From Phones to High End PCs: How to Scale

How to develop & scale your game technology across vastly different platforms.

The notes under the slides may or might not contain the information I actually managed to say during the talk ;)

/me

- Aras Pranckevičius
- Rendering dude at Unity
- @aras_p

I'm graphics architect and resident troublemaker at Unity Technologies. Also on Twitter.

I don’t know!

So the frank answer to the question – how to scale across vastly different platforms – is that I don’t know! At Unity, all I’ve seen is how we grew it from one weird platform (it was Mac only at the time) to a dozen or so platforms, while sharing a lot of code and technology. And I’ve seen what things we or our customers have to do to scale their games across several different platforms, or within one varying platform (like a PC).

So that’s what I’m going to talk about, but keep in mind that everything beyond this point might be false and/or stupid.

Overview

- Many platforms
- Design challenges
- Technical side
- Scaling the content

So the overview of the talk is:
The existence of & need for targeting multiple platforms.
Design challenges while doing that.
Technical side, mostly how to develop cross platform technology.
And finally, the part with some pictures, is how to scale actual game content once you have said cross platform technology.

Many platforms!

- More diverse platforms
  - Mobile, consoles, PCs, web
    - Someday... microwaves?
  - Often have to target >1 platform

Now, many platforms. This shouldn't be a surprise to anyone. It looks like the world is not moving towards a single platform; what happens is that there are more and more diverse platforms.

The platforms are wildly different, and big groups probably could be mobiles, consoles, PCs... perhaps web is a platform on it's own. There are many more, for example set top boxes, gambling machines, internet TVs and so on. If we'd believe all the "gamification" buzz, someday the microwaves and the toilet seats will have games on them.

It's very nice to exclusively target one platform, but quite often a developer can't afford that due to limited market size. Even within a single "platform group" like mobile, you probably want to target at least iOS and Android. Which is way more than a single hardware configuration!

This talk:

- Smartphones, tablets
- Desktops, consoles
- Web (Unity, Flash, HTML5/WebGL)

In this talk I'll only be touching several platforms, namely smartphones & tablets (of iOS and Android variety), desktops and consoles, and web.

By web, I mean something that could do 3D games, using Unity, upcoming Flash 11, or HTML5/WebGL. I'll skip on Adobe Director here, because at least from here it seems to be pretty much dead.

This talk:

- Won’t cover some others

There are various other platforms that I just won’t cover in this talk. I can’t cover everything; also, not everything is supported by Unity so I wouldn’t know what to actually say there.

Challenges

Now, challenges in targeting multiple platforms.

Game Design

- Input schemes
- Attention span / play lengths
- Player skill sets
- Monetization / pricing
- Distribution / marketing

One immediate challenge mostly lies in how to design a game for different platforms. A lot of stuff is different here! Some of the obvious differences could be:

Vastly different input schemes; touch vs. controller vs. mouse.
Typical game session lengths; on a console you can expect the player to spend 2 hours in one go; good luck with that on a phone.
Somewhat related; different platforms have different player skills. Something like mobile or web typically has much fewer players who know everything you're supposed to know in a modern shooter game. And vice versa, a console player might not be highly trained in estimating trajectories of bird launches.
And the list goes on; even the business side of different platforms is very different.

Game Design

- All very different between platforms!
- **BIG CHALLENGE**

And all this is a big, big challenge!

Why bother

- Which game is the same across different platforms?
- Need to adjust to the platform

It's such a big challenge in how to design a single game across vastly different platforms, that I don't even know if anyone should bother.

I mean, I haven't seen a single good game that is the same across mobile, consoles and web, for example. There are games that share the same universe or story; or share same art assets, but successful games have their design tailored to the platform.

However

- Can share skills the team has learned
- Can share tech. between several games

At Unity, what we very often see is that people pick a small set of target platforms for a single game. For example, “web” or “mobile”. They love that a lot of stuff they learned in Unity is easily transferable between platforms, and that they can easily reuse art assets and code between platforms, but very often a single game targets relatively small set of platforms.

So even if your current game is for one or two platforms only, you might want to make the next one for different platforms. And a lot of code, tools and workflows can actually be shared.

Technical Side

So let’s get to the technical side, how to develop multiplatform technology

Easy

- Did I say it already?
- Relatively easy part

It’s actually not very hard ;)

Codebase

- ~95% code shared
- Most support C/C++, asm
- Web trickier!
- WP7 .NET only, makes us ignore it

At Unity, absolute majority of our codebase is cross platform. This isn't hard since all (or actually, most) platforms we care about support C and C++ and some form of assembly code. So just write most of your code in cross platform way and you're done.

Now, web is trickier because there's no widespread way of writing native code to the web. However web is a large & important platform, so I'll talk about it in a second.

Some other platforms, like Windows Phone 7, support only .NET code. All this decision achieves is that it makes us ignore the platform. If any platform holders read this: seriously, native code access or GTFO. You are already starting with really, really strong competition in the field. How do you plan to compete with Apple when they do allow native code and you don't?!

Targeting Web

- Most flexible: your own plugin
- Also, most pain
- Installing it, support, backwards compatibility

So, targeting the web. Most flexible way is writing your own browser plugin. A plugin can do pretty much anything it wants; it's piece of native code running in the browser, with very little sandboxing done. So technically, saying "console quality graphics in the browser!" when it's done via a plugin is nothing fancy. It's not much different to a standalone game running on the same machine, except it happens to draw pixels inside the browser's window.

However, writing your own plugin is a nightmare. Both due to technical reasons - suddenly you have all these slightly different browsers to support, and getting your plugin to be widespread will be a huge challenge. Also, you will have to maintain backwards compatibility pretty much forever. Want to fix a bug and release a new version of your plugin? Guess what, some weird game might be there that depends on the bug being present. It's a lot of headache.

And on plugin penetration side, you'll always have "but Flash is more popular" or "it's not a standard". Actually, you will get both. We get over 3 million of new plugin installs each month and pretty good installation rates (meaning quite a lot of players that go to your game will end up already having or installing the plugin), but it's still very, very long call from Flash.

Targeting Web

- Cross compilation?
- C/C++ -> AS3/JS/CIL
- Alchemy, Emscripten, Mandreel, custom LLVM backend, ...
- Lower performance than native
- Multicore, SIMD hard or impossible

So another option of how to target the web could be cross compilation. It sounds totally nuts, but there are technologies that can take almost arbitrary C++ and spit out ActionScript or JavaScript or .NET on the other side. So with that you could target Flash or HTML5 or WP7 for example.

Any platform specific APIs, like D3D, obviously can't be translated into ActionScript. So you'd have to somehow call into Flash APIs from your cross compiled code. But it is certainly doable.

Of course this way the code would not run at the original speed. It could run maybe 2 to 10 times slower, which is not awesome but might be good enough.

Some things would currently be impossible, for example good multicore utilization or SIMD. Maybe someday Flash or HTML will have ways of using that.

Targeting Web

- Google Native Client
- Chrome only
- C/C++, GLES2.0, audio buffer, input, ...

A way of targeting web that I'd really like to be widespread is Google's Native Client technology. Basically it's a way, with special compiler and some runtime support, to prove that the code will not do anything that might be a security risk. Turns out it is possible to do this, with only a small cost in performance.

Native Client right now is Chrome only, and not enabled by default in the current stable version of Chrome.

But it has APIs to do OpenGL ES 2.0 rendering, mixing into an audio buffer, processing input events, loading files from the internet and so on. Pretty much everything that you need to make a game actually.

Most of code...

- Relatively high level
- Cross platform!

So, back to cross platform code. Majority of lines of code is relatively high level, cross platform code. Ta-da!

...plat specific code

- Rendering interface
- Audio buffer
- Input
- I/O
- SIMD Math
- ...

There are parts that are necessarily platform specific, like actual rendering, audio, input, I/O APIs and so on. However, these are quite small pieces of code, well maybe except rendering.

Similar: Render

- Shader based
  - Fixed function in old PCs, GLES1.1
  - GPU Compute: not yet, will come
- Shaders, geometry, textures, render targets, device state

But even when the APIs you need to call are different; the “design” of the platform specific systems is often quite similar.

For example, rendering. These days, if you ignore low end platforms and don’t want to push high end platforms to the limit, you can design a renderer around DX9/OpenGL ES 2.0 functionality. Which would be shaders, geometry buffers, texture data, render targets and device state.

Going for low end platforms you’d have to add fixed function vertex and pixel processing, and make a lot of fancy stuff optional. Going for high end platforms, you’d have to somehow incorporate GPU compute, tessellation, more flexible resource views and so on. So that would actually complicate cross platform renderer design.

Similar: Multicore

- Multicore on all platforms we care
- Except Web without plugins
- Still need 1 core path for mobile
- PS3... somewhat different to others!

Likewise, multicore CPUs. These days everything is multicore. Except, well, Web, unless you write your own plugin. You still need to maintain an efficient single core path for mobiles and the Wii.

Now of course PS3 is somewhat different, but as long as your build up your stuff in “job” style, it’s not totally different. If you go for other high end platforms, you might want to do some GPU compute stuff in a similar fashion, or maybe not. I don’t really know much there ;)

Similar: SIMD

- Good support on many platforms
  - NEON, VMX, SSE
- Except Web without plugins...
- Still need scalar for mobile

Things like SIMD are again quite similar between platforms. Mobiles have NEON which is awesome, consoles have VMX which is good, and PCs have SSE which, well, it does SIMD as well.

Like usual, Web without plugins is different. And you still need to have a “fallback” scalar implementation for mobile because...

Well because even some relatively new ARMv7 platforms do not have NEON support. And there are quite a lot of older ARMv6 platforms which do not really have any real SIMD support.

Similar: Audio

- Just final audio buffer
- Do mixing yourself
  - We use FMOD

All platforms you care about can provide an audio output buffer for you, where you can write final mixed sound data. So just do the mixing yourself and call it a day. We use FMOD which does support all platforms we care about, and has some platform optimized mixing routines.

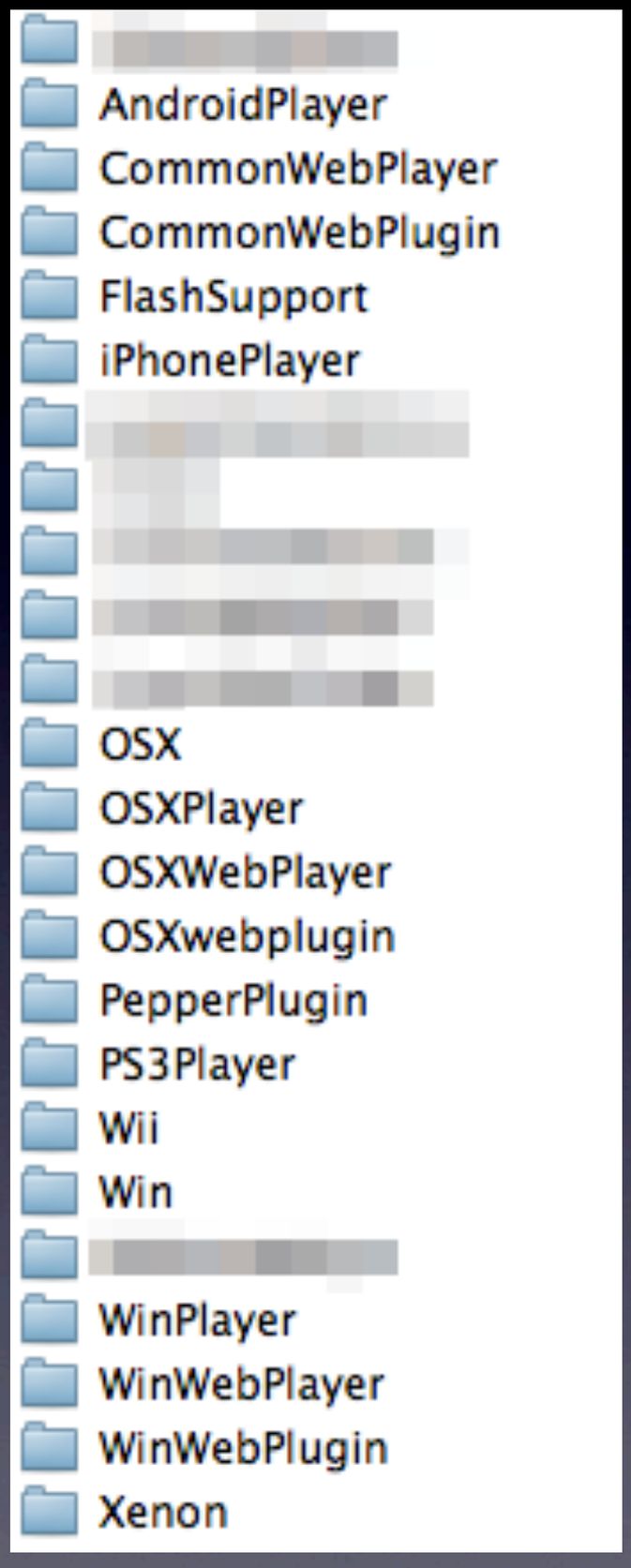
What we do

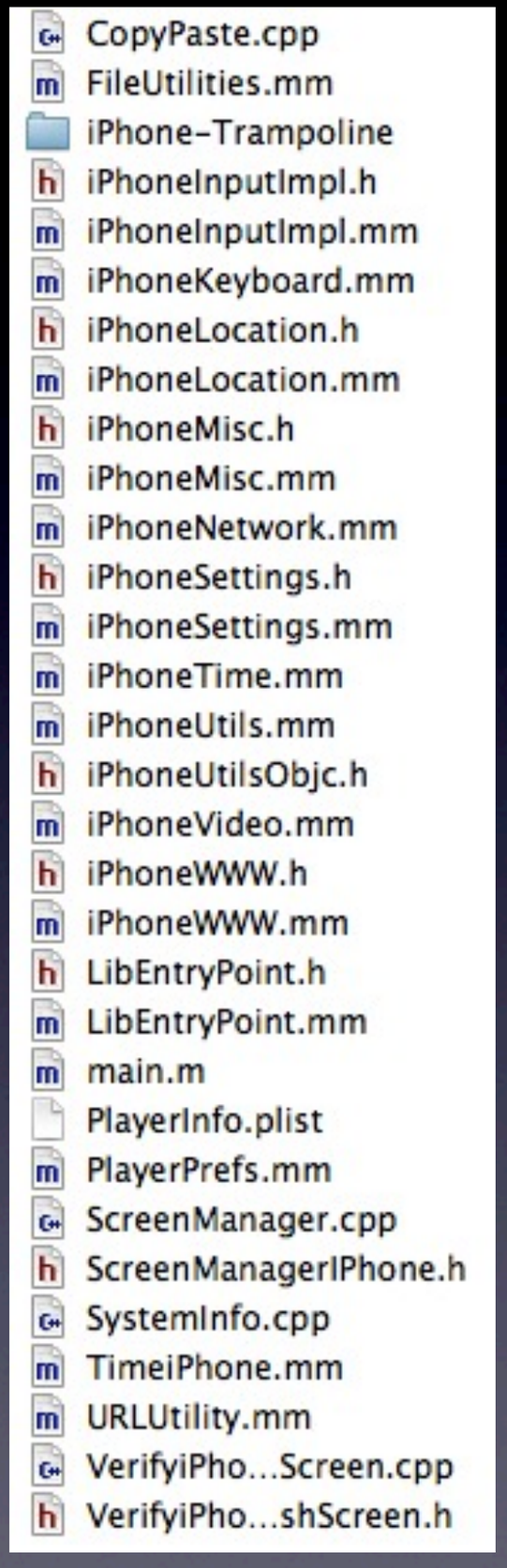
- No fancy abstractions
- Interface in common .h file
- Link to needed .cpp file
- Folder of stuff per platform

How we actually write platform specific parts of Unity:

We do not do fancy abstractions with platform implementation factories, virtual interfaces and whatnot. A simplest approach is to have the common interface in a header file – which can be just a bunch of global functions – and implement them for each platform. Link the needed implementation into final executable.

We throw all platform specific stuff into one folder per platform, except the low level graphics interface.



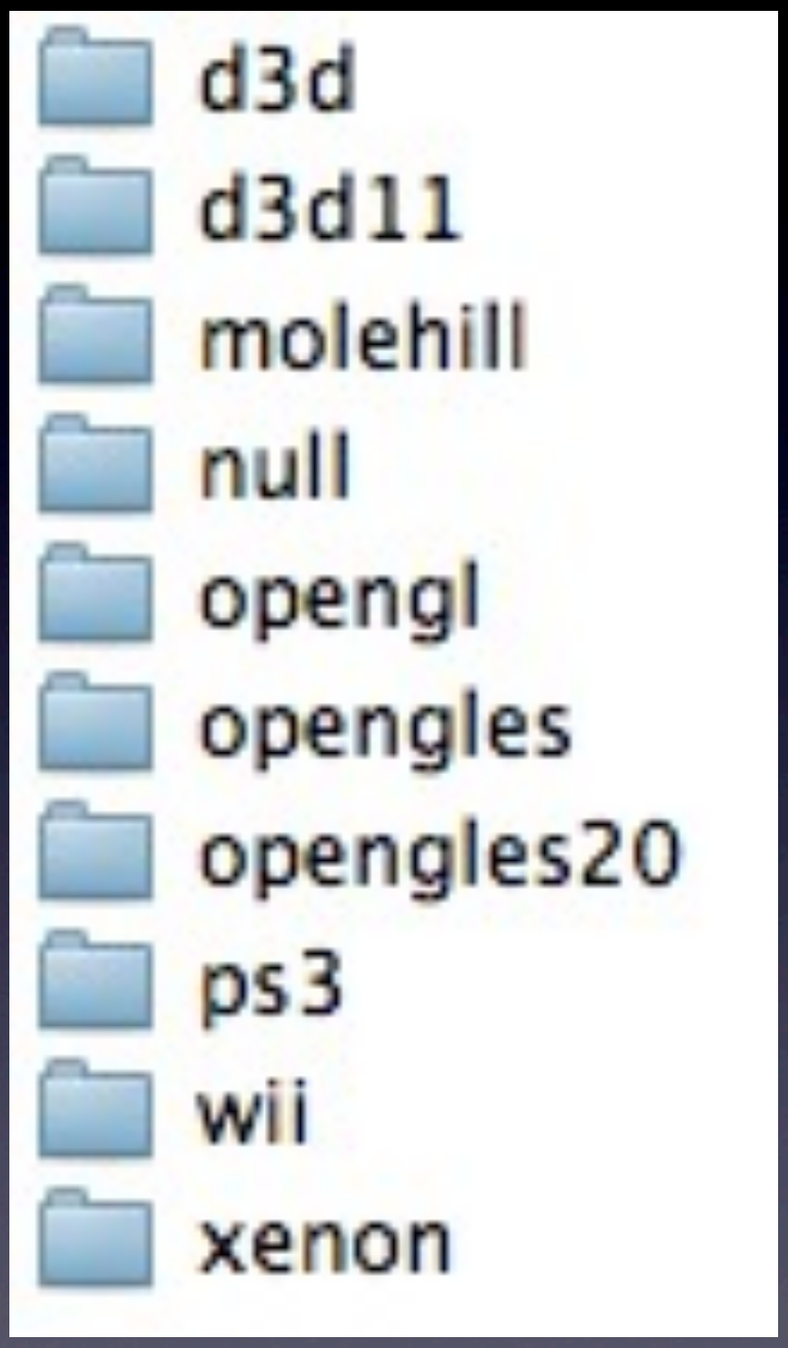


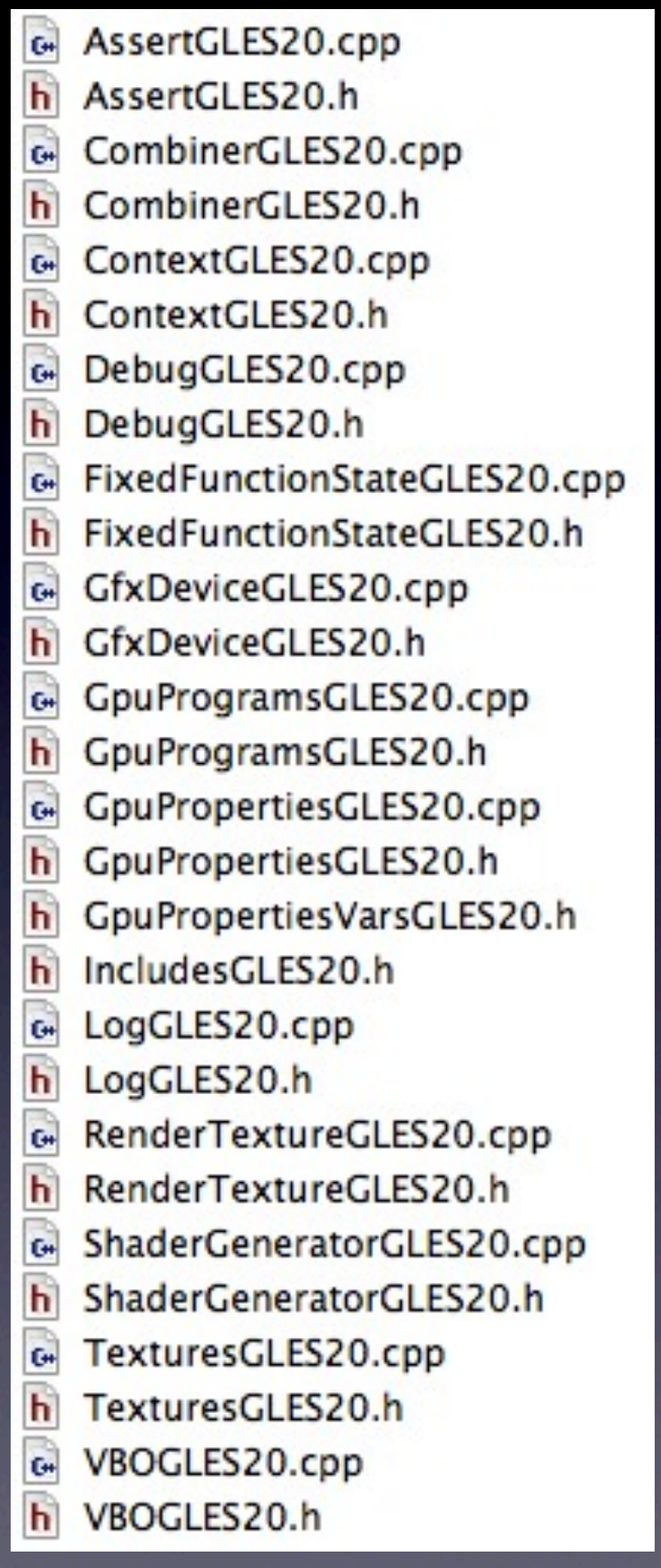
Like this. So you have a bunch of folders for each platform; and inside some folder you have bunch of files implementing one or another piece of functionality.

What we do

- Rendering (GfxDevice)
  - Largest platform specific piece
  - Some shared between platforms
    - E.g. OpenGL ES 2.0

The low level rendering part, we do it a bit differently. Mostly because it is quite large pieces of code, and they are more tied into the actual rendering API than to a specific platform. OpenGL ES 2.0 is pretty much the same in iOS, Android and Native Client for example, and we have one implementation for all of them.





But again, it’s just some folders and files... easy ;)

GfxDevice size

- Source code size:
- OpenGL ES 2.0: 240 KB
- OpenGL ES 1.1: 120 KB
- Xbox 360: 120 KB
- Direct3D 9: 300 KB (!)
  - Lots of code for pre-DX9 HW

However it's quite a lot of code. For example, here the current C++ source code size of our implementations. Surprisingly enough, D3D9 is the largest, but that's because it covers the widest hardware capability range, from essentially DX6 stuff to DX9 with fancy extensions. If you can not support pre-DX9 hardware, I'd advise you to do that, a lot of blood and tears will be saved.

Testing / Support

I said that developing for multiple platforms is easy, and it indeed is. However, testing and properly supporting them is not that easy.

More hassle

- For developers
- Different compilers / env
- Can accidentally break others
- Get everyone at least 2-3 platforms
  - Win, Mac, iOS/Android

For the developers, it's slightly more hassle. Different platforms have different compilers and environment, and you can accidentally break other builds. What we do is get all developers at least 2 platforms, for example everyone gets a PC and a Mac; while many developers also have iOS or Android device set up for development. In the ideal world everyone would also have one or two console devkits, but getting those is a hassle on it's own.

CI

- Continuous Integration helps a lot!
- Here, TeamCity with 40 build/test machines

What really helps of course is having a Continuous Integration setup. As soon as someone pushes a commit, a build farm notices it and builds everything it can with the change. So you get notified when you break a build on another platform quite soon, with full compiler log output.

A Continuous Integration farm also then runs various test suites on various platforms, which help to catch errors of a kind of “it compiled just fine, but crashed or did something wrong”.

At Unity we use TeamCity with something like 40 build/test machines. TeamCity is not perfect, but way better than everything else we tried, especially at complex build setups.

mac-3d-test-08	403	Enabled	Feature Branch - Kinect :: Test 3DAssetImportTest - Mac	Importing mesh #8: 'Assets/Meshes/FBX/Uncategorized/Pig_embed.fbx'	overtime: 45m:30s	Stop
mac-gen-vm-01	436	Enabled	Feature Branch - MemoryManager :: Build AndroidPlayer	Building Android player	13 Jul 11 15:53 (10m:14s)	Stop
mac-gen-vm-02	453	Enabled	Trunk :: Build MacDevelopmentWebPlayerInstallerForRecordPlaybackSystem	Building target: MacWebPlayer	2m:45s left	Stop
mac-gen-vm-03	475	Enabled	PhysX :: Android	Running	9m:59s left	Stop
mac-gen-vm-04	470	Enabled	PhysX :: OS X (universal)	Running	overtime: 57s	Stop
mac-gfx-test-14	398	Enabled				Idle
mac-gfx-test-15	389	Enabled				Idle
mac-ios-vm-05	467	Enabled	Topic Branch - Wi3x :: Test IntegrationTests iOS	Running	overtime: 4m:23s	Stop
mac-ppc-09	222	Enabled				Idle
mac-ppc-10	247	Enabled				Idle
mac-test-vm-11	431	Enabled	Trunk :: Test IntegrationTests FullSuite - Mac	Running	53m:24s left	Stop
mac-test-vm-12	416	Enabled				Idle
mac-test-vm-13	418	Enabled				Idle
mac-test-vm-16	440	Enabled	Topic Branch - Mono :: Test FunctionalTestsRunningInEditor - Mac	Running	13 Jul 11 15:55 (8m:48s)	Stop
mac-test-vm-17	446	Enabled	Unity 3.4 :: Test IntegrationTests FullSuite - Mac	Running	44m:58s left	Stop
mac-test-vm-18	440	Enabled				Idle
win-3d-test-07	522	~~Disabled~~				Idle
win-3d-test-08	591	Enabled	Unity 3.4 :: Test 3DAssetImportTest - Windows	Tests failed: 25, passed: 148	overtime: 5m:51s	Stop
win-console-vilnius-12	484	Enabled	Feature Branch - Wii (Stable) :: Test GraphicsFunctionalTests - Xbox 360	Waiting for process to finish	overtime: 2m:02s	Stop
win-gen-vm-01	446	Enabled	Topic Branch - Core :: Build WindowsEditor	Building WindowsEditor	13m:44s left	Stop
win-gen-vm-02	452	Enabled	Topic Branch - Core :: Build WindowsDevelopmentWebPlayer	Running	20m left	Stop
win-gen-vm-03	444	Enabled	Topic Branch - Core :: Build GLESEmu	Running	7m:22s left	Stop
win-gfx-test-12	586	Enabled	Topic Branch - Mono :: Test GraphicsFunctionalTests GLESEmu 2.0 - Windows	Running	13 Jul 11 15:55 (8m:43s)	Stop
win-gfx-test-13	597	Enabled	Topic Branch - Online Services :: Test GraphicsFunctionalTests GLESEmu 1.1 - Windows	Building player	8m:28s left	Stop
win-test-vm-17	399	Enabled	Feature Branch - Wii (Stable) :: Test IntegrationTests FullSuite - Windows	Running	1h:28m left	Stop
win-test-vm-18	402	Enabled	Topic Branch - Mono :: Test EditorFunctionalTests - Windows	Running	13 Jul 11 15:55 (8m:28s)	Stop
win-test-vm-19	399	Enabled	Topic Branch - Mono :: Test FunctionalTestsRunningInEditor - Windows	Running	13 Jul 11 15:56 (7m:38s)	Stop

Here's a screenshot of which machines are building what on TeamCity, with everything being too small to actually see.

Device Zoo

- Consoles are easy!
- PCs more of a problem...
- Mobile quickly becoming the worst

Now, testing the different platforms is quite hard. Consoles are easiest in comparison, because it's pretty much that any Xbox 360 is the same as any other, more or less. PCs, well not so much, they are very different. And mobile platforms are quickly becoming the worst for testing, with new funky kinds of (Android) devices appearing each week.

A small zoo

So for PCs, you have a small test zoo with different GPUs, installed OS versions and whatnot. It's easy to setup multiple OSes on a single machine, and relatively easy to swap different discrete GPUs. To test integrated GPUs you will pretty much have to buy a machine per GPU.

A real zoo!

Small part of Unity's zoo

These two photos from Martin Shultz of Decane

On phones and tablets, you will pretty much have to buy each hardware configuration you care about, multiplied by the number of OS versions you care about.

Some of those devices are more painful to automatically test on, for example on iOS you can only transfer test results via wireless connection. If there's a wireless glitch on your test farm, you'll get failing builds.

The Scaling Part

Now finally, onto the “how to actually scale” part

Vast differences!

- Computing power

Most obviously, there are huge differences in computing power of different devices. Even within same platform, a lot of variation.

PCs:

- 1 to dozens CPUS
- 128 MB to 16 GB RAM
- 1 to 200 GB/s VRAM BW
- 0.3 to 30 Gpix/s pixel fill
- **100X** difference!

For example, on PCs. People have wildly different configurations.

Source: Unity's hardware stats http://unity3d.com/webplayer/hardware-stats

So that's 100 times performance difference within a single platform. It's really hard, or probably impossible, to cover all that with a single game, while providing "ok" experience.

Majority, however:

- 1 to 4 CPUs
- 1 to 4 GB RAM
- 3 to 12 GB/s VRAM BW
- 0.5 to 6 Gpix/s pixel fill
- **10X** difference

However, if we ignore outliers of the usual distribution curve, say lowest 5% and highest 5% of the market, the differences become much smaller.

Something like 10 times the performance difference. Which is still a lot, but not “impossible” to cover.

Mobile:

- Some problematic
- High res screens, weak GPU
  - iPhone 4, iPad 1 etc.
- Too many pixels to paint!
- Same problem on low end PC GPUs

Mobile platforms have similar variation these days. One particularly weak area in many of them is high resolution screens coupled with a “meh” GPU. It’s often just fine for 2D games, but gets really challenging once you go 3D; any actual overdraw, transparencies or more complex shaders is death.

Scaling 10X

- Not very hard
- Look / run okay on baseline
- Up: add optional stuff
- Down: remove non critical stuff

So, scaling to cover 10 times performance variation. To me, it seems like a sensible approach is deciding on a baseline configuration, and making sure the game looks and plays well on that.

And then you add optional stuff that's essentially more eye candy to cover more powerful platforms -- which is scaling up from the baseline.

If you need it, you can also try targeting lower than the baseline, by removing some of non critical stuff.

Scaling up

- Mostly easier IMHO
- Won't push high-end to the limit
- For actual games, not future-fancy tech demos!
- Sad: uses all extra power for eye candy

Scaling up is somewhat easier I think, and you'll end up with a game that looks & feels good on all configurations you care about.

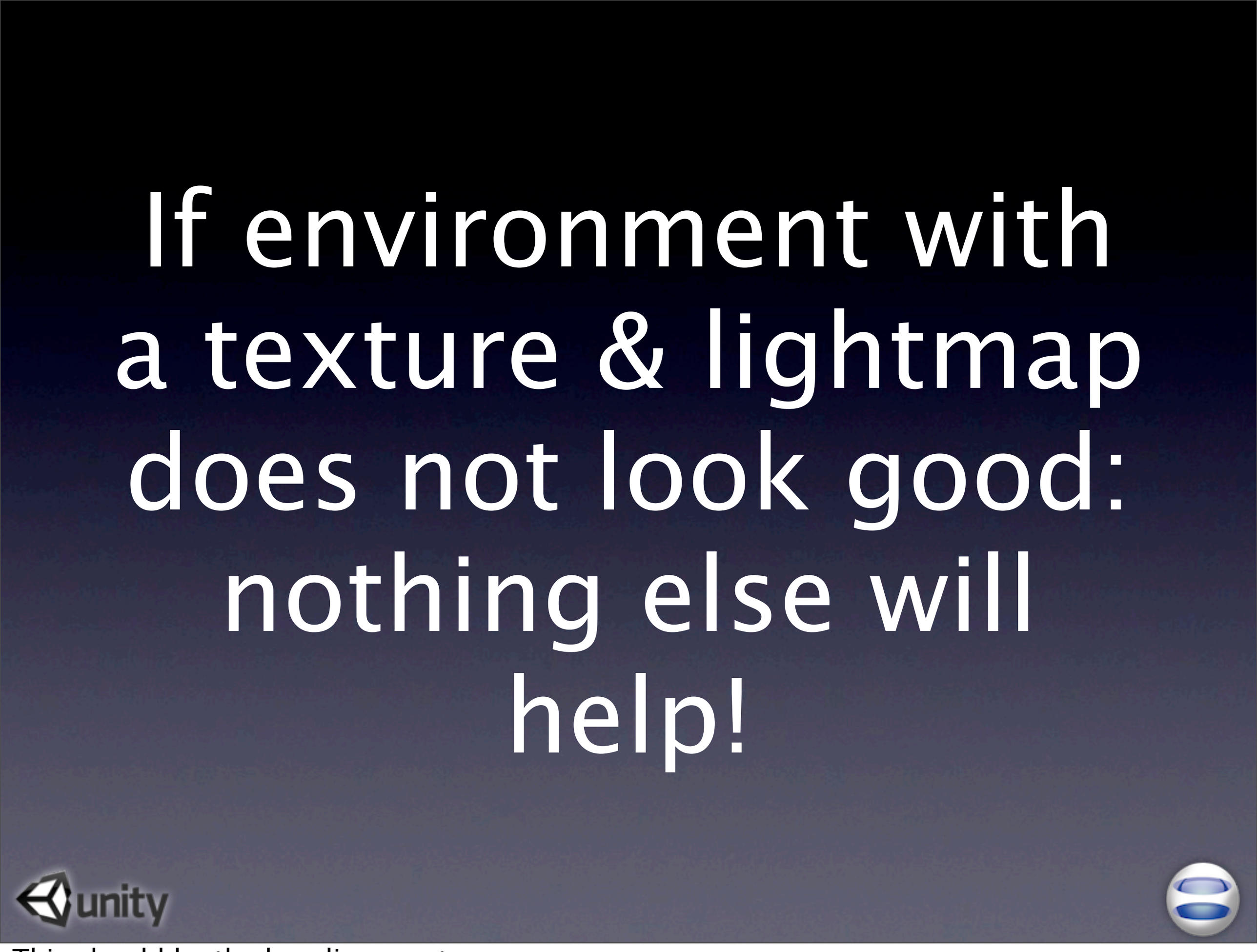
It will not, however, push the high end configurations to the limit, unless you invest extraordinary amount of effort. Which is fine for actual games, but not for some tech-demos that would be supposed to show how the future might look on hypothetical devices.

It's a bit sad to use all this extra power on eye-candy features only. You could try to somehow make a game better with some extra computing power, but this again goes into game design challenges. Having more enemies, or more complex gameplay physics, or smarter AI actually changes the game, so you'd end up with slightly different games depending on the power available. Which may or may not be fine.

Example

- Will be using Shadowgun by Madfinger Games
- Planned for Sept 2011
- Smartphones & tablets

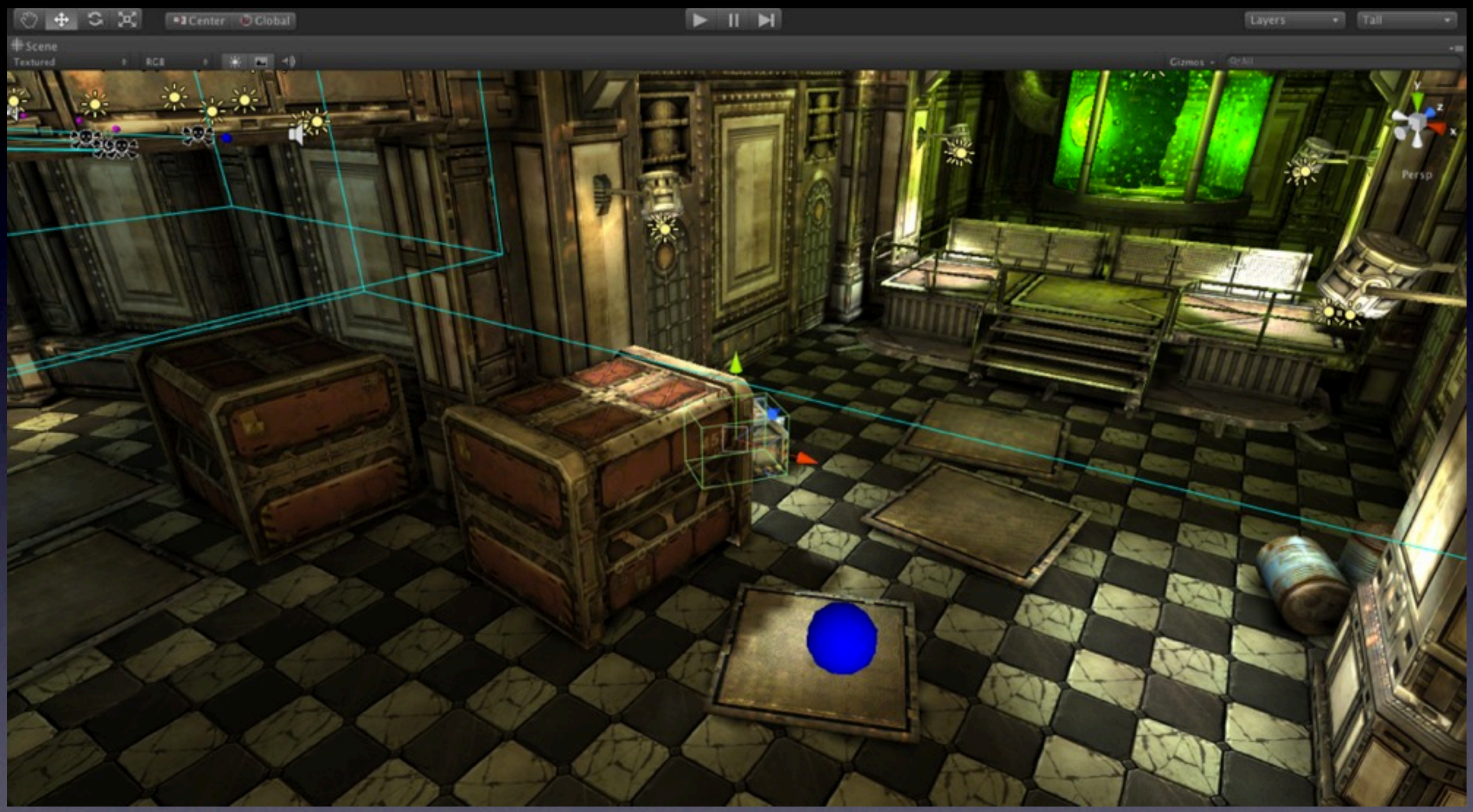
Now, a quick example of one Unity game; Shadowgun by Madfinger Games. The game could be summarized as Gears of War type of shooter for mobiles. It's still in development, planned to launch quite soon. Madfinger got some advice & bits of technology from us, but a lot of stuff they came up with themselves.



This should be the baseline mantra.

If environment with just a texture & a lightmap does not look good, then nothing else will help.

Which means, a game has to look good even with super simple technology. You can always add more fancy stuff on top of this base, but the base has to already be good. If the underlying artwork is crap; or lights are not placed in a meaningful way; or level design is boring, then no amount of shaders, post processing or anti-aliasing will make it shine.



The environment here is pretty much a texture multiplied by a lightmap.

Baseline

- 30 FPS on iPad 1 / Tegra 2
- At 1024x768, iPad 1 one of weaker configs in pixel processing on mobile
- Similar for Tegra 2, Android tablets generally higher res

So, the baseline of Shadowgun is running 30 FPS on iPad1 and Tegra2.

Now, iPad1 is really weak on pixel processing power, so it’s not exactly easy to achieve 30 with a complex looking 3D game.

Same applies to Tegra 2, especially since Android tablets tend to have even higher resolutions.

Can’t afford much

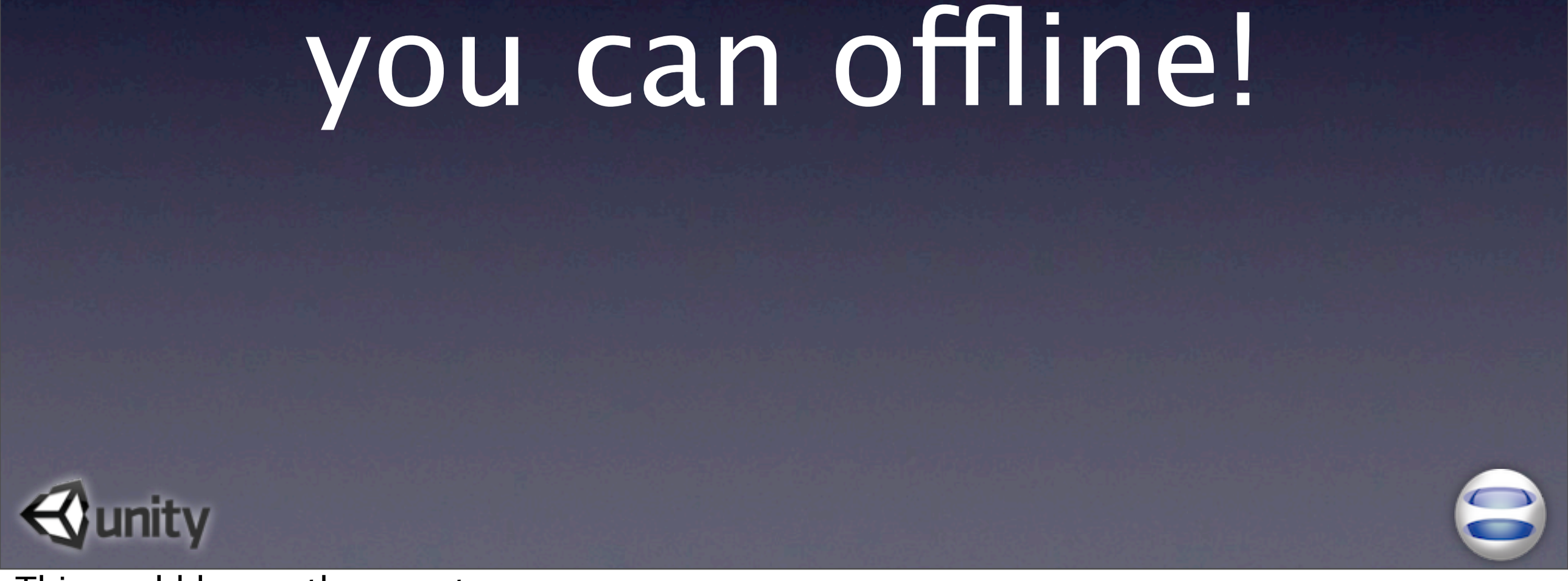
- GPU limited on pixels
- Make tech really simple
- Use good artwork instead!
- Bake all you can

So the recipe how to achieve the baseline is to make everything as simple as possible. It was discovered from the start that the pixels are going to be the bottleneck, so make them be simple.

Use good artwork instead!

And bake everything you can offline, so the pixels appear to be complex when in fact they aren’t.

Move all fancy stuff you can offline!



This could be another mantra:

Move everything you can into offline tools.

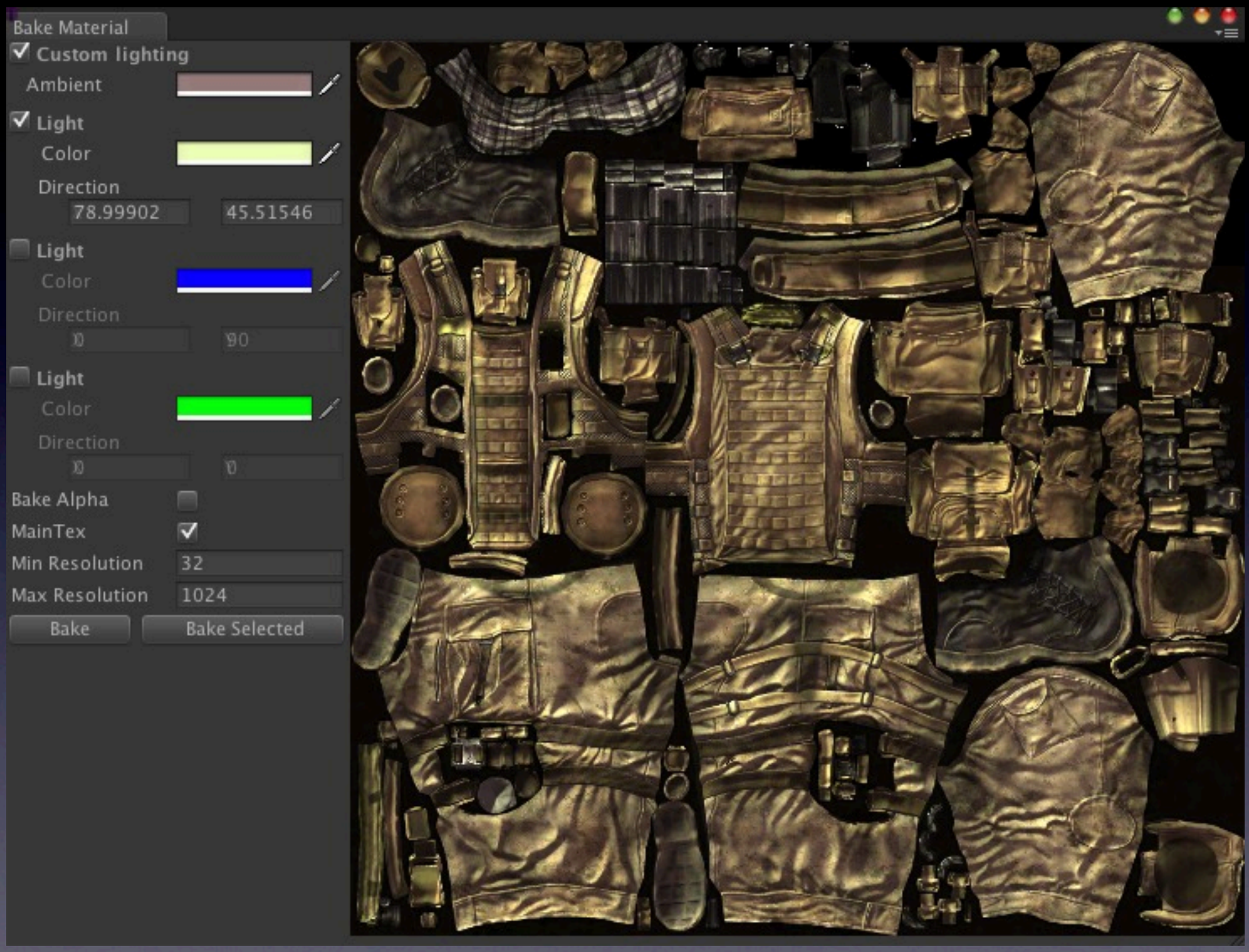
Precompute, bake, compute platform optimized data, whatever. This is somewhat more annoying for you as a game developer, but all the time you have spent baking stuff is something that does not have to be computed multiple times per second by millions of players.

Environment

- Texture, lightmap
- Complex materials baked into a texture
- Fake per vertex specular

Shadowgun's environment objects are basically a texture and a lightmap. They are authored as more complex objects, with multiple texture layers, normal maps and whatnot, but all the material complexity is "baked" into a single texture. This is not correct since some light directions have to be approximated at baking time, but oh well.

Specular is faked, it always comes from a direction somewhere close to the camera. Computed per vertex.



You just get a complex looking texture in the end, and 99% of the players won't be able to tell it's not actually bumpmapped at runtime.

This material baking tool for Unity is released publicly by the way: http://www.unifycommunity.com/wiki/index.php?title=Bake_Material_to_Texture

Characters

- Custom BRDF via (N.L*0.5+0.5,N.H) lookup texture
  - iPad1, faster than single highp pow()!
- Normal maps
- Per vertex SH lighting
  - Light probes, muzzle flash, explosions

Characters in Shadowgun actually use normal maps. Shading is done with a trick from 2001 era, which is encoding lighting response into a small lookup texture. Turns out doing this is much faster than even a standard Blinn-Phong, and allows more complex BRDFs to be put into the lookup texture.

There are spherical harmonics light probes baked around the level, and lighting from them is applied per vertex. Additional realtime lights, like muzzle flashes or explosions, are also fed into the spherical harmonics data, so it’s no extra cost on the GPU at all.

So again, some nice editor tools for artists to tweak the character lighting and bake the final texture. At runtime, it's just one dependent texture read in the shader.

Fx

- No postprocessing, no fog
- Fog planes, light shaft, glow card geometry
- Scale down (move along normals) in vertex shader when close: dramatic fill savings!

Effects wise, everything is really simple. There's no image postprocessing going on, at all. No fog either.

Where these things are needed, they are emulated by manually placed fog planes, light shafts, glow cards and so on. More work for artists, but less work for underpowered GPUs. Well actually, their artists like this because it enables them to have more control than for example postprocessing.

Again, one simple trick that dramatically saves fillrate: when you get close to a fog plane or a glow card, fade it out and scale it down in the vertex shader, so that it does not cover majority of the screen. Shadowgun moves vertices along the normal; artists edit normals so this scaling down happens exactly how they want it.

Fx

- Liquids, caustics: two distorting textures

Places that have effects like caustics and liquids: just some extra shader that combines two moving textures. All is fake!

Fx

- Vent shafts/shadows: rotating bright/dark geometry

Vent shafts and shadows: that's just some bright and dark geometry with alpha blending. Rotate it, and the press goes "spectacular lighting effects" ;)

Scaling up target

- iPad 2, still 1024x768
- 4-8X faster GPU
- 2x CPU cores
- 2x more RAM
- One of best configs in current mobile

Ok, so all that runs 30 FPS on iPad1. The current target to scale up to is iPad2.

Same resolution, but massively more powerful GPU. Not ten times, but hey ;)

Scaling up target

- Tegra 3 (Kal-El)
- Not out yet; can't publicly talk specs ;)
- But expect it to be strong!

Another scaling up target, upcoming Tegra 3 chip. Likewise, a good step up from Tegra 2.

Scaling up

- Target 60 FPS on iPad2/Tegra3
- Anisotropic filtering
- More fx (particles, cloth, shadows, parallax)
- MSAA

Currently scaling up in Shadowgun adds easy eye candy.

Run at 60 FPS, use anisotropic filtering on a lot of surfaces, use more eye candy effects (particles, cloth, shadows, parallax shaders), maybe some anti-aliasing.

Scaling up

- Quickly becomes GPU bound again at 60 FPS
- Open question what to do with spare CPUs?
  - Not even fully using 1 core of iPad2!

That is enough to make it GPU bound again.

However, most of that made it more expensive on the GPU, but almost same cost on CPU. How to scale up there?

I really don’t know. It could try having smarter AI or better physics or whatnot, but all that again is not pure eye candy as it changes the game.

Tiny Wish Here

- Offloading GPU work back to CPU not easy on mobiles...
- People do it all the time on PS3...
- Want to do the same on mobile!

A little step aside, there's a big need for be able to juggle tasks around between CPU and GPU depending on which one is overloaded in the game. If we want to keep the game the same as the baseline, there's not much stuff we can do with extra CPU power on a more powerful configuration!

On consoles that's a bit easier because of lower level access to the hardware, but on mobiles this is hard. Someone should solve it one way or another.

Scaling up to PC?

- Shadowgun not targeted at PCs...
- ...but let's do some guesswork

Now, what are the ways Shadowgun could be scaled up even more, for example to PC performance levels?

Note that this game is not targeting PCs, so I'm just doing wild guesses here.

Up to PC, GPU

- Higher resolution
- Postprocessing!
  - Color correction, DOF, motion blur, SSAO, MLAA
- Blobs -> actual shadowmaps
- Aniso + trilinear filtering
- More particles / fx

On the GPU it's relatively easy to burn performance on eye candy.

PCs tend to have high resolutions, boom, 2 to 5 times more pixel work.

Image postprocessing can burn a lot of power. And I mean, really a lot!

Fake blob shadows could be replaced with actual shadowmaps.

And so on.

Up to PC, GPU

- Could go linear space HDR lighting + tonemapping
- Would probably need texture/ lighting tweaks to be good

I guess it could also do proper linear space lighting at high dynamic range, followed by tonemapping.

However, this might be not trivial to apply, because changing to linear space does change a look of the game a lot. Level lighting could need retweaking, perhaps even some textures would have to be repainted.

So not trivial to do.

Up to PC, CPU

- More light probes
- More debris
- Effect physics (cloth, water, ...)
- Software occlusion culling
- ...other changes would need gameplay tweaks?

On the CPU side there are several eye candy things it could do:

More light probes, more debris, more effect only physics like cloth and water interactions.

Again, putting more meaningful load on the CPU is hard without affecting the gameplay.

Scaling down?

- Kind of last resort
- When baseline “looks/runs good” still too expensive

So that’s with scaling up. As a kind of last resort measure, you could also try scaling the game down from your baseline.

Scaling down

- Drop texture mip levels
- Drop mesh LODs
- Drop particles/fx/shadows/...
- Drop normal maps
- ...

Making everything blurry, making models have too visible edges, making explosions look bad and so on. Basically making sure it runs acceptably fast, at expense of looking worse than the artists have intended.

In upcoming Unity's demo project, we actually drop from a 3D top down shooter to a text mode adventure game if we detect really, really crappy hardware ;)

So, in Shadowgun...

- No fancy tech at runtime!
  - Basic render, particles, light probes, skinning, navmesh, physics, audio, scripts
- Scale up with eye candy on GPU

So, in the Shadowgun case, there's not a lot of technology at runtime. There are basics you can expect to be in any 3D game, and then a lot of editor side tools and workflows that enable it to cheat and bake. Scaling up is done by adding more eye candy that primarily taxes the GPU.

Summary

- Tech is easy
- Game design much harder
  - Maybe you **don't** need to make same game on all platforms?

Technology is relatively easy. A lot of code can be shared, and scaling to something like 10 times performance difference is possible by just adding some eye candy.

The game design parts are much harder. Gameplay has to be tailored to the platform; so if the platforms are very different you might end up doing several slightly different games. Which is fine, you can still share a lot of underlying tech of course.

Thanks!

- Slides up soon
- @aras_p
- blogs.unity3d.com
- aras-p.info

So that's it!

Slides will be soon, watch my twitter stream or Unity/my blogs.