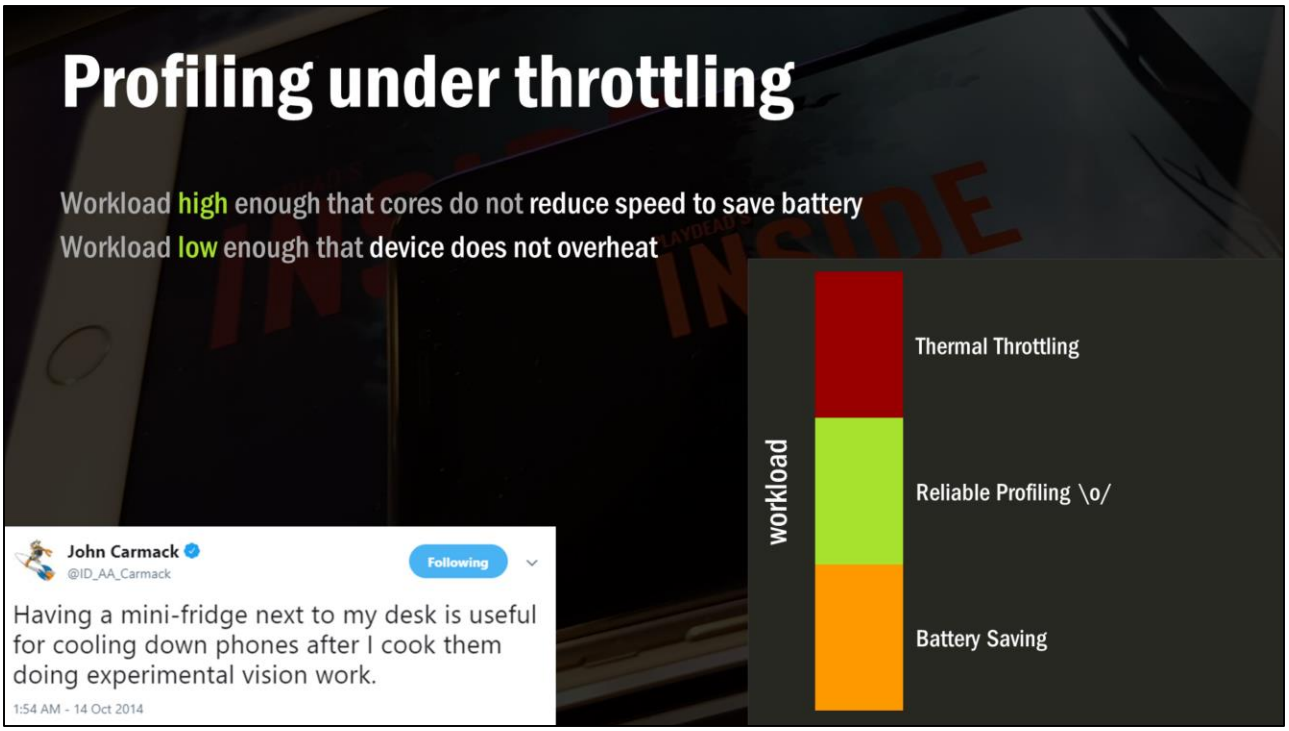


Our external QA, **VMC, had come back** complaining about **uncomfortably** high device temperature…

Like all good engineers, we went out and purchased an **industrial-spec thermometer**... to figure out what “uncomfortable” meant in numbers.

Turns out you have to target below device-spec... at **human spec** (feels unconfortably warm at around 40c)

As an example, iPhone X, can run at 60fps with no framedrops
but gets too hot for puny human comfort.
(also, if also running OLED at full brightness, at some point it turn down brightness to reduce temperature)

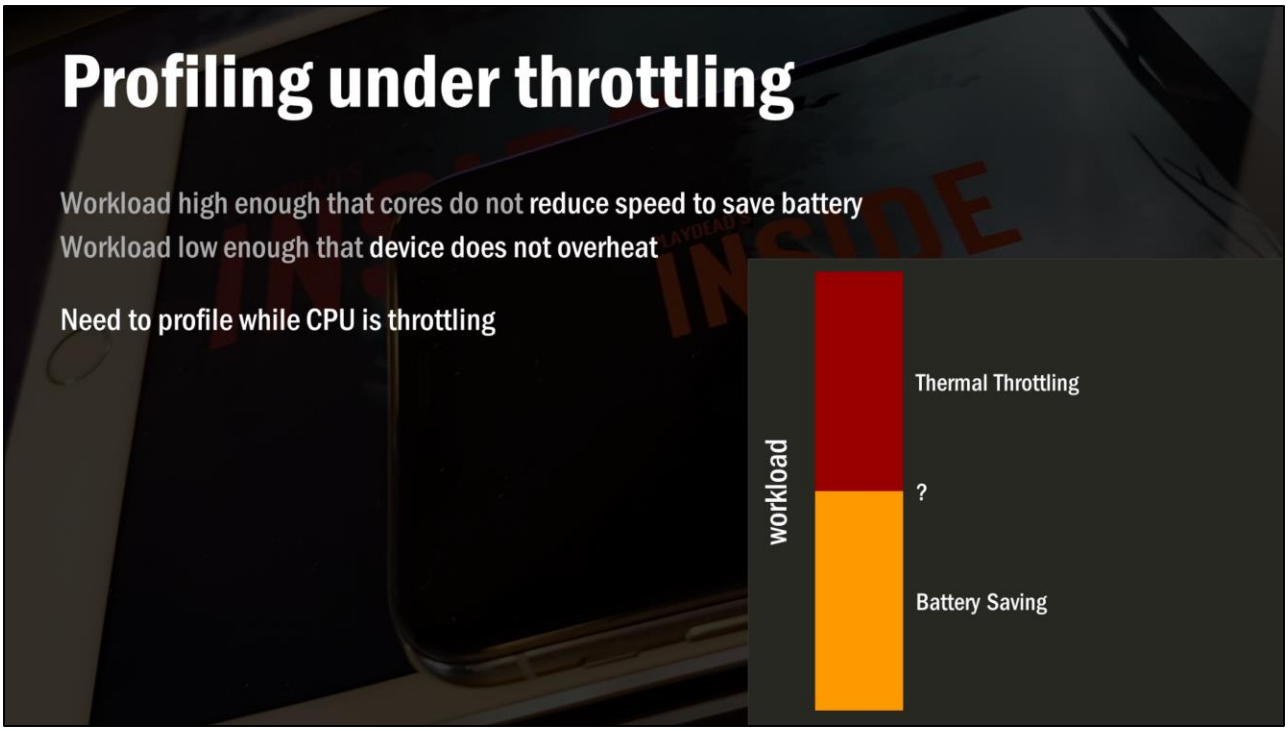


Different approaches to solving this

- PC: Nvidia has a way to disable the GPU throttling / boost
- Carmack: Small fridge with Phone
- Apple-engineer on-site at playdead: shove device in a bowl of water - the hw can is certified for it ;D (we didn't, also don't recommend that to others who pay for their own hw…)

https://twitter.com/id_aa_carmack/status/521826400257728512?lang=en

(a way to get something like this, is to start with **a cool device**
boost the performance by running some intense workload - then profile before frq reduces again)

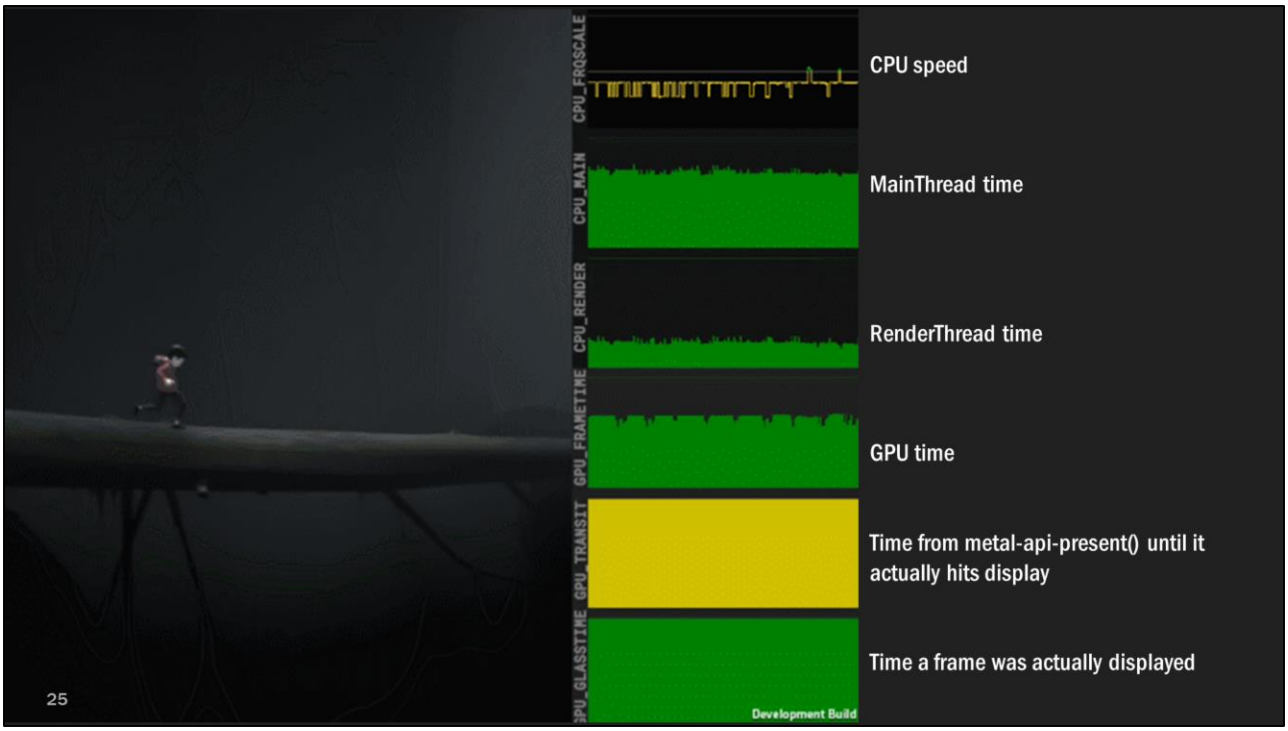


No guarantee that it ever runs at full speed, or doesn't change speed – makes sense to ALWAYS save as much battery as possible
- again, not primarily a gaming device

Implications is, that you may optimize something, and, to save battery, it will adjust cpu-speed – **resulting in the same profiling-timings**.
(intel has tool to see power-use, which again would be handy)

Don't necessarily need to know WHY it is throttling, just that it is, and how much

It IS possible to know how you are doing thermally - Apple has an API to tell you the current **thermal bracket**
https://developer.apple.com/library/content/documentation/Performance/Conceptual/power_efficiency_guidelines_osx/RespondToThermalStateChanges.html



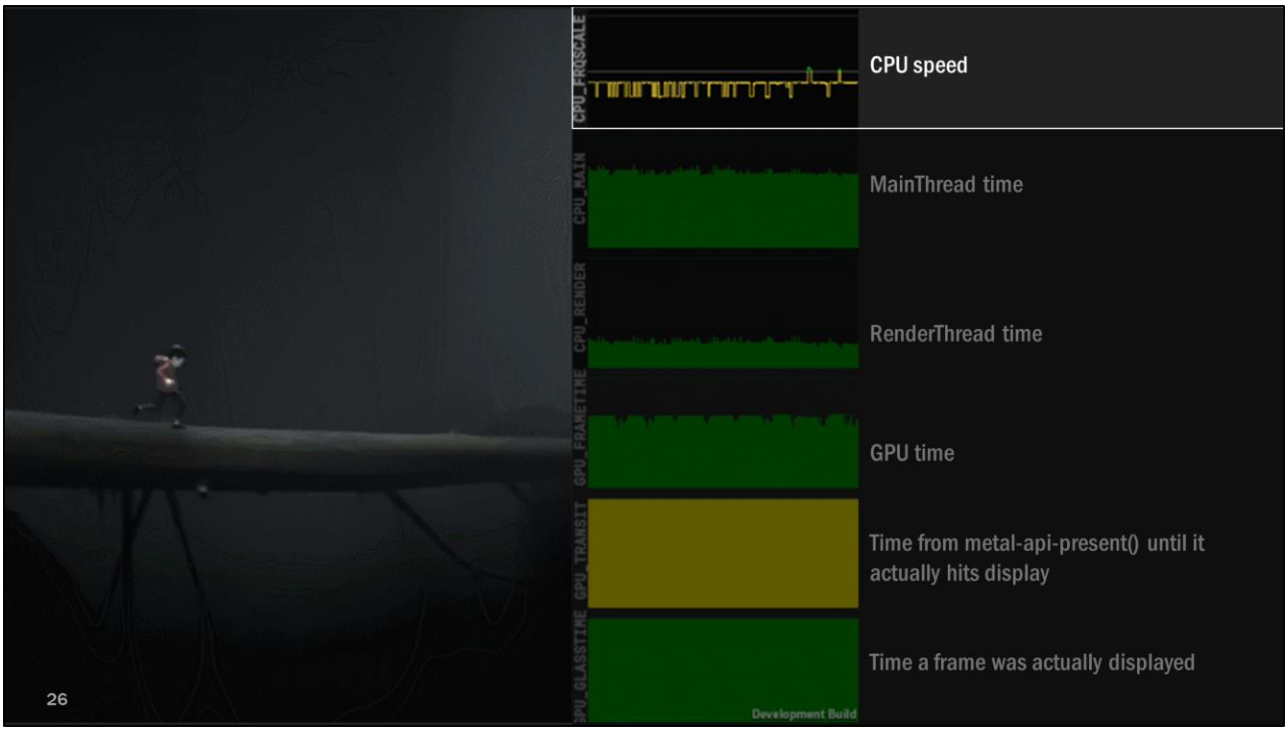
We build a runtime performance-overlay

…see spike and adjustment in frequency to compensate

Regular timers for CPU
CommandEncoder start- and end-times for GPU-profiling

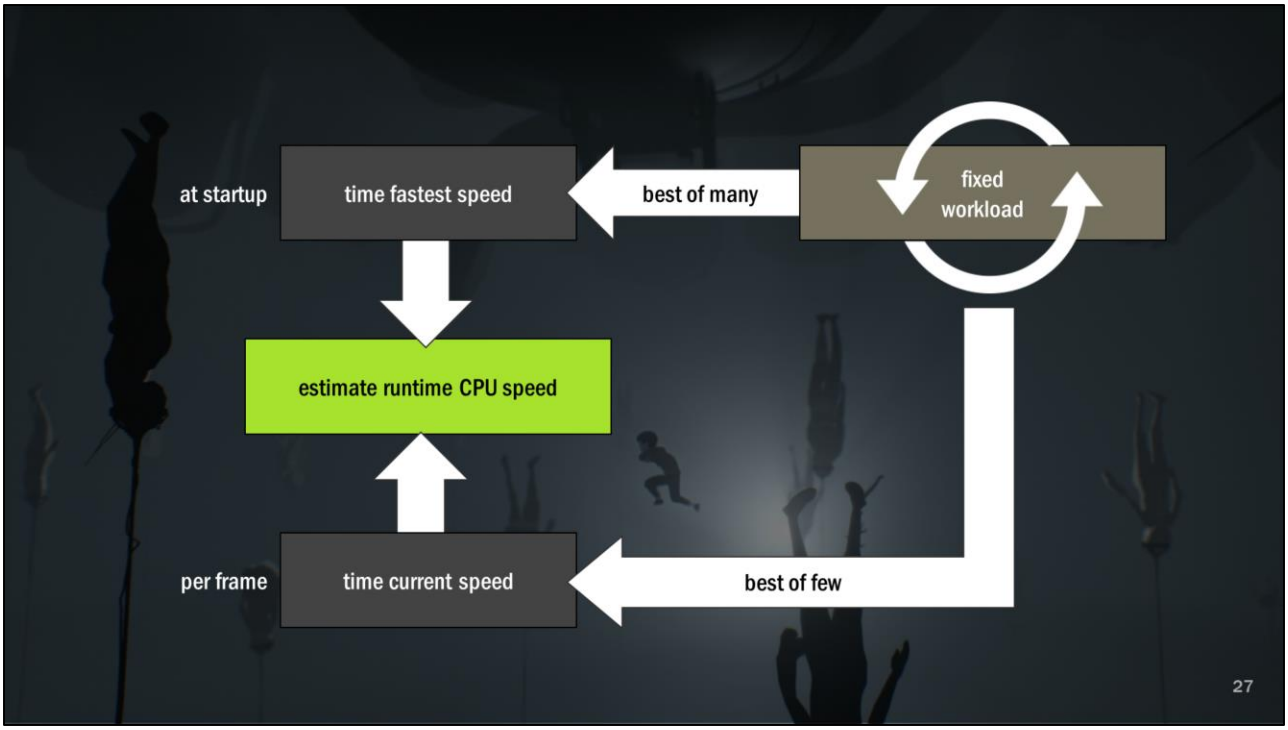
CPU_FRQSCALE – current_speed / peak_speed
CPU_MAIN (thread) – timing for main-thread
CPU_RENDER (thread) – timing for render-thread
GPU_FRAMETIME - timestamps on command-encoders (no finer granularity available)
GPU_TRANSIT - time from api-present-submit until image actually presented (shows latency / depth of swap-chain / queued up frames)
GPU_GLASSTIME (how long was an image actually displayed... actual framedrops)

https://developer.apple.com/documentation/metal/mtlcommandbuffer/1639924-gpustarttime
https://developer.apple.com/documentation/metal/mtlcommandbuffer/1639926-gpuendtime



...could potentially divide profiled times with CPU-speed to get objective performance (we haven't though, so can't speak to how useful that would be...)

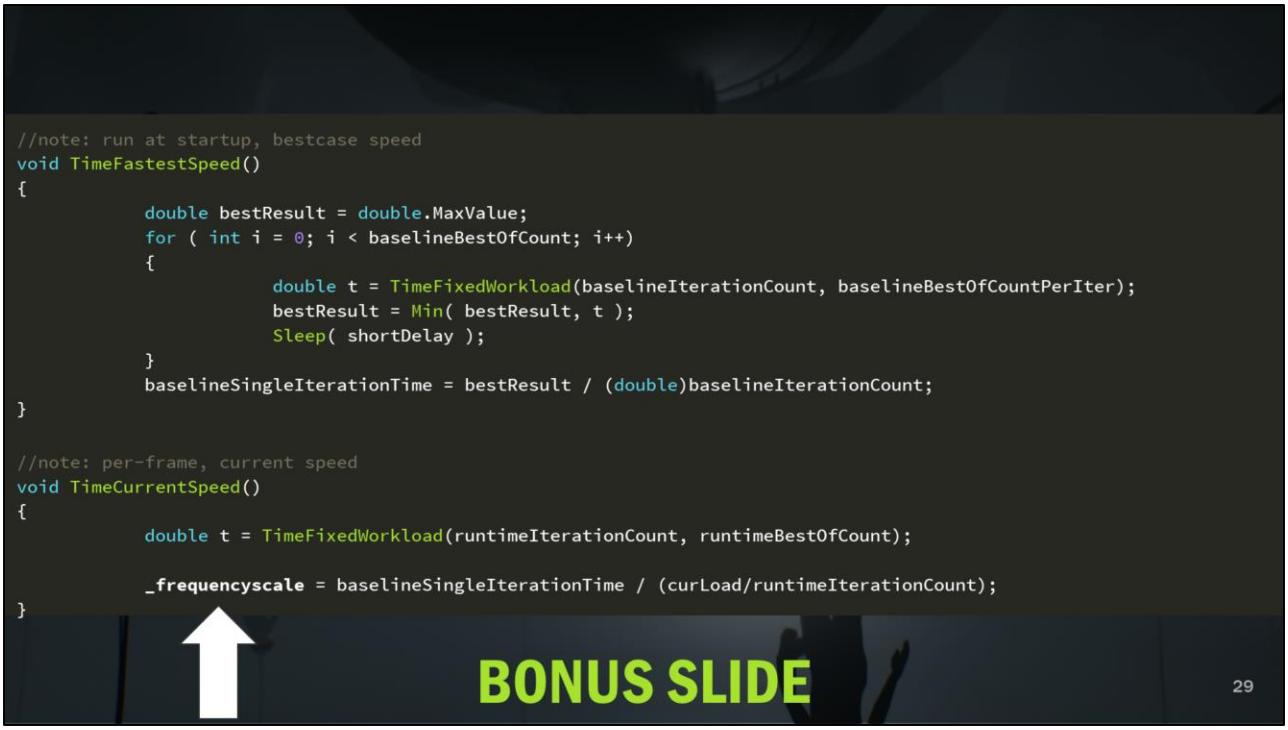
Mostly good enough to know that it happened (along with delta-time)



To get peak-performance:
Important to run over a **considerable period of time**

- frequency **adjustment not instant** (this is the entire problem)
- The device might be doing something else, pre-empting our game... (again, not dedicated gaming device)
- ...all in all, just trying to hit the **peak-performance** the device can possibly run at!

To compensate for instability, do multiple runs, take the best one

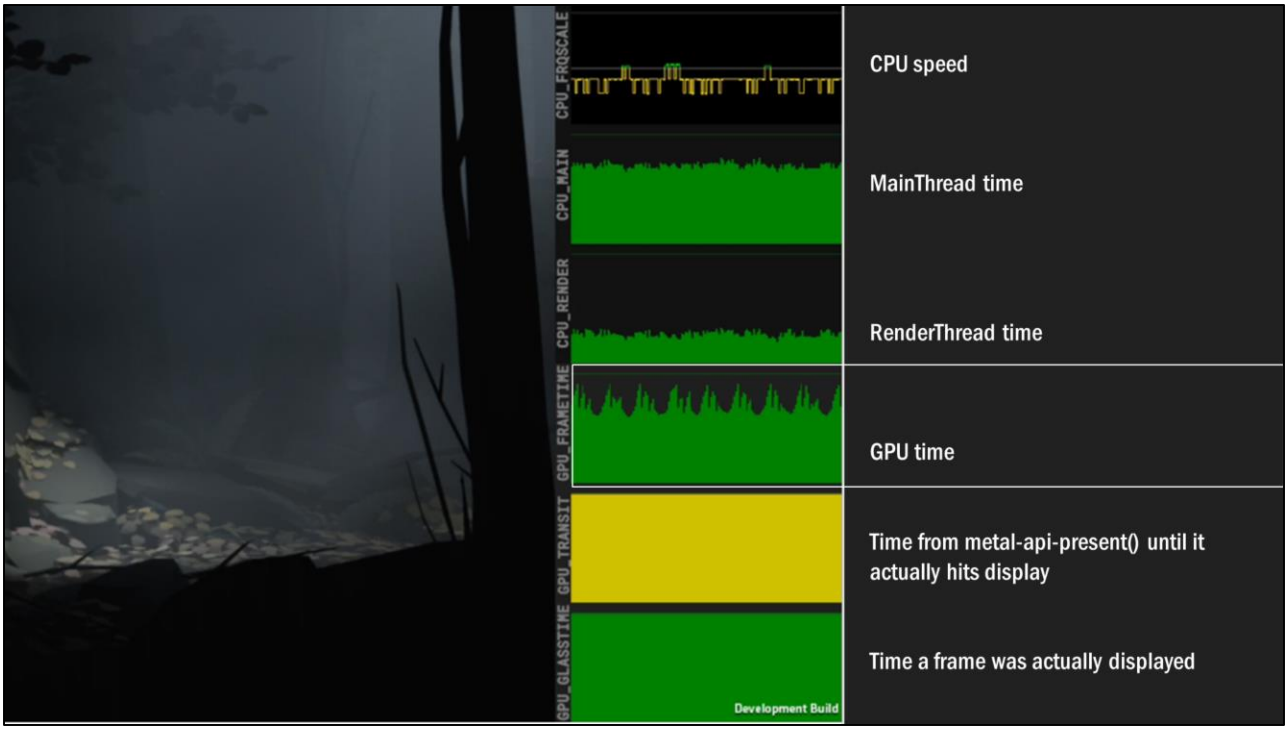


At startup, measure the baseline speed

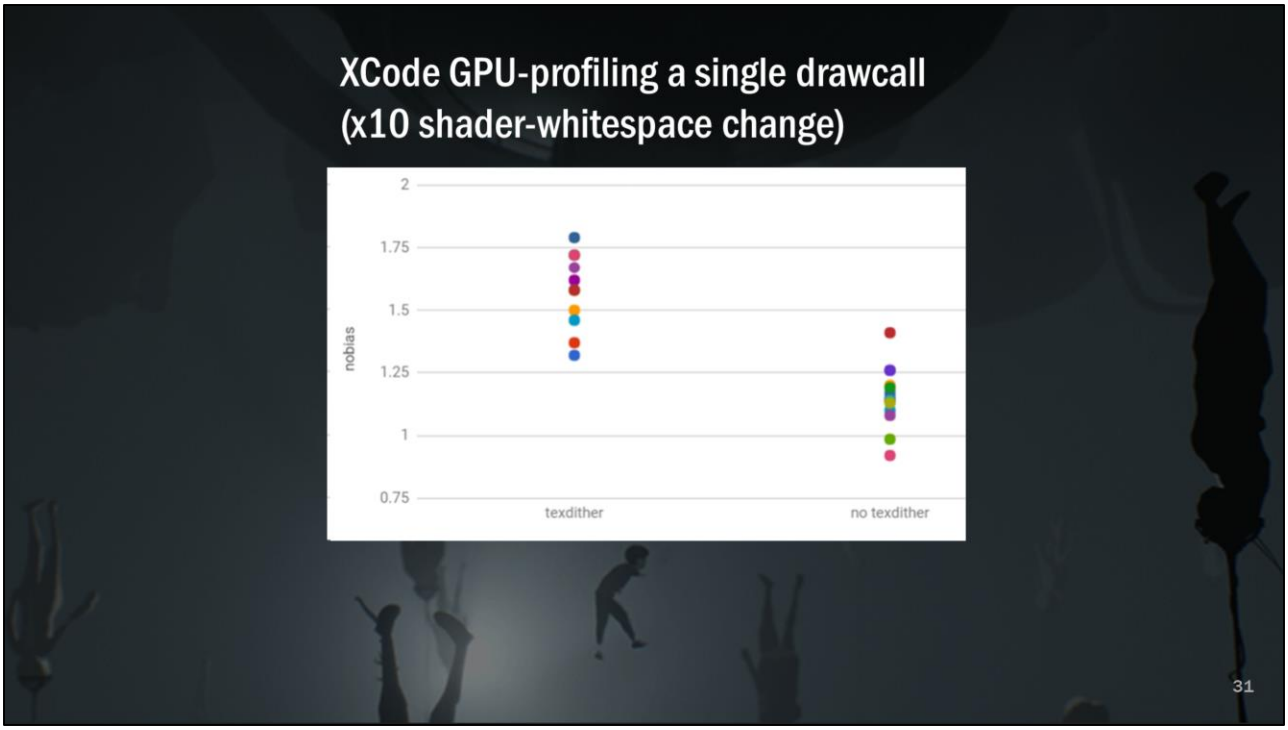
- run workload **multiple times** to make sure CPU is at max frequency
- pick best-case speed (ie fastest/lowest time)

At runtime: Again, run multiple times, take best of runs

Only single core, non-mainthread work does not contribute to frequency-scheduler...



…do it for GPU too?
(looking at 30Hz atv4k)

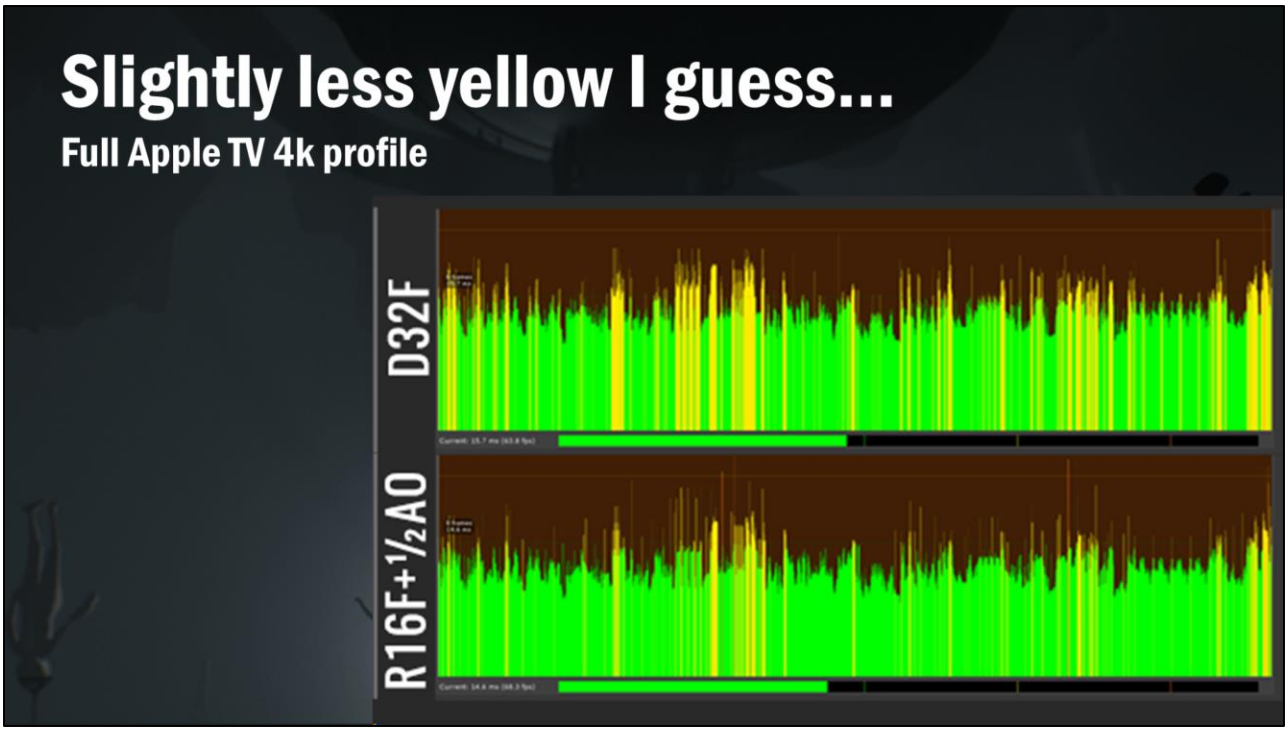


Presents an interesting issue when profiling in xcode

We initially thought it was just all over the place – then started looking at stats for profiling-results

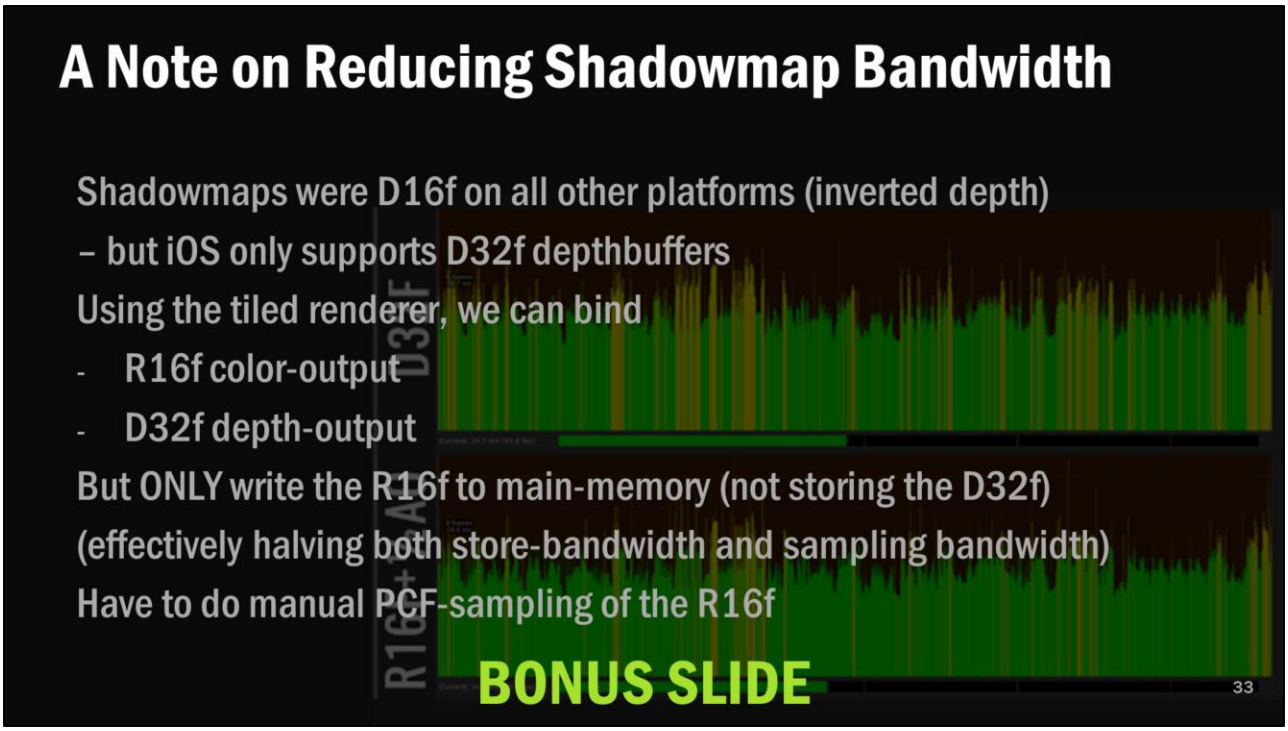
On the left: timing for a single drawcall, in a ~20ms frame. Takes around 1.6ms on average, profile has variance of around 0.5ms.
On the right: same drawcall with all dithering removed (around half the ALU in the shader)

Looking at it statistically the change is reasonably clear – looking at a single profile from each, it is impossible to say anything for certain about the optimisation.
Doing this check takes around 10 minutes.

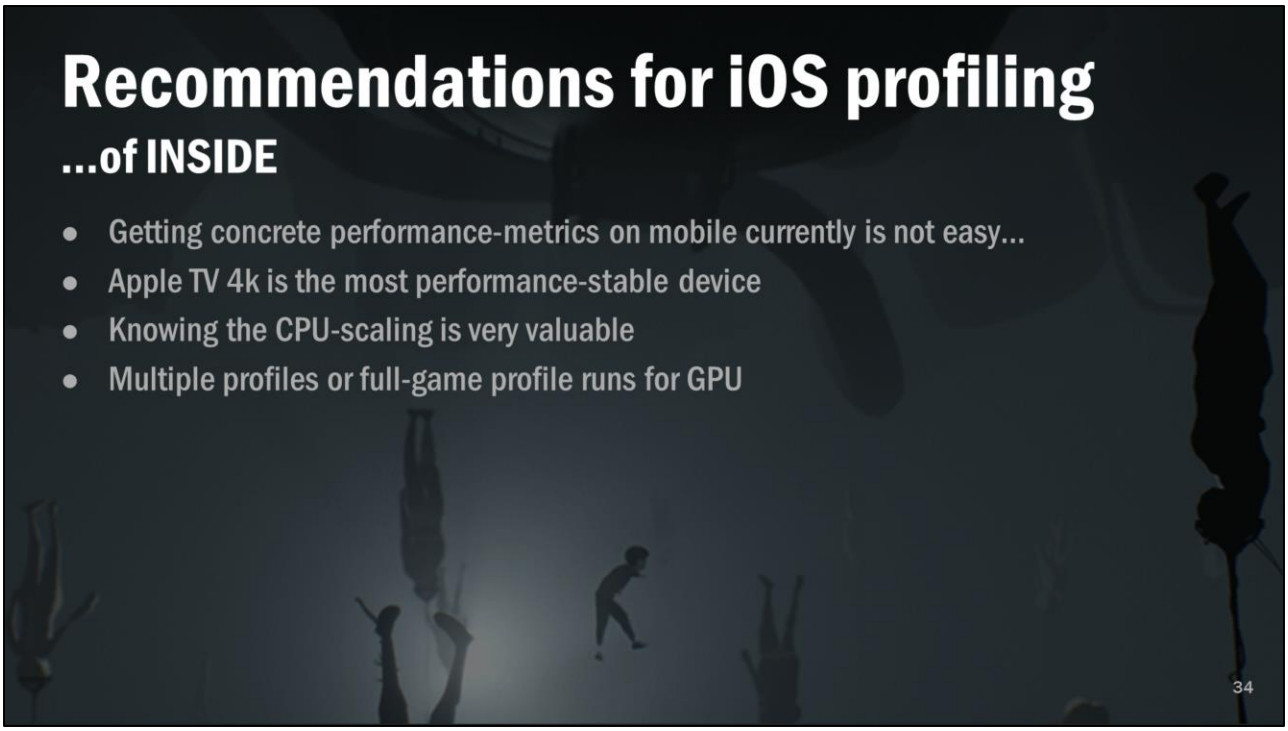


What we **ended up** looking at full captures
Example of optimisation – test we did on using float16 shadowmaps with manual comparison

Very hard to do micro-optimisations
Not very good – best we could do :(

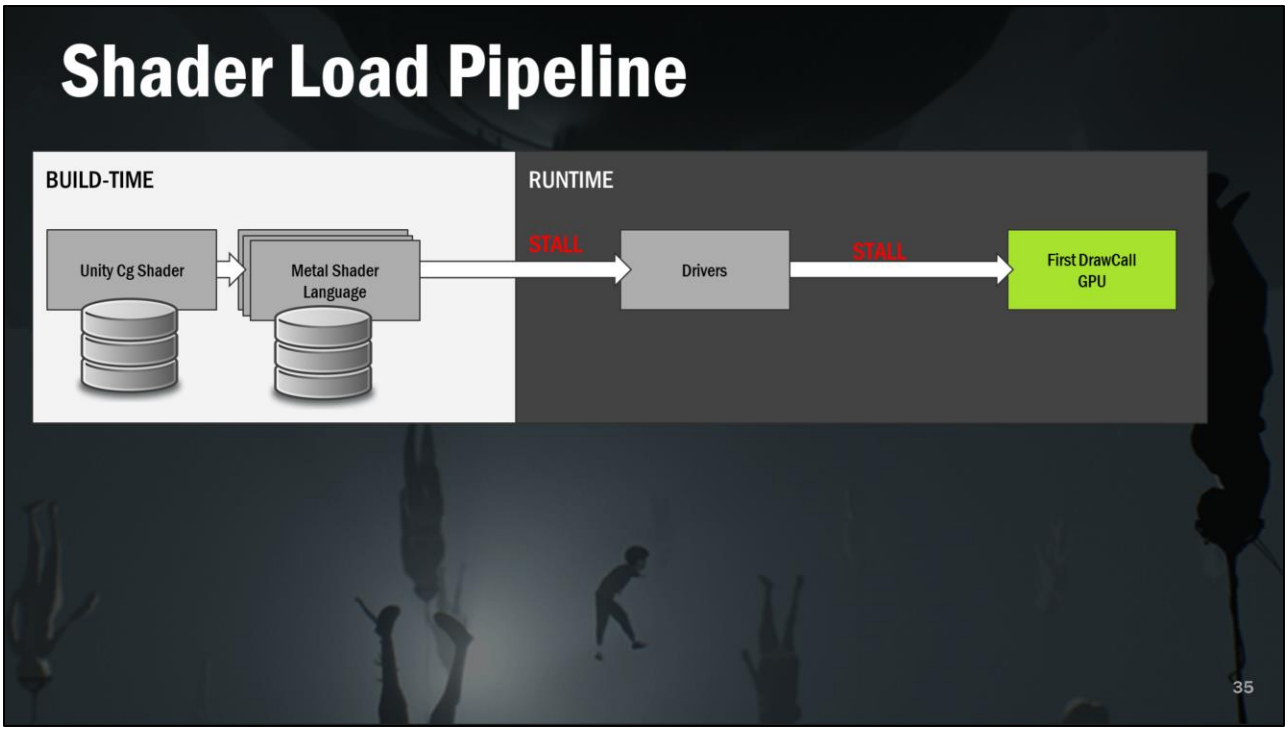

A Note on Reducing Shadowmap Bandwidth
Shadowmaps were D16f on all other platforms (inverted depth)
– but iOS only supports D32f depthbuffers
Using the tiled renderer, we can bind
- R16f color-output
- D32f depth-output
But ONLY write the R16f to main-memory (not storing the D32f)
(effectively halving both store-bandwidth and sampling bandwidth)
Have to do manual PCF-sampling of the R16f
BONUS SLIDE
33

TODO: stats



ATV4k: active fan => thermal throttles less

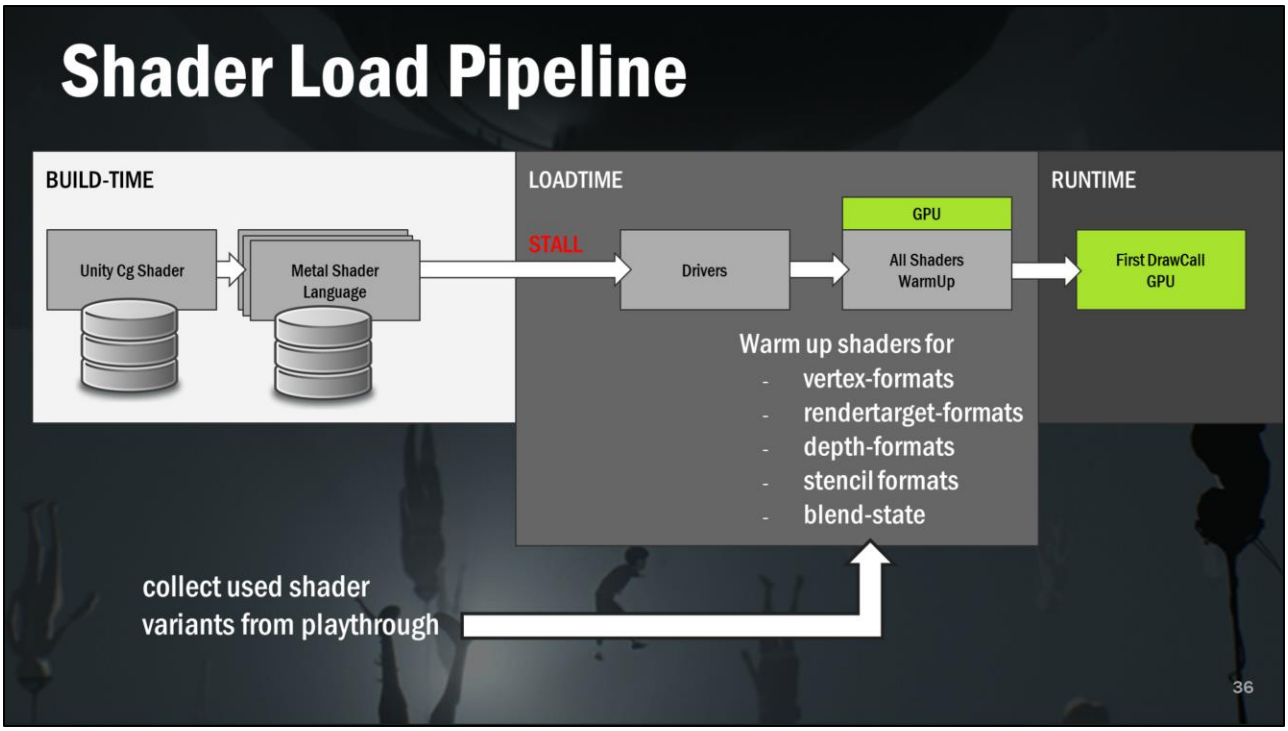
…also Metal-traces in Instruments are pretty useful for actual system-wide performance



First thing actually shows in Instruments Metal-traces: CPU stall in driver-thread (shown as “shader compile”)

On **FIRST LOAD**, driver recompiles to **Shader IR,** cached for the future…

On first Drawcall: **Driver recompiles shader IR to match state**

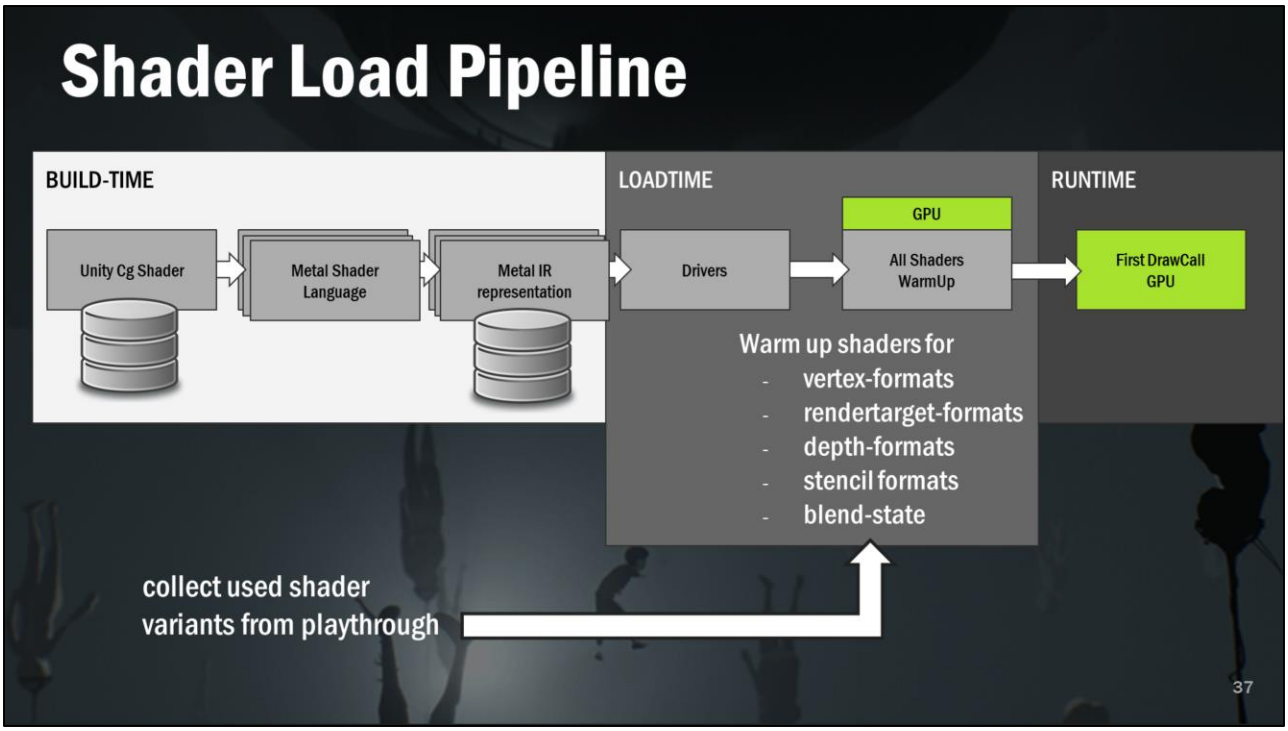


Before build:
Collect all shader-variants used in the game (the ones actually used, takes too long to warm-up ALL possible shaders)
- We did many, many playthroughs… advantage: Game was final! Otherwise should be in build-pipeline… somehow… (probably only something you want to do while optimising, so on a reasonably final game)

On Game-Load:

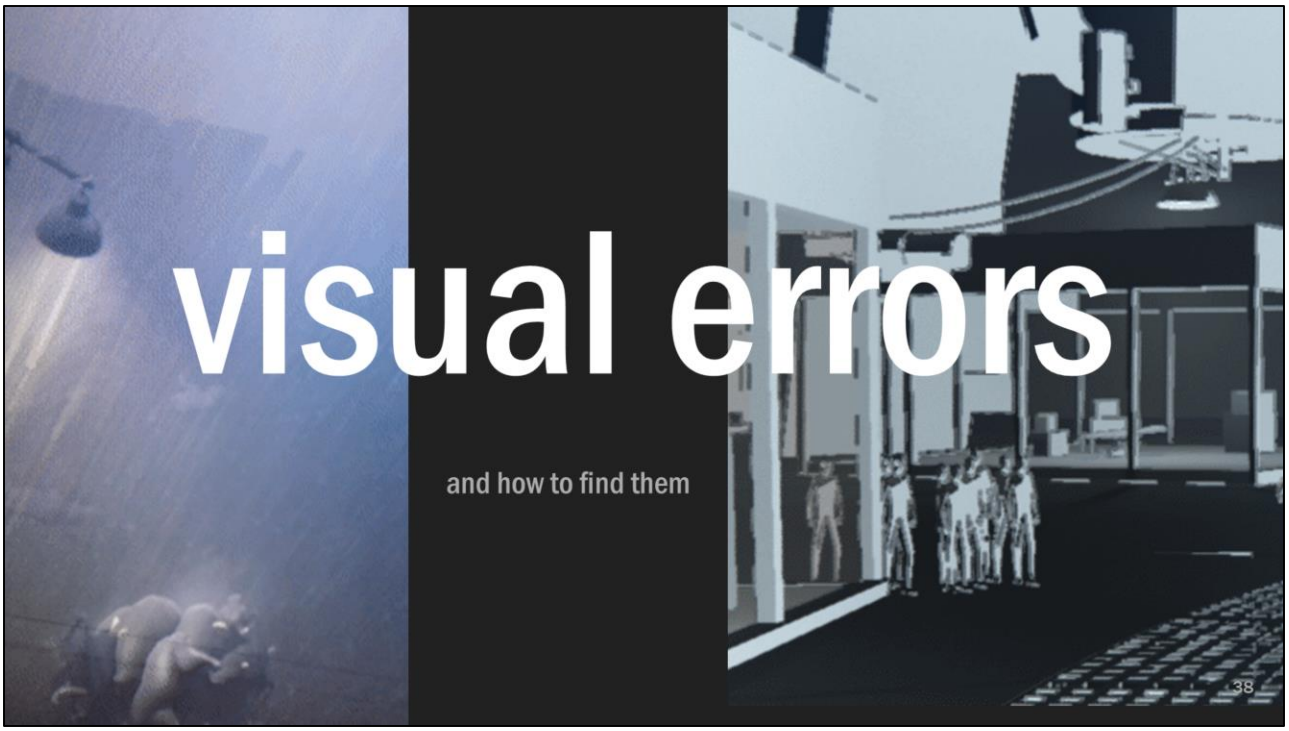
1. Pre-load all shaders
   (unity shader-variant cache, list built from manual playthrough)
2. Pre-warm all shaders (with vertex-format, rendertarget-format) - also collected from playthroughs. “warm up” means to render a triangle with the given shader-variant to an offscreen rendertarget, exposing the gpu-driver to the shader for the first time (forcing it to recompile the IR and cache it…)

=> No jit driver-patchups \o/



Saving Metal IR to disk just to speed up **FIRST load, subsequently from internal ios shader-cache (driver?)**
…became important as we are loading ALL SHADERS, warming them up too

Separate IR for iOS / tvOS



A lot of the time spent porting starts here: Most bugs makes something look different in the game...
(…or something running slow…)

Automated Screenshots Tool

INSIDE we (Erik) built this awesome tool that would load every 5 meters from a playthrough, and take a screenshot. VERY useful for finding bugs between platforms

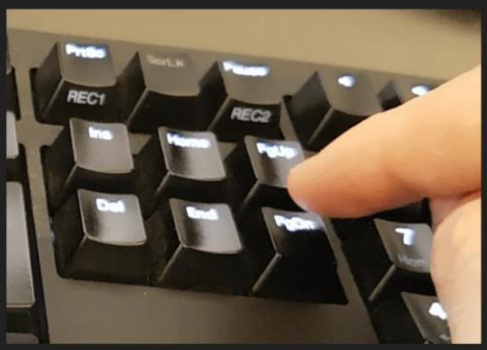
we used around 1300 locations for full game

Not SO easy to compare them...


A)
41

...a lot of flipping back and forth between images.

1300 images, 2-3 hours of...



Ideally needs to be done for every major change

Does works though, and it is what we did for consoles to ensure visual parity between all platforms on release

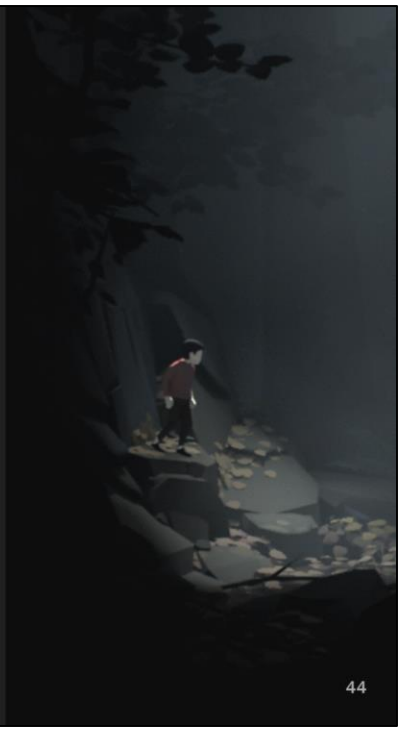
Automatic comparison tool

Single set of **test**-images

Multiple sets of **reference**-images…

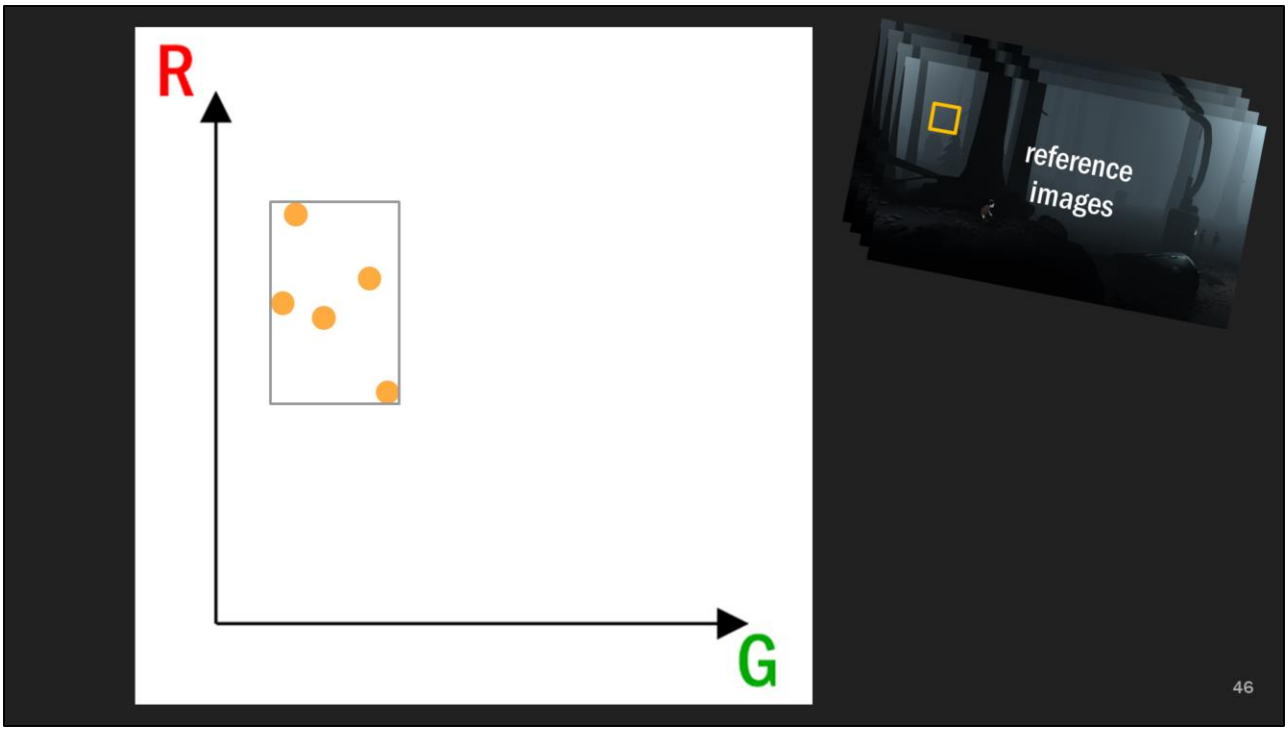
game is almost deterministic

- except particle-system start-times…
- and time-based procedural animations (e.g. rain)…
- and per-run settings (e.g. albino-type)…



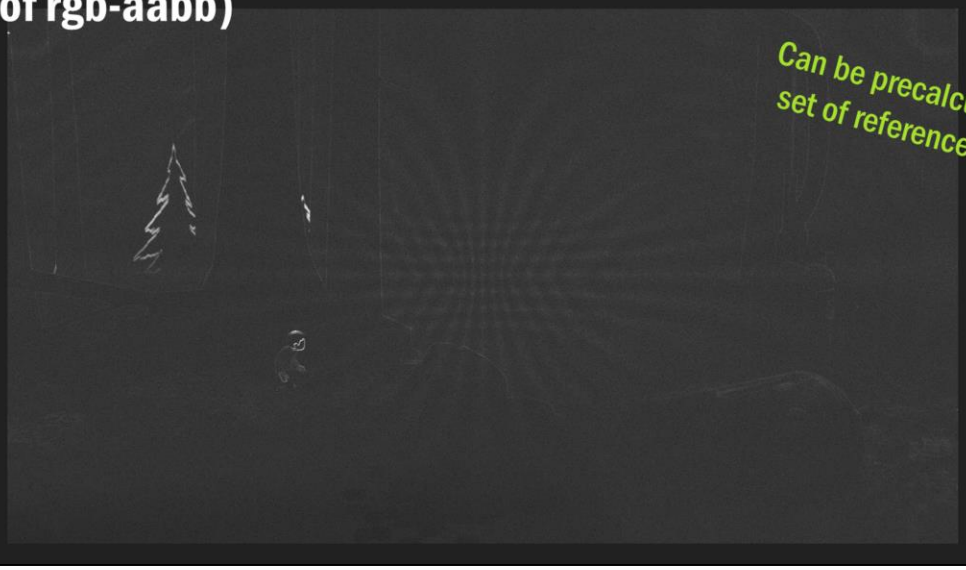
“How different is this image from the reference images?”

- for every pixel, build RGB-aabb of that-pixel in reference-images
- distance from test-image pixel-RGB, to ref-aabb
- calculate a metric from image pixel-distances

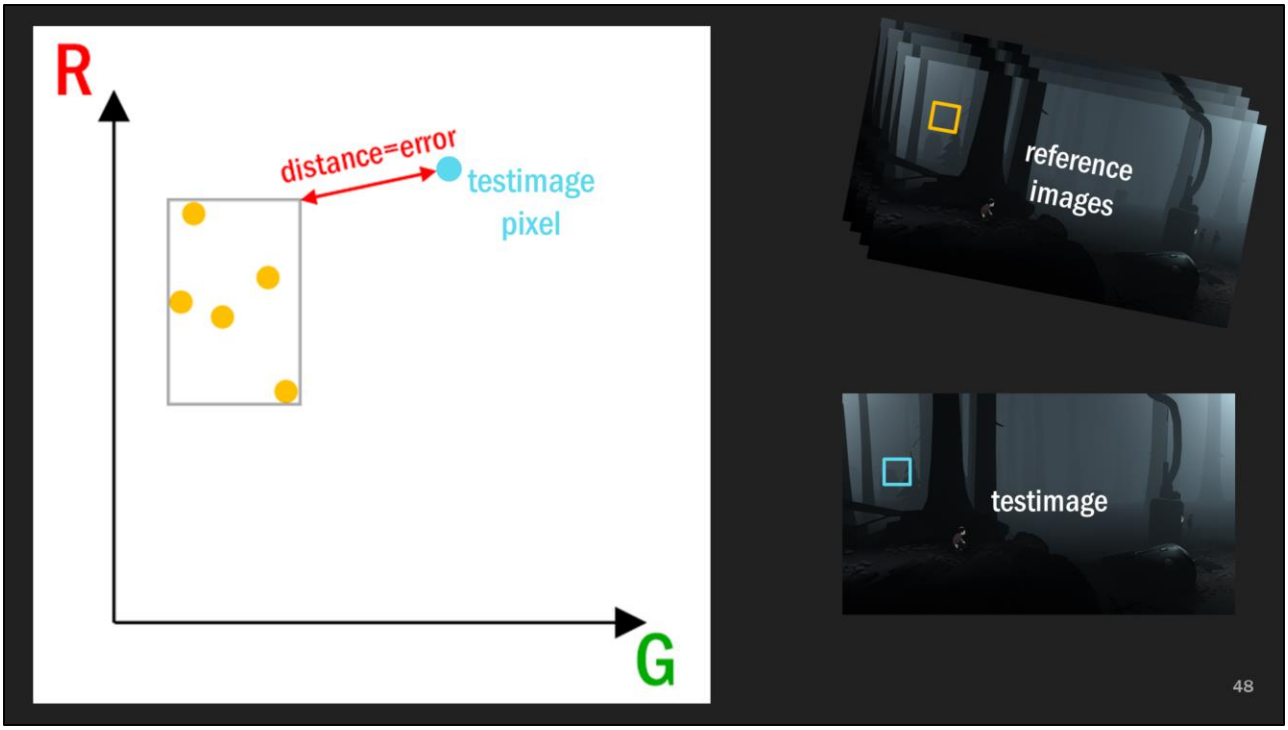

R
G
reference
images

Reference Image Confidence Map

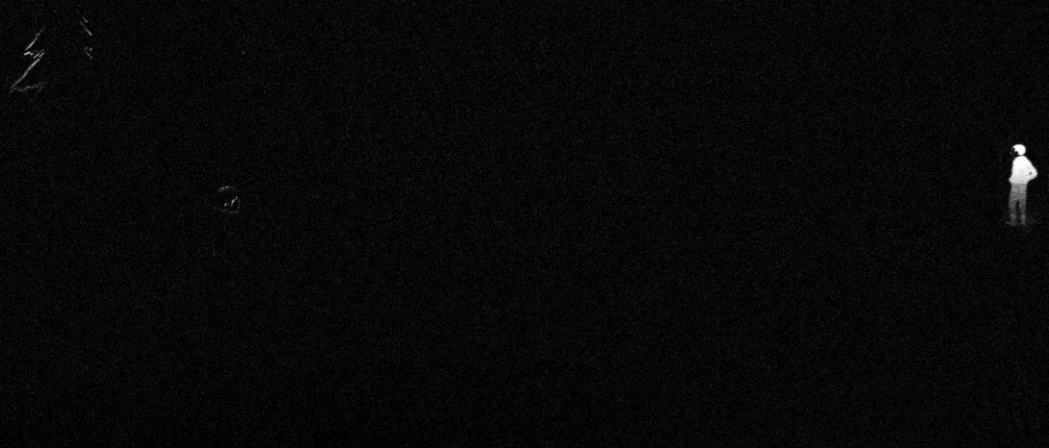
(size of rgb-aabb)

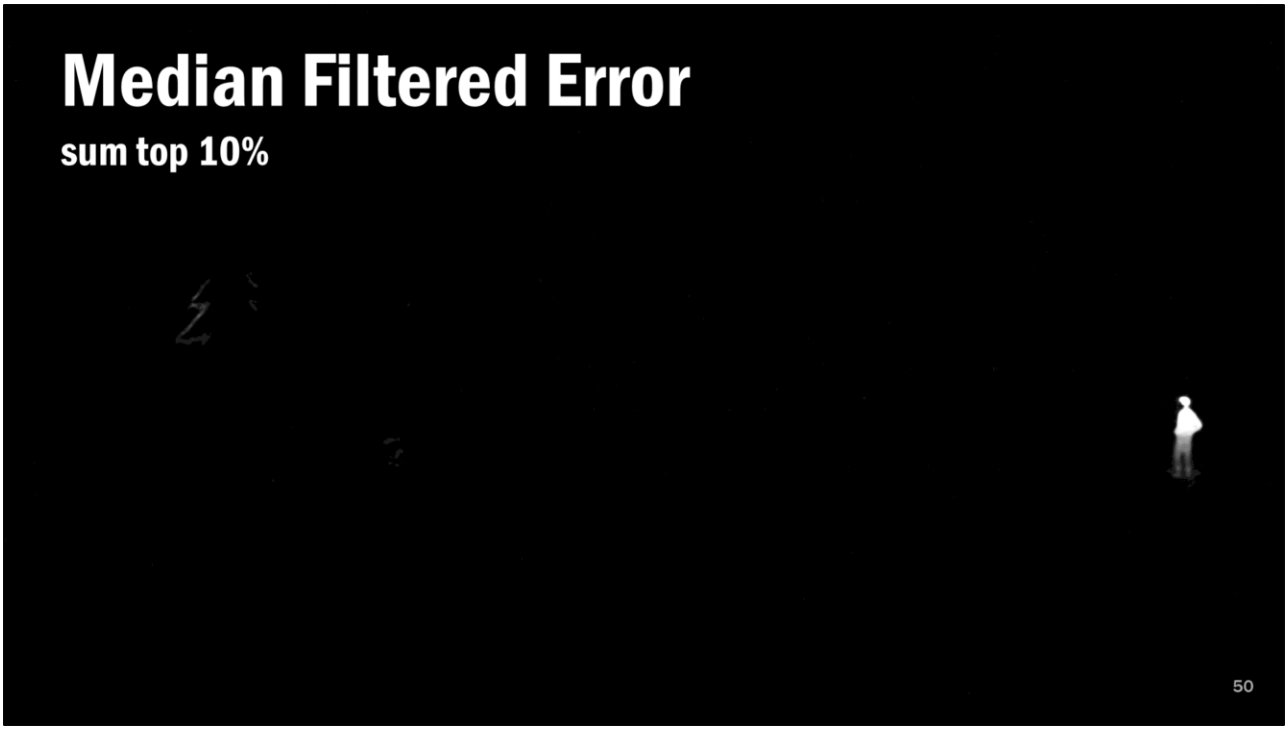


Can be precalculated per set of reference images


R
distance=error
testimage
pixel
G
reference
images
testimage

Error:
Testimage pixel-distance to reference-AABB





…then sum top 10% pixels as the IMAGE-error

Calculating an error-metric

We have a measure for the usual **variance per pixel**

Need to know the **variance of an entire image** too

- test a single reference image against the calculated reference-image AABB

Calc **error-metric** by **comparing testimage-score to reference-score**

Confidence image does not work for large moving things

EXAMPLE: TreeLeaves are vertex-animated, particles - both are non-deterministic. Need to know the “usual” error in an image, to know if a testimage is outside the normal range.

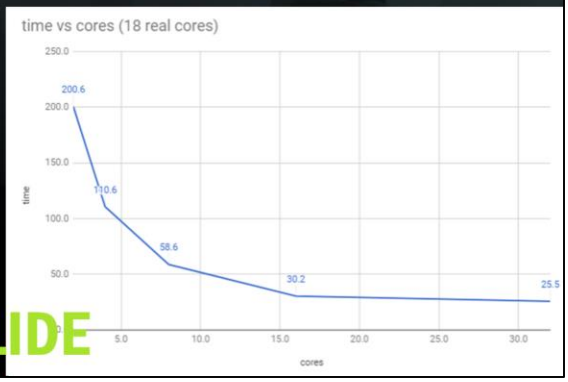
Optimise It

just a little

- Trivially **parallelizable**: batch into num_test_images/**num_cores** per thread
- **Precalc** reference-image aabb (save-to-disk compressed, e.g. zlib): **x4**
- Faster median-filter by bucketing: **x6**

~1hour => ~30seconds (on 16 cores)

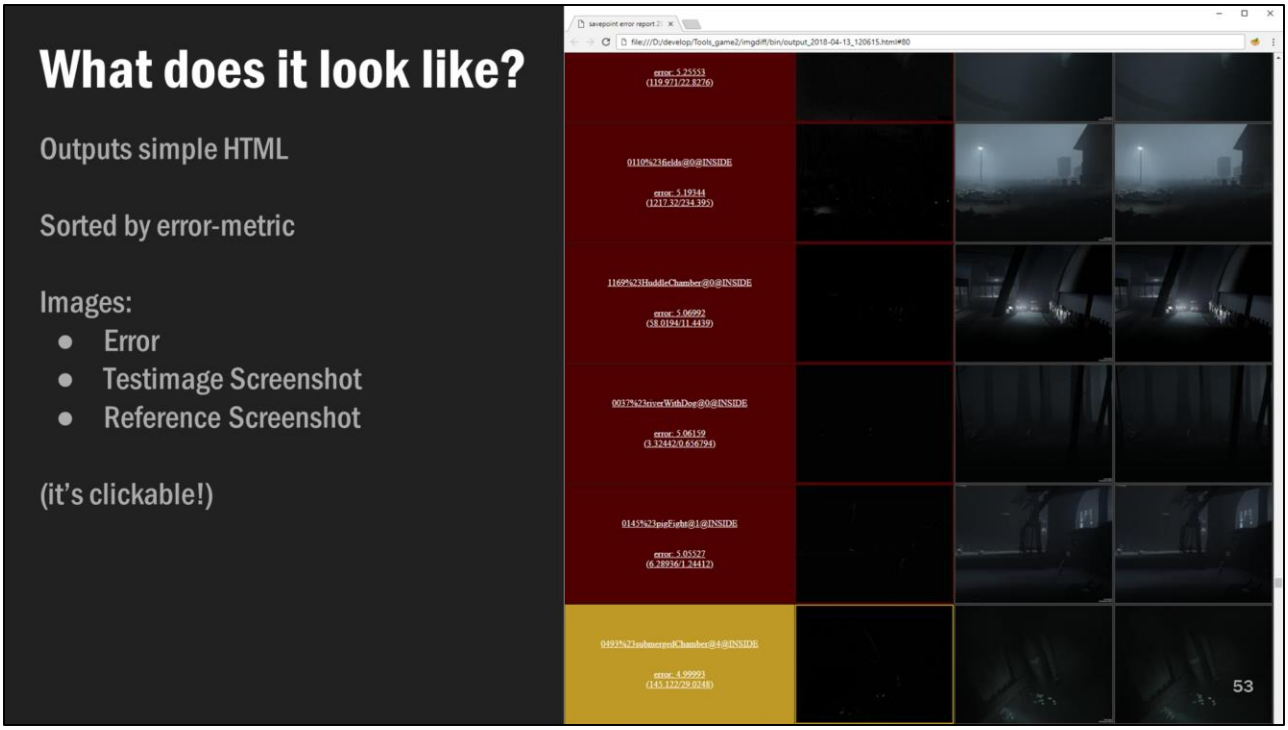
(1300 test-images, 10 sets of reference images)



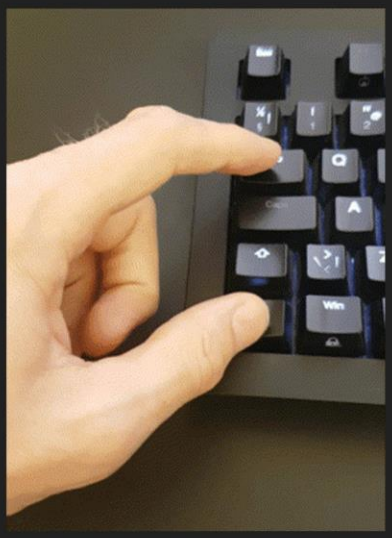
BONUS SLIDE

This will run quite often, makes sense to spent a bit of time optimising tools too...

Calculations on images from a single location is of course independent of other locations
concurrency::parallel_for(int(0), num_images, [&]{int imgidx})

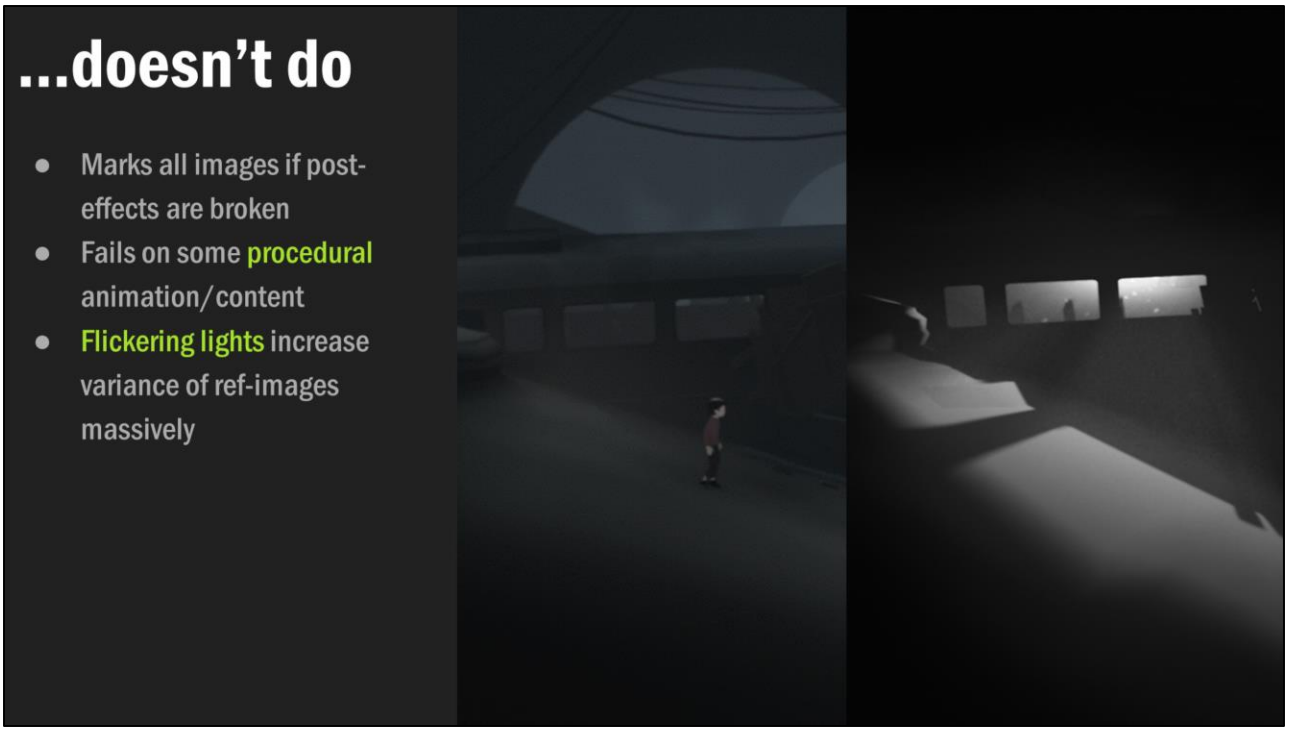


...inspect the errors by opening the two images in new tabs, and then...



54

Err, tab between them... for manual inspection
Admittedly, not conceptually a massive improvement :)
...but arguably with far fewer images now as tool marks images we don’t need to look at

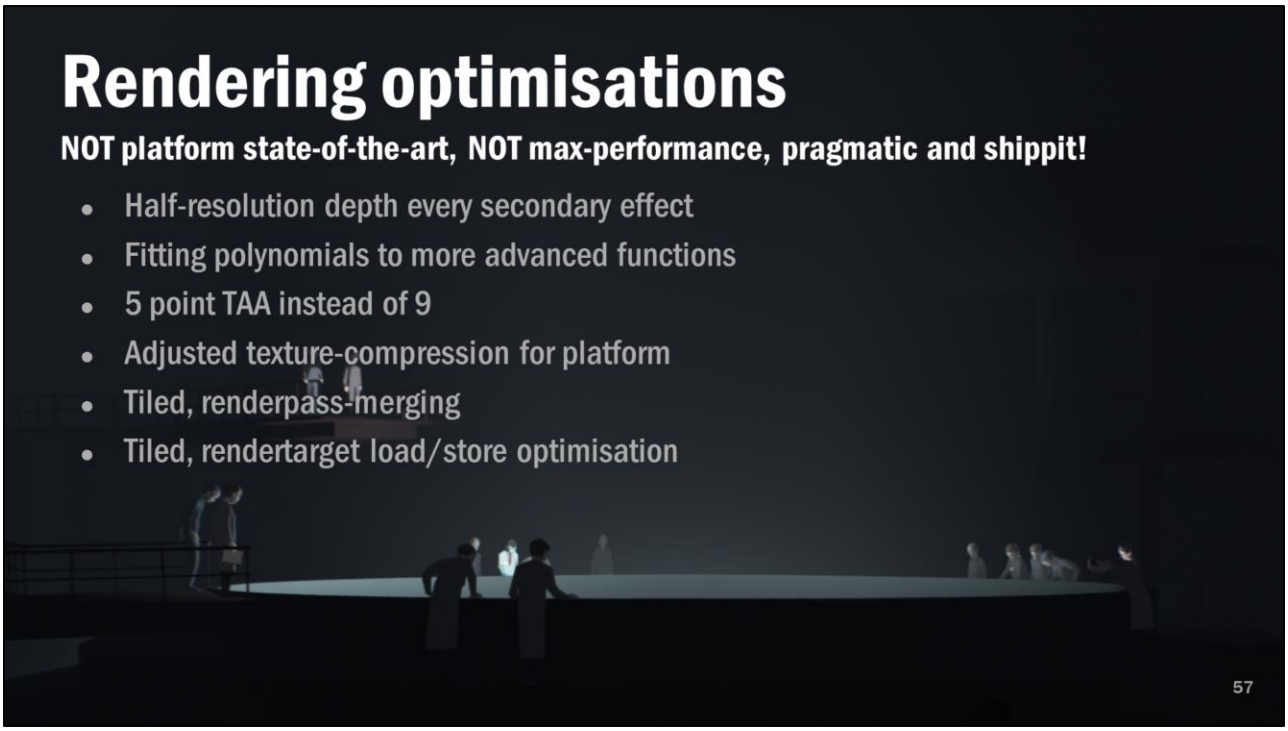


(inverse) confidence image on the right
Very low confidence caused by flickering
can't find error in pixels that are **valid in any color**

Non-deterministic anim should be caught in reference-images, sometimes not)

- Albinos are seed'ed on startup-time
- Some animations seeded since startup-time
  (reference images all run on same computer!?!)

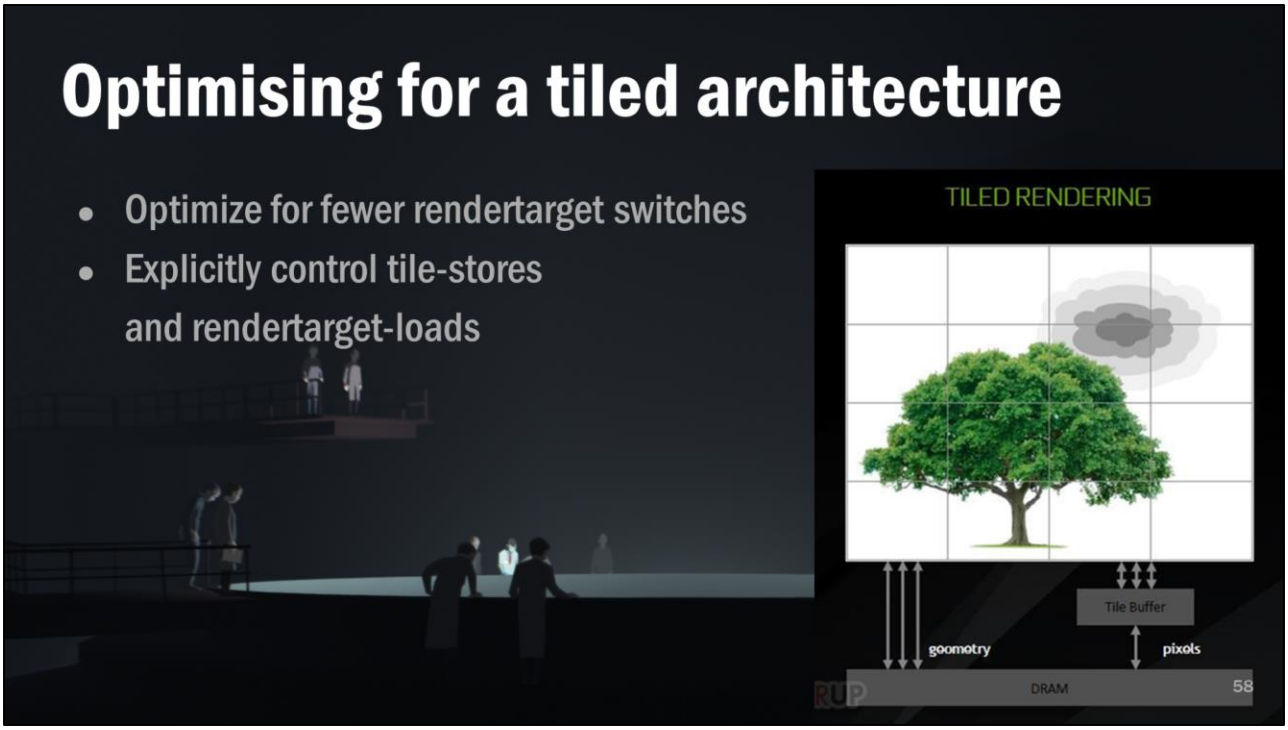

RENDERING
yay!
56



Most optimisations were pretty basic and not too interesting…

As we just **scaled resolution** to run
Only goal for rendering then becomes improving resolution...
With **TAA** better resolution translates to "**less blurry**"

Also reduced TAA to single path and optimised that path
(sometimes you literally can't see the forest for all the trees… or codepaths)

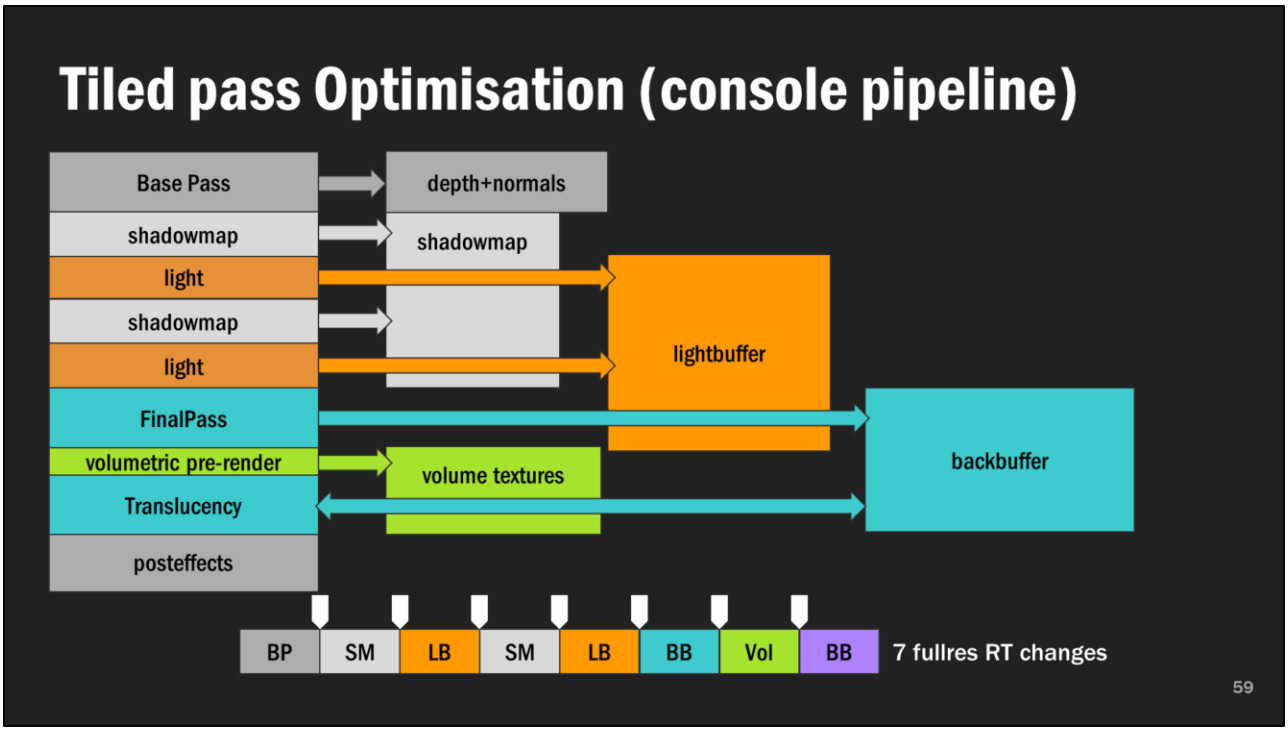


Not going into the details of how tiled-rendering affects performance, just what we did to adapt inside for it

For the purpose of this presentation, consider the Tile a fast local cache on the GPU, that we want to use as much as possible – while flushing or restoring it to main memory (ie the rendertarget) as LITTLE as possible.

Some of the things presented here required changes to the Unity Engine source – in the future of Unity, 2018.1 and forward, it should be accessible through the **Scriptable Render Pipeline**.

--
”rendertarget switches” from a tiled point of view is renderpasses

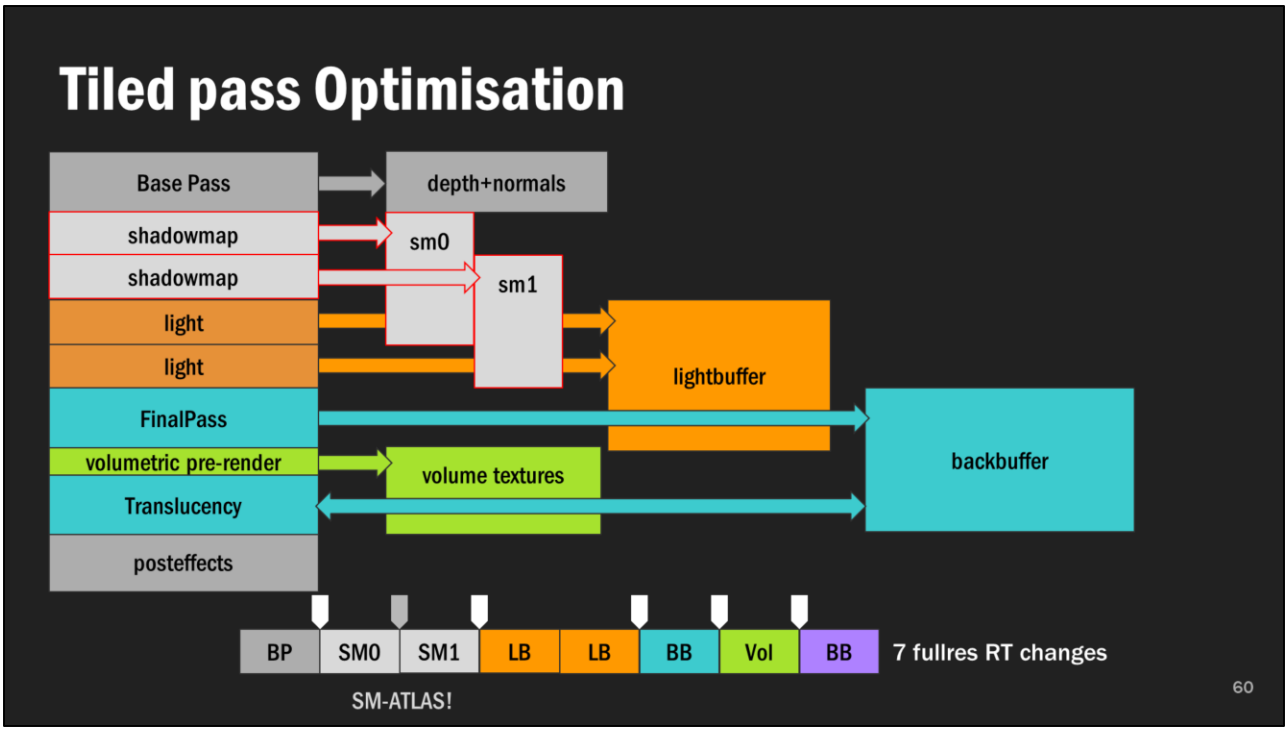


Go from optimising for RT-lifetimes / memory
to optimising for RT-changes

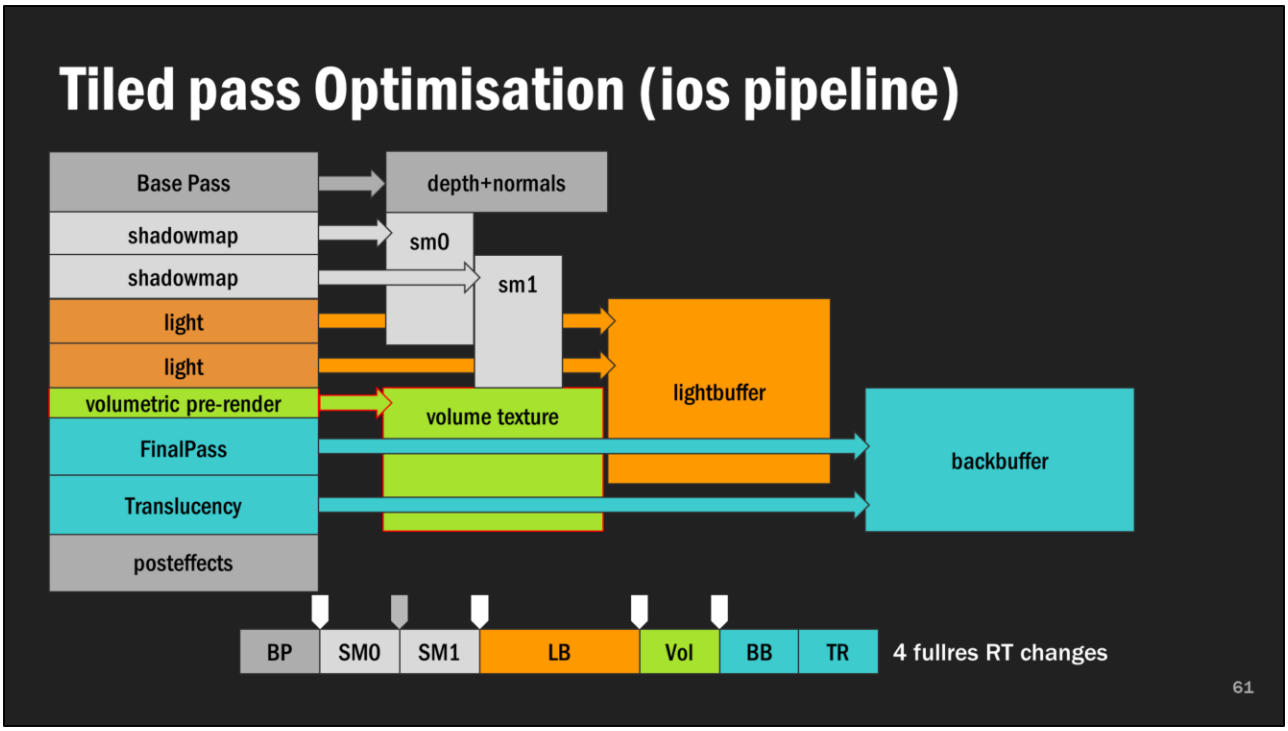
- Moved GCube PreRendering - afterfinal=>afterlight
- Pre-Rendered Shadowmaps

...some lights are light-decals

some shadowmaps also used for volume rendering

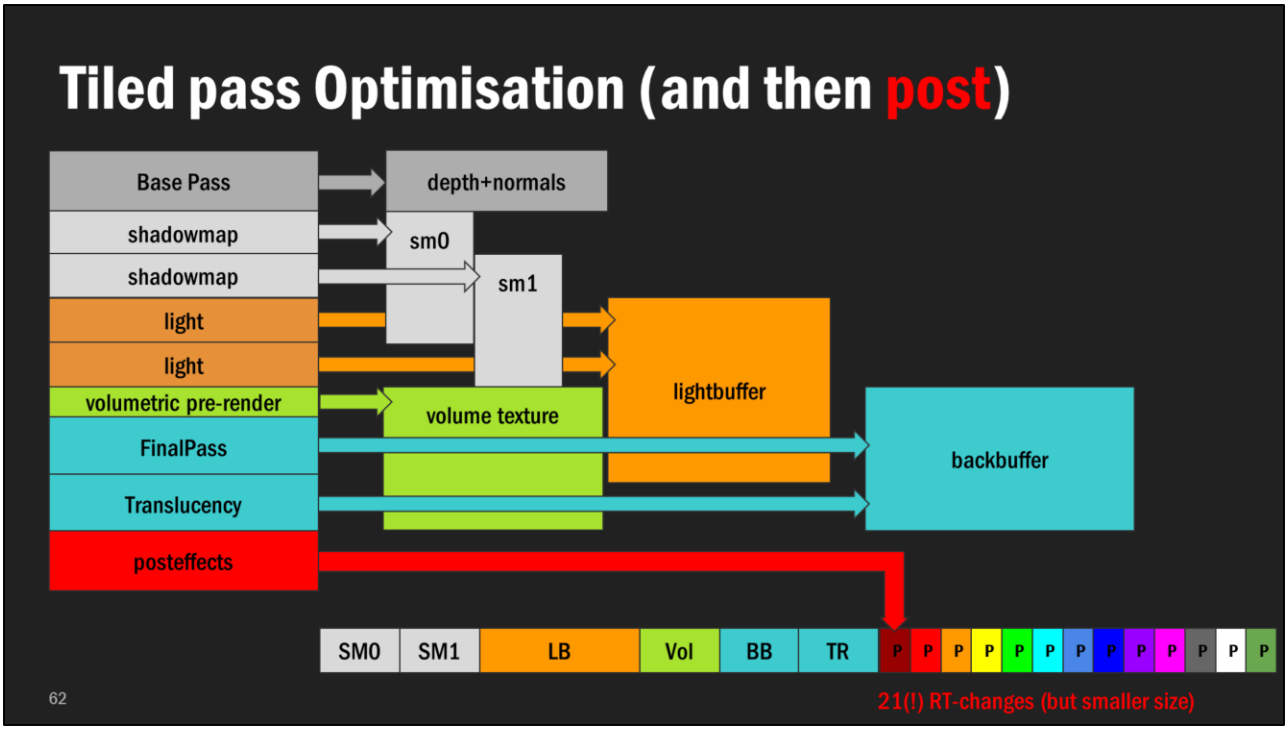


Shadowmap swaps could be reduced by packing in atlas



Possible to do tricks to merge more passes, outputting to MRT but only writing to a single RT at a time (discarding writes to the other)... afraid it would break in the future (as it doesn't port to other platforms right now)

TODO: link



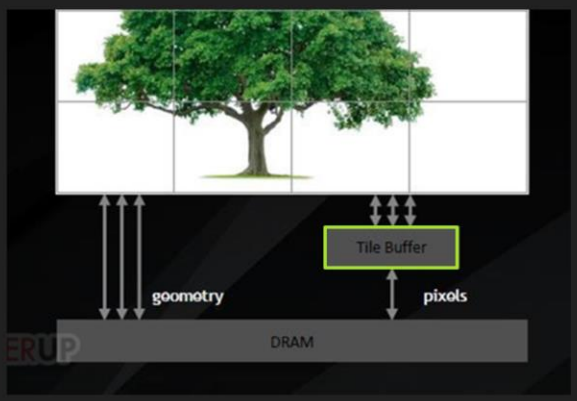
Bloom

Did **NOT** see a significant **GPU** overhead
BUT
Creates a Metal command-encoder per pass
=> **2.5ms** CPU renderthread driver-overhead

Converting passes to **compute** appears to get rid of that overhead entirely

Tiled load/store Optimisation

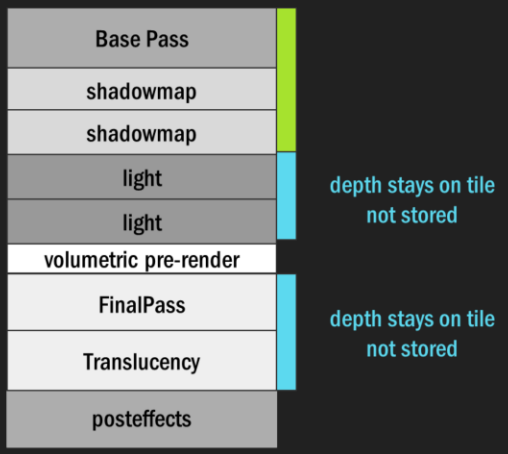
- Manually control what data is load/stored between Tile and RenderTargets
- Unity Defaults to both Load and Store bound Z-buffer (and stencil-buffer) when used



63

- creates a lot of unnecessary zbuffer-stores when z only used for comparison and not modified
- Zbuffer, as opposed to color-buffer, is often bound as rendertarget for **depth-comparisons** only – a bound color-buffer is almost always rendered to and should be flushed.

Tiled load/store Optimisation



Depth-buffer LoadStoreAction

- lighting => load/dontcare
- finalpass => load/dontcare
- translucency => load/dontcare
  - had to change some rendering

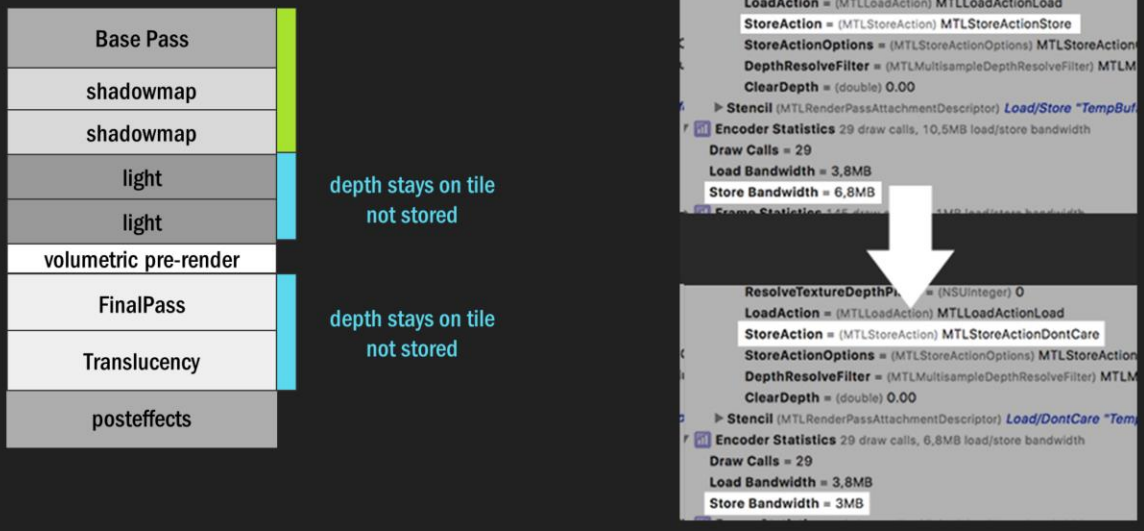
Important to clear rendertargets

Color / Depth / Stencil are controlled separately

ENTIRELY handheld – you can break stuff here.
We had translucency-passes legitimately rendering into the depthbuffer (which is ordinarily fine) that broke when we disabled tile-store in translucency pass.

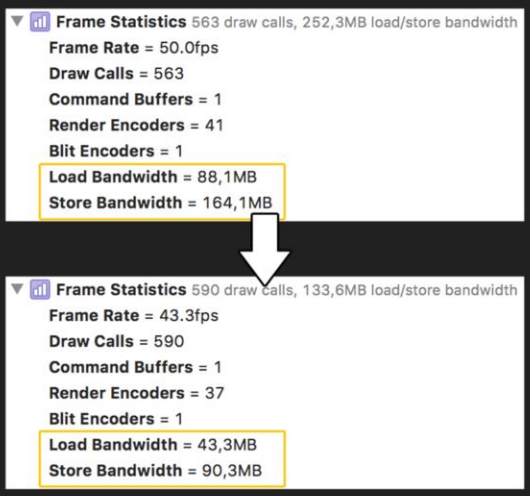
Also important to clear rendertarget rather than just overwrite them, to indicate that rt-data need not be loaded into tile-mem…

Tiled load/store Optimisation



Use xcode to verify that what you did actually worked
(renderpass descriptor)

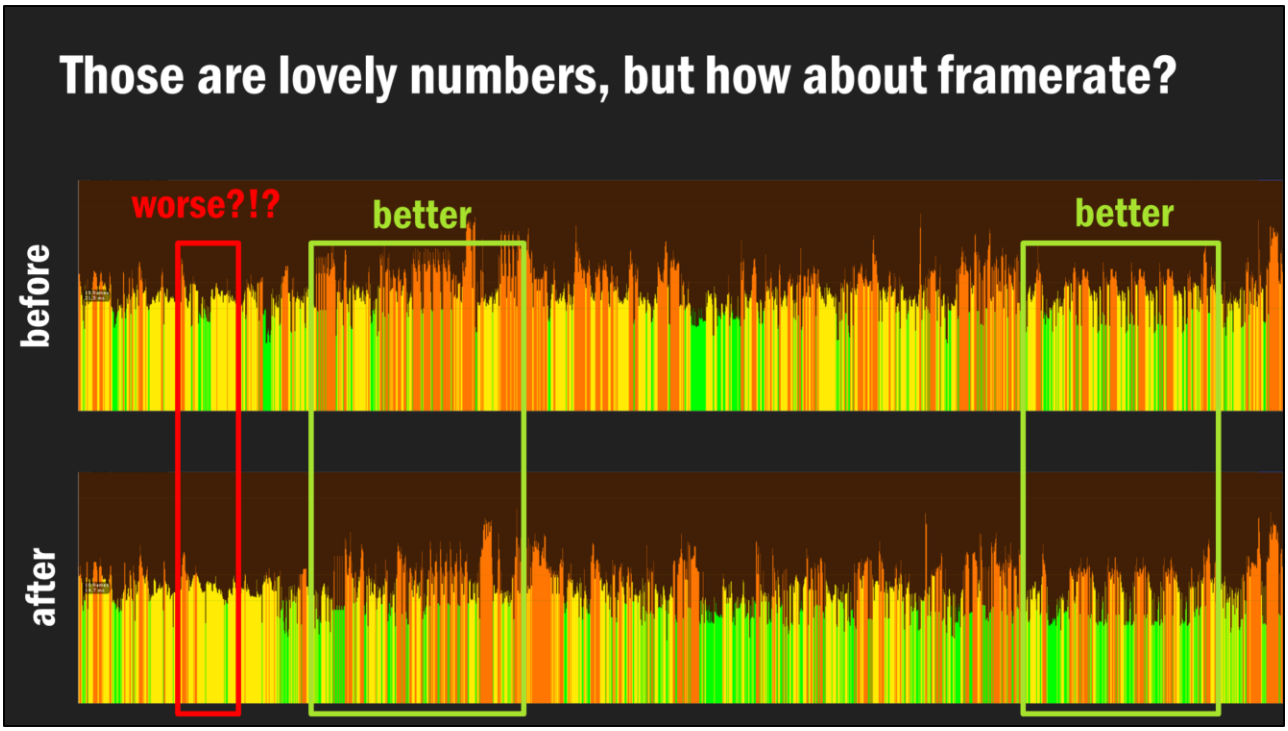
Pass + LoadStoreAction Optimisation



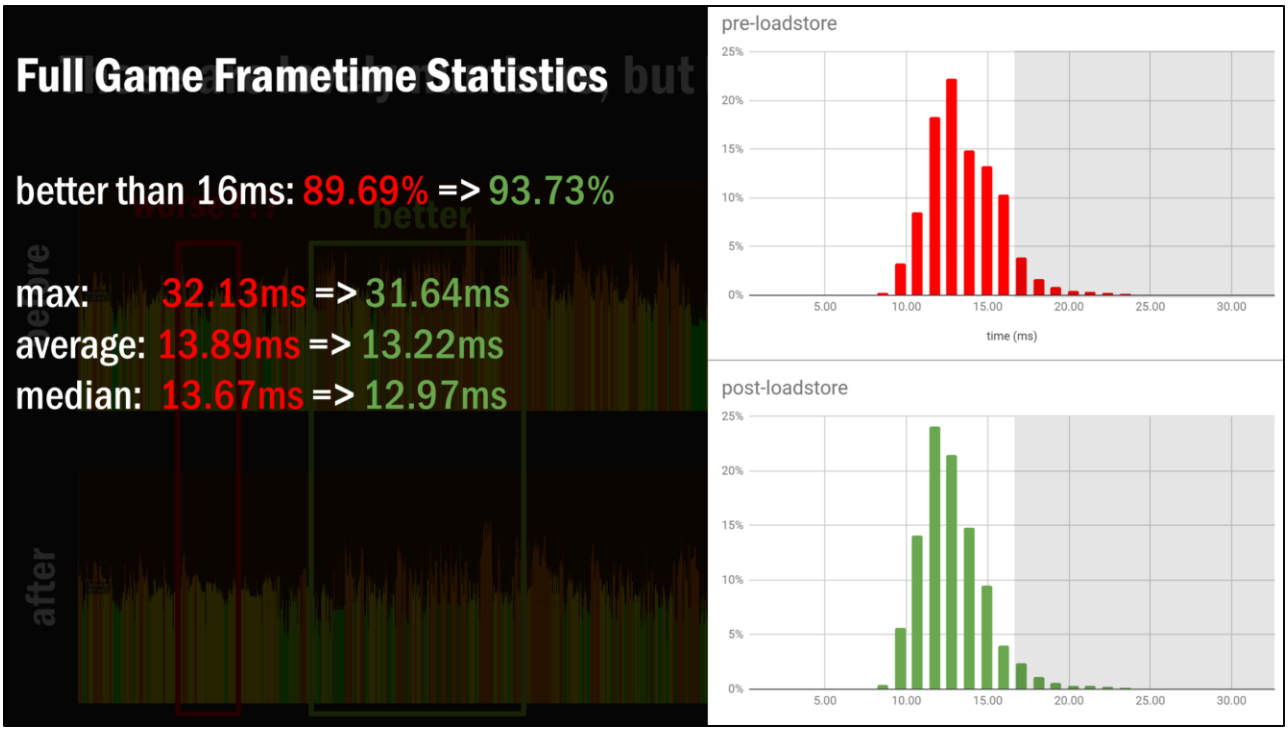
66

If all goes well, nothing changes visually how do we confirm that we made a difference?
Xcode fortunately has counters for tiled bandwidth (but not for e.g. texture-bandwidth…)

Apple tv profile



Mostly better...
No theoretical reason it should be worse, so probably a safe optimisation... though the area that got worse should probably be investigated....
(improves some of the worst places)



Terrible way of comparing anything – but was how we looked at optimisations during production.

Did some stats for this presentation to have something more tangible to relate to...

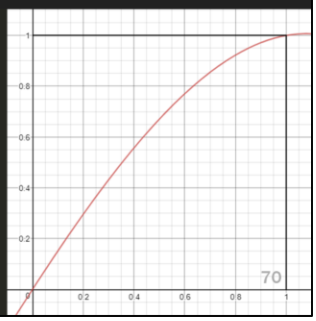
- Just above half a millisecond saved (not ”a couple”, no matter what I said during the presentation...)
- Important bit of the histogram is that there are now significantly fewer values JUST below16ms

Max: -0.49ms => **1.5%** better
Avg: -0.67ms => **4.8%** better
Med: -0.7ms => **5.1%** better

Mobile Screen Brightness

- Tailored for INSIDE - most of the game is in <25% brightness range
- Brightening on mobiles to spare battery from user increasing backlight
  - Same SoftMin-filter as used for dynamic colorgrading in posteffects
    Remains mostly linear in lower range
  - Fitted polynomial with python: scipy.optimize.minimize(..)
    (make sure endpoints match exactly)

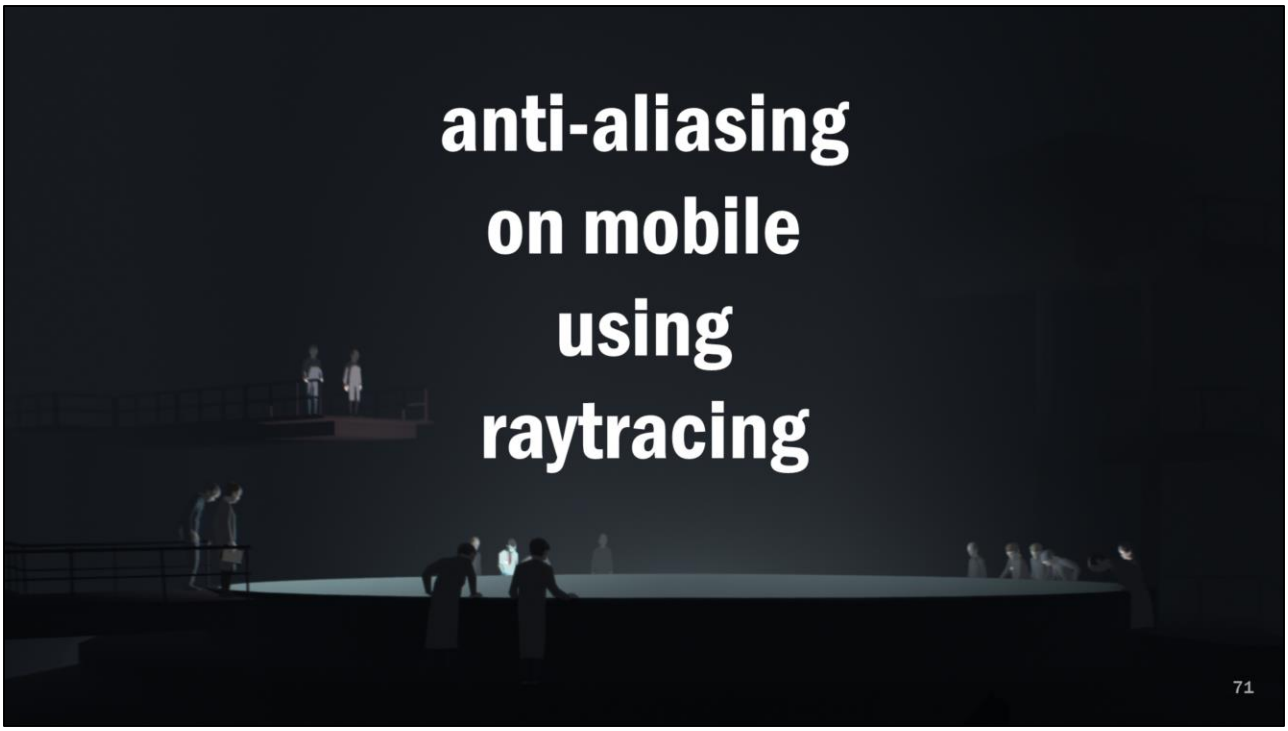
```
//note: approximates remap + softmin with 1
half3 mobile_brightness_adjust( half3 c )
{
    return c*(c*(c*(-0.278737793718h)-0.264172956765h)+1.54291075048h);
}
```

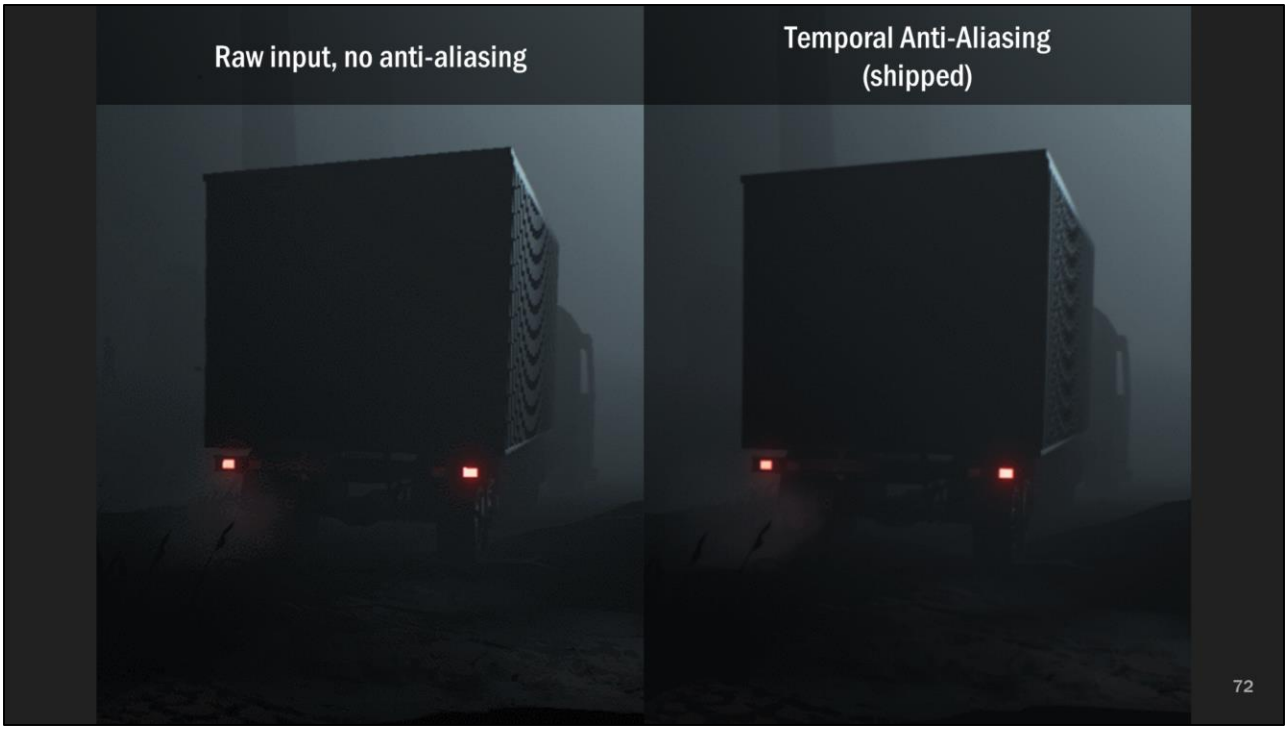


Three reasons
- Get it BRIGHT enough
- Save battery from by reducing backlight (non oled devices)
- Reducing heat from backlight may **improve performance**!!

Fitting a polynomial to function is an approach we use many places to optimise functions

Does it have dynamic input? Often good enough to fit two solutions to either extreme and linearly-interpolate...

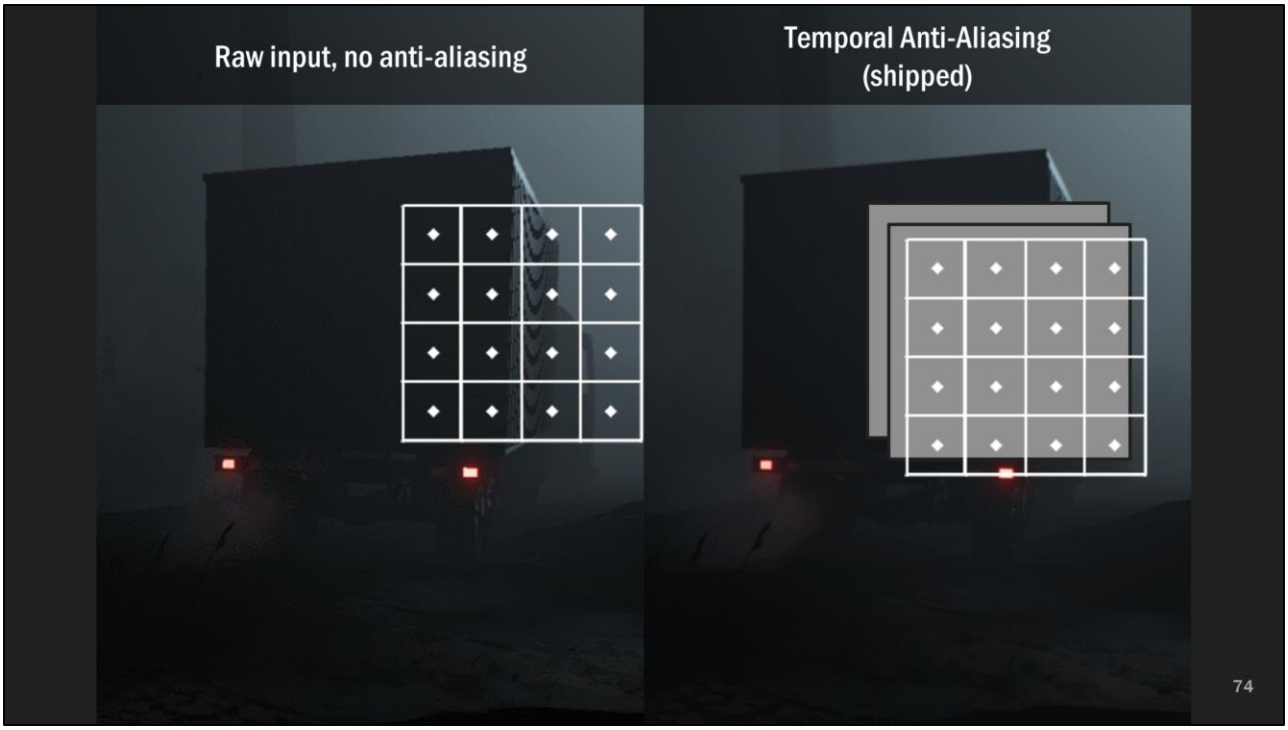

anti-aliasing
on mobile
using
raytracing



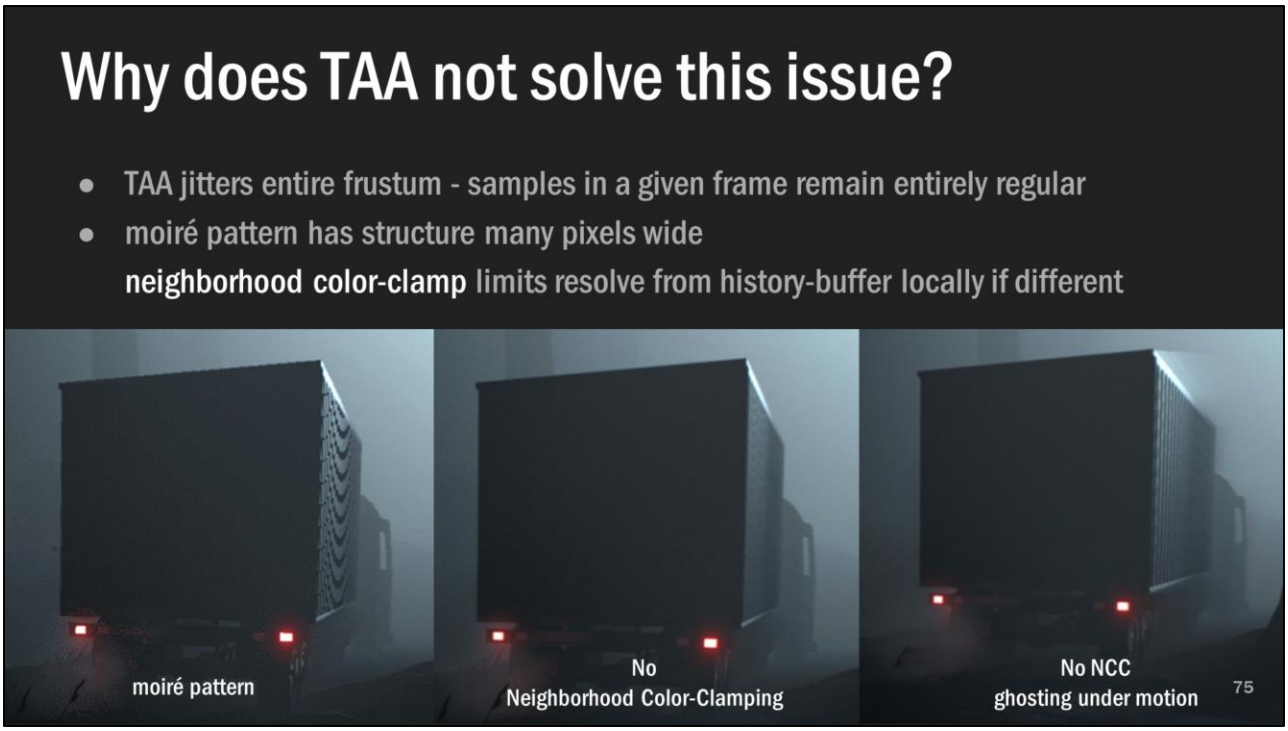
We **had this one place** in the game that we never got around to fixing…
(two actually, but I am not going to tell you the other one until we fixit)

Fundamentally an undersampling problem
- High frequency content sampled too little
- entirely uniform sampling each frame

...worse at **lower resolution** (e.g. mobiles)



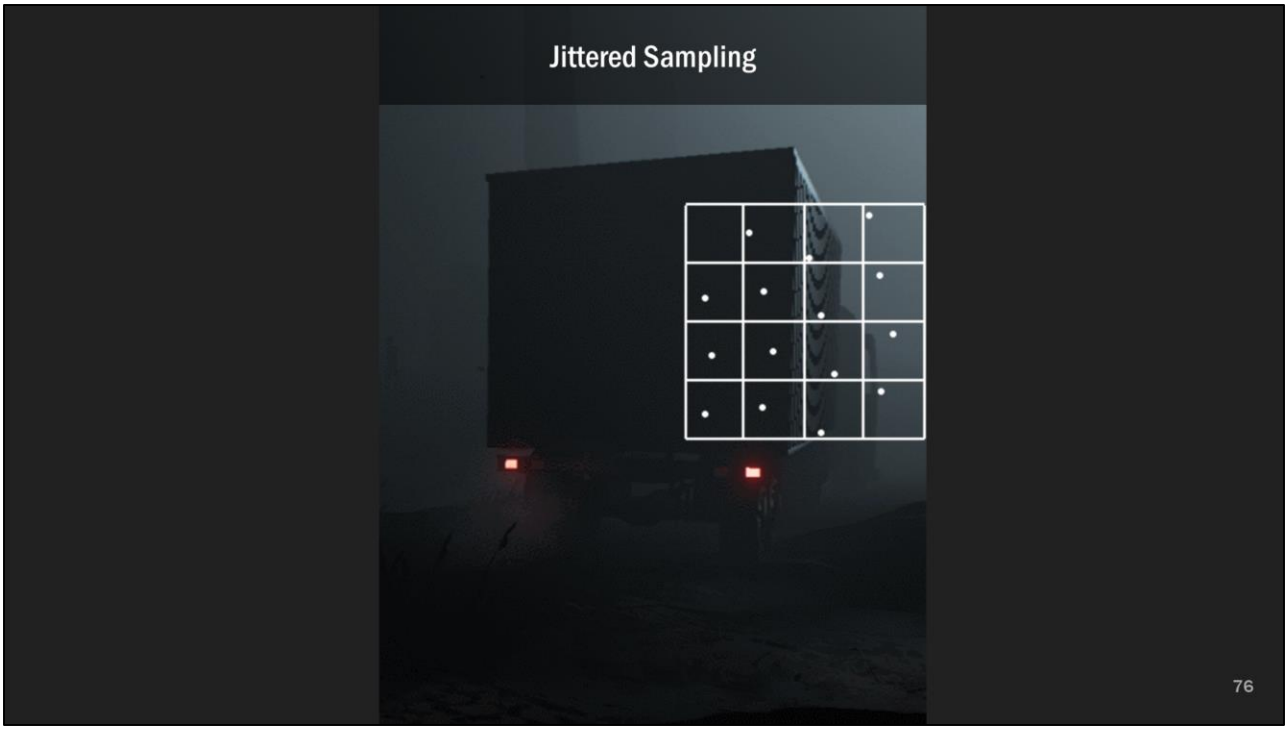
TAA still has uniform sampling per frame



Each frame STILL sees undersampling artefacts (moiré patterns)

Wide pattern => causes the TAA pixel-neighbourhood to only see similar pixels – REJECTING anything in the history buffer not identical to it

(not affected by limited feedback or history-rt precision)



Jittered grid
To make sure each **pixel-neighborhood** sees a lot of **variance**

we are converting **pattern to noise**

-
…get TAA to accept history by uniformly forcing more local variance – essentially, we want **noise rather than patterns**!

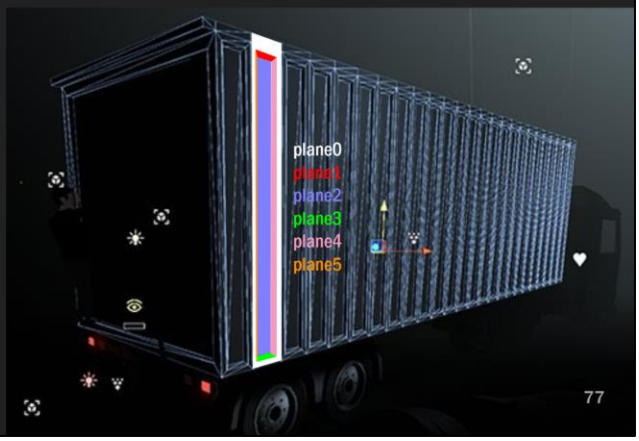
Procedural modelling and raytracing a truck (side)

Goal: Convert Aliasing-to-Noise

Solution:

Jitter geometry sample-location per pixel

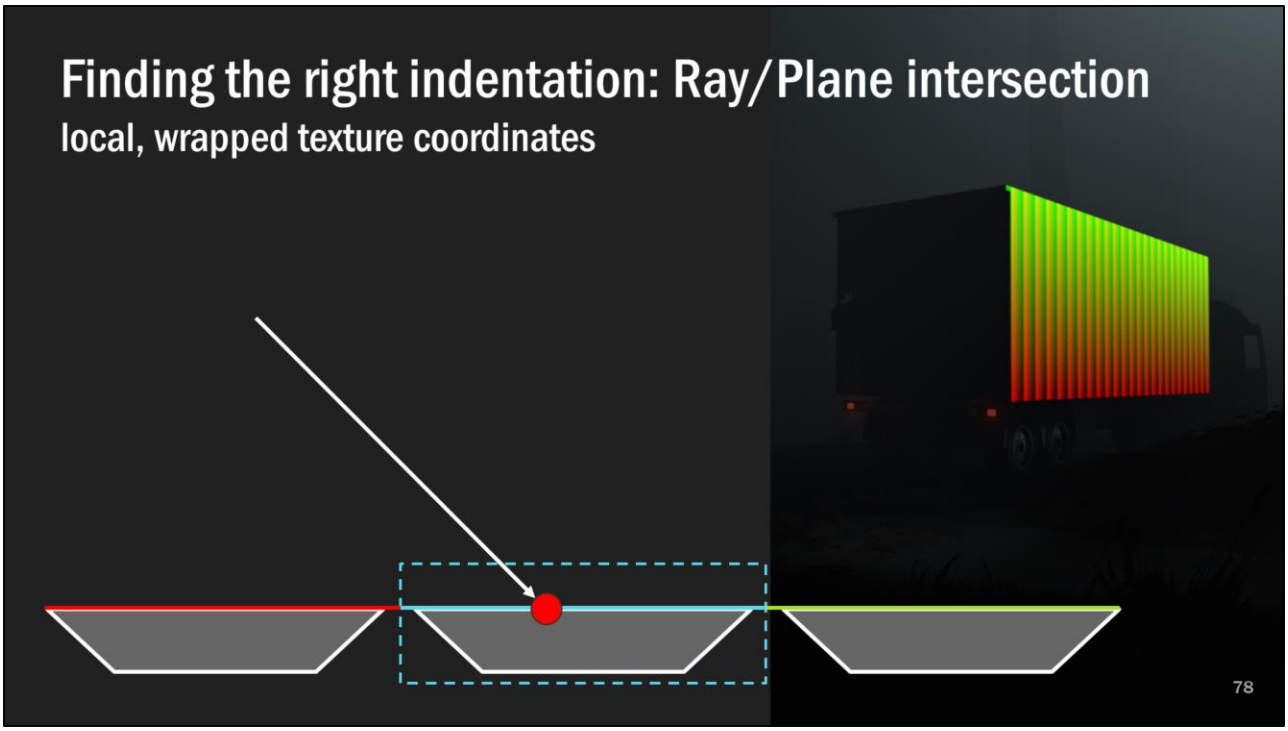
- Raytrace hardcoded planes
- **Per pixel** blue-noise ray-jitter
  => TAA color-neighborhood has **higher variance**
- Forward-rendered "patch-up"
  - hardcoded lights / decals interacting with the surface – all code duplicated here :(



77

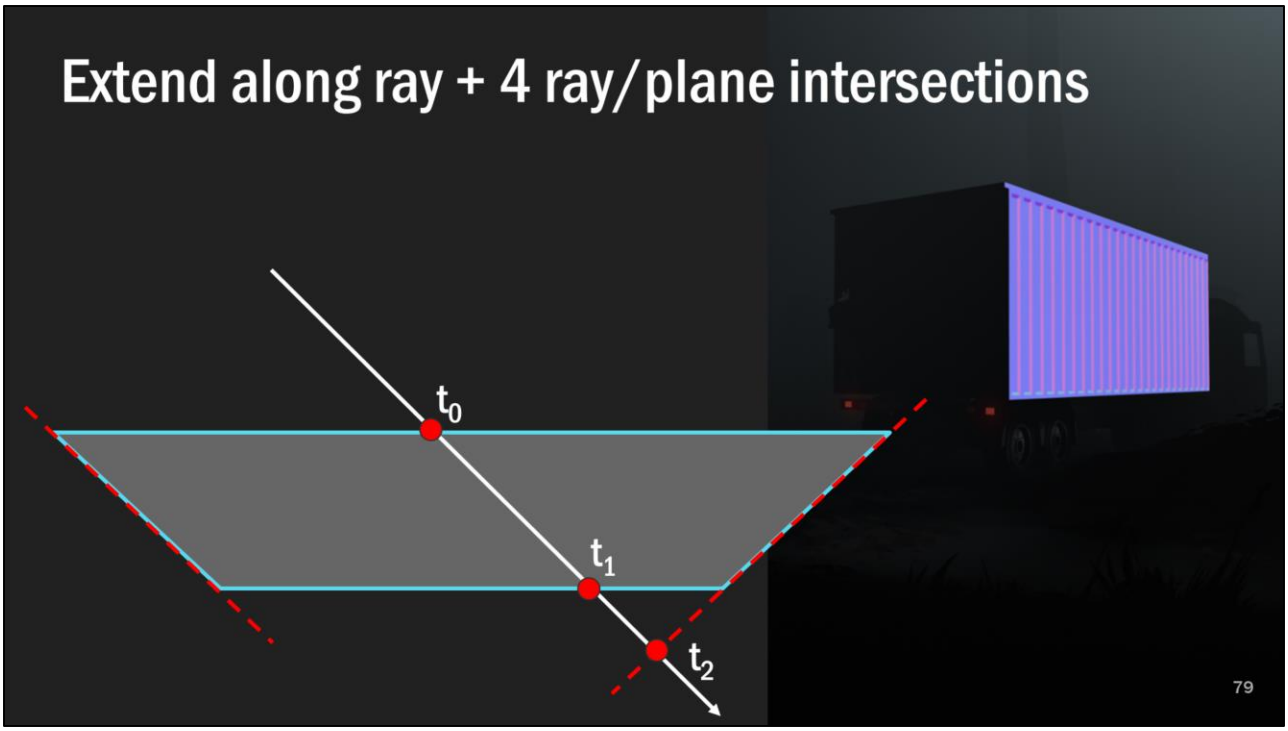
Can't do this using rasterization – we CAN by raytracing...

So procedurally model the truck-side



observation: can not hit other indentations along the ray

Find the right groove by simply wrapping uvs



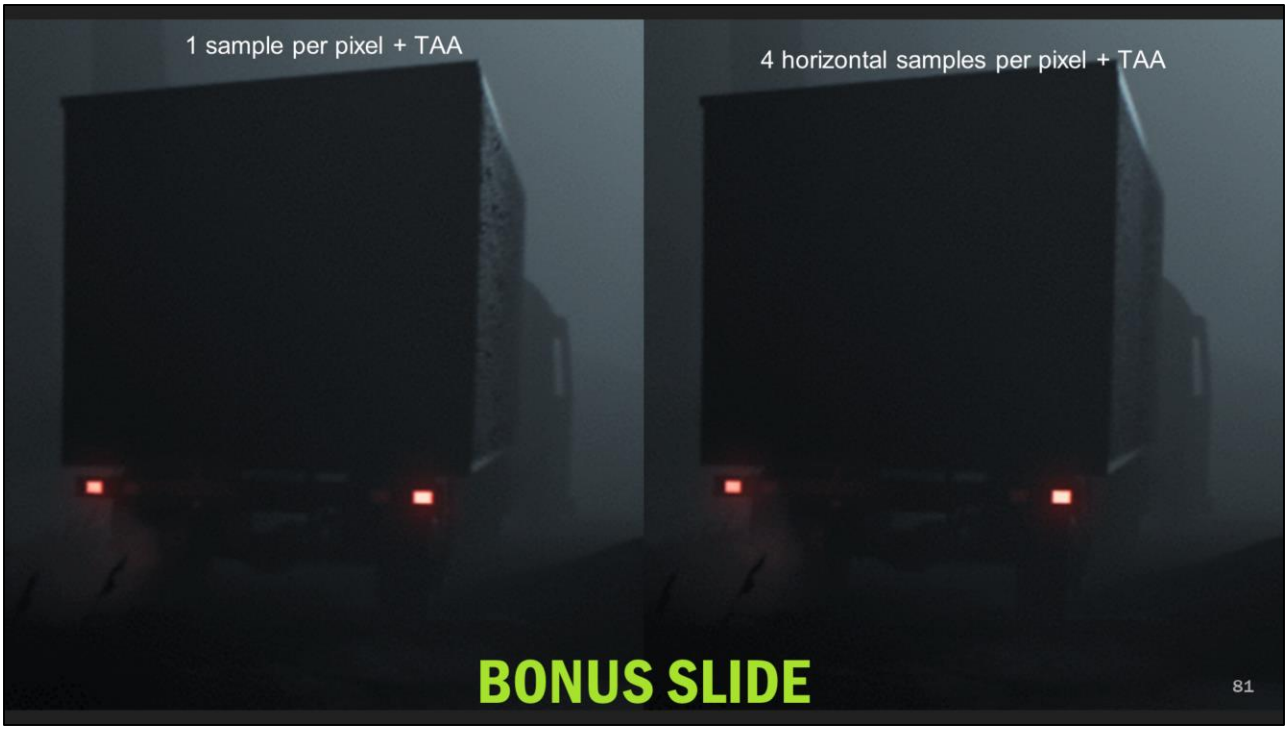
Observation is that all intersections are in the range [t0;t1]
(which is the same as min(intersections[])

Calc t1 directly from ray

In this context, the TAA primarily is just an accumulation buffer

Jittering removes pattern, higher frequency noise -> neighborhood color clamping now accepts a wider range of colors

:´(:| :D



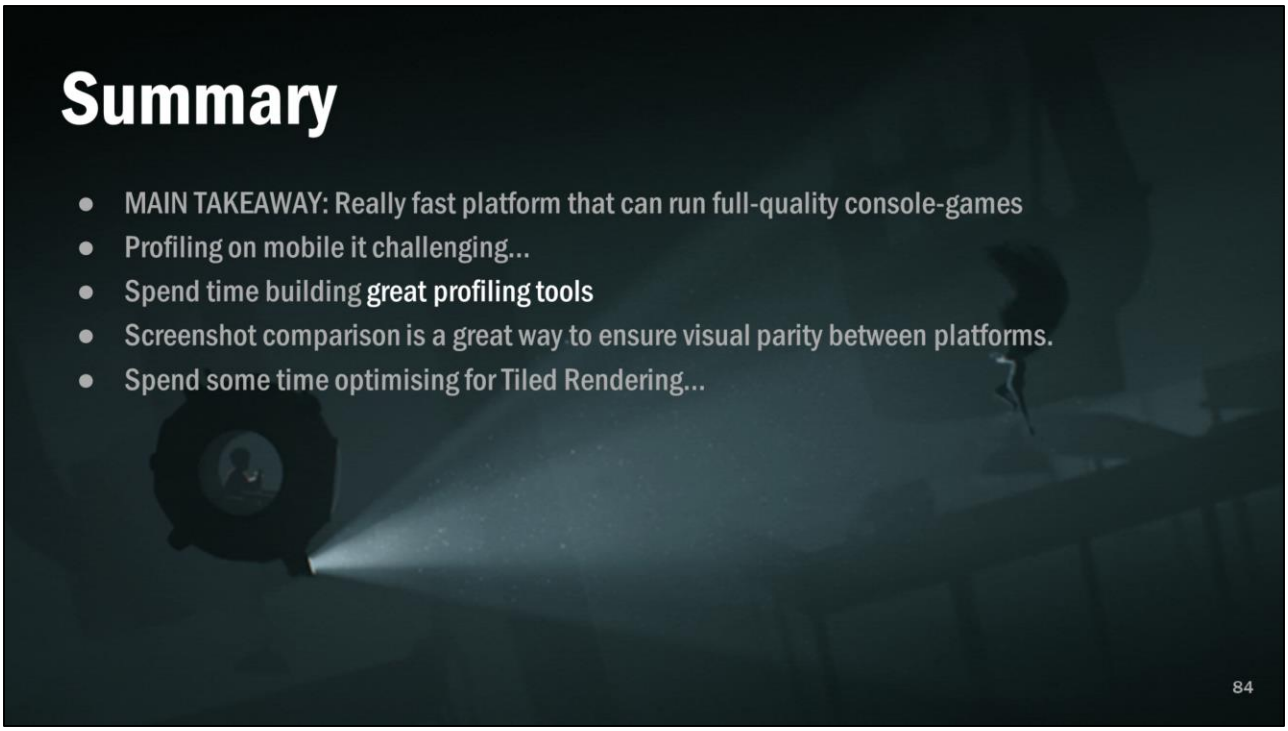
Samples 1 vs 4 – despite the jitter + taa, a single sample was not enough

~~Cutting~~ Optimizing Corners!

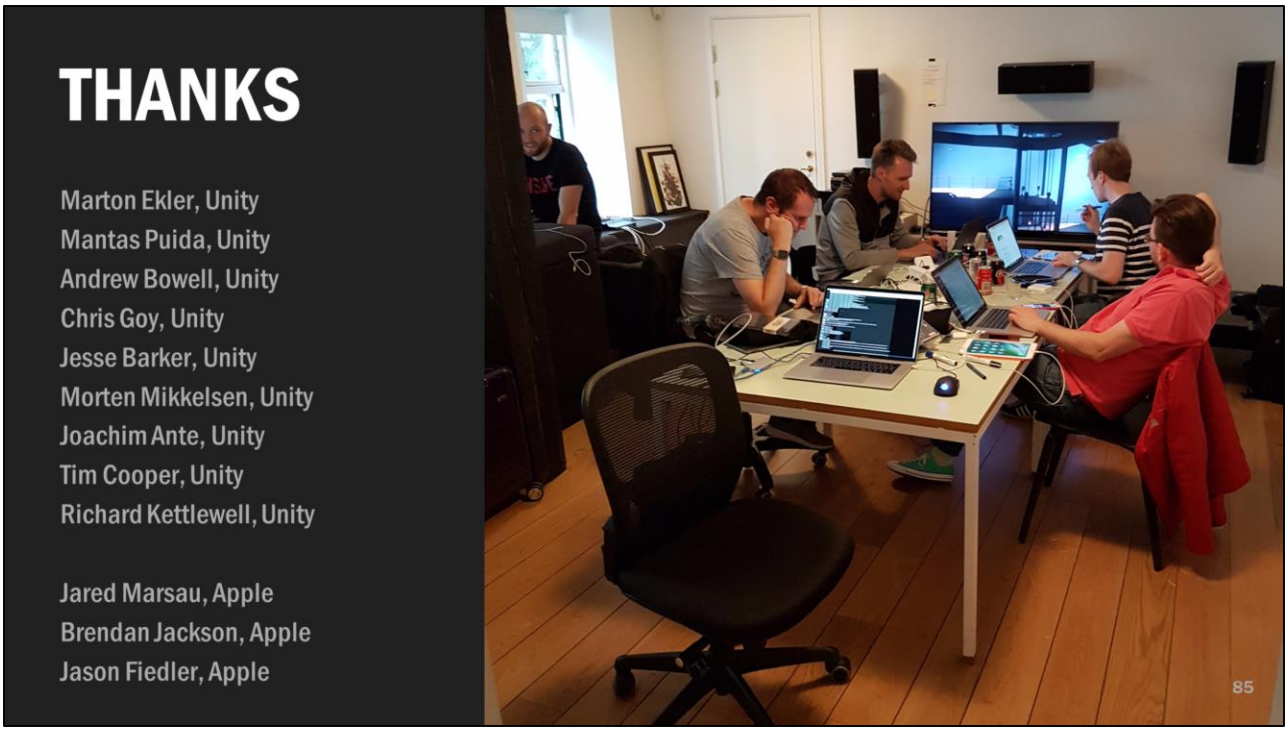
- Specific content only aliases horisontally...
  => multi-sample in **X only** (4 samples was enough)
- **Approximate** normal-independant functions by evaluating at polygon-surface
  - Calculate light/decal intensities at geometric plane
    (ie sample projected textures at surface-level)
  - Normal-dependant calculations have to be done for each sample, e.g. specular/fresnel

Side note: Method could be implemented using **texture-based displacement** mapping instead (e.g. Parallax Occlusion Mapping, Relief-mapping...)


SUMMARY
83



Profiling tools, great investment EVERY time you ship



...and thanks to you for listening.
I hope some of this made sense, maybe even interesting - possibly even useful in the future...
and if at any point during this presentation you thought **you could do better**

…please do :)


Questions?
(remember: There are no stupid questions, only stupid people.)
MacBook Pro
command
option

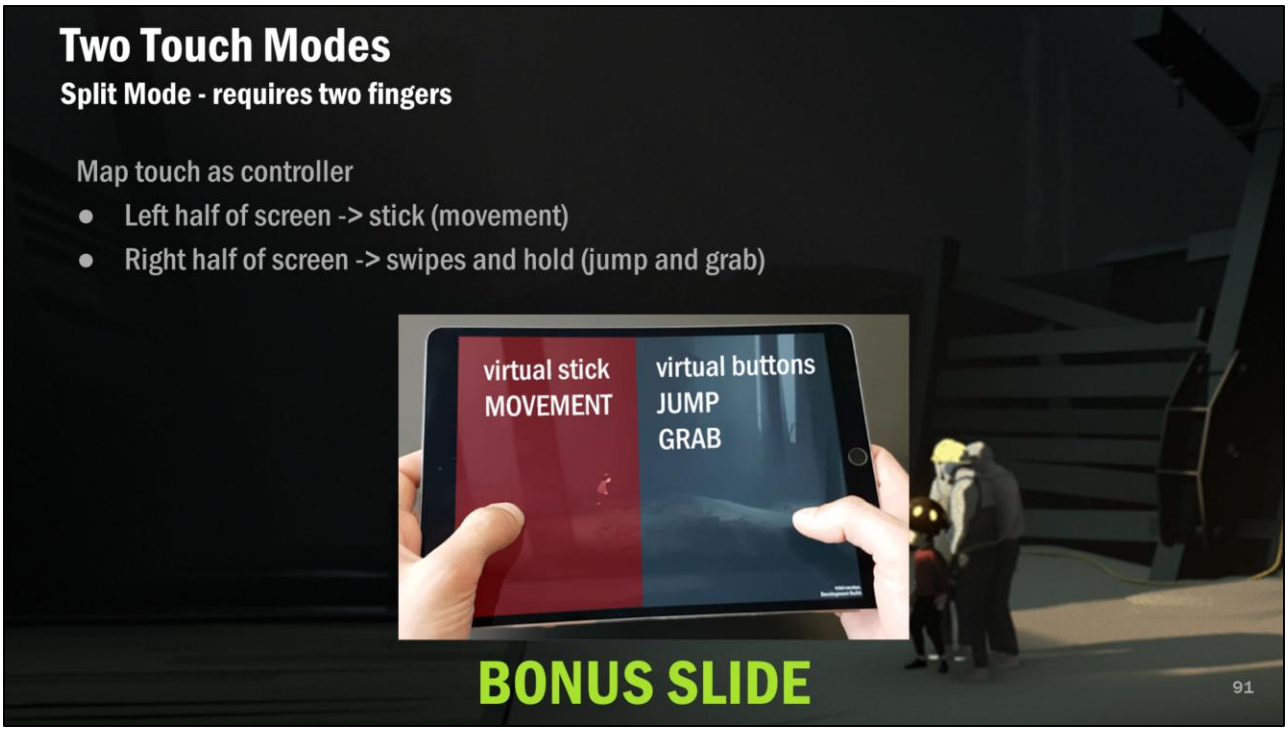
References

https://github.com/playdeadgames/publications/tree/master/INSIDE

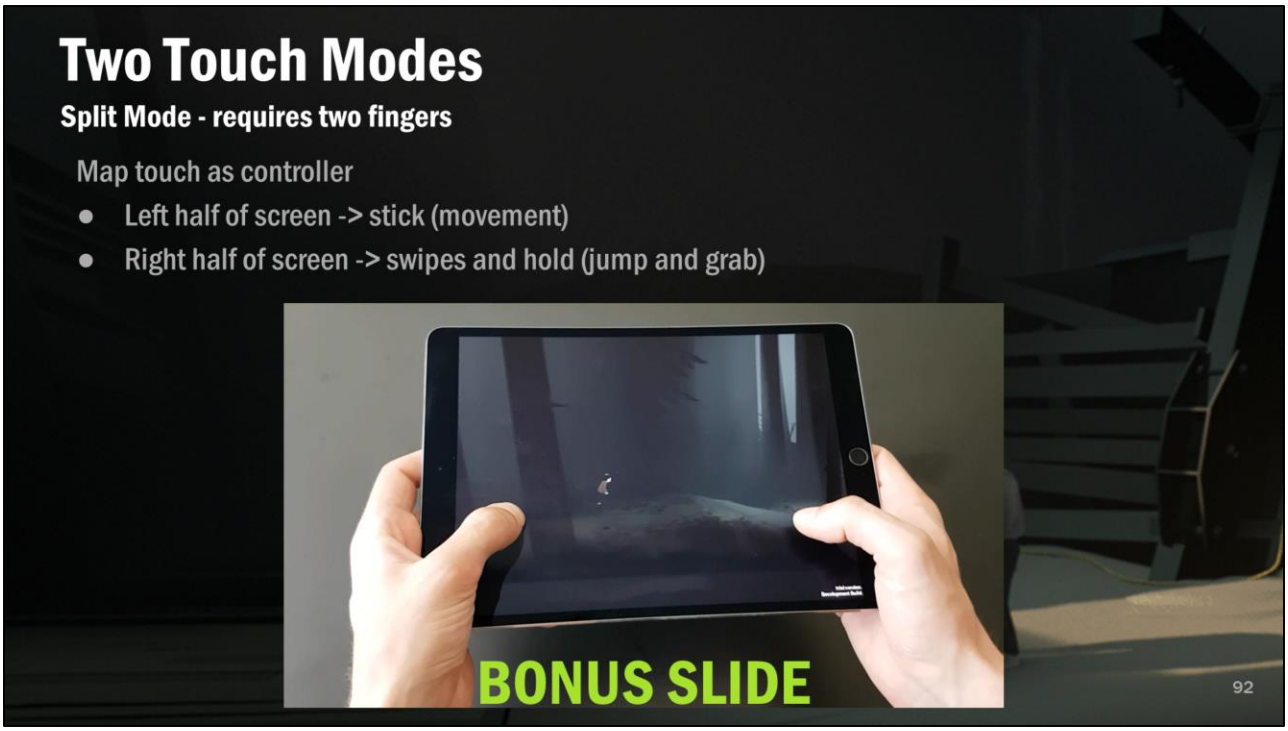
http://www.gdcvault.com/play/1025324/Automated-Testing-and-Profiling-for

https://developer.apple.com/videos/play/metal/7/

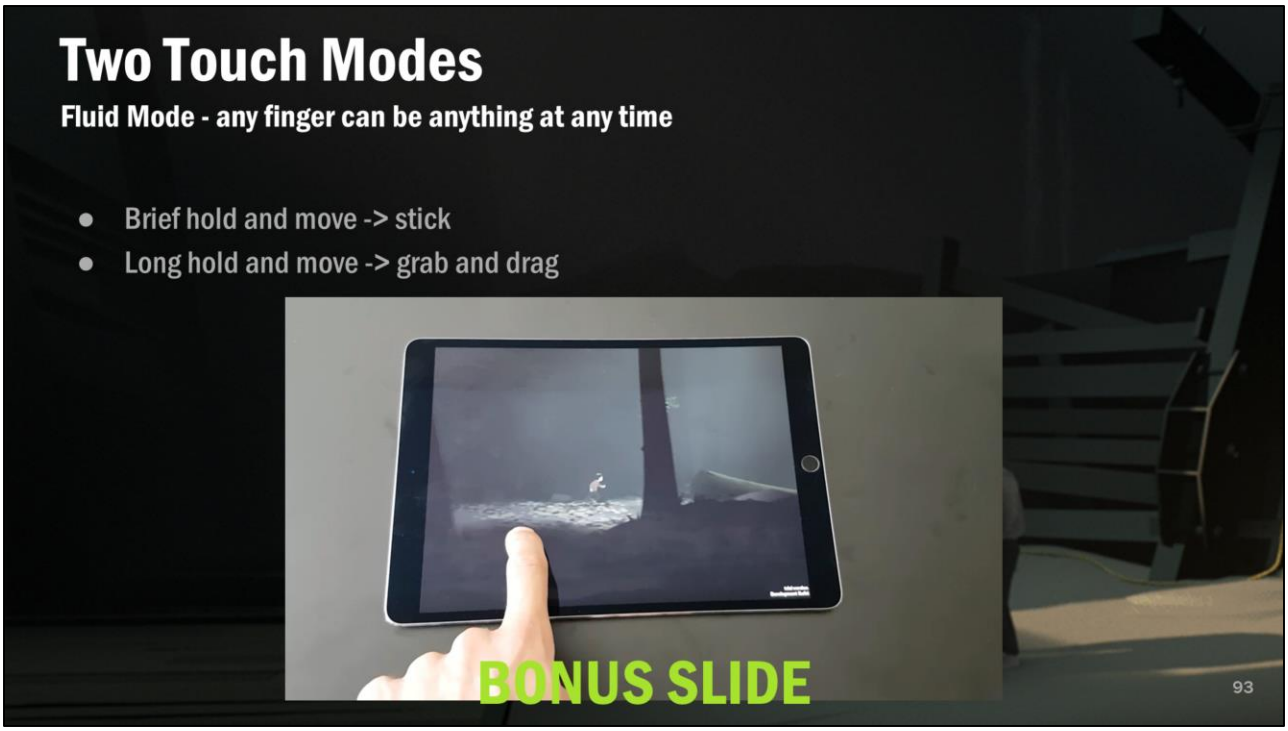
BONUS SLIDES



all touches/fingers are treated equally and a state-machine is tracking their individual behavior from touch-start to touch-end.
Their states will all be combined.
If one finger moves right and another left, the boy will stand still as a result.



all touches/fingers are treated equally and a state-machine is tracking their individual behavior from touch-start to touch-end.
Their states will all be combined.
If one finger moves right and another left, the boy will stand still as a result.

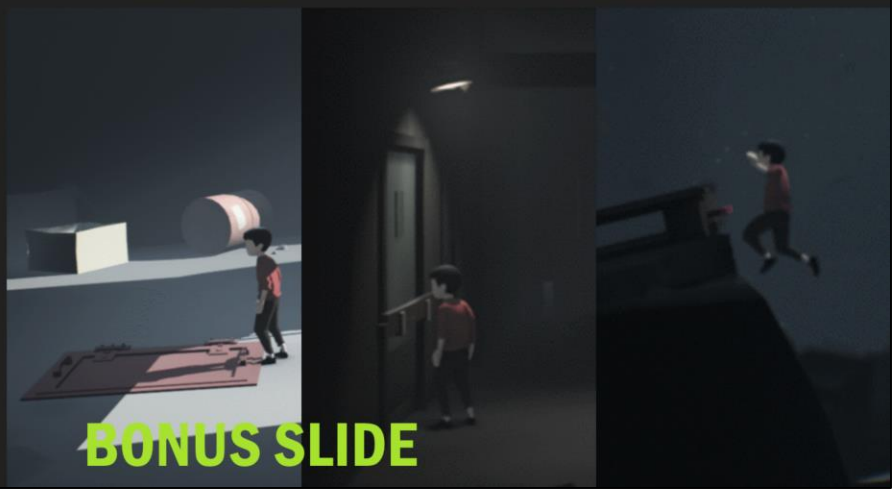


Robust

Hold device with a thumb ON-SCREEN

Touch Content Changes: Swipe to grab

Stick towards something to grab (hatch, door, underwater raft)



Make it easier to play on touch

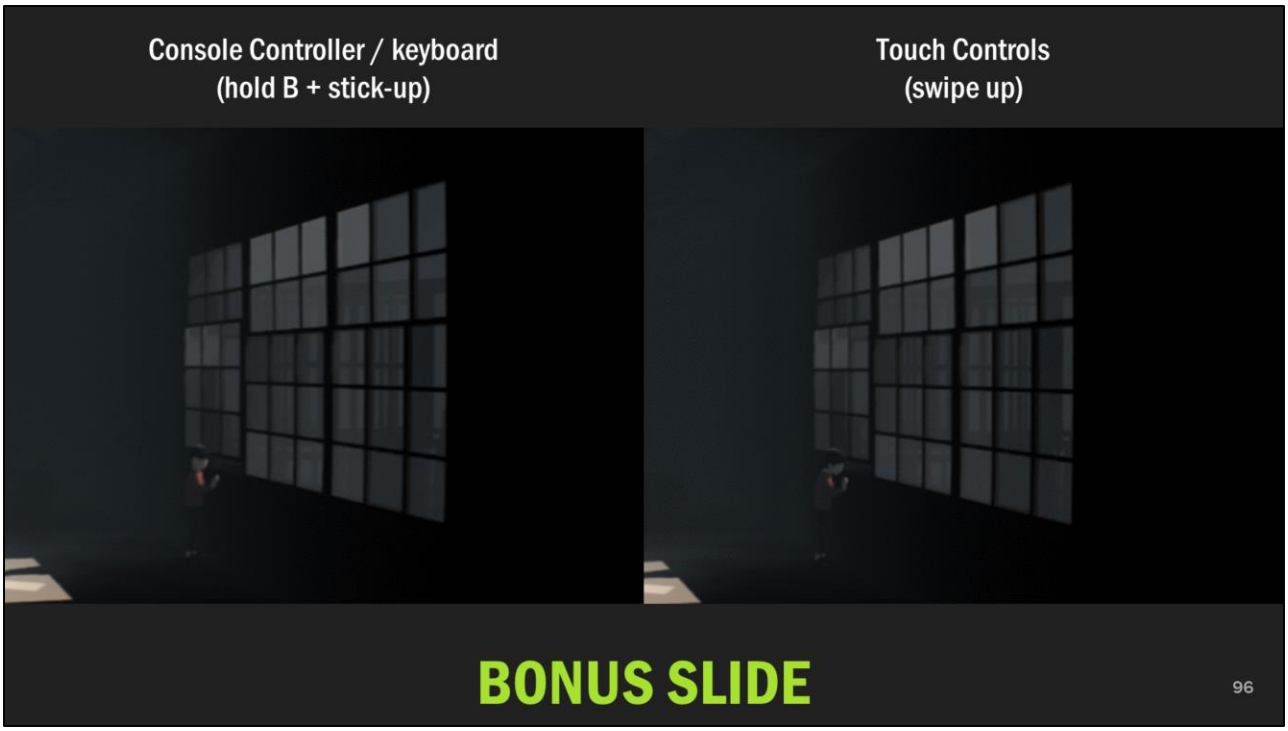
Swipe _towards_ object to grab: left, down…

(very useful underwater... especially while drowning)
(sidenote: Not on objects the character moves around, so only works on things that blocks movement)

Make it feel better

Started with jetcrate and spread to the rest of the game
Other examples: pulling lever, opening window, pulling out raft, pulling secrets

No flick-stick on console-controllers

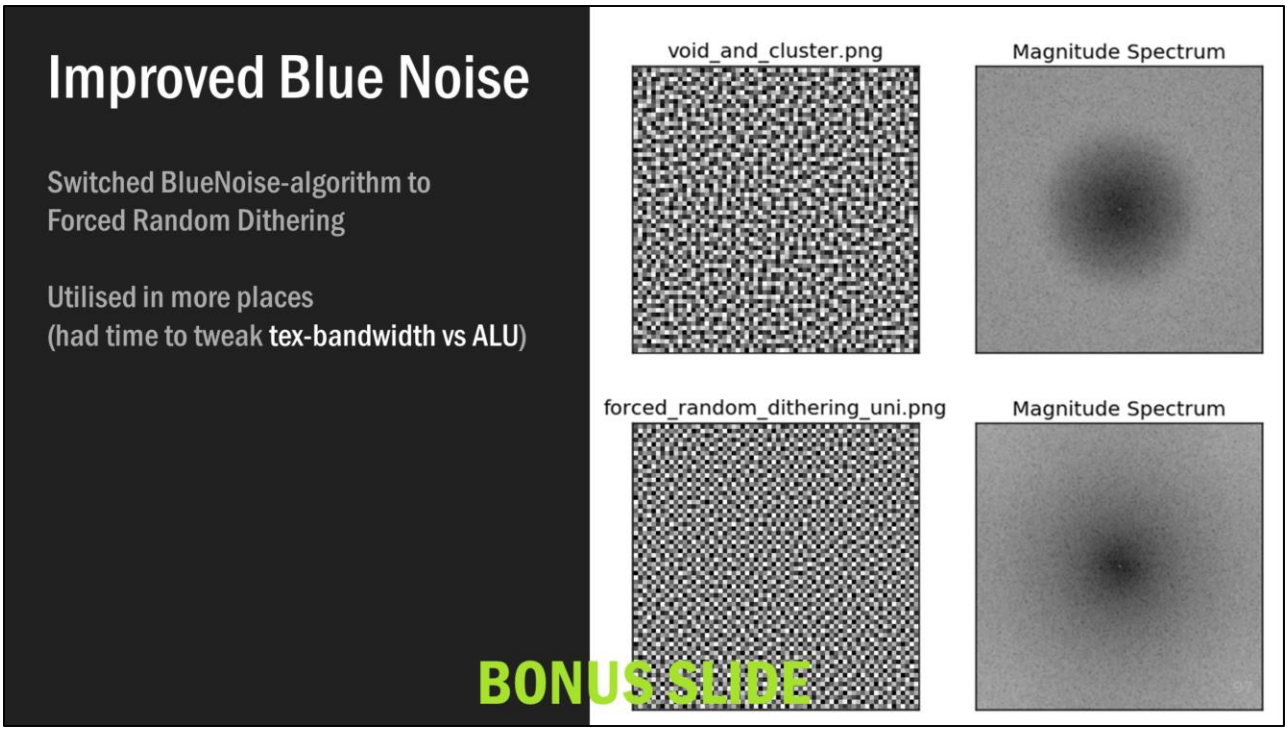


Tried auto-complete, ends up waiting for triggered animation to end. loses feeling of interaction :(
Worked a lot better swipe-wise – keeps interaction

Short as possible animation while still **prioritising** the feeling of interaction

Very **painful** change, since we really liked the weight of the controller input
Balance between matching the swipe, while not looking too instantaneous

Challenging change when timing was important (Sentry gun in city, Sentry gun in sewers)



Fourier spectrum on the right, shows that the transition seems softer... (don't have objective numbers on this or 2D plot...)

https://github.com/pixelmager/BlueNoiseGenerator/tree/master/Reference

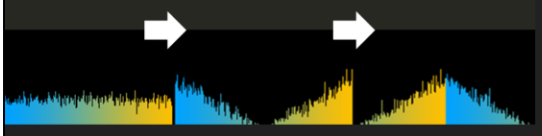
https://ieeexplore.ieee.org/document/413512/
Also: Graphics Gems V (
https://books.google.dk/books?id=ekGjBQAAQBAJ&pg=PA297&dq=forced+random+dithering&hl=en&sa=X&ved=0ahUKEwjDj53ws8PaAhWL26QKHRXADr0Q6AEIJzAA#v=onepage&q=forced%20random%20dithering&f=false)

https://www.cg.tuwien.ac.at/research/publications/1994/Purgathofer-1994-FRD/TR-186-2-94-15Paper.ps.gz
https://www.vrvis.at/publications/pdfs/PB-VRVis-2017-012.pdf
https://pdfs.semanticscholar.org/0f83/600b0ec569cc6668530ff82dd654dcaf8ed8.pdf

Triangular Remapping of Blue Noise

Shipped INSIDE on other platforms with subtly broken blue-noise

```
//note: faster, but only for unordered noise
//note: range [-0.5;1.5]
float remap_to_tripdf__tf( float x )
{
      float orig = x*2.0 - 1.0;
      float f1 = orig*inversesqrt( abs(orig) );
      return f1 - sign(orig) + 0.5;
}
```



f1 result

```
//note: slower, works with ordered noise
//note: range [0;1[
float remap_to_tripdf__erp( float x )
{
            float r2 = 0.5 * x;
            float f1 = sqrt( r2 );
            float f2 = 1.0 - sqrt( r2 - 0.25);
            return (x < 0.5) ? f1 : f2;
}
```

f1 f2 result

BONUS SLIDE

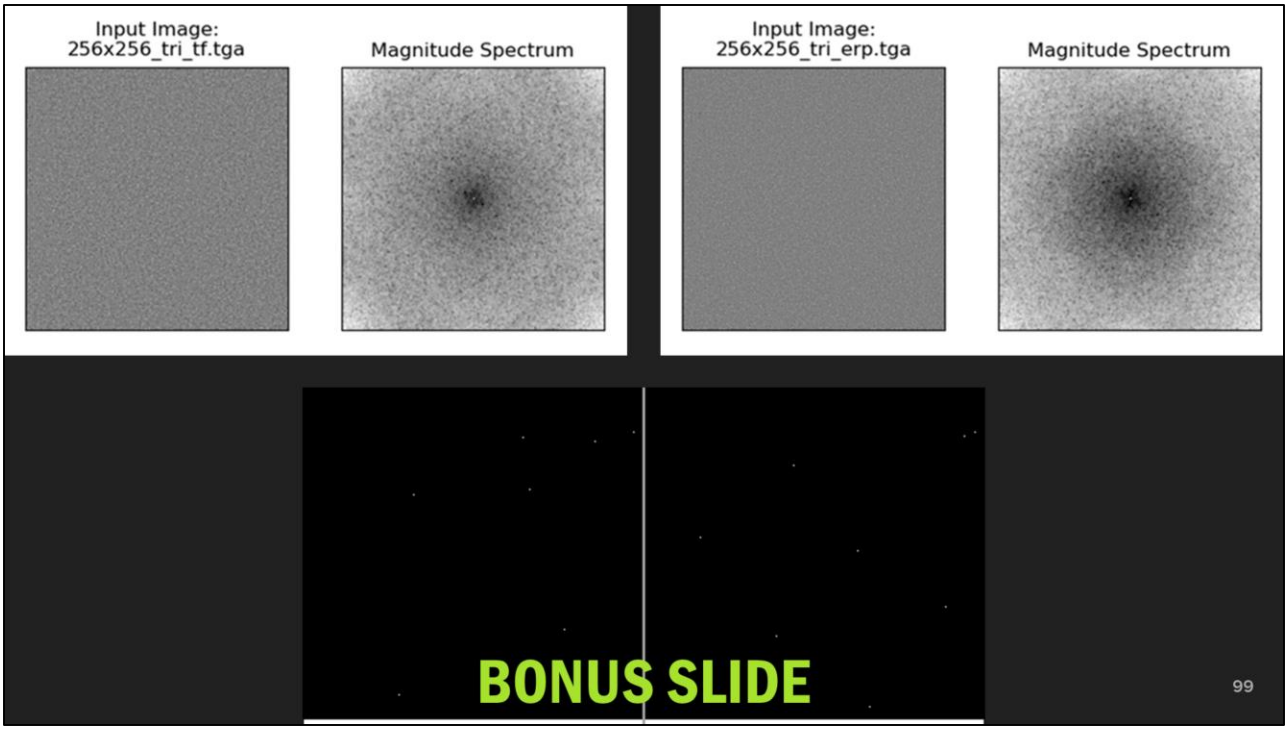
98

[best dithering, that we use to remove banding, requires a triangularly distributed blue noise]

https://www.shadertoy.com/view/4t2SDh

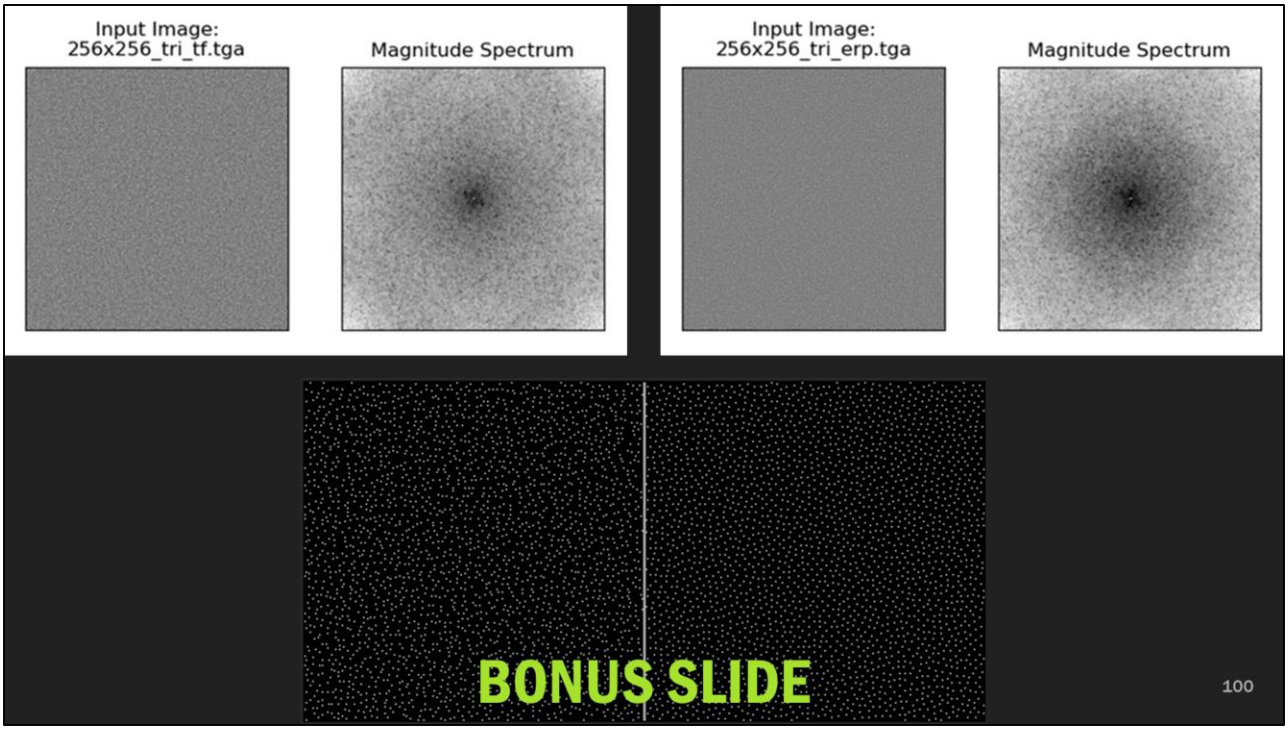
https://www.shadertoy.com/view/Mt2XW1

ERP is Erik Rodriges Pedersen from Playdead



…hard to get objective metrics for small changes like this

Goto tools are fourier spectrum as shown on top
thresholding as shown bottom



Looking at specific points around the middle... erp-version is more uniformly spread

Does not make a big difference for dithering - which is why it got through to begin with. Most of the image is still low discrepancy, just overlayed