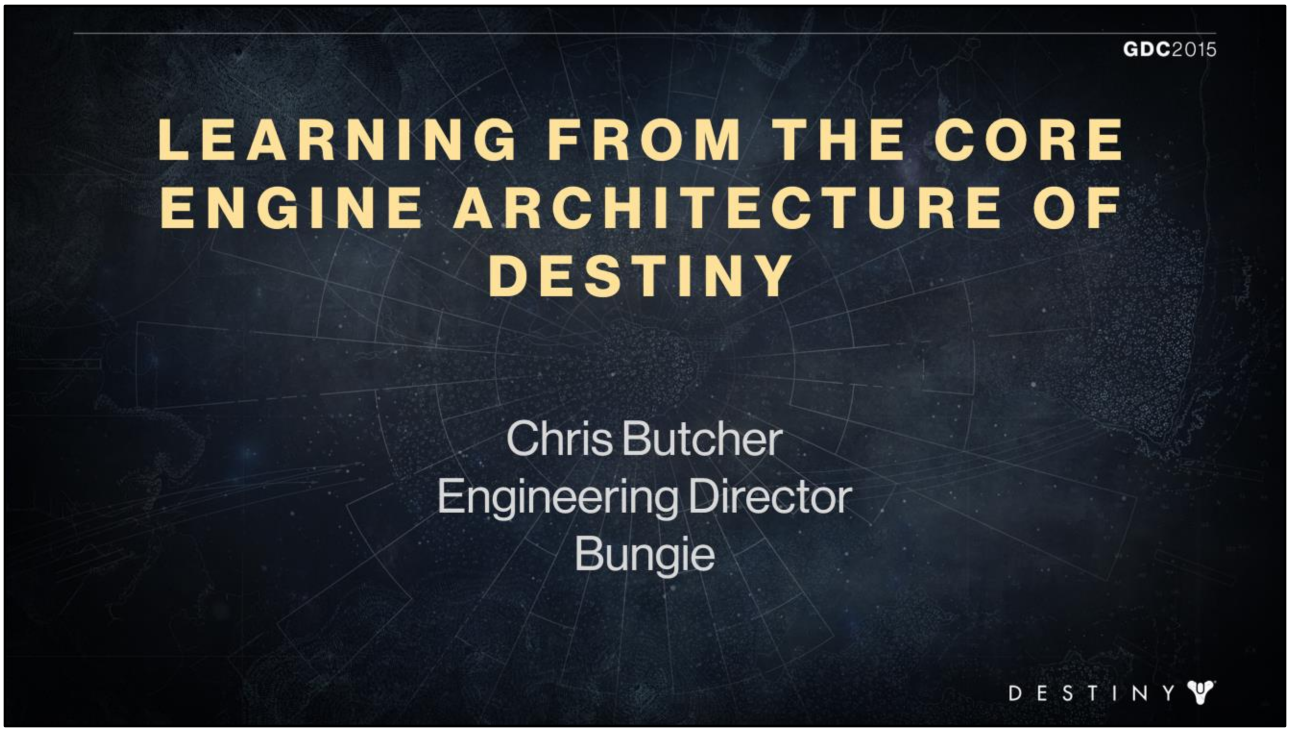


GDC2015

LEARNING FROM THE CORE ENGINE ARCHITECTURE OF DESTINY

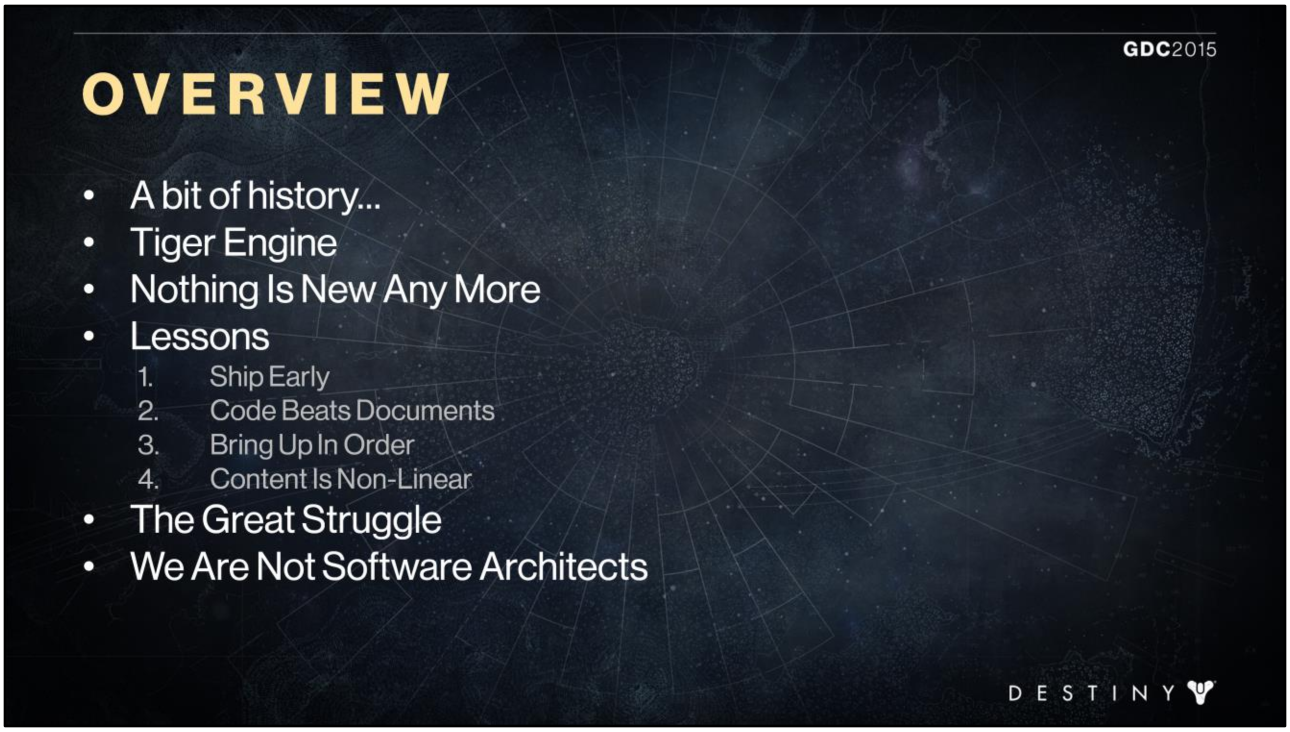
Chris Butcher
Engineering Director
Bungie

DESTINY

We'll get started in a few minutes. But before then, I'd like to ask everyone to silence their cell phones please. These slides will be available online, there's a download link at the end of the presentation, so don't bother taking pictures of every slide. And there will be time for questions at the end.

Hi everyone, and thanks for coming today.

My name's Chris Butcher, and I'm one of the engineering directors at Bungie.



GDC2015

OVERVIEW

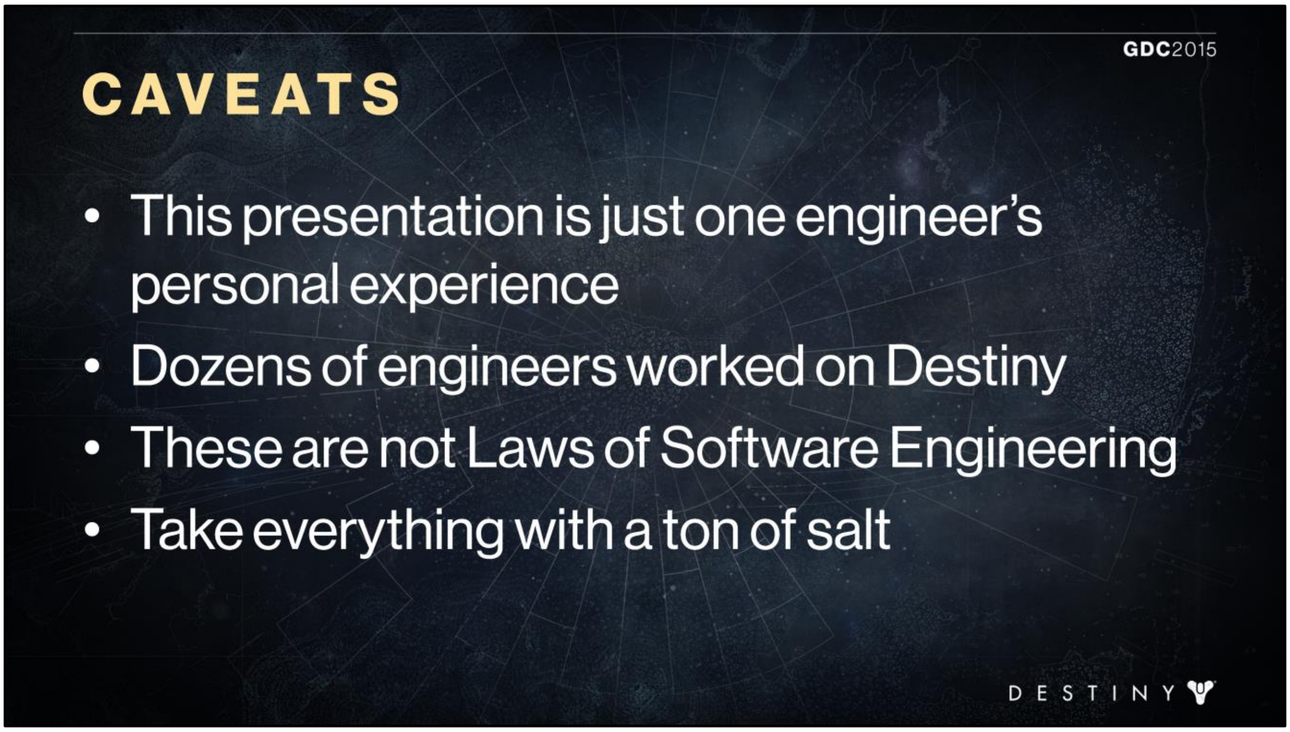
- A bit of history…
- Tiger Engine
- Nothing Is New Any More
- Lessons
  1. Ship Early
  2. Code Beats Documents
  3. Bring Up In Order
  4. Content Is Non-Linear
- The Great Struggle
- We Are Not Software Architects

DESTINY

So, basically this format of this presentation is: “what I wish I had known before we got started”.

We’re going to start with a bit of history, then take a tour through the development of the Tiger Engine that powers Destiny.

We’ll cover some lessons that we learned along the way, and then wrap up with a discussion of the great struggle against software complexity, and why I believe none of us are truly Software Architects any more.



CAVEATS

GDC2015

- This presentation is just one engineer's personal experience
- Dozens of engineers worked on Destiny
- These are not Laws of Software Engineering
- Take everything with a ton of salt

DESTINY

Up front I'd like to set some expectations if that's ok.

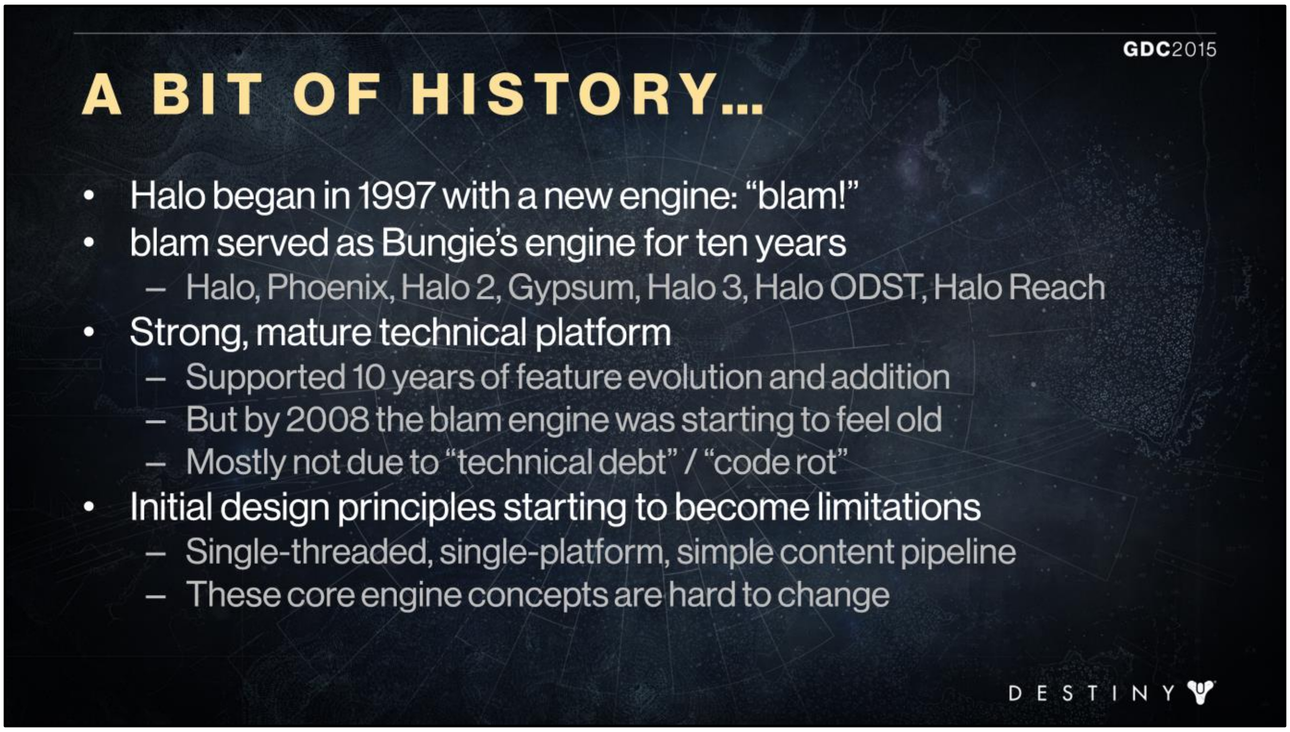
I've worked as an engineer at Bungie for fifteen years. I've spent a lot of time talking to other people about game engines, and I've evaluated lots of game engines as well. **But, the Bungie engine is the only one I've actually used professionally.** That makes it hard for me to have an outside perspective sometimes, so keep that in mind.

** Also, I am only one of many engineers that worked on Destiny. So I don't want to take too much credit for any of this. Except anything that's overly complicated. That's probably my fault.

** Lastly, this presentation is just some subjective thoughts about the craft of software architecture. Any time I state something is a FACT, I just mean it's my best guess at a principle, not that I've discovered some immutable Law of Software Engineering.

** So basically … you should take this whole presentation with a metric ton of salt.

Here’s about a million tons of salt.



GDC2015

A BIT OF HISTORY…

- Halo began in 1997 with a new engine: “blam!”
- blam served as Bungie’s engine for ten years
  - Halo, Phoenix, Halo 2, Gypsum, Halo 3, Halo ODST, Halo Reach
- Strong, mature technical platform
  - Supported 10 years of feature evolution and addition
  - But by 2008 the blam engine was starting to feel old
  - Mostly not due to “technical debt” / “code rot”
- Initial design principles starting to become limitations
  - Single-threaded, single-platform, simple content pipeline
  - These core engine concepts are hard to change

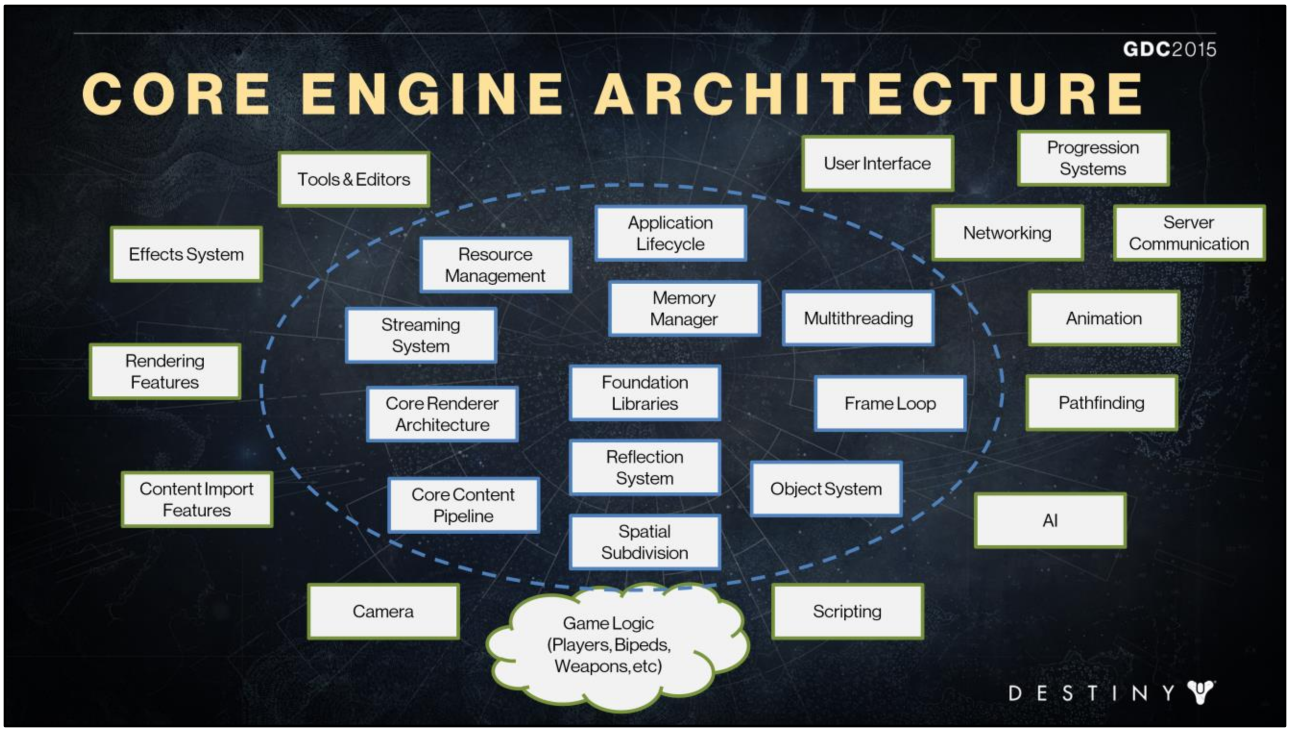
DESTINY

Let’s start with a bit of history. In 1997 Bungie started working on Halo, and a new engine to power it, code named “blam”.

We used this engine for all of the Halo games, and also Phoenix and Gypsum which were two game explorations that were built using the blam engine but didn’t ever see the light of day.

** The blam engine was a strong and mature platform on which we built 10 years of features. But by 2008 it was starting to feel old. It wasn’t really due to an accumulation of bad code or “technical debt”.

** Instead, what was happening was that a decade earlier on the first Halo game we’d set the initial design principles of the engine, which worked really well back them, but they didn’t apply so well to the modern world. A single main thread, a single platform - these were key simplifying assumptions of the engine, but they were starting to become restrictive – and they were very hard to change. We could see it was going to be increasingly difficult to compete in the future.



The “core engine” is my term for the framework on which engine features are built. It’s the stuff inside the line.

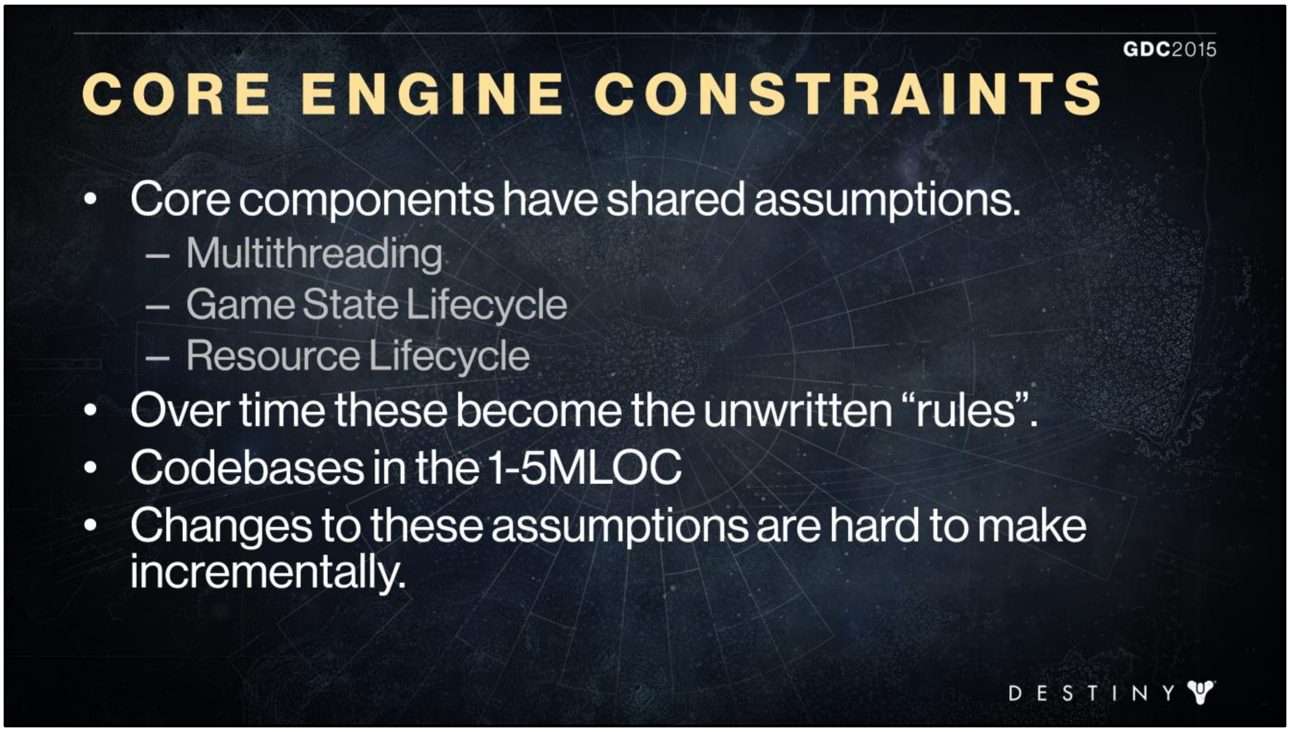
The core engine tends to be tightly coupled, and its components call directly into one another.

- This makes it hard to abstract them away from each other using interfaces. Not because you can’t put an interface on the code, but rather because there are shared assumptions; about data formats, object lifetime, state management and the performance characteristics that are hidden behind the interface.
- For example the resource manager and the environment streaming system have to be designed together because of implicit rules about how resources load and instantiate.

Outside the core, I call the next level the “feature layer” of an engine. It’s what’s outside the line.

- In general, these feature components **tend** to be more loosely coupled.
- For example, camera systems, pathfinding, networking and rendering features are almost completely independent.
- These feature components tend to talk to each other through interfaces for abstraction.
- They tend to use data formats, object lifetimes, interface systems, message protocols… code constructs that are defined by the core components.

There really isn't a hard distinction about what's in the core layer and what isn't. This is a qualitative distinction, it's a spectrum.



GDC2015

CORE ENGINE CONSTRAINTS

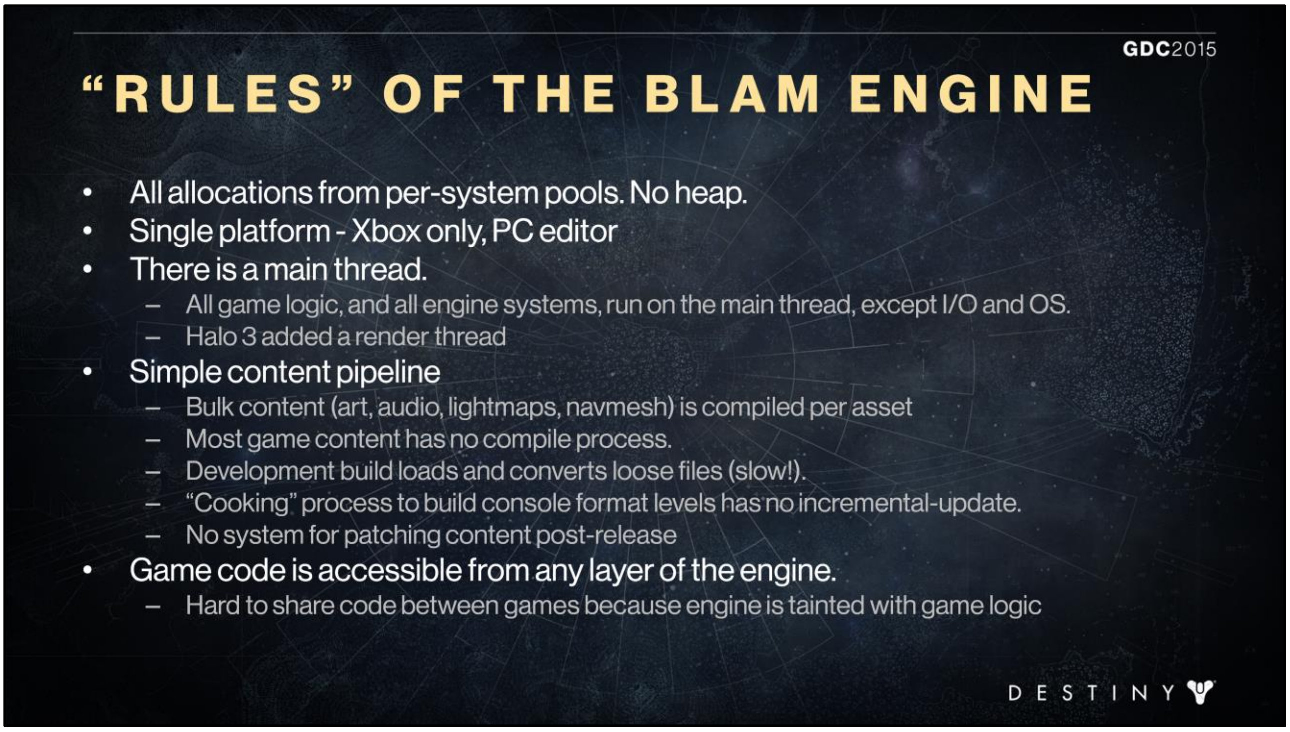
- Core components have shared assumptions.
  - Multithreading
  - Game State Lifecycle
  - Resource Lifecycle
- Over time these become the unwritten “rules”.
- Codebases in the 1-5MLOC
- Changes to these assumptions are hard to make incrementally.

DESTINY

So why is this interesting? Because these shared assumptions of your engine’s core components turn into the “unwritten rules of your engine”. Particularly about things like multithreading, and data lifecycle.

Every component has to follow these rules, otherwise it will be unstable or buggy or hard to work with.

** Engine codebases are really large these days. So these rules are very hard to change in a global fashion. You can extend the core components and add many new features on top of them. But evolutionary change to these core engine properties is very difficult.



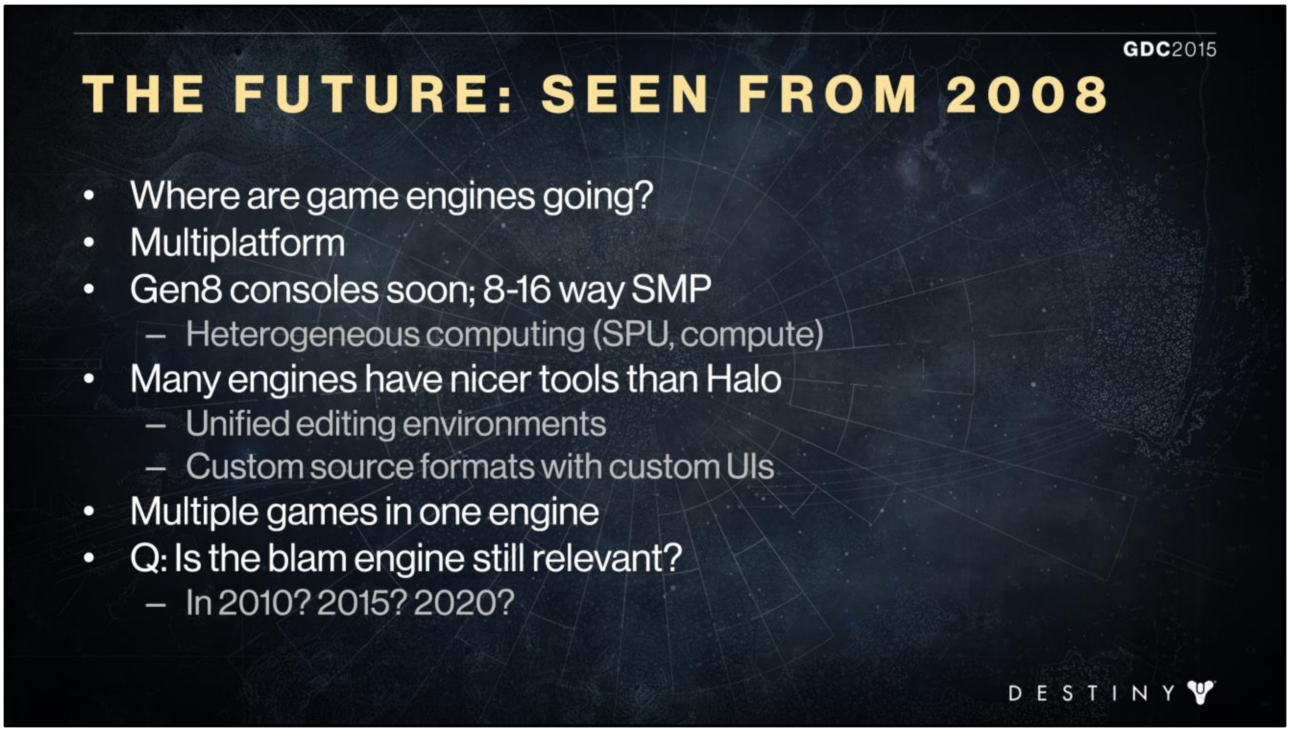
GDC2015

"RULES" OF THE BLAM ENGINE

- All allocations from per-system pools. No heap.
- Single platform - Xbox only, PC editor
- There is a main thread.
  - All game logic, and all engine systems, run on the main thread, except I/O and OS.
  - Halo 3 added a render thread
- Simple content pipeline
  - Bulk content (art, audio, lightmaps, navmesh) is compiled per asset
  - Most game content has no compile process.
  - Development build loads and converts loose files (slow!).
  - "Cooking" process to build console format levels has no incremental-update.
  - No system for patching content post-release
- Game code is accessible from any layer of the engine.
  - Hard to share code between games because engine is tainted with game logic

DESTINY

So what are the rules of the blam engine?

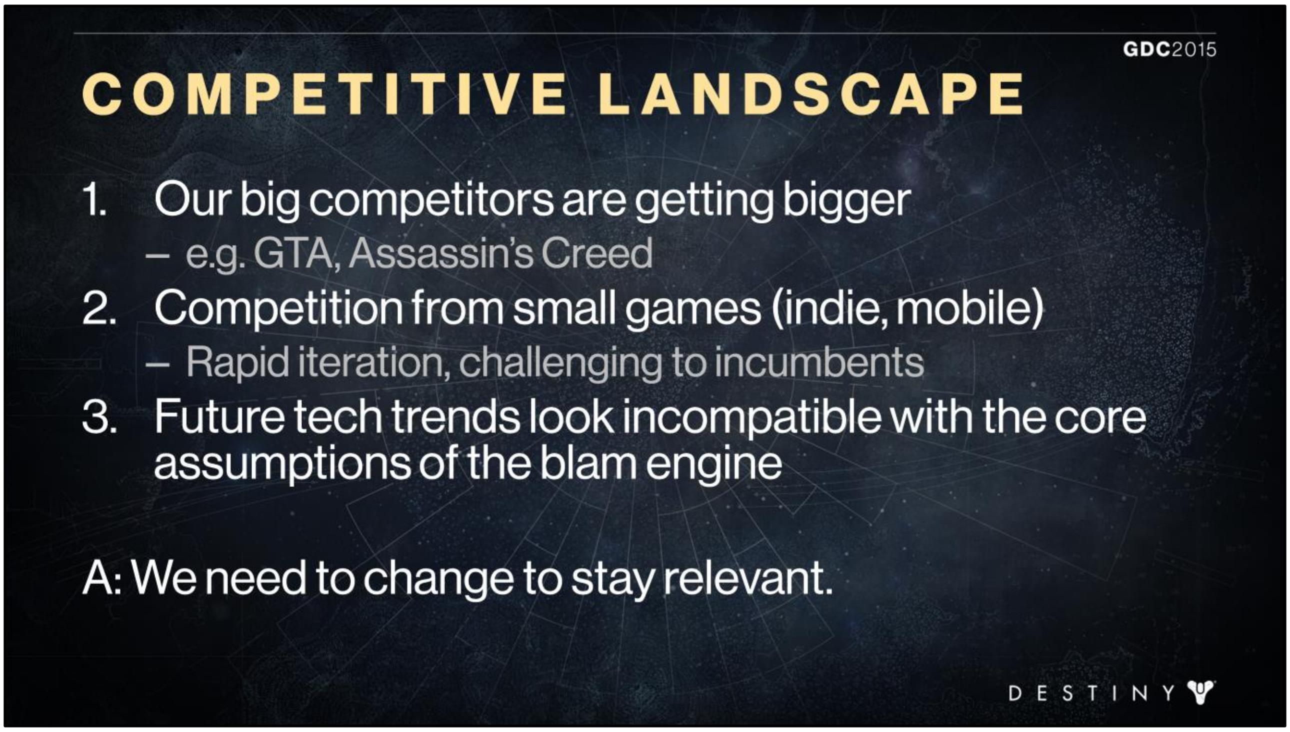


GDC2015

THE FUTURE: SEEN FROM 2008

- Where are game engines going?
- Multiplatform
- Gen8 consoles soon; 8-16 way SMP
  - Heterogeneous computing (SPU, compute)
- Many engines have nicer tools than Halo
  - Unified editing environments
  - Custom source formats with custom UIs
- Multiple games in one engine
- Q: Is the blam engine still relevant?
  - In 2010? 2015? 2020?

DESTINY


GDC2015

COMPETITIVE LANDSCAPE

1. Our big competitors are getting bigger
   – e.g. GTA, Assassin's Creed
2. Competition from small games (indie, mobile)
   – Rapid iteration, challenging to incumbents
3. Future tech trends look incompatible with the core assumptions of the blam engine

A: We need to change to stay relevant.

DESTINY

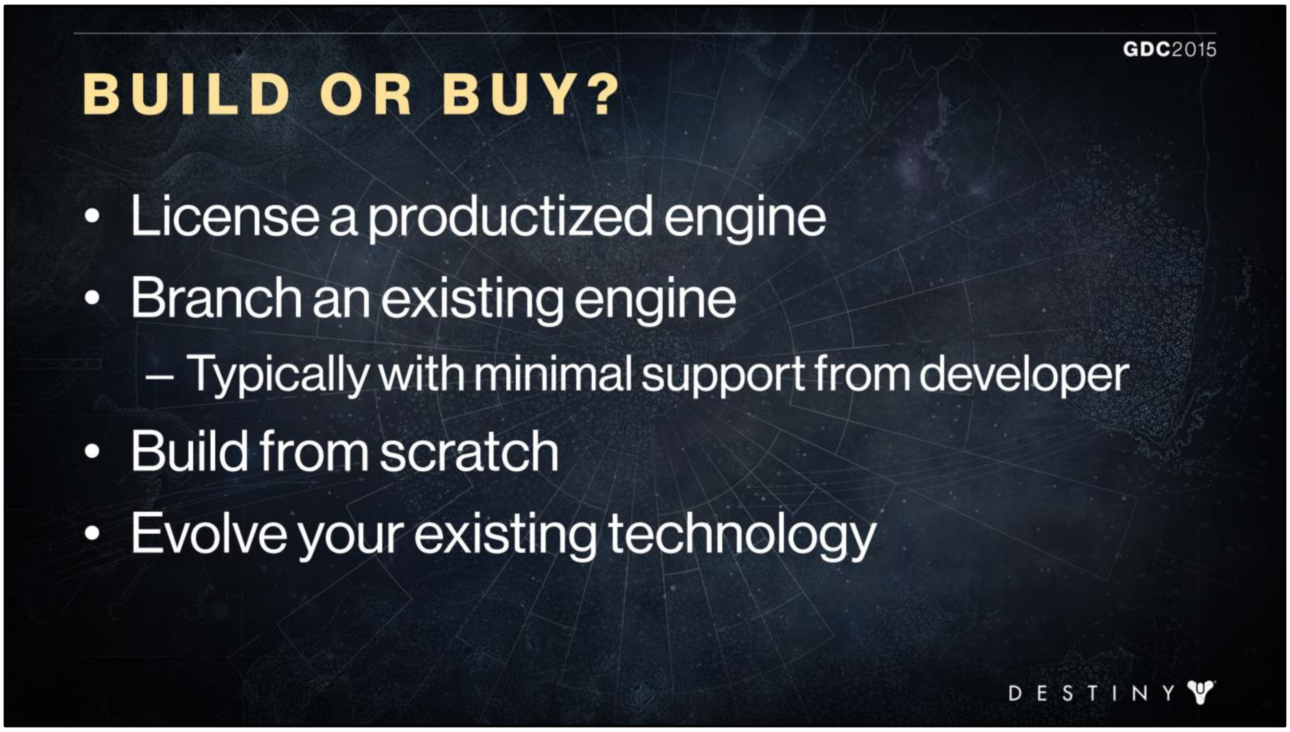
Three main factors in the competitive landscape back in 2008.

** First: The big games that are our competitors are getting bigger. More spectacle! More content! More specialized art teams!

** Second: Small games are on the rise, and competing for our business. Indie teams or mobile games can iterate rapidly, and they're eating away at the niche that small console games exist in.

** Third: When we look at technology in the future it seems that the core assumptions of our engine might start holding us back.

** These factors all lead us to conclude that we need to change something about our engine technology in order to stay relevant.


GDC2015

BUILD OR BUY?

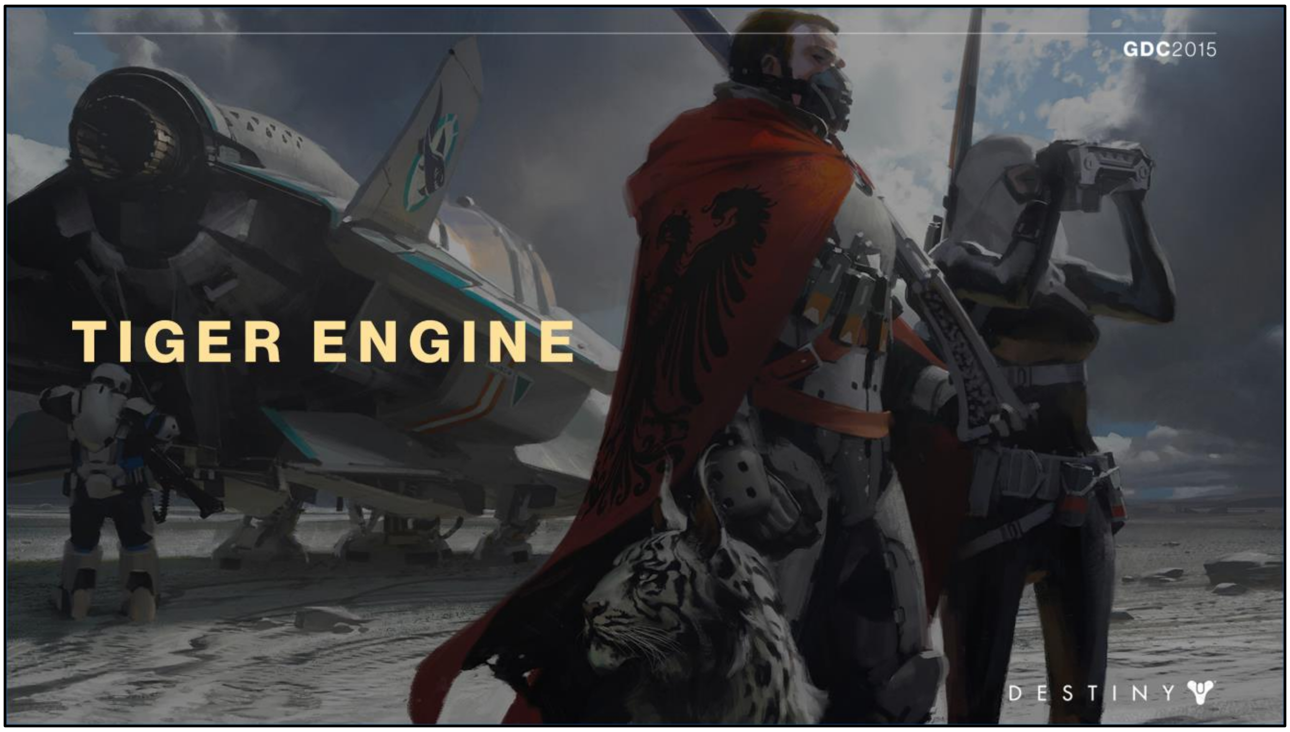
- License a productized engine
- Branch an existing engine
  - Typically with minimal support from developer
- Build from scratch
- Evolve your existing technology

DESTINY

Ok, so, suppose you've identified that you need to make a shift in engine strategy. You have options to choose from.

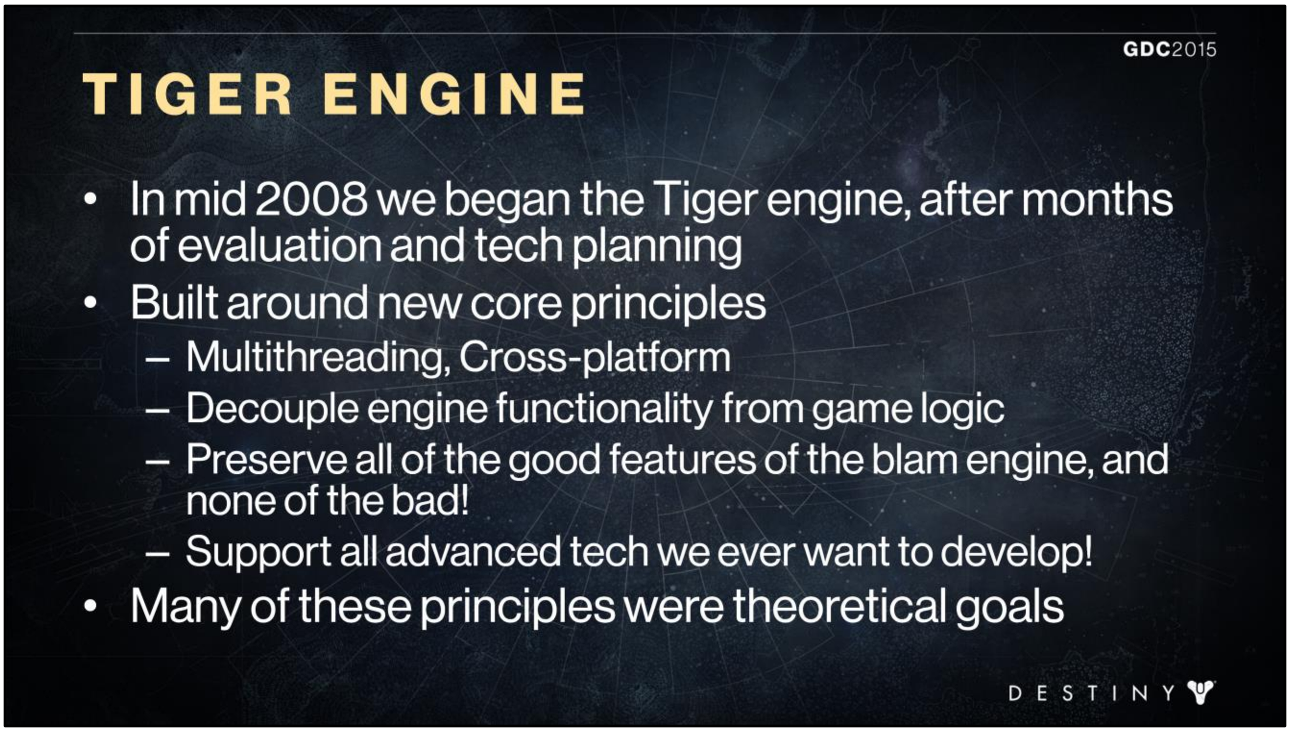
- ** You can license an engine that is a product with continuous support, such as Unreal or CryEngine. (Remember Unity wasn't around back in 2008.) You have the option here to add your own components, but in this scenario you aren't going to make changes to the core assumptions of the engine, because that would prevent you from taking continuous integrations and receiving support from the engine provider.
- ** Another option is to branch from another engine, either within your company or perhaps an external company. This model isn't favored as much these days, but it was common during the 1990s for example with ID software engines.
- ** Or, you could start from scratch. This is rare in the modern age. The only documented example I know of is Remedy developing the Alan Wake engine. [See Markus Maki's GDC11 presentation] This was a focused game – single player, single platform, they opted out of a lot of requirements that are present in a game like Destiny. Also, it was a 5-year effort for them. which for us wasn't an option at the time.

** So pretty quickly we settled on an evolutionary approach. There were big pieces of Halo technology that we knew we needed to preserve for Destiny, specifically the gameplay framework and networking. We had spent the last ten years solving the problems of making a satisfying networked action game and we just didn't know if we could replicate it in a new engine.



TIGER ENGINE

We called this effort to evolve our technology the "Tiger Engine", because we had this cool piece of concept art for Destiny that had a Space Tiger in it, and what is more awesome than that.



GDC2015

TIGER ENGINE

- In mid 2008 we began the Tiger engine, after months of evaluation and tech planning
- Built around new core principles
  - Multithreading, Cross-platform
  - Decouple engine functionality from game logic
  - Preserve all of the good features of the blam engine, and none of the bad!
  - Support all advanced tech we ever want to develop!
- Many of these principles were theoretical goals

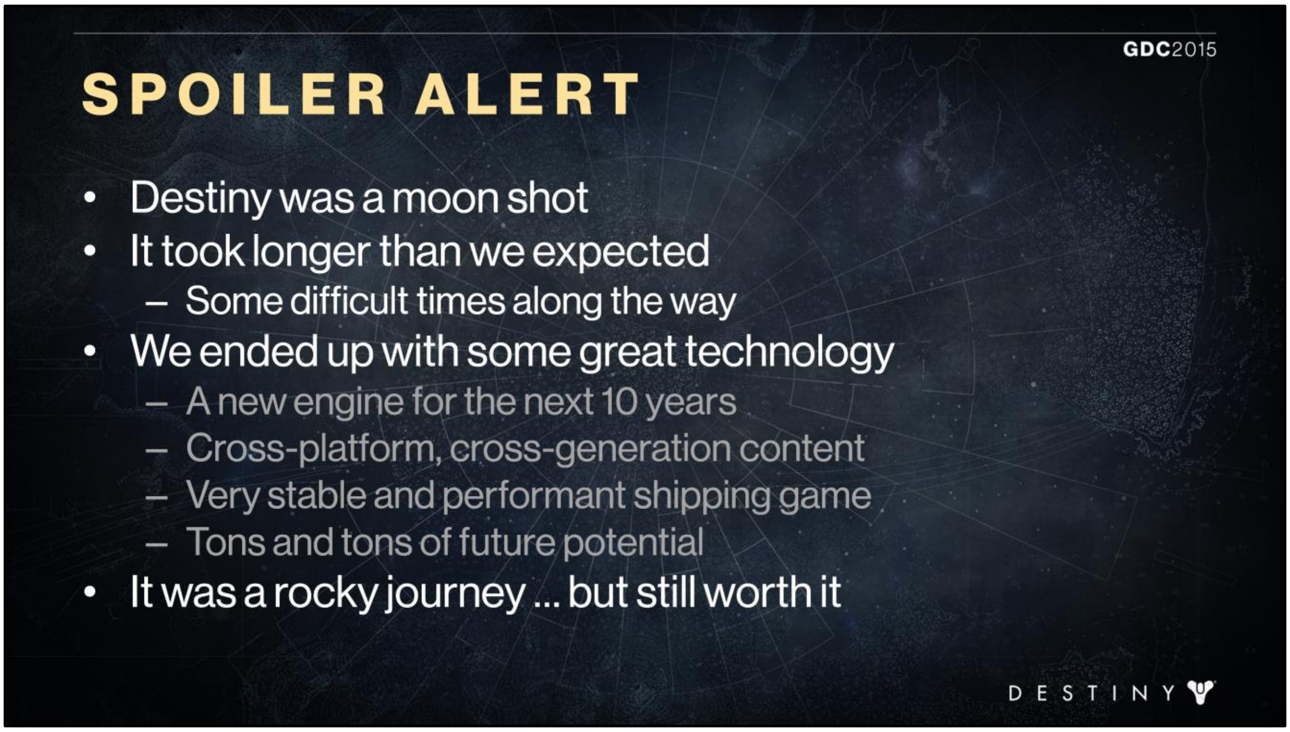
DESTINY

So after some months of evaluation and technology planning, we started the Tiger engine in mid 2008.

** We'd identified some core principles that we wanted from our technology.
- It needed to be pervasively multithreaded and cross-platform.
- We wanted to divide the codebase into layers, so we could decouple the engine functionality from the game logic.

** We were also pretty idealistic. We wanted to preserve all of the good features of the blam engine, and none of the bad features! And we had all of these great ideas for advanced technology that the engine had to be able to support!

** A lot of these principles were just theoretical goals when we started, and they became a lot clearer over the course of development.


GDC2015

SPOILER ALERT

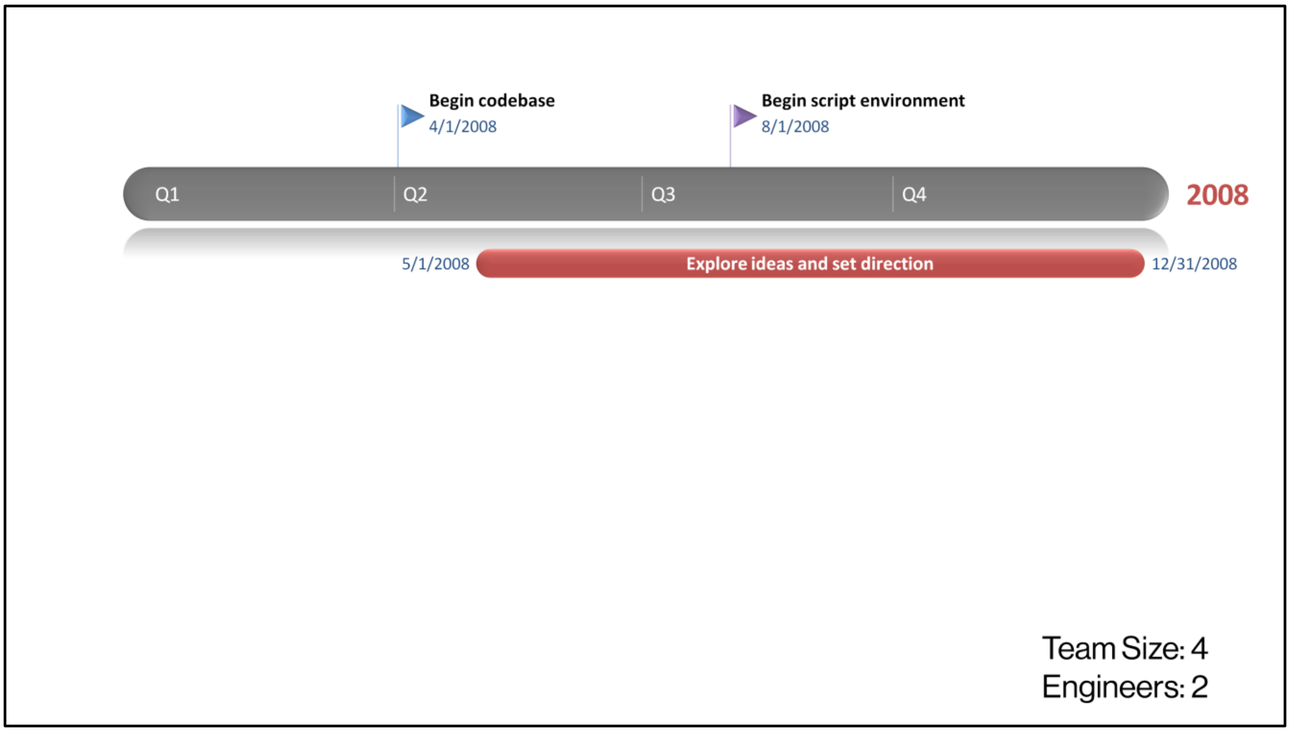
- Destiny was a moon shot
- It took longer than we expected
  - Some difficult times along the way
- We ended up with some great technology
  - A new engine for the next 10 years
  - Cross-platform, cross-generation content
  - Very stable and performant shipping game
  - Tons and tons of future potential
- It was a rocky journey ... but still worth it

DESTINY

When we come to GDC we're all looking for the real gritty truth, so I'm not going to sugar-coat anything about the development Destiny. As you'd expect, everything took longer than we expected, and there were some really difficult times along the way.

** But it's not all negative. We did a lot of great work and wound up with some really awesome technology.
- We have a new engine that will last us for 10 years
- It's cross-platform and cross-generation so our designers make one game and then it just ships on 4 platforms
- The final game is very stable, very performant, and it has a huge amount of future potential

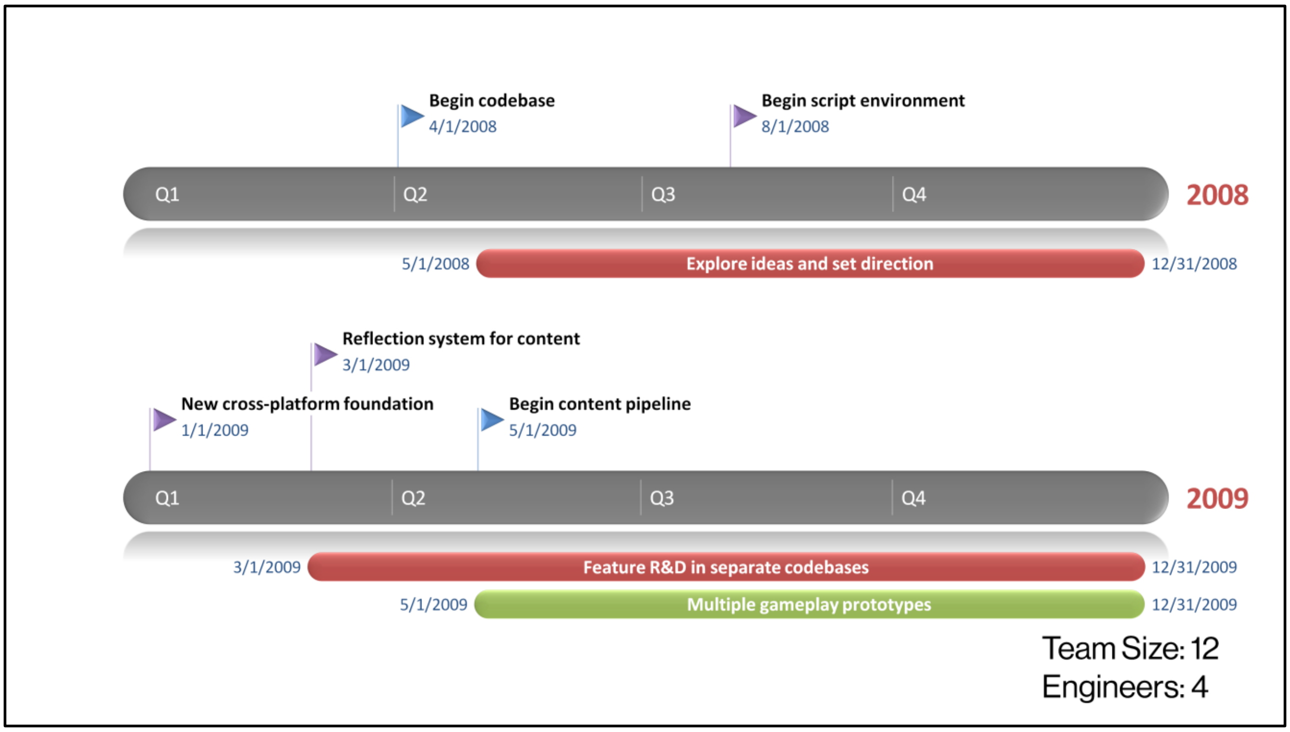
** It was a rocky journey developing all this technology, but despite all the difficulties we encountered, it was still worth it.



Team Size: 4
Engineers: 2

So now I'm going to walk you through a simplified timeline of the development of Destiny. My goal here is to provide just enough context for us to discuss the architectural lessons that we learned.
I'm only going to talk about the technology development, and won't touch at all on art or design concept development.

April: Begin with a branch of the Halo Reach codebase
Explore ideas and set direction for tiger engine
August: Build script environment for gameplay experimentation



Build a new cross-platform foundation for tiger core components

- Extract toolbox functions from Reach, write new multithreading primitives.

Reflection system and new source content format
Break ground on content import pipeline

Multiple independent efforts in feature R&D (graphics, pathfinding, animation, etc)

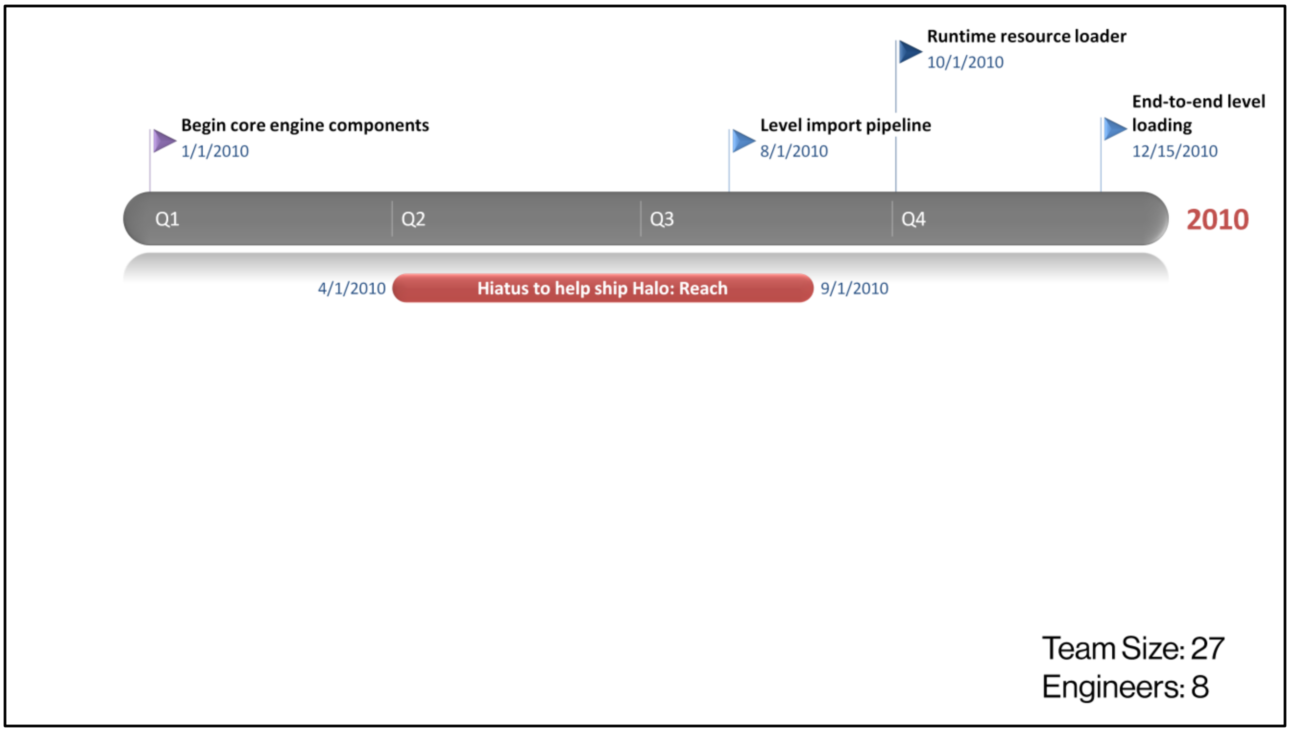
Multiple game prototypes in new gameplay/script environment

- These are experimental gameplay “mods” on top of Halo Reach content

EOY: A few new tiger components running inside the legacy engine codebase

[21:21:56] Enabling Manual Camera

This is an example of what the feature R&D codebases looked like in mid 2009. They were totally standalone from any of our engines or gameplay prototypes. This allowed rapid iteration on potential features for Destiny without being dependent on engine components that weren't written yet.



In 2010 we begin standing up those core engine components.

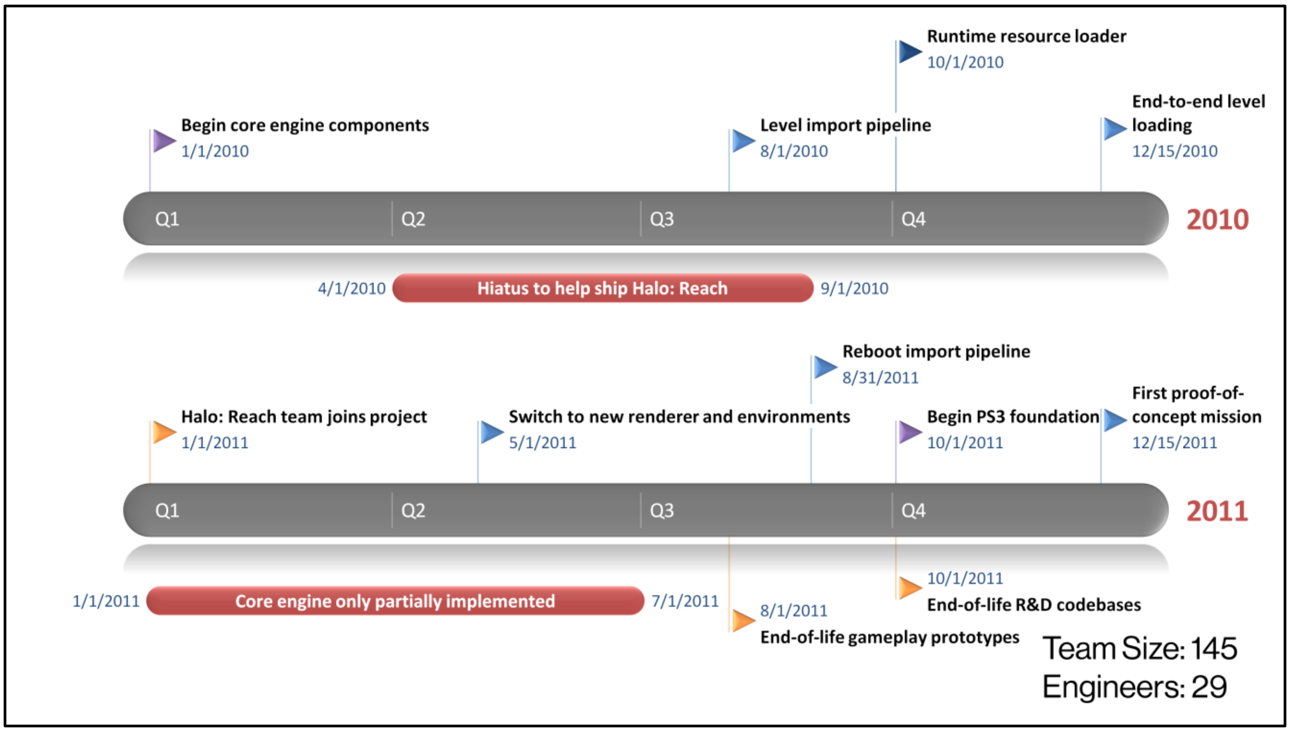
In the middle of the year there's a large hiatus when half of the Destiny engineering team returns to help ship Halo: Reach

Meanwhile we stand up the new level import pipeline

- New editor framework
- Requires all-new import pipelines for environment, geometry, object system, and gameplay

EOY: Basic test levels can be authored end-to-end,

all the from source content into the new runtime resource loader
- New engine components now connect mostly to each other rather than legacy engine



Around this time the Halo: Reach team joins the project

Many new engine components are in a partially-implemented state

- Most features you want to work on need new core functionality written to support them
- Engineering efforts get fragmented, as multiple teams move into separate R&D codebases to get their work done

In May we bring the new renderer and environment system out of their R&D codebase and into the main line

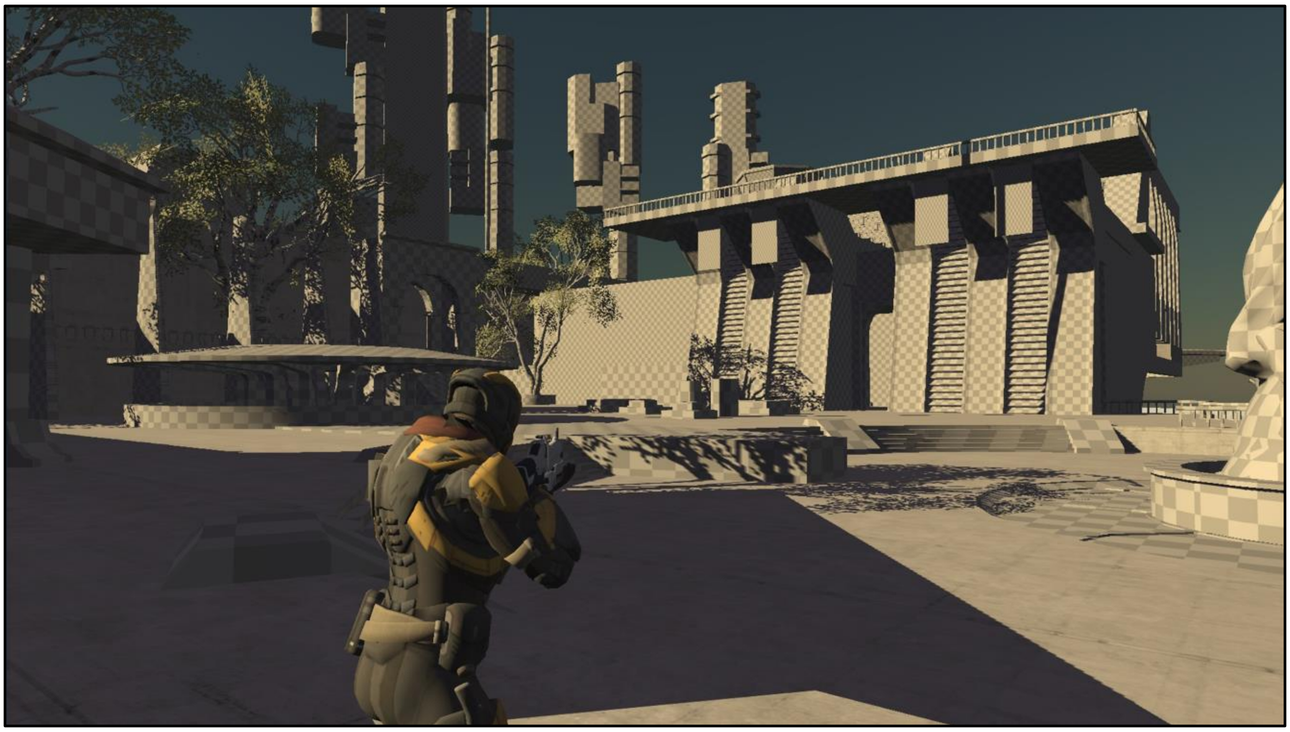
- You can still run legacy game prototypes but they aren't

compatible with these new systems
- Game prototypes continue for a few months before we deprecate them

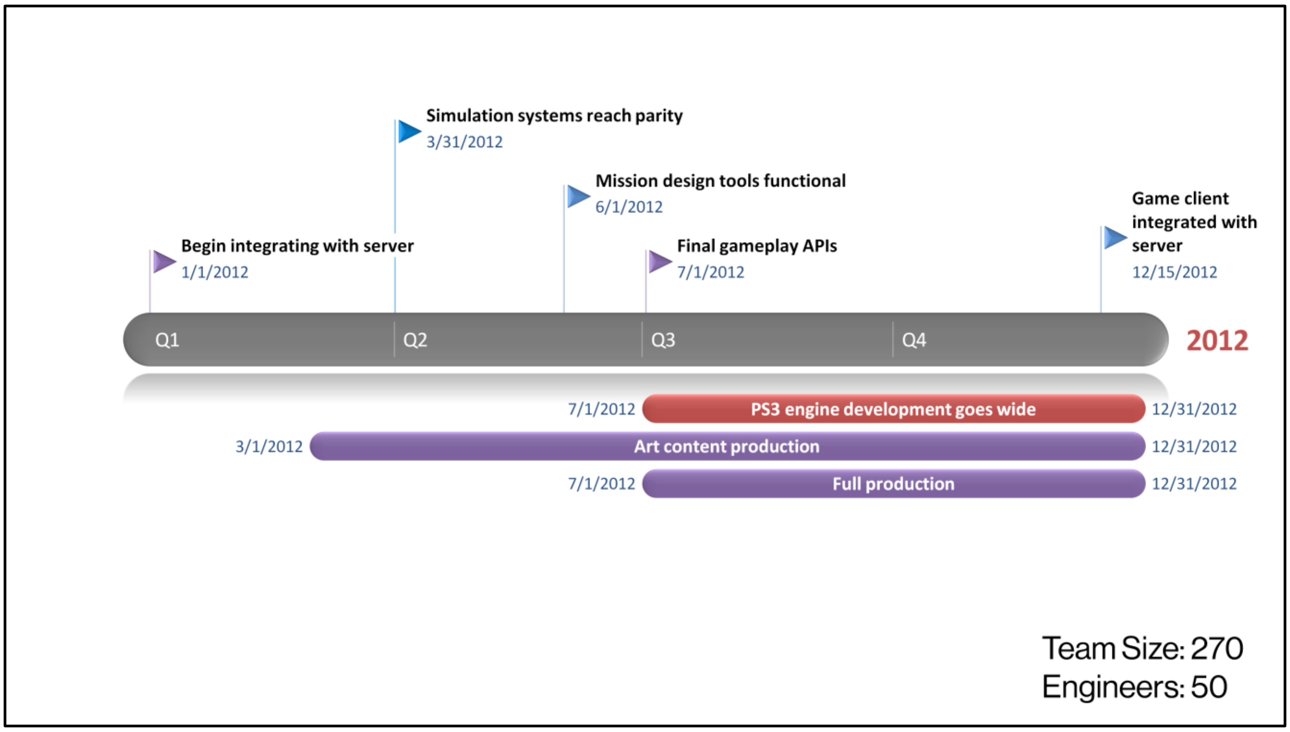
In August we have an onslaught of new content being created and our systems are not scaling to handle it.
We have to reboot the content import pipeline to unblock art content production. I'll talk about this more later.

In October PS3 foundation development begins with small team

EOY: First 'proof of concept' mission with client networking but no servers
- All R&D and gameplay codebases reunited into single mainline



This is an example of the tiger engine systems as they start to be put together around the middle of 2011.
Artists and designers are able to try out their ideas in test levels in the new engine but no production content can be created yet.



2012 is the year when all of the separate pieces come together.

We start integrating client engine with server architecture which prior to this had been independent.

March: Art content production can begin.
New game simulation systems reach parity with legacy engine, so we can port all our prototypes

June: Mission design tools are functional, true game production can begin

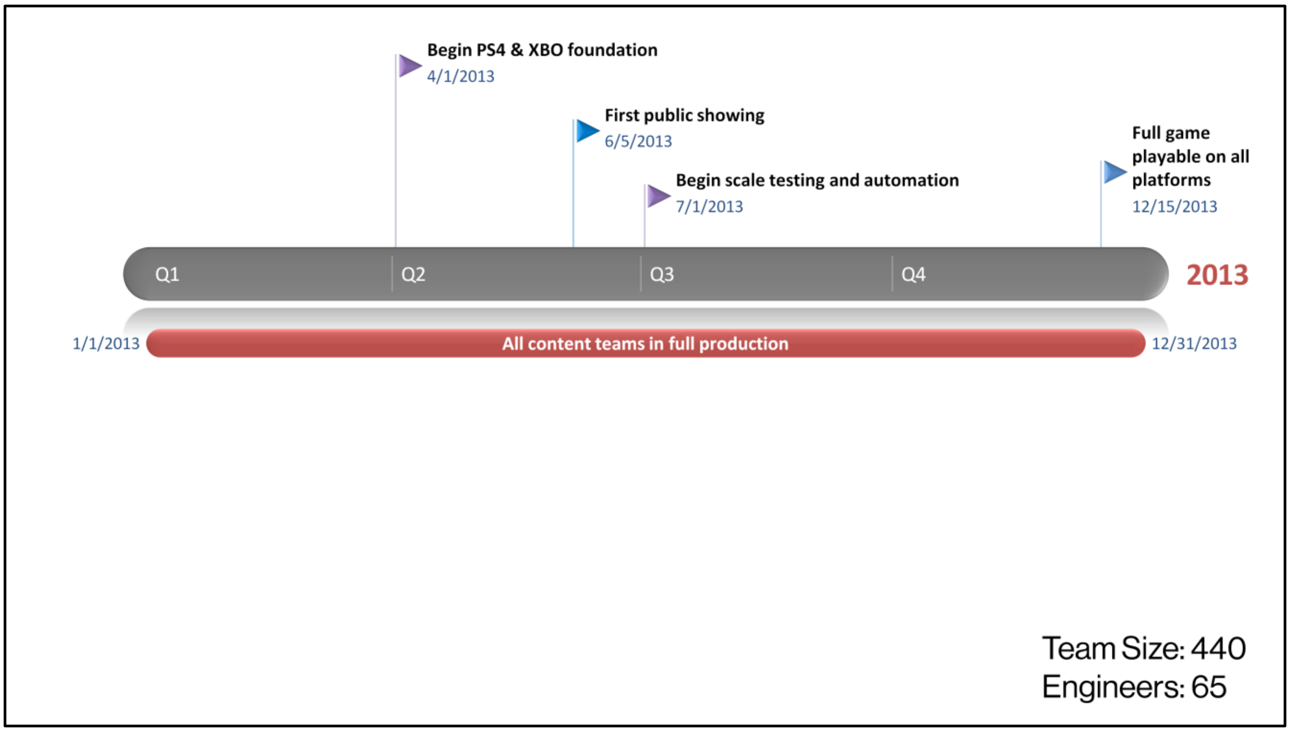
Around this time, gameplay features can use final APIs

- No longer need any rewrites between here and ship

PS3 engine development had started in late 2011 with a small team. By late in 2012 we were ready to put a ton of engineers on converting all our features and writing SPU code. I’d estimate the PS3 engine took about 25 engineer-years in total over the project, not including the content creation costs.

EOY: Game client integrated with network and service architecture

This is where we end up at the end of 2012. We can make representative experiences. Take a look at the difference one year makes. Everything can come together quicker than you think.


Begin PS4 & XBO foundation
4/1/2013
First public showing
6/5/2013
Begin scale testing and automation
7/1/2013
Full game playable on all platforms
12/15/2013
Q1
Q2
Q3
Q4
2013
1/1/2013
All content teams in full production
12/31/2013
Team Size: 440
Engineers: 65

Engineering moves to support gameplay features
All content teams in full production

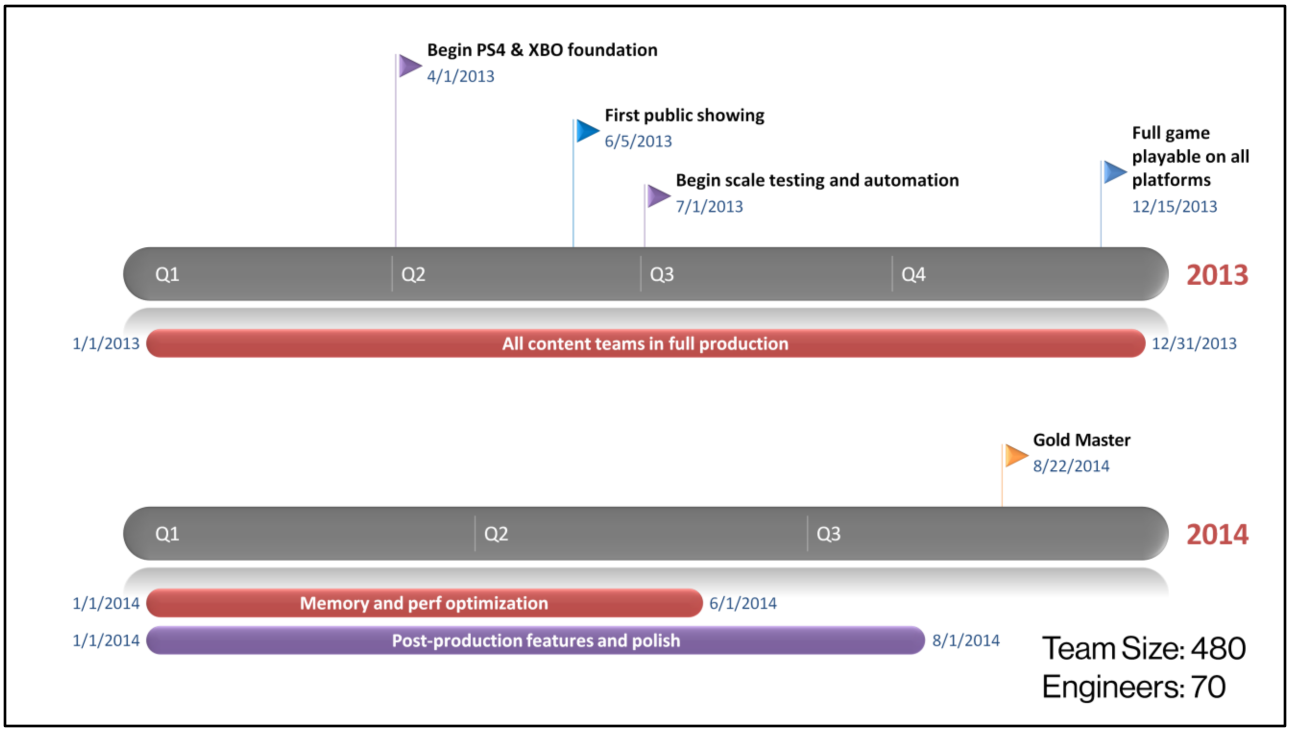
- Continuous optimization of content pipeline to meet new content demands

April-November: Bring up engine on PS4 and XBO

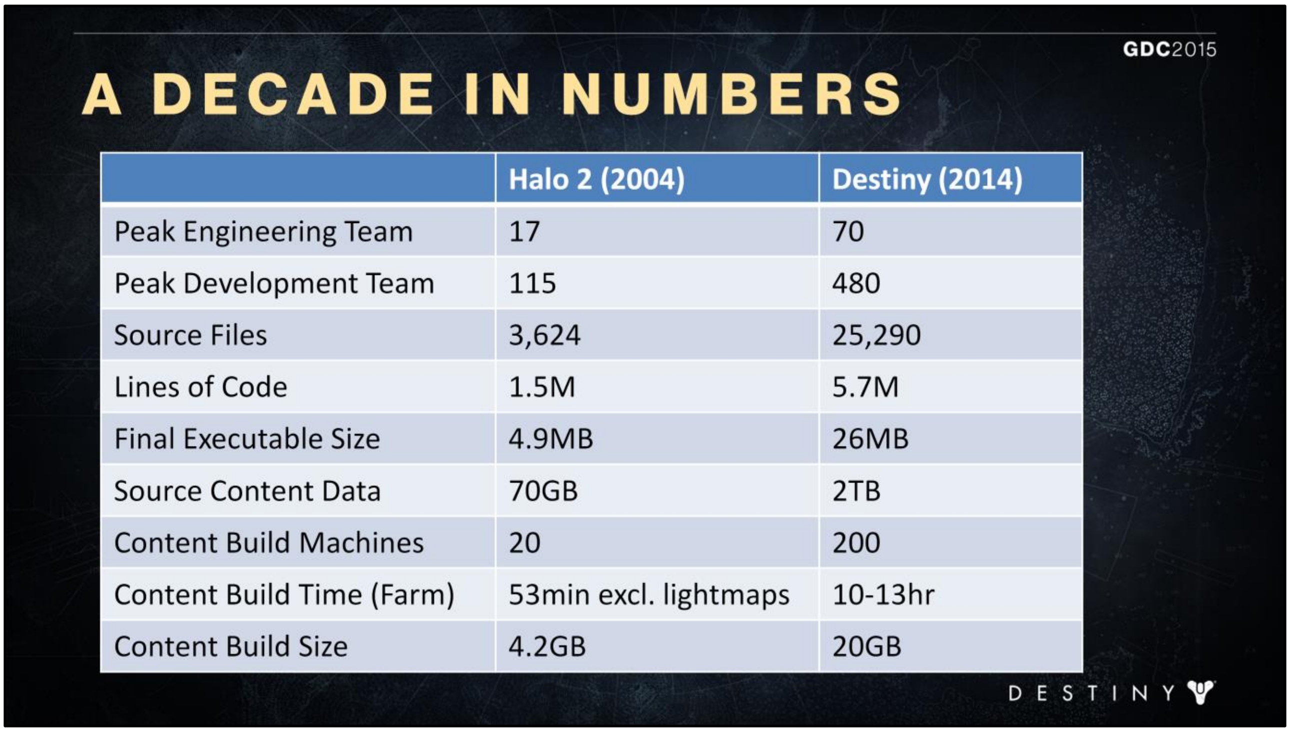
- Next-gen platforms went from 0 to 80% functionality in ~6 months (mostly because our engine features and content were planned for them from day 1)

E3 2013: first public showing
Large-scale testing and automation begin

EOY: Full game playable on all platforms



Engineering post-production focused on memory and performance
Content post-production: build missing features, polish the game
Bring all features to shipping bar on 4 platforms
August: Go Gold



GDC2015

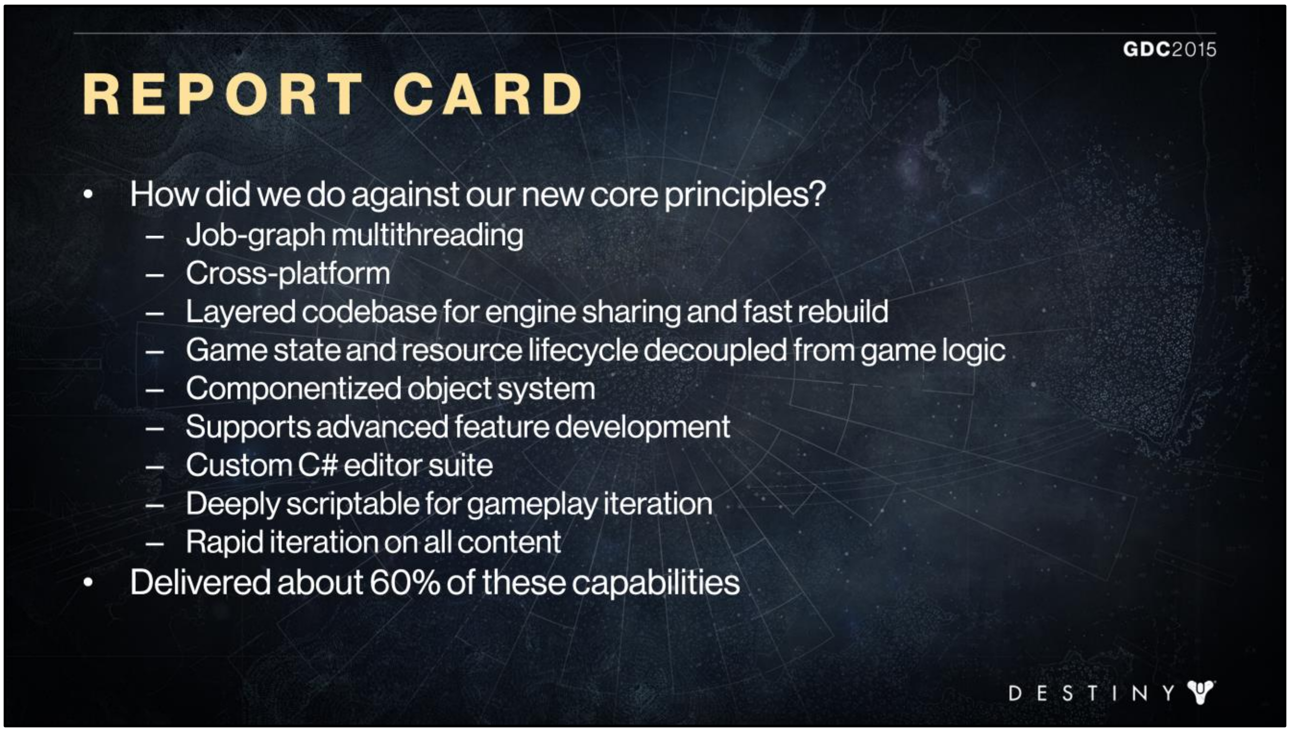
A DECADE IN NUMBERS

	Halo 2 (2004)	Destiny (2014)
Peak Engineering Team	17	70
Peak Development Team	115	480
Source Files	3,624	25,290
Lines of Code	1.5M	5.7M
Final Executable Size	4.9MB	26MB
Source Content Data	70GB	2TB
Content Build Machines	20	200
Content Build Time (Farm)	53min excl. lightmaps	10-13hr
Content Build Size	4.2GB	20GB

DESTINY

So finally, we were done. We ended up with a really large project, by far our biggest.

The codebase clocks in at around 5.7MLOC, this size includes both client game engine and tools, but not bungie.net or the Destiny service.



GDC2015

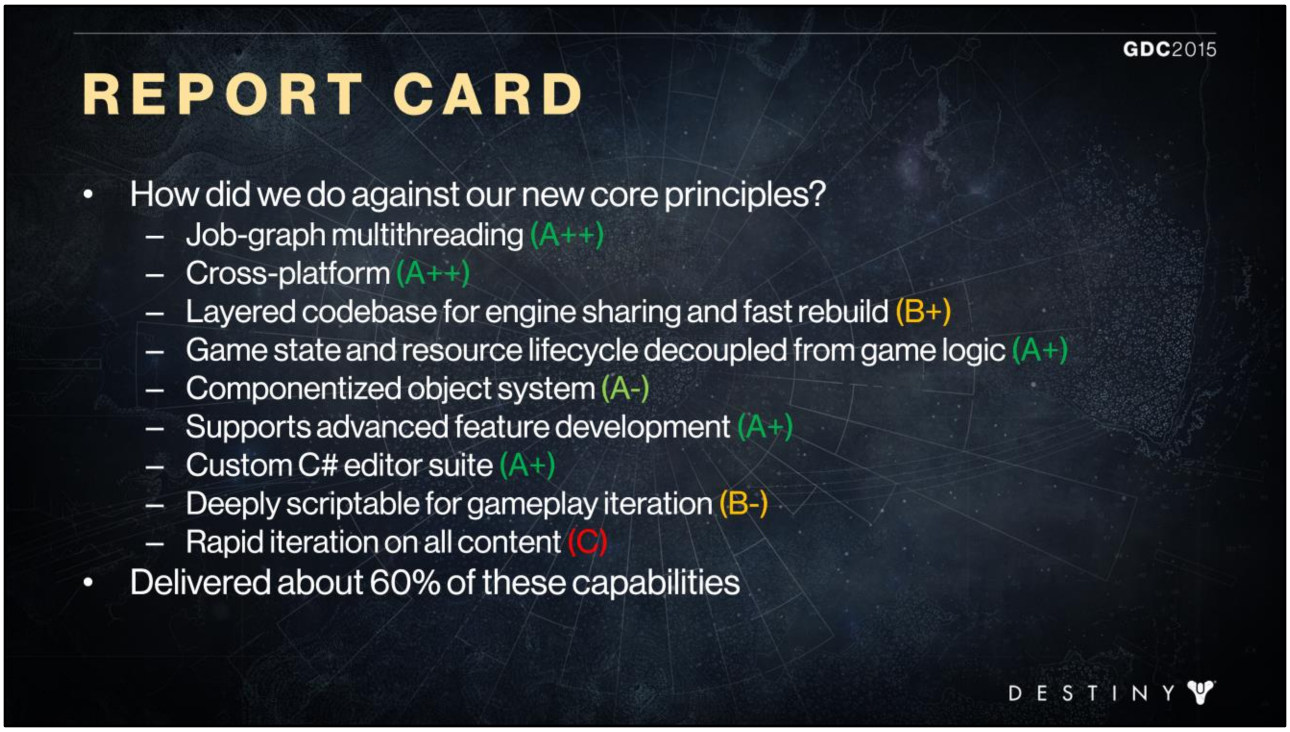
REPORT CARD

- How did we do against our new core principles?
  - Job-graph multithreading
  - Cross-platform
  - Layered codebase for engine sharing and fast rebuild
  - Game state and resource lifecycle decoupled from game logic
  - Componentized object system
  - Supports advanced feature development
  - Custom C# editor suite
  - Deeply scriptable for gameplay iteration
  - Rapid iteration on all content
- Delivered about 60% of these capabilities

DESTINY

So how did we do against our new core engine principles? We had a ton of goals.

** Well, overall, I'd say we delivered about 60% of the capabilities that we were aiming for. Some of them ended up better than others.



GDC2015

REPORT CARD

- How did we do against our new core principles?
  - Job-graph multithreading (A++)
  - Cross-platform (A++)
  - Layered codebase for engine sharing and fast rebuild (B+)
  - Game state and resource lifecycle decoupled from game logic (A+)
  - Componentized object system (A-)
  - Supports advanced feature development (A+)
  - Custom C# editor suite (A+)
  - Deeply scriptable for gameplay iteration (B-)
  - Rapid iteration on all content (C)
- Delivered about 60% of these capabilities

DESTINY

A lot of stuff wound up working really well – like our multithreading, our cross-platform engine, our editor and tools.

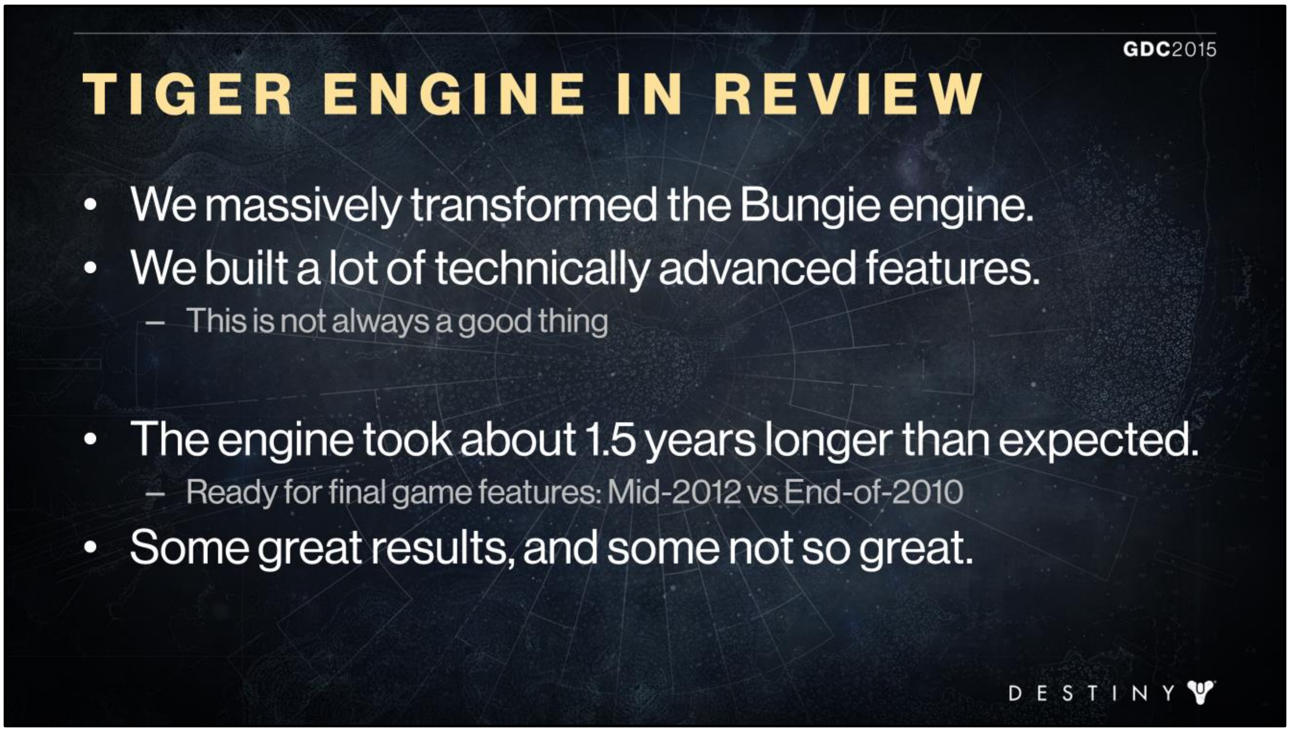
However, our layered codebase structure ended up messier than we intended. It compiles very slowly and ends up being a time sink on our engineering team. We probably need to split up the codebase into DLLs by layer, which take a really long time now given the size of the codebase. We now realize that we should have done this up front, but we didn't.

Our componentized object system is super powerful, but it's kind of hard to use both for engineers and content creators.

Our scripting environment works, but it isn't very efficient either for prototyping or implementing production gameplay. This is mostly a consequence of iteration times and performance. I'll talk about this later on.

But the big one is that our content iteration times are pretty bad. You can be looking at minutes for small changes and tens of minutes for big changes. One of the pillars of the engine was meant to be rapid content iteration, and we struggled with this for the whole course of the project. It's very hard to solve this problem during active development, but we've made big strides toward fixing it after release. I'll talk about this more later on as well.

We don’t have time for a full postmortem, so that’s about as far as I’m going to go into these specific issues for now.


GDC2015

TIGER ENGINE IN REVIEW

- We massively transformed the Bungie engine.
- We built a lot of technically advanced features.
  - This is not always a good thing
- The engine took about 1.5 years longer than expected.
  - Ready for final game features: Mid-2012 vs End-of-2010
- Some great results, and some not so great.

DESTINY

So, to review what we've learned so far:
- ** we undertook a massive transformation of our engine
- ** we built a lot of features that are highly technically advanced – which is both good and bad
- ** And it took about one-and-a-half years longer to build the engine than we originally thought

** And like we discussed, we ended up with some great results and some not so great results.

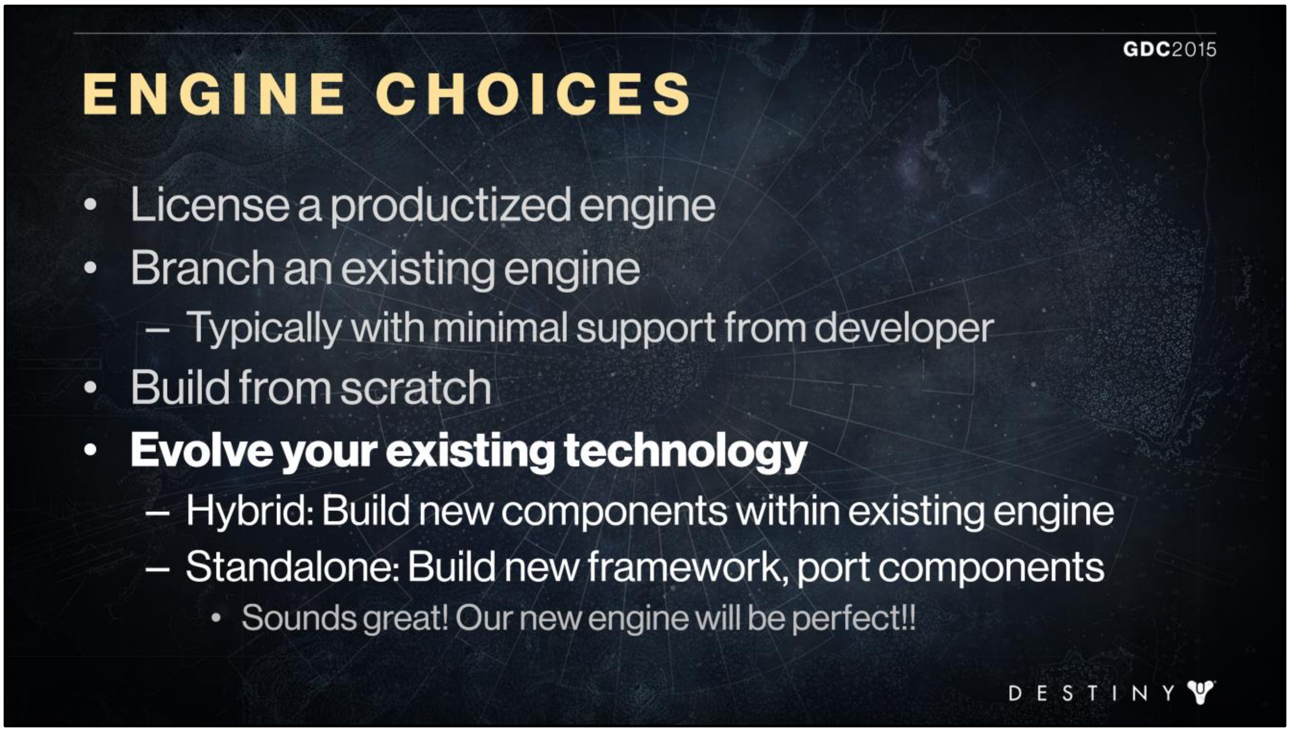
Now we'll take a look at some lessons that emerged from this extended development process.

GDC2015

NOTHING IS NEW ANY MORE

DESTINY

The first lesson is that Nothing is New Any More.



GDC2015

ENGINE CHOICES

- License a productized engine
- Branch an existing engine
  - Typically with minimal support from developer
- Build from scratch
- **Evolve your existing technology**
  - Hybrid: Build new components within existing engine
  - Standalone: Build new framework, port components
    - Sounds great! Our new engine will be perfect!!

DESTINY

Remember we were talking about engine choices and we identified that we wanted to evolve our existing technology.

There are two main approaches.
** You can try to build your new components within the context of an existing engine. But how would that even work? You'd be dealing with all of the complexity of the old code PLUS all of the new code at once. That sounds like that would be really slow and inefficient. We shouldn't do that!!
** I know - instead we should construct a new framework that stands alone, and then take the pieces we want from the old engine and bring them across. That sounds amazing and beautiful. Our new engine will truly be a work of art!

GDC2015

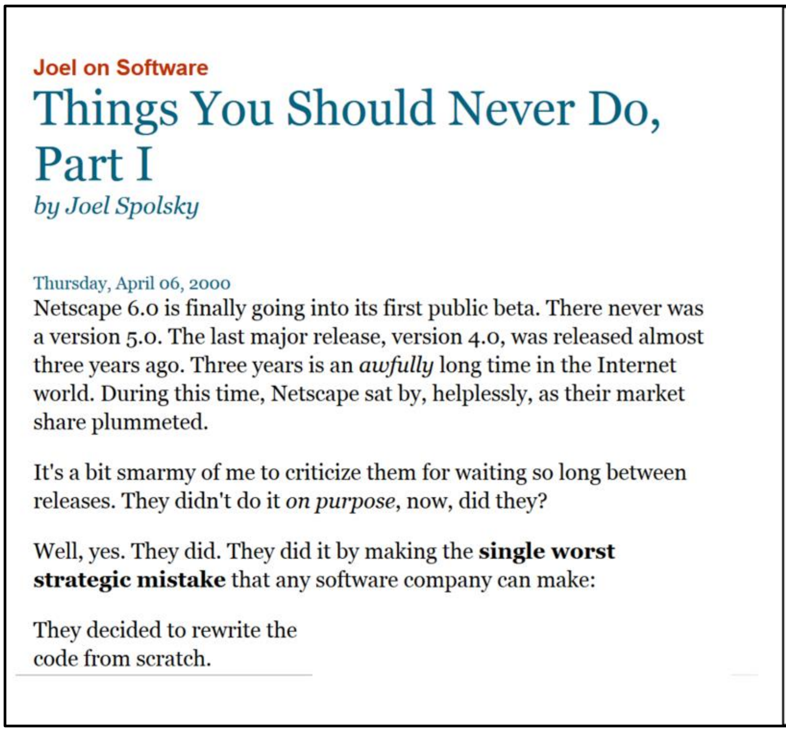
ENGINE

- License a p
- Branch an
  - Typically   loper
- Build from s
- Evolve you
  - Hybrid: B   sting engine
  - Standalo   omponents
    - Sounds   !!

DESTINY

Well… no. It’s a trap.

The problem we face here is well-known. Programmers love to write new code. It’s our favourite thing in the world. It colours our judgment in every way … so much that it is very difficult to make rational decisions about “rewrite vs extend”. Even when we’re aware of this emotional bias, we are still subject to it. So yes, the “standalone” approach where we port existing code into a newly written framework sounds beautiful, it’s incredibly enticing, but it is a huge trap.



Joel on Software

Things You Should Never Do, Part I

by Joel Spolsky

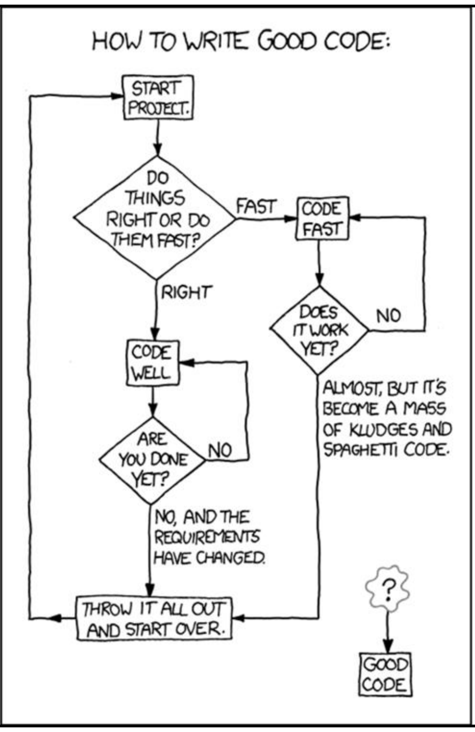
Thursday, April 06, 2000

Netscape 6.0 is finally going into its first public beta. There never was a version 5.0. The last major release, version 4.0, was released almost three years ago. Three years is an *awfully* long time in the Internet world. During this time, Netscape sat by, helplessly, as their market share plummeted.

It's a bit smarmy of me to criticize them for waiting so long between releases. They didn't do it *on purpose*, now, did they?

Well, yes. They did. They did it by making the **single worst strategic mistake** that any software company can make:

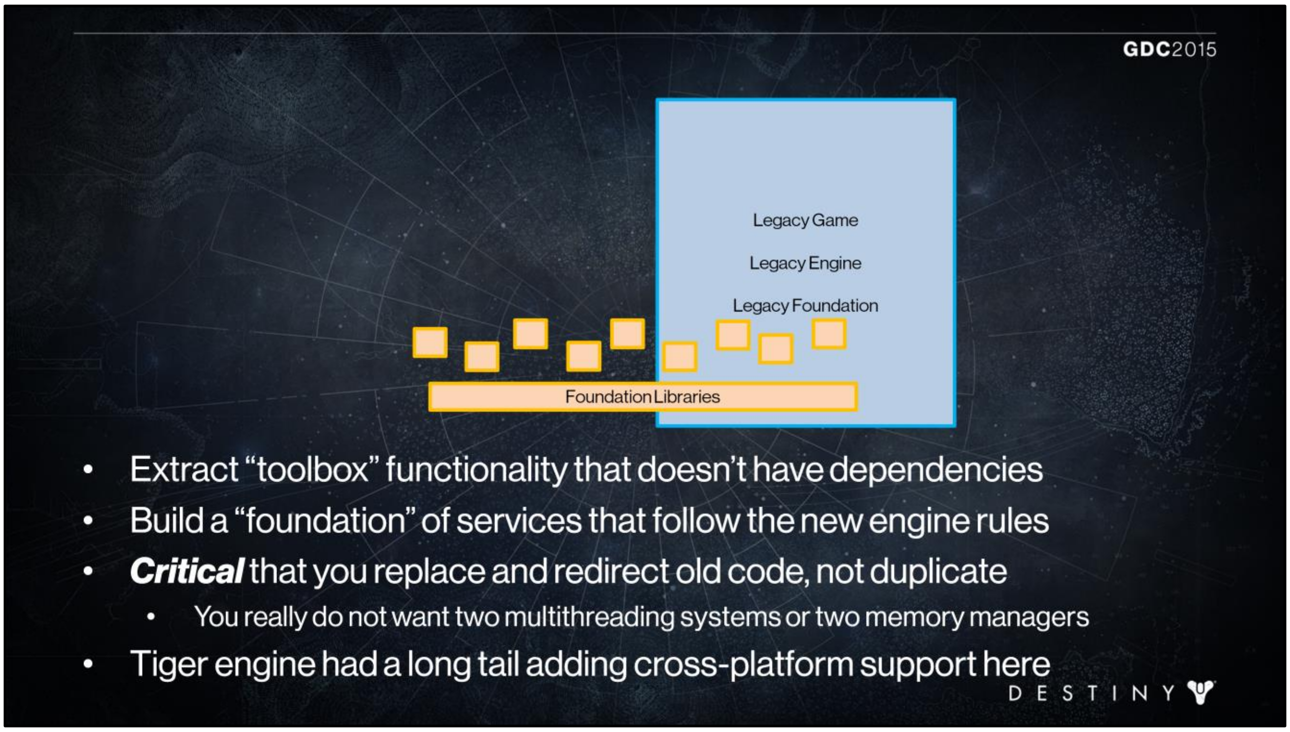
They decided to rewrite the code from scratch.


HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
DOES IT WORK YET?
NO
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
RIGHT
CODE WELL
ARE YOU DONE YET?
NO
NO, AND THE REQUIREMENTS HAVE CHANGED.
THROW IT ALL OUT AND START OVER.
?
GOOD CODE

This is not a new concept. Here you can see two famous examples of writing on this subject. Joel Spolsky and Randall Munroe. Hopefully everybody is familiar with their work.

Now, you could form the opinion that new code is never the answer, and we are just doomed to legacy systems forever.

But there is another way, and it’s what we did with the Tiger engine.



GDC2015

Legacy Game
Legacy Engine
Legacy Foundation
Foundation Libraries

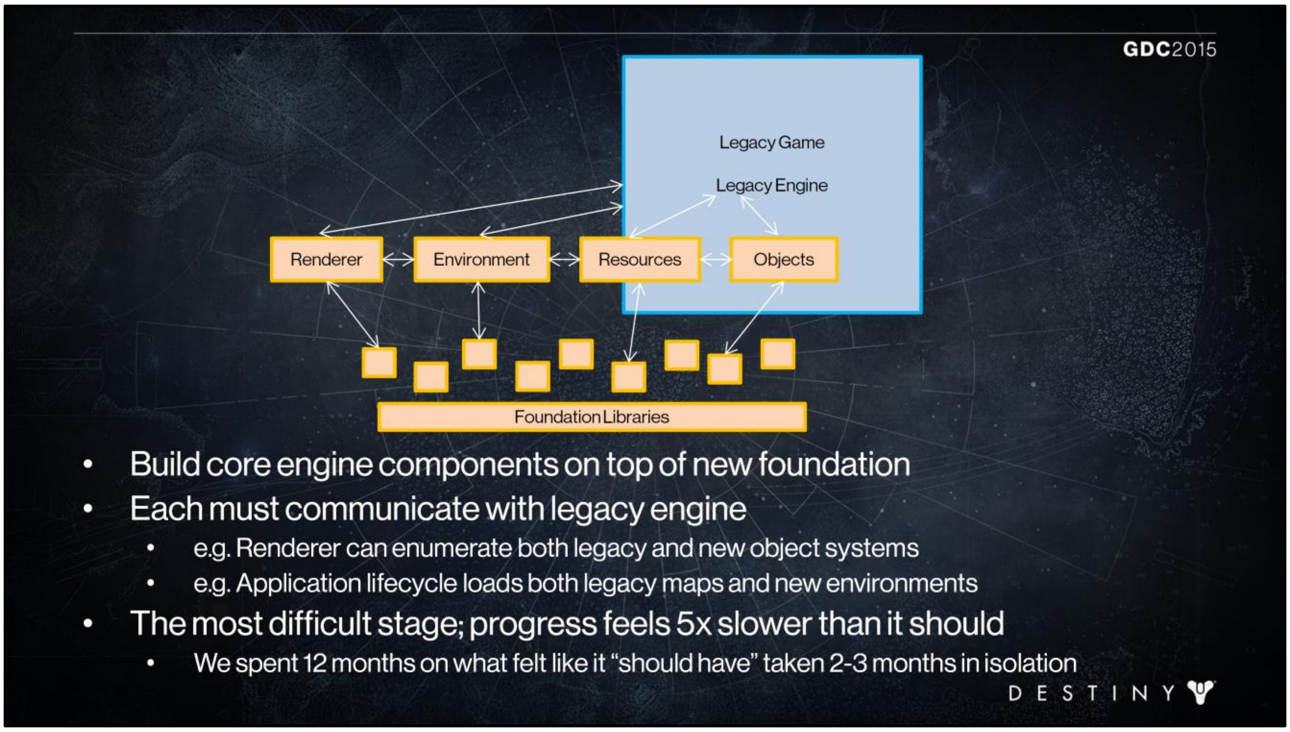
- Extract "toolbox" functionality that doesn't have dependencies
- Build a "foundation" of services that follow the new engine rules
- ***Critical*** that you replace and redirect old code, not duplicate
  - You really do not want two multithreading systems or two memory managers
- Tiger engine had a long tail adding cross-platform support here

DESTINY

In this approach, you start by extracting functionality from your legacy engine that can stand alone, "toolbox" type code.

You need to extract and **REPLACE** the old code, rerouting the engine to talk to the new foundation. If you don't do this then you are adding too much complexity. It's ok to replace a mature system with a 40% complete implementation at this point, as long as you can get the job done and then come back later to handle edge cases.

** Now for the sake of simplicity I'm going to present this as a number of sequential stages. But what actually happens is that all of these stages have a long tail before they are complete, and they overlap with later stages.

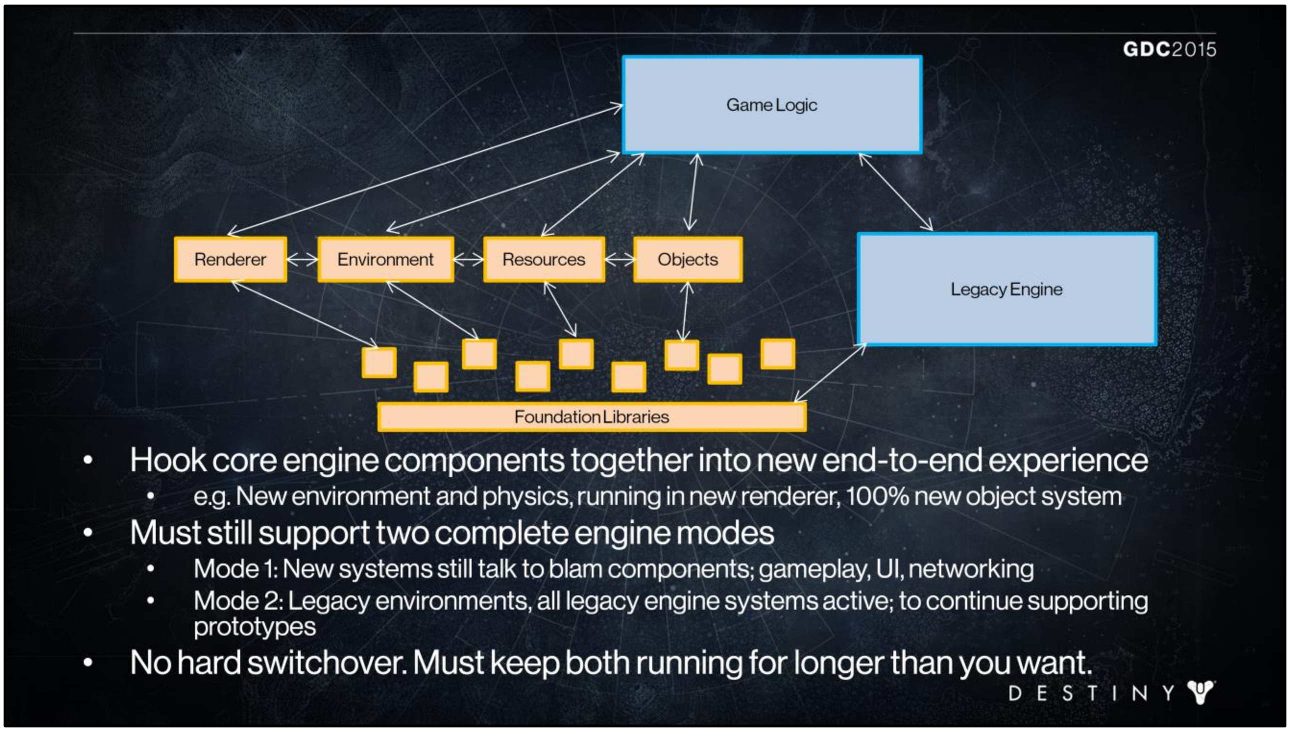


- Build core engine components on top of new foundation
- Each must communicate with legacy engine
  - e.g. Renderer can enumerate both legacy and new object systems
  - e.g. Application lifecycle loads both legacy maps and new environments
- The most difficult stage; progress feels 5x slower than it should
  - We spent 12 months on what felt like it "should have" taken 2-3 months in isolation

The next stage is when you start building your new core engine components on top of this new foundation layer.

Each of these components will follow the new engine rules internally, and communicate with the legacy engine through shims. For example your new renderer has to be able to draw legacy objects (even if it just shows them as colored spheres) and your new object system has to collide with both legacy environments and your new environment representation.

** This is really hard. This approach adds such a lot of complexity and time. We spent over a year here in this stage and progress felt very, very slow – emotionally it felt about 5x slower than it "should". But eventually we started to get traction and the new components started to get completed and then connect together.

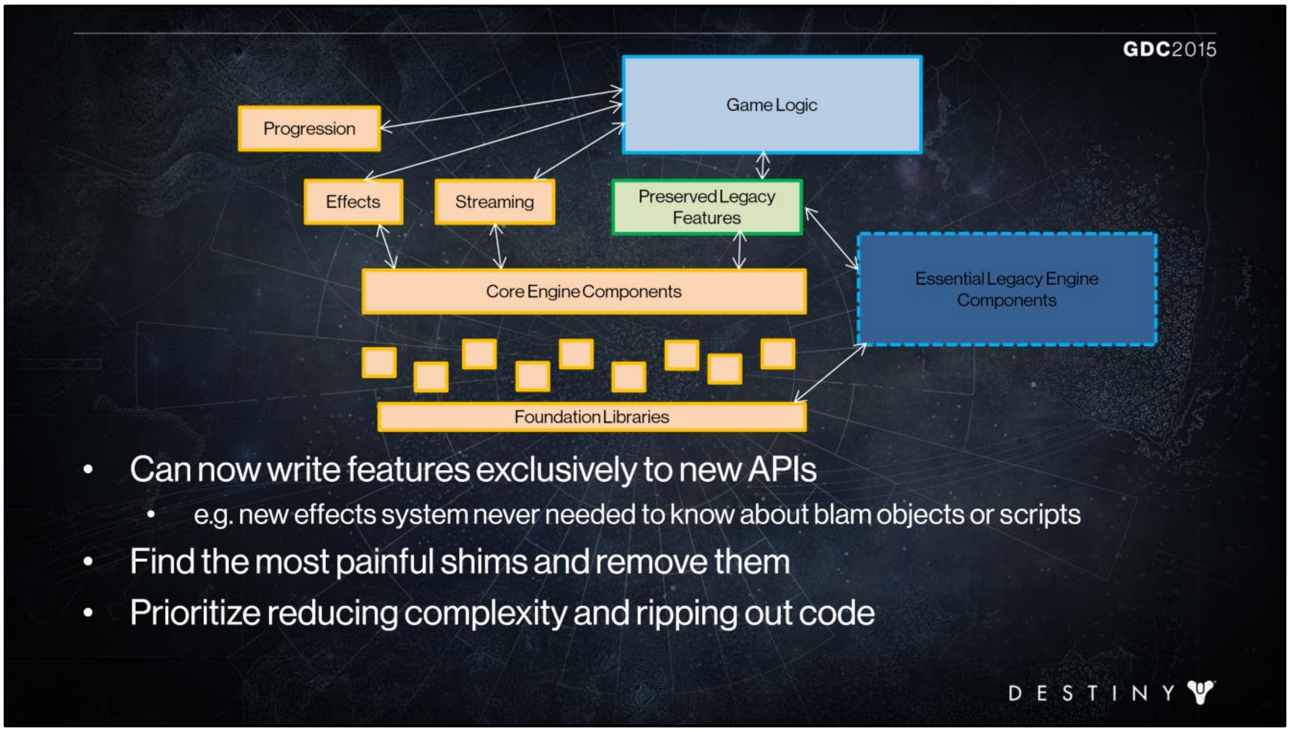


- Hook core engine components together into new end-to-end experience
    - e.g. New environment and physics, running in new renderer, 100% new object system
- Must still support two complete engine modes
    - Mode 1: New systems still talk to blam components; gameplay, UI, networking
    - Mode 2: Legacy environments, all legacy engine systems active; to continue supporting prototypes
- No hard switchover. Must keep both running for longer than you want.

The next stage is exciting because you can start to see real progress. You can run the engine in a mode where these new systems are all active and they all have primary control. The new renderer is displaying new environments with a new physics system. You can start building very basic new content.

** At this point you need to retain the ability to switch at runtime between two modes. One with the new systems and one with the legacy systems – so you can run a legacy map that uses the old renderer and environments, and old object system. That way you can still develop your prototypes that rely on the legacy gameplay content.

** Again, this is really painful, because you have to keep this stuff running for longer than you want.

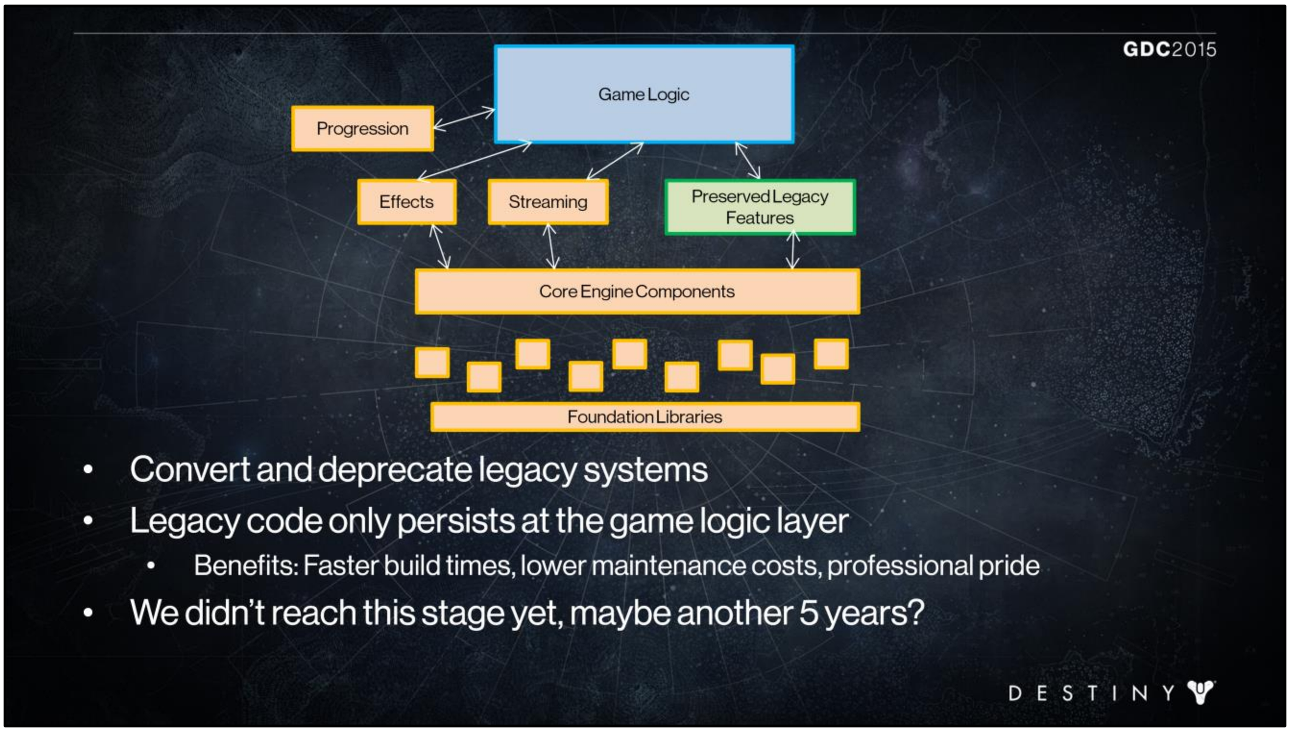


- Can now write features exclusively to new APIs
  - e.g. new effects system never needed to know about blam objects or scripts
- Find the most painful shims and remove them
- Prioritize reducing complexity and ripping out code

In the next stage, now that you have the core engine up and running, your engineers can start writing features exclusively to the new APIs.

For example our new effects system was written entirely on top of the new object system and never knew anything about the old one. This is where your progress starts to take off.

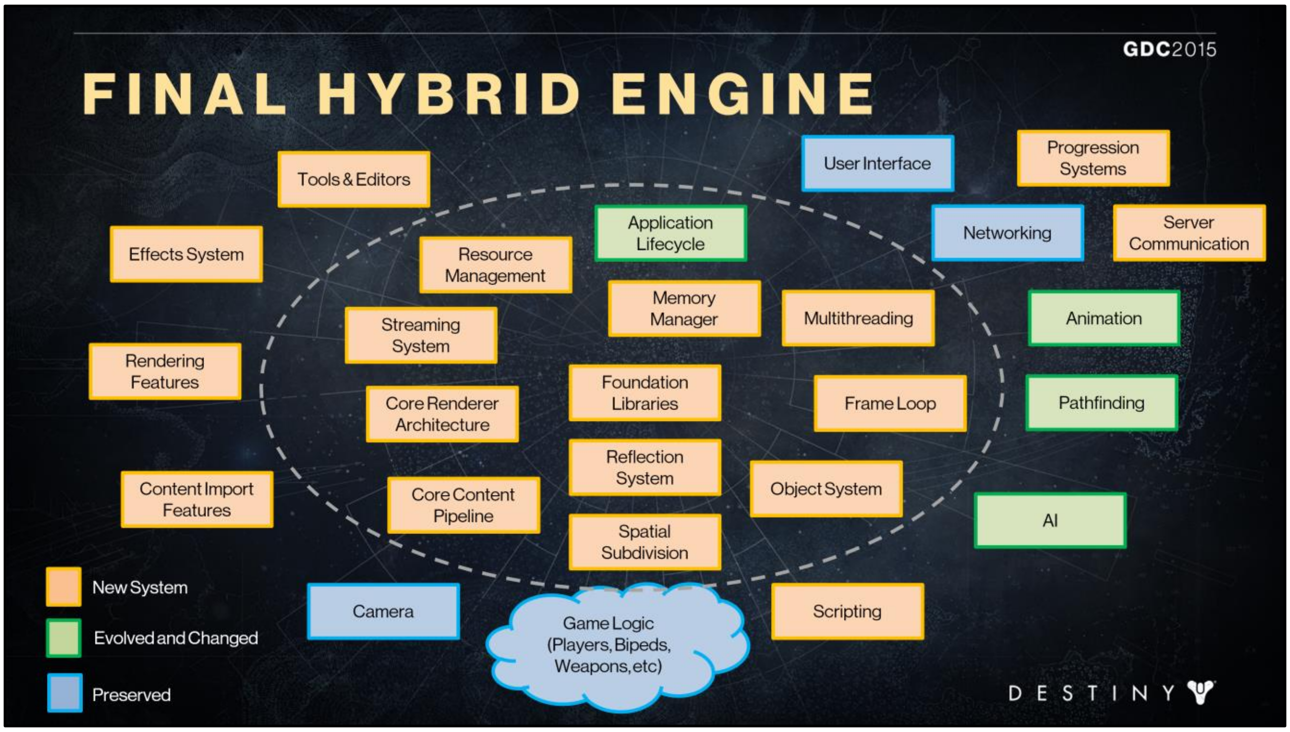
** At this point the legacy engine components are deprecated but still exist in the codebase. You can start removing them if you have time, but there are likely lots of legacy features that you want to continue using into the future so you have to keep around some pieces. The priority here is to reduce complexity by ripping out old dead code. But you have to deal with diminishing returns as you do so.



- Convert and deprecate legacy systems
- Legacy code only persists at the game logic layer
  - Benefits: Faster build times, lower maintenance costs, professional pride
- We didn't reach this stage yet, maybe another 5 years?

At some point you have deprecated enough of the legacy systems that you can make a surgical strike and remove all the old code except the highest level game layer. This is great because it reduces your build times, your maintenance costs, and you can start to feel good about your codebase again.

** Well, we didn't reach this stage yet. I'm not sure if we ever will – but the important point to note here is that you're looking at a flexible path.
At every stage along the way, you have options, and you can choose what's pragmatically right for your project in the next 3 or 6 months.
You're continuously able to make cost/benefit choices about where you spend your time.



This hybrid approach took 3.5 years, and some of the stages along the way were really difficult. We **changed almost all of the core rules of our engine** while working with over 5 MLOC in our codebase, and preserving critical gameplay functionality that couldn't be broken.

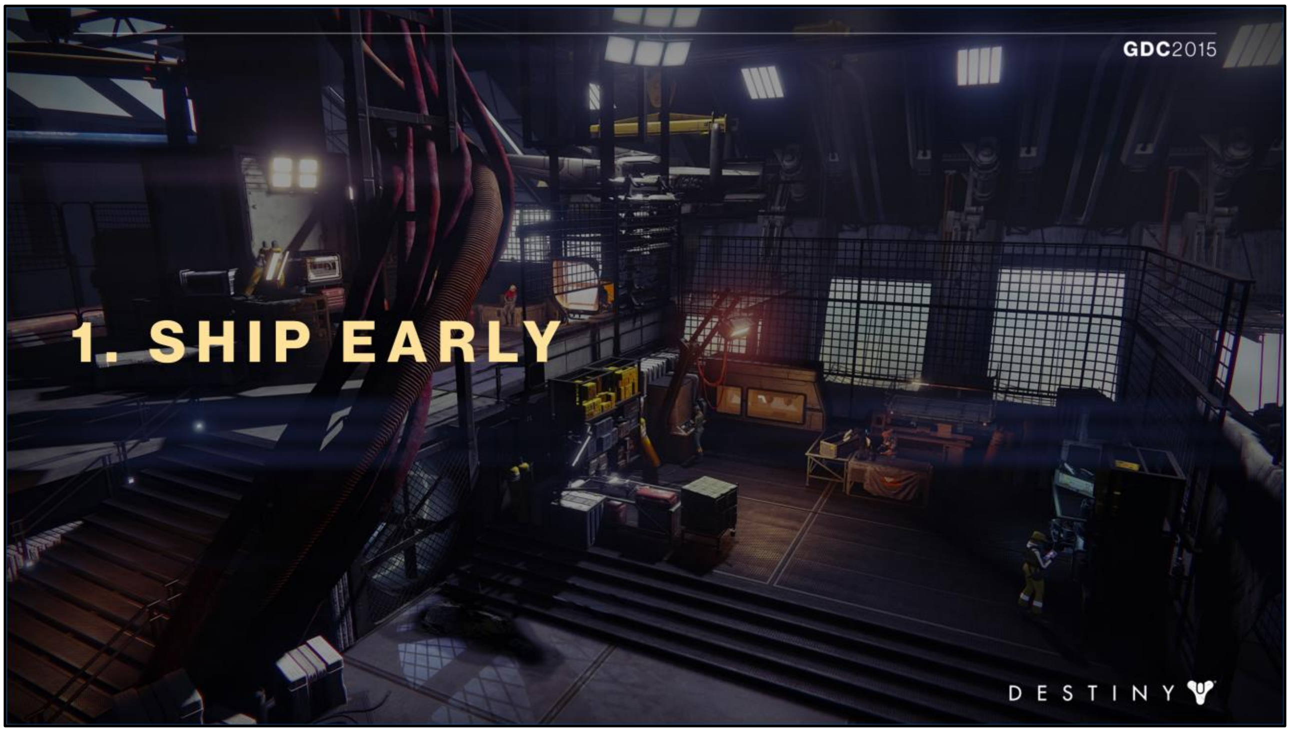
You pay a massive tax in order to have this flexibility. We don't have a control group that built this same engine from scratch. So we don't really know exactly what we gave up, but it sure felt like our engine development was maybe 2-3x times slower than it "should have been".

In the end, the systems in orange here are all new for Destiny. A lot of this stuff is very advanced and could not have been built without changing the core components that it relies on.

The green components came from Halo Reach but they were evolved so much that they're essentially completely changed.

And in blue are the components that were preserved from Halo Reach and then extended for the game of Destiny. The important components in blue here are the **networking stack**, and the **game layer** – each of these is a critical investment that we just couldn't afford to lose.

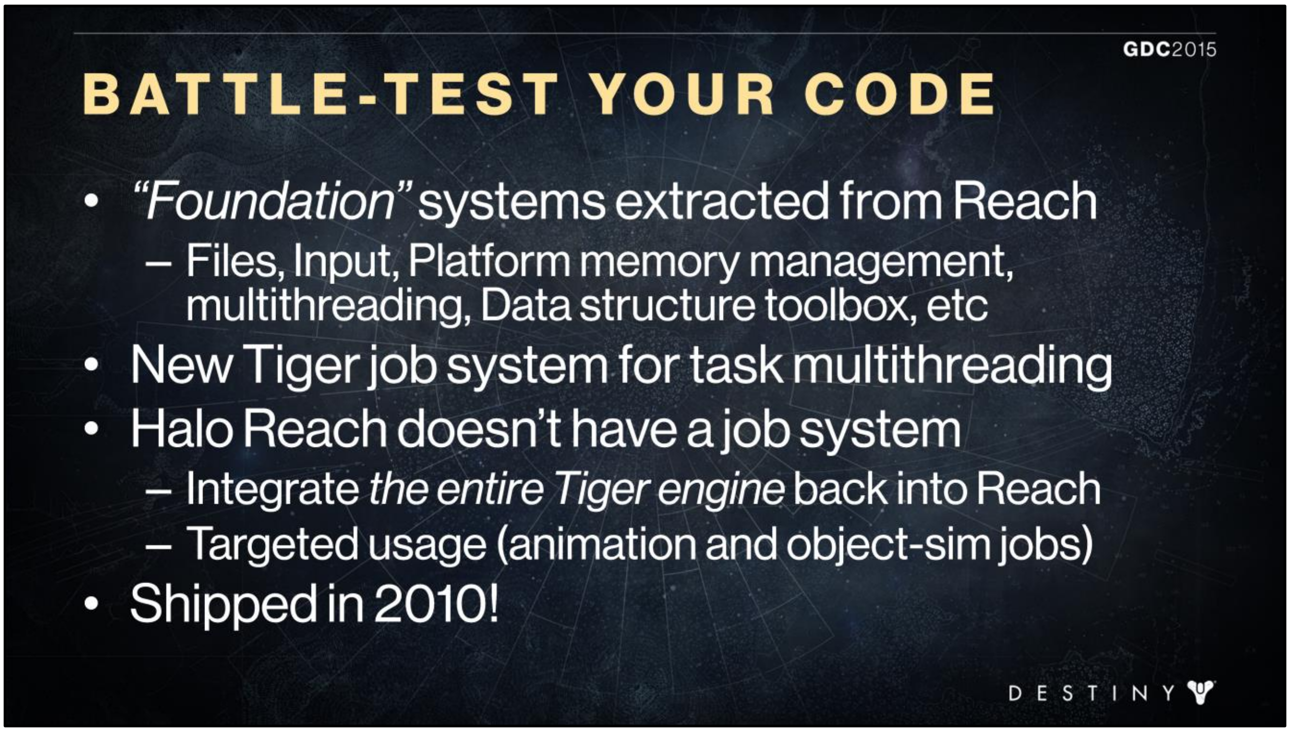
Overall, this was a really long process and we spent a lot of engineering time doing it. But I don’t know any other way we could have ended up with the results that we did.


GDC2015

1. SHIP EARLY

DESTINY

Now we’re going to shift gears, and look at a few specific lessons from the development process.



GDC2015

BATTLE-TEST YOUR CODE

- *"Foundation"* systems extracted from Reach
  - Files, Input, Platform memory management, multithreading, Data structure toolbox, etc
- New Tiger job system for task multithreading
- Halo Reach doesn't have a job system
  - Integrate *the entire Tiger engine* back into Reach
  - Targeted usage (animation and object-sim jobs)
- Shipped in 2010!

DESTINY

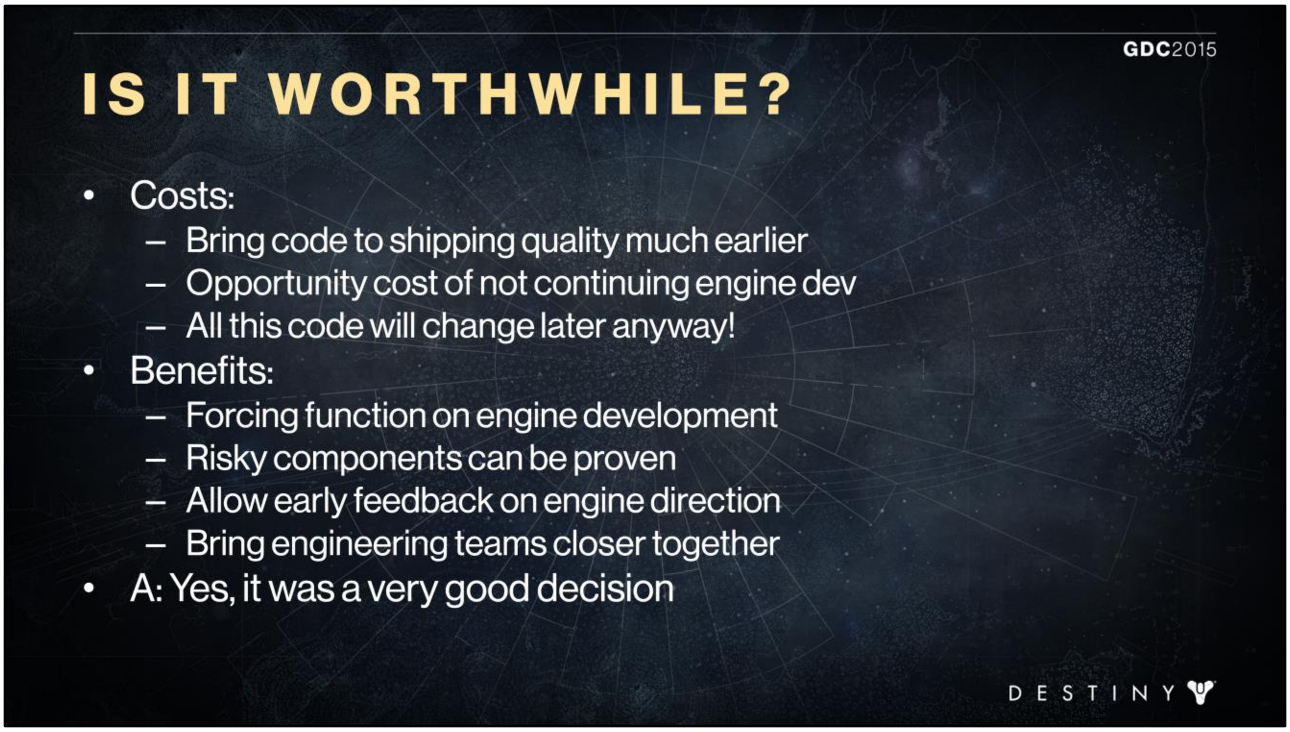
We already talked about how the first stage of the Tiger engine was to extract low-level code from the Halo codebase and refactor it.

This created a new cross-platform *foundation* layer, responsible for systems like files, platform memory management, reusable data structures, and a job system for task based multithreading.

** Wait a minute … Halo Reach is being developed in parallel and it doesn't have a job system. That would be really helpful to take better advantage of the X360's computing power.

So, we decided to integrate the whole Tiger engine, such as it was, back into Halo Reach. This replaced the foundation layer underneath the engine, and enabled us to use the job system for a few targeted purposes. We ended up converting the animation system and part of our object physics simulation to use it.

** So this tech shipped 4 years before the rest of Destiny, in 2010!


GDC2015

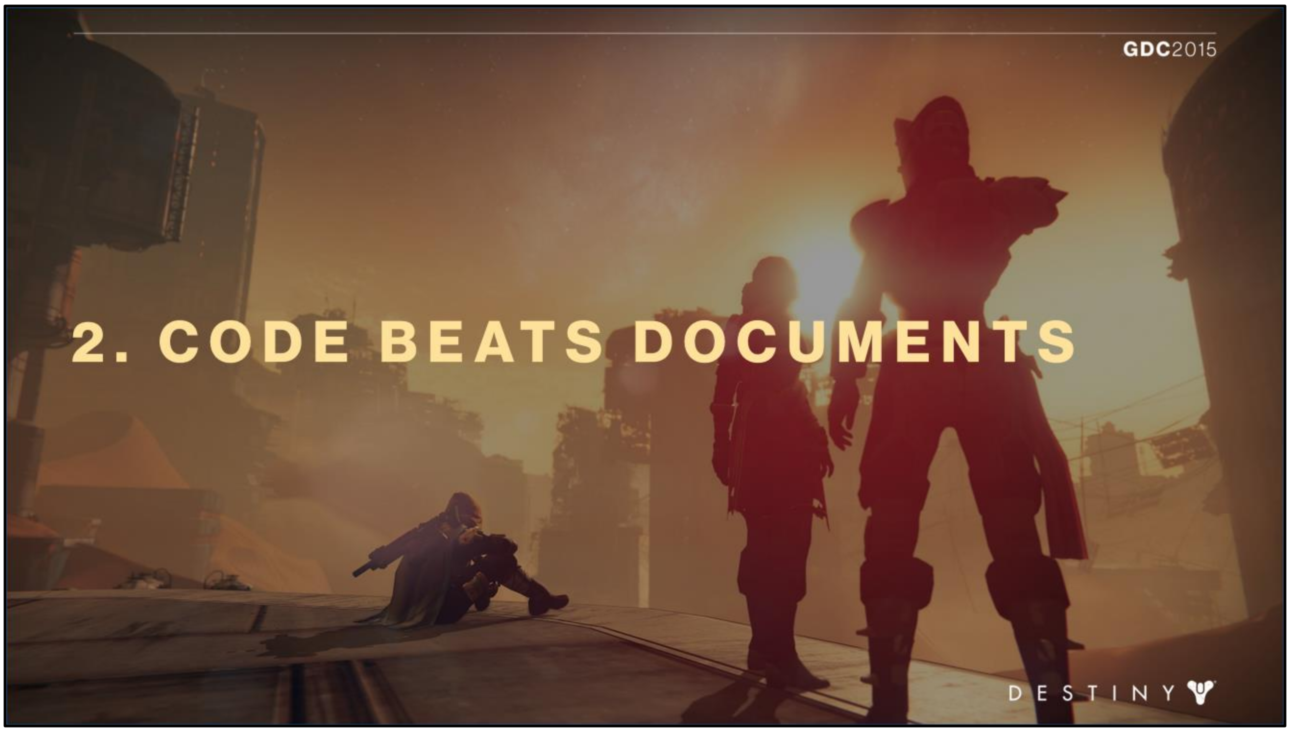
IS IT WORTHWHILE?

- Costs:
  - Bring code to shipping quality much earlier
  - Opportunity cost of not continuing engine dev
  - All this code will change later anyway!
- Benefits:
  - Forcing function on engine development
  - Risky components can be proven
  - Allow early feedback on engine direction
  - Bring engineering teams closer together
- A: Yes, it was a very good decision

DESTINY

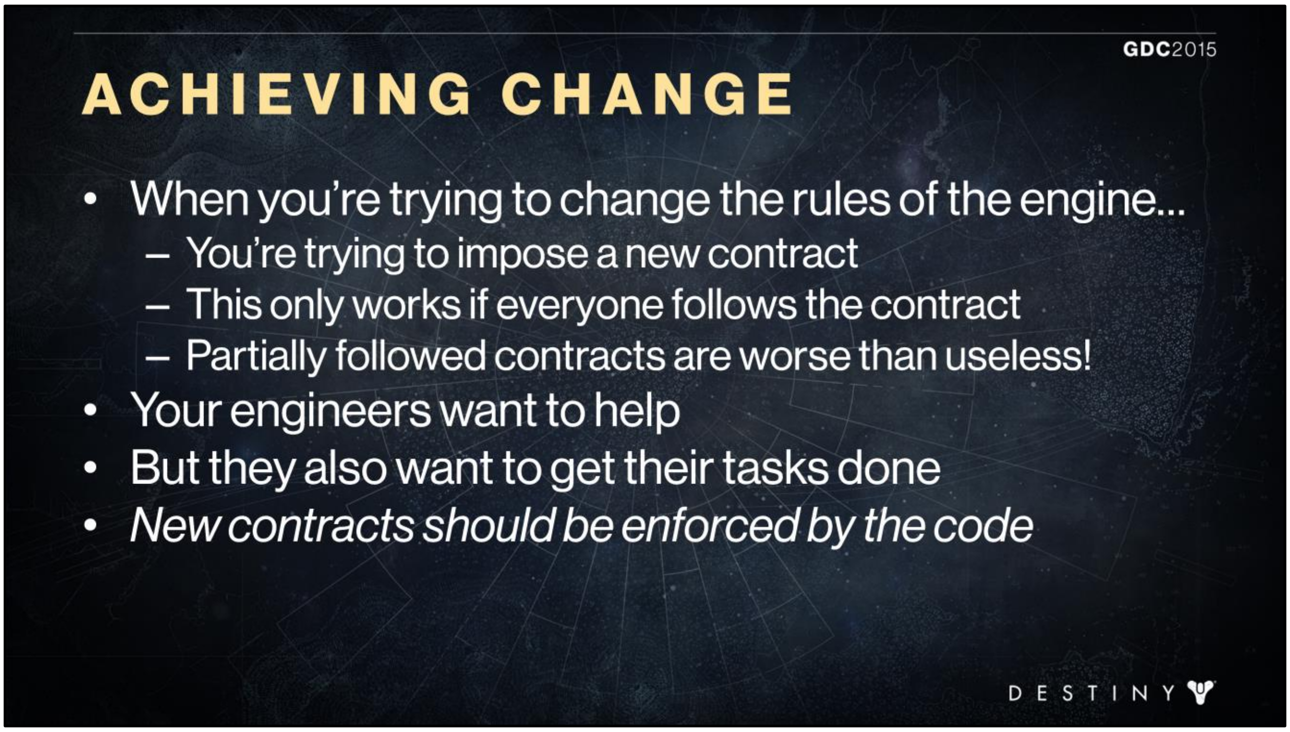
Was it a good idea? As with anything there were costs and benefits.

** With hindsight, even though all of these layers changed massively in the following four years, this turned out to be an important forcing function and one of the best decisions we made.


GDC2015
DESTINY

2. CODE BEATS DOCUMENTS

Next we'll look at another lesson, about how you can achieve change in a large team.


GDC2015

ACHIEVING CHANGE

- When you're trying to change the rules of the engine…
  - You're trying to impose a new contract
  - This only works if everyone follows the contract
  - Partially followed contracts are worse than useless!
- Your engineers want to help
- But they also want to get their tasks done
- *New contracts should be enforced by the code*

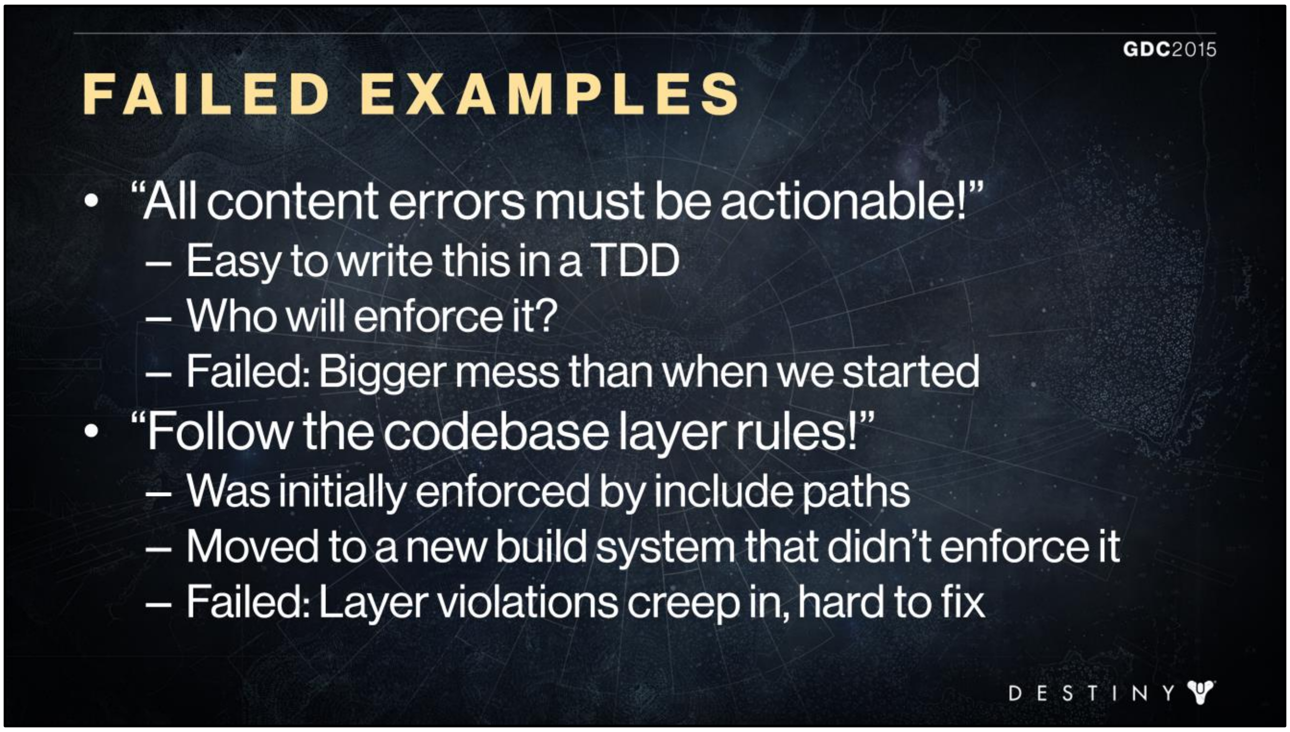
DESTINY

Suppose you're in a large team and you're trying to change the rules of the engine.

** This requires you to impose a new contract, potentially on a lot of existing code, or new code being written. This only works if all of the code successfully gets converted. A partially converted contract would be worse than useless because it introduces confusion, uncertainty and complexity.

** Your engineering team wants to help. [If they don't, you have larger problems that are outside the scope of this presentation.] But they also want to get their tasks done and they likely aren't on the exact same page about the goals of the new engine.

** In this situation, our experience tells us that the new contract you're trying to achieve must be **enforced by code**. You can write about it, or talk about it, as much as you want. But without code enforcement, all that talking is just not going to work.



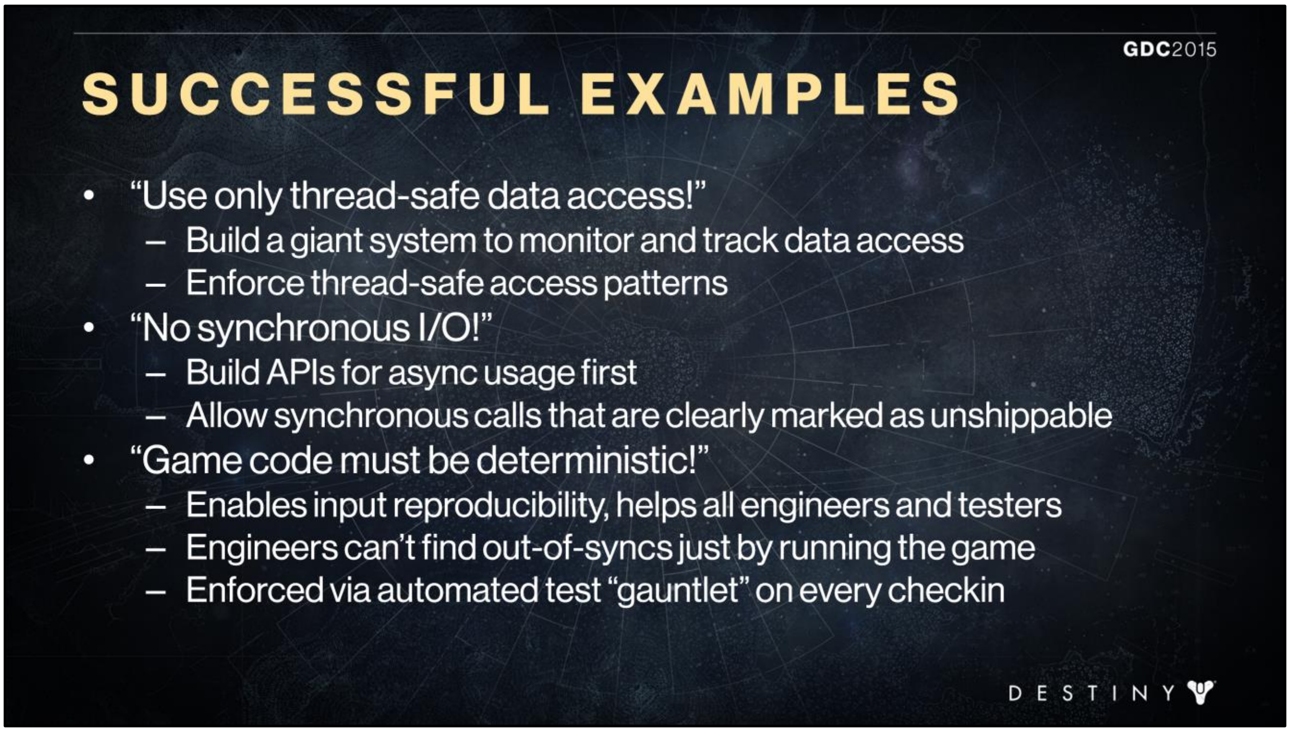
GDC2015

FAILED EXAMPLES

- "All content errors must be actionable!"
  - Easy to write this in a TDD
  - Who will enforce it?
  - Failed: Bigger mess than when we started
- "Follow the codebase layer rules!"
  - Was initially enforced by include paths
  - Moved to a new build system that didn't enforce it
  - Failed: Layer violations creep in, hard to fix

DESTINY

Here are some examples of rules that we tried to follow which weren't successful.



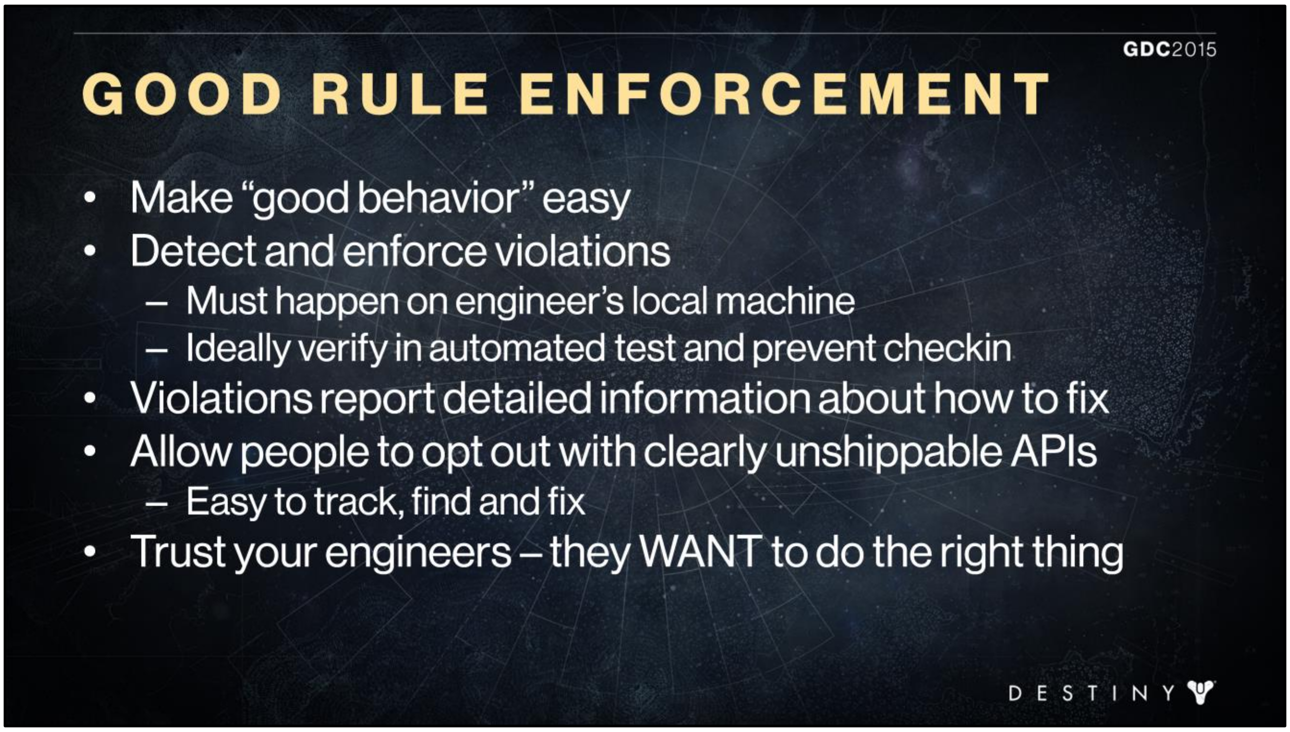
GDC2015

SUCCESSFUL EXAMPLES

- "Use only thread-safe data access!"
  - Build a giant system to monitor and track data access
  - Enforce thread-safe access patterns
- "No synchronous I/O!"
  - Build APIs for async usage first
  - Allow synchronous calls that are clearly marked as unshippable
- "Game code must be deterministic!"
  - Enables input reproducibility, helps all engineers and testers
  - Engineers can't find out-of-syncs just by running the game
  - Enforced via automated test "gauntlet" on every checkin

DESTINY

Here are some examples of rules that we did successfully change across the codebase.


GDC2015

GOOD RULE ENFORCEMENT

- Make "good behavior" easy
- Detect and enforce violations
  - Must happen on engineer's local machine
  - Ideally verify in automated test and prevent checkin
- Violations report detailed information about how to fix
- Allow people to opt out with clearly unshippable APIs
  - Easy to track, find and fix
- Trust your engineers – they WANT to do the right thing

DESTINY

So what does good enforcement of an engine rule look like? The overall idea is to make following the rules the "easy choice".

** That means you need a code system to detect anyone violating the rules, and stop them. Usually by an assert or a build break. This enforcement needs to work on an engineer's local development machine, while they are testing their code, not at some later time on a build farm. Ideally you'll also have enforcement that runs as part of your automated checkin tests so you can prevent violations from being checked in to the source tree.

** Any violation needs to report DETAILED information about how to fix it. And not just information for experts! Your audience is a grumpy programmer who doesn't know anything about this system, and they have to be able to understand what's going on and be given step-by-step instructions for how to fix it.

** Sometimes it's not possible to fix a violation. Maybe it's a subtle problem. Maybe there's a playtest build and you just have to bypass the enforcement temporarily. For a number of reasons any enforcement needs to have a local opt-out mechanism. Not a global opt-out – that's very important – otherwise the system gets checked in turned off, and then the onus is on you to get back down to zero errors before you can turn it on again. These local opt-outs needs to be easy to track, find and fix.

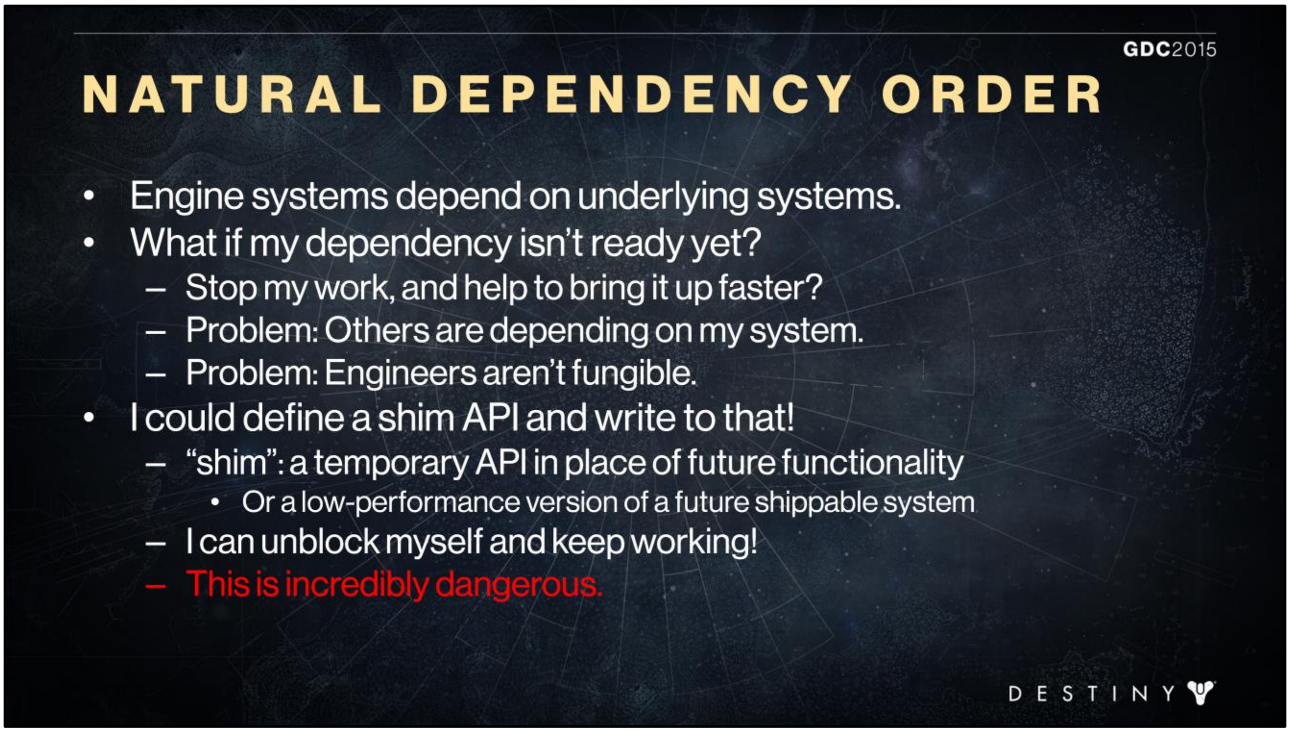
** The overall message here is Trust, but Verify. Your engineers want to do the right thing, you just have to make it easy for them.

3. BRING UP IN ORDER

GDC2015

DESTINY

Now we're going to talk about implementation order of engine systems.



GDC2015

NATURAL DEPENDENCY ORDER

- Engine systems depend on underlying systems.
- What if my dependency isn't ready yet?
  - Stop my work, and help to bring it up faster?
  - Problem: Others are depending on my system.
  - Problem: Engineers aren't fungible.
- I could define a shim API and write to that!
  - "shim": a temporary API in place of future functionality
    - Or a low-performance version of a future shippable system
  - I can unblock myself and keep working!
  - This is incredibly dangerous.

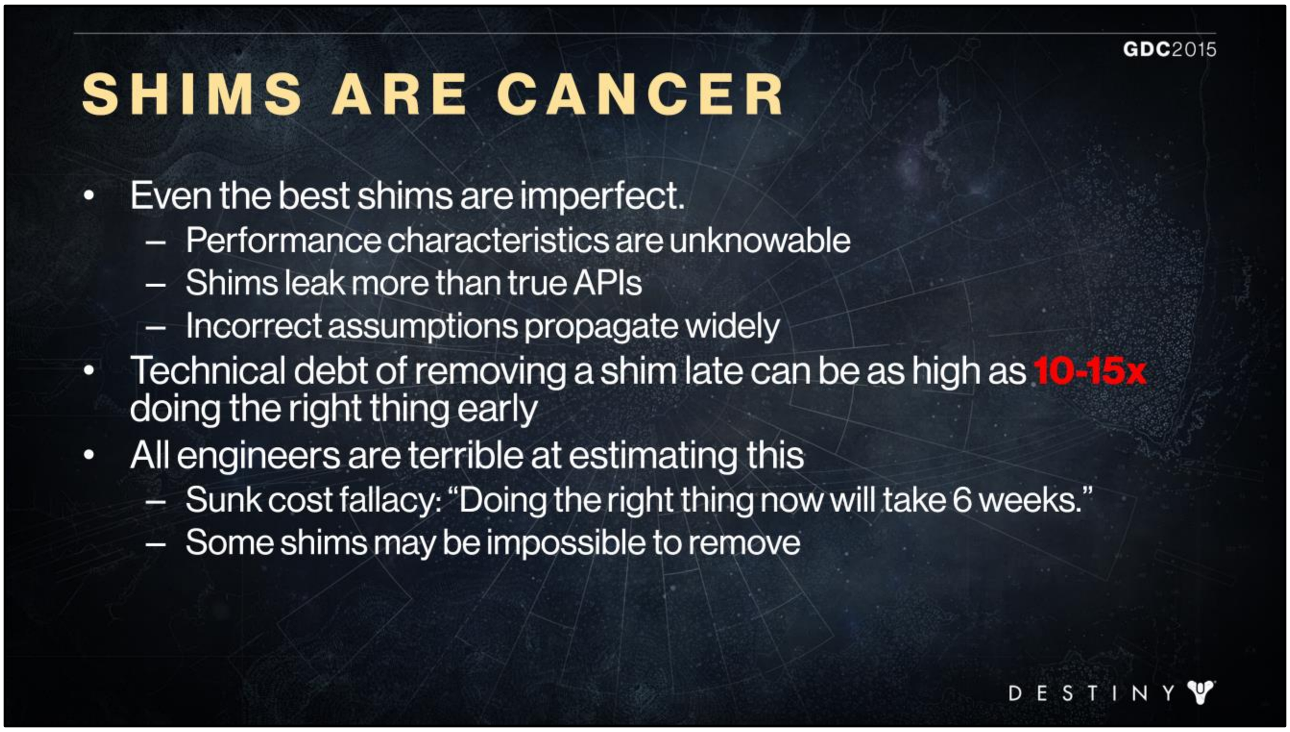
DESTINY

Any new engine system depends on underlying technology. Suppose I'm working on a new feature, and I depend on the widget system. But Bob is still working on the widget system and it's not ready yet. As an engineer I have a couple of options.

** I can stop my work, and help Bob out. Maybe the two of us can bring up the widget system faster, and then I'll be unblocked.
** Okay, but designers are depending on **my** work, and I don't want to slow them down.
Also, I don't understand very much about widgets, so I wouldn't be anywhere near as effective as Bob is at writing widget code.

** This is a pretty common problem. One way to solve it is for me to work around this dependency by writing my own shim API. This could be a new temporary API that I can call into as a placeholder for Bob's future widget system. Or, maybe there is a widget prototype that exists and I can use it, but it's kind of crappy, a low-performance version that will eventually get removed and replaced with a later production-ready version.

Either way, I can unblock myself and keep writing code and making decisions! Great!
** Well, this is actually incredibly dangerous.


GDC2015

SHIMS ARE CANCER

- Even the best shims are imperfect.
  - Performance characteristics are unknowable
  - Shims leak more than true APIs
  - Incorrect assumptions propagate widely
- Technical debt of removing a shim late can be as high as **10-15x** doing the right thing early
- All engineers are terrible at estimating this
  - Sunk cost fallacy: "Doing the right thing now will take 6 weeks."
  - Some shims may be impossible to remove

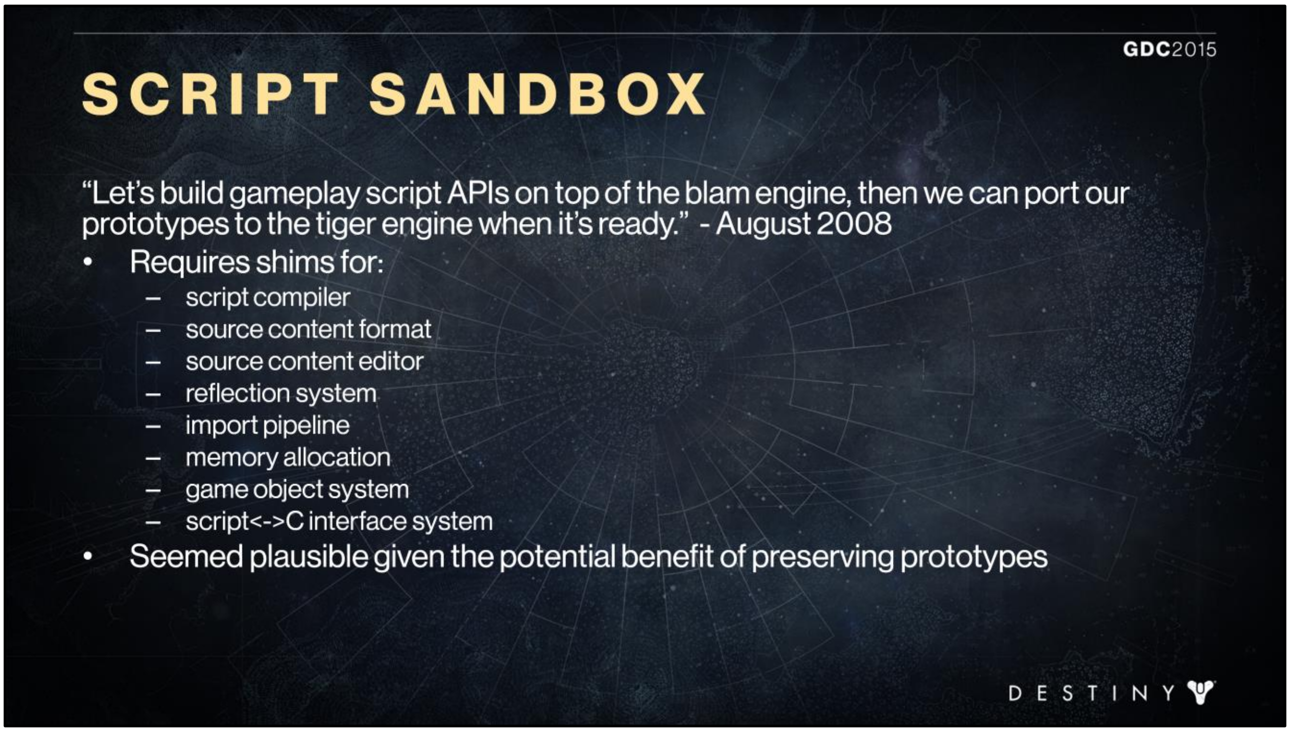
DESTINY

The reason it's dangerous is because you are building a system on top of something that isn't real.
- If you don't have actual code backing your API, then you don't know what patterns in your high level code are structurally unshippable because they can never be made performant.
- Another reason is that shims tend to have leaky abstractions. They can have edge cases and failure modes that cause complexity to leak out into your higher level system.
- But the main reason is that shims are just full of incorrect assumptions – about data structures, or calling conventions, or data lifetime.

** All of these traits can propagate very quickly throughout a codebase. The technical debt incurred by removing a shim late can be just astronomical.

** And it's very easy for us as engineers to underestimate these costs. I'm sure you've heard these phrases before:
- "I totally planned for that. We can just drop that system in whenever it's ready."
- "Doing the right thing now will take 6 weeks. We don't have time for that in the current milestone."
- And my favourite: "Well, I guess it's too late now. We're going to have make the minimum hacks required to ship it."

We do have some examples where shims were effective. But the examples where it failed are really bad. Let’s take a look at one of those.



GDC2015

SCRIPT SANDBOX

"Let's build gameplay script APIs on top of the blam engine, then we can port our prototypes to the tiger engine when it's ready." - August 2008

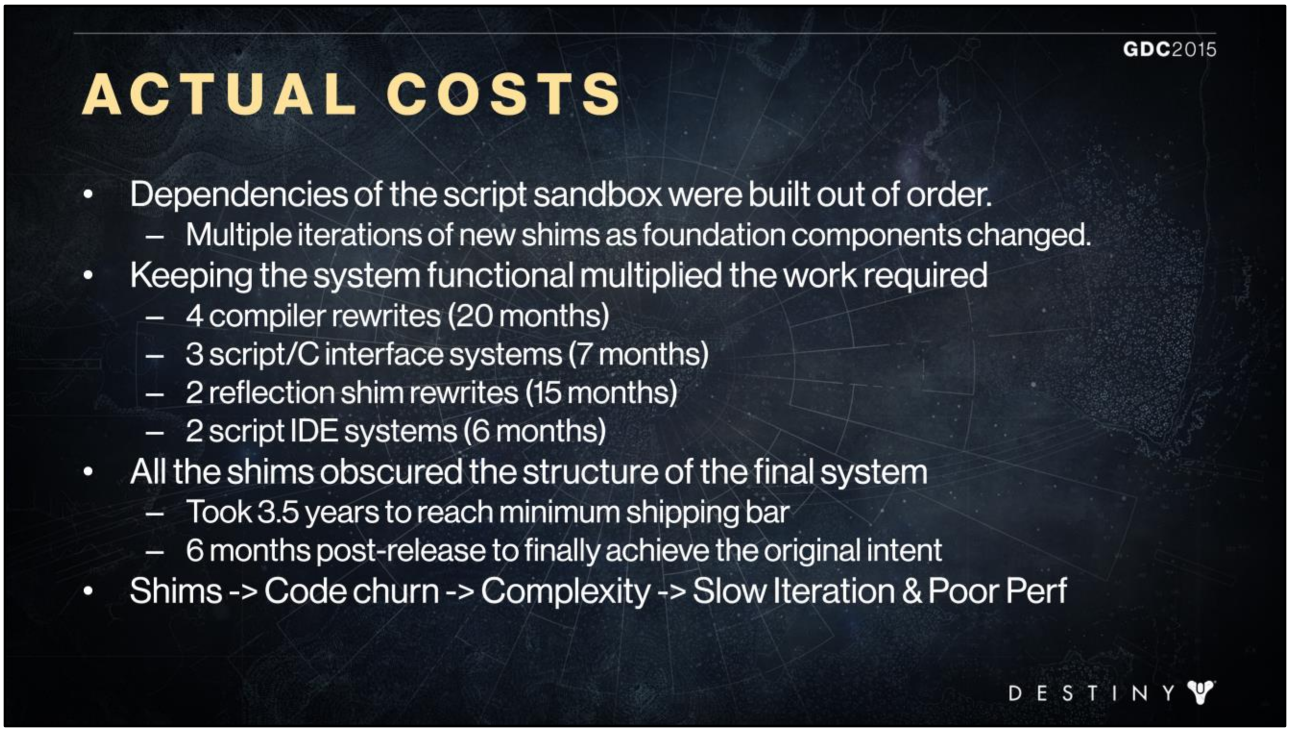
- Requires shims for:
  - script compiler
  - source content format
  - source content editor
  - reflection system
  - import pipeline
  - memory allocation
  - game object system
  - script<->C interface system
- Seemed plausible given the potential benefit of preserving prototypes

DESTINY

You might remember from earlier that we built a scripting engine very early on in the Tiger project. The goal was to build some gameplay script APIs on top of the Halo Reach engine, then do a bunch of gameplay investigations, and port these prototypes by replacing the engine APIs with the Tiger engine when it was ready.

** Well, in order to do that we have to make a lot of assumptions all the way up and down the engine stack. We need shims for : X, Y, Z

** That's a lot of assumptions. But, we thought it seemed plausible, particularly given the potential benefit of preserving all our prototypes forward into the future. Besides, if it wasn't working we could always throw away the prototypes.



GDC2015

ACTUAL COSTS

- Dependencies of the script sandbox were built out of order.
  - Multiple iterations of new shims as foundation components changed.
- Keeping the system functional multiplied the work required
  - 4 compiler rewrites (20 months)
  - 3 script/C interface systems (7 months)
  - 2 reflection shim rewrites (15 months)
  - 2 script IDE systems (6 months)
- All the shims obscured the structure of the final system
  - Took 3.5 years to reach minimum shipping bar
  - 6 months post-release to finally achieve the original intent
- Shims -> Code churn -> Complexity -> Slow Iteration & Poor Perf

DESTINY

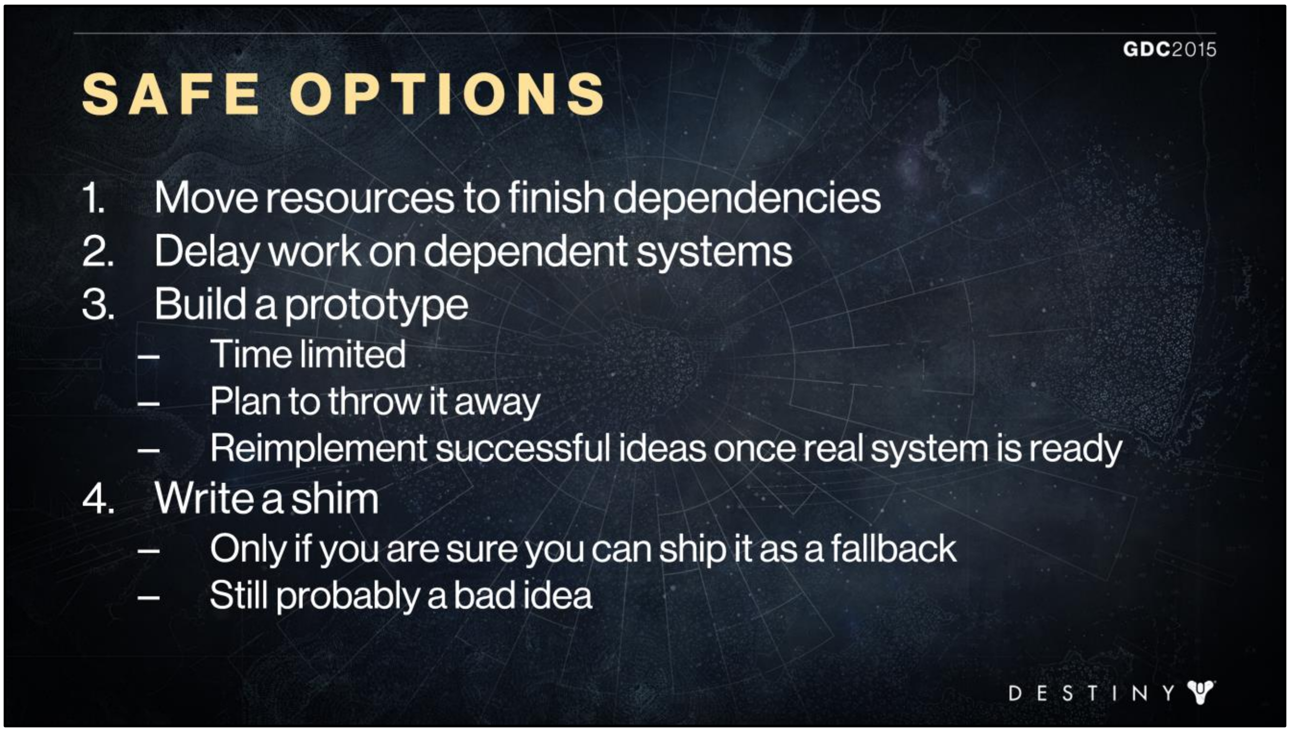
Well, what actually happened is that all the engine dependencies underneath the script sandbox wound up being built out-of-order. So it wasn't just that the Tiger engine APIs arrived late. Rather we would replace a shim, and then realize we needed to replace it again later because that shim needed to get rewritten due to ANOTHER shim underneath it changing.

** We basically ended up writing 4 separate script compilers, in 2008, 2010, 2012 and 2014.

The details aren't super important, and they are very hard to quantify anyway. The main point of this slide is to convince you that I'm really not kidding about technical debt that costs 10-15x what you would expect.

** At every point along the way we were making very calm, rational decisions based on sunk costs and what we felt remained to be shippable. But the costs kept creeping up on us, because all of the shims obscured the structure of the final system. We always felt we were a lot closer to a shippable system than we really were.

** We ended up bogged down in this vicious cycle of shims leading to code churn, leading to complex systems, which resulted in slow iteration times and poor performance.


GDC2015

SAFE OPTIONS

1. Move resources to finish dependencies
2. Delay work on dependent systems
3. Build a prototype
   – Time limited
   – Plan to throw it away
   – Reimplement successful ideas once real system is ready
4. Write a shim
   – Only if you are sure you can ship it as a fallback
   – Still probably a bad idea

DESTINY

Ok, so the obvious question is, how can we handle this situation safely? Well, there are some patterns that we've found to be safe.

- ** Like we talked about, we can move resources around. I can go help Bob finish the widget system.
- ** Or, we could just delay my system. I can go and work on something else until Bob is ready for me.

** Now we get into more complex options. I could build a prototype of my system. We're not talking about a game prototype but an engine prototype here. This only seems to be useful if you put tight constraints on it. You have to limit the time you spend. You have to plan to throw it away and reimplement any successful ideas, after your engine dependencies are ready. Given those constraints, it does seem to work, but it's still less efficient than just waiting.

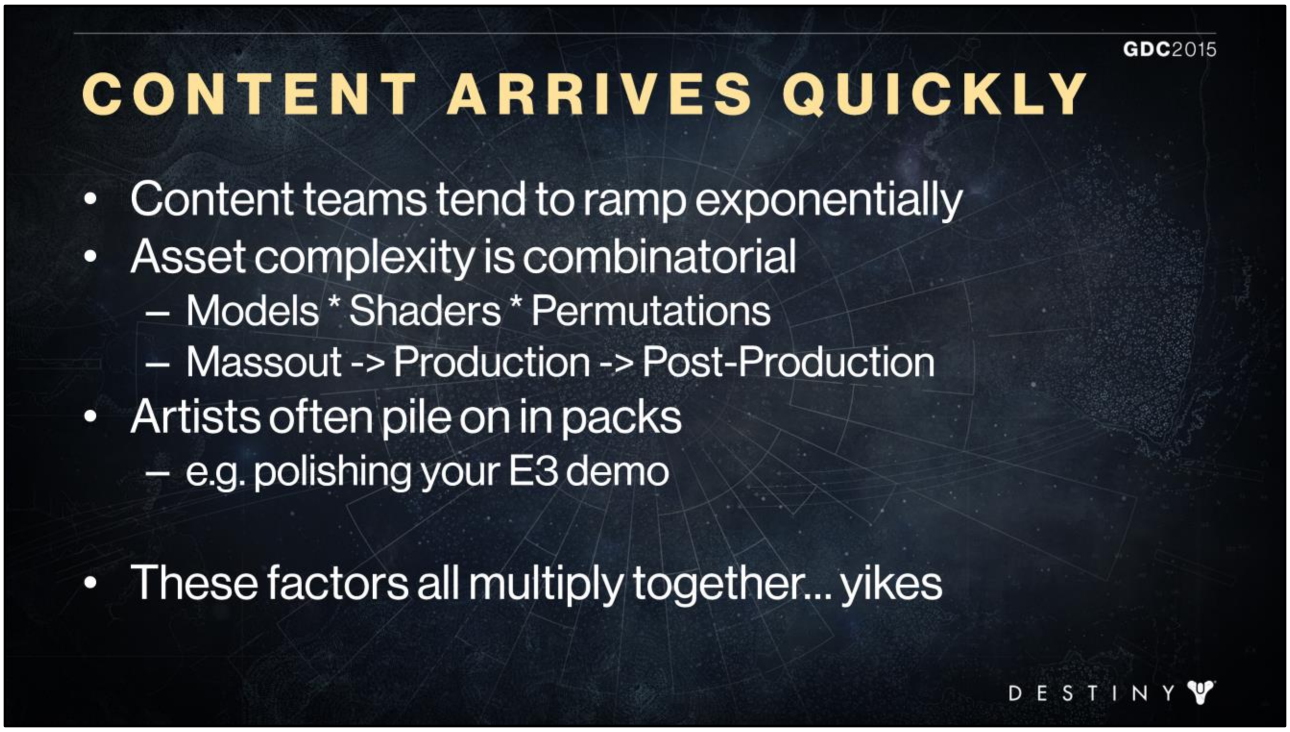
** And, of course, you can write a shim to unblock yourself. The trick here is to be absolutely certain that you can ship this shim if necessary, as a fallback. And even then, it's still probably a bad idea and you really shouldn't do it.

GDC2015

4. CONTENT IS NON-LINEAR

DESTINY

The next lesson we’re going to talk about is how to handle content scaling.



GDC2015

CONTENT ARRIVES QUICKLY

- Content teams tend to ramp exponentially
- Asset complexity is combinatorial
  - Models * Shaders * Permutations
  - Massout -> Production -> Post-Production
- Artists often pile on in packs
  - e.g. polishing your E3 demo

- These factors all multiply together... yikes

DESTINY

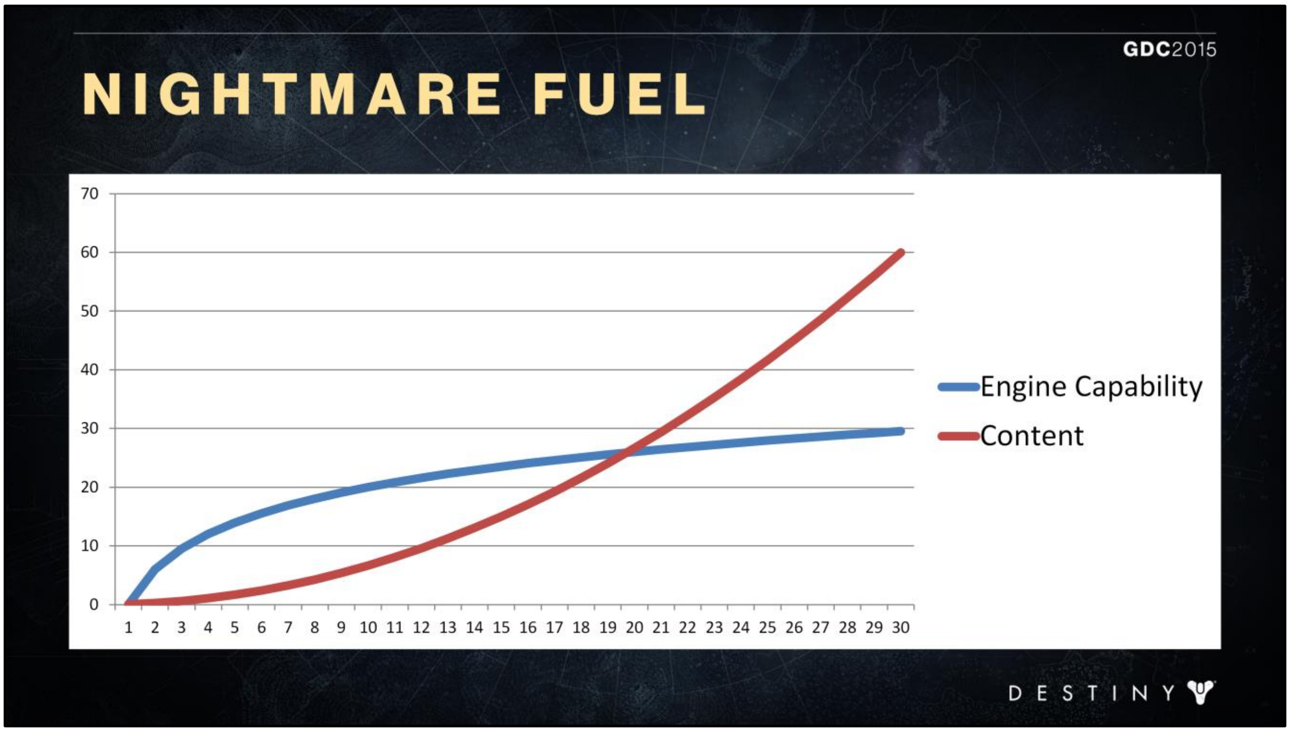
The volume of source content in your game is hard to predict ahead of time. But it's important to try and predict this volume because it has a big impact on the scalability of your content processing pipeline. One thing we can definitely say for sure - it tends to arrive very quickly.

** One reason for this is that your content team is likely to ramp up as you move into production. If you're part of a multiple project studio, you can go from a small number of artists and designers to a large number quite fast.

** Another factor is the complexity of individual assets.
It's possible for you to build a combinatorial system that has complexity proportional to (models x shaders x permutations) and not realize it at first.
** Your content will go through several changes in fidelity as you go from massout to production, and then from production to post-production.
So as you add, for example, more shaders and more permutations per model, you can end up with breathtaking increases in asset complexity, again, very quickly.

** The last factor is that artists tend to travel in packs. At least at Bungie, our artists are very good at bringing a team to bear on a single experience and making it look amazing. For example an E3 demo or something like that. This can also increase asset complexity very quickly.

** All of these factors multiply together to do something like … this…



NIGHTMARE FUEL

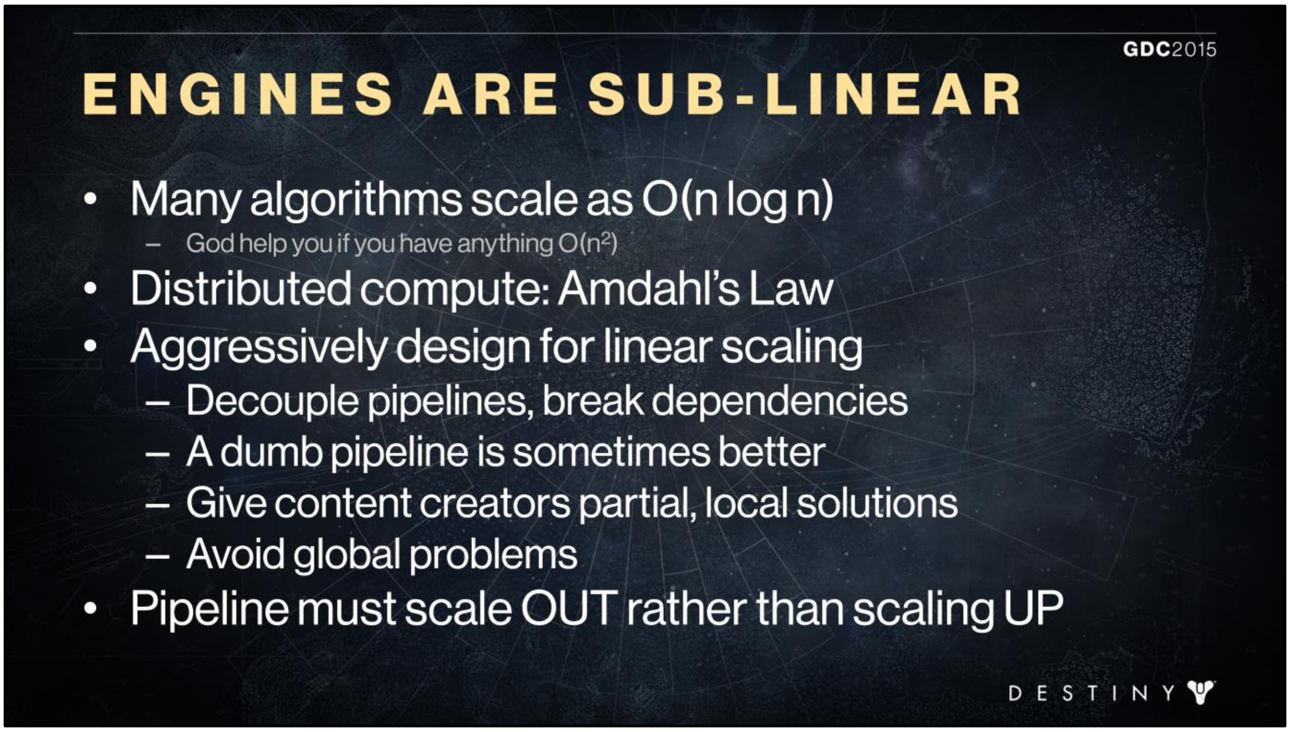
GDC2015

DESTINY

Everyone raise your hands if this graph gave you a flashback to some disastrous situation that you encountered in your career.

Just like climate change, this is a problem that our monkey brains are very bad at solving.
By the time you're in the danger zone, it's already too late to fight these trends, you are stuck trying to mitigate, and unable to keep pace with the incoming volume of content.

Every engine MUST think about this up front and try to get out ahead of the curve.



GDC2015

ENGINES ARE SUB-LINEAR

- Many algorithms scale as O(n log n)
  - God help you if you have anything $O(n^2)$
- Distributed compute: Amdahl's Law
- Aggressively design for linear scaling
  - Decouple pipelines, break dependencies
  - A dumb pipeline is sometimes better
  - Give content creators partial, local solutions
  - Avoid global problems
- Pipeline must scale OUT rather than scaling UP

DESTINY

The big problem here is that while content is super-linear, engine capacity tends to be sub-linear.
Many algorithms used in graph operations scale as O(n log n), which isn't great.
You can try to use distributed computing but you'll quickly run into Amdahl's Law.

** What you can do is aggressively design your pipeline to scale linearly.
- This means decoupling your content types and trying to break dependencies between them.
- ** Sometimes it may be better to have a dumb content pipeline and push a bit more work to the runtime engine or content creator. If you can keep several types of content isolated and prevent them from talking to each other, it may not be convenient for your content creator, but it could be the right thing to reduce overall system complexity.
- ** Another strategy is that sometimes you can get your content creators to use a partial workflow, for example instead of implementing an expensive incremental lightmap computation, just give your artists the choice of whether to re-bake lightmaps or not.
- ** All these strategies boil down to trying to avoid global optimization problems.

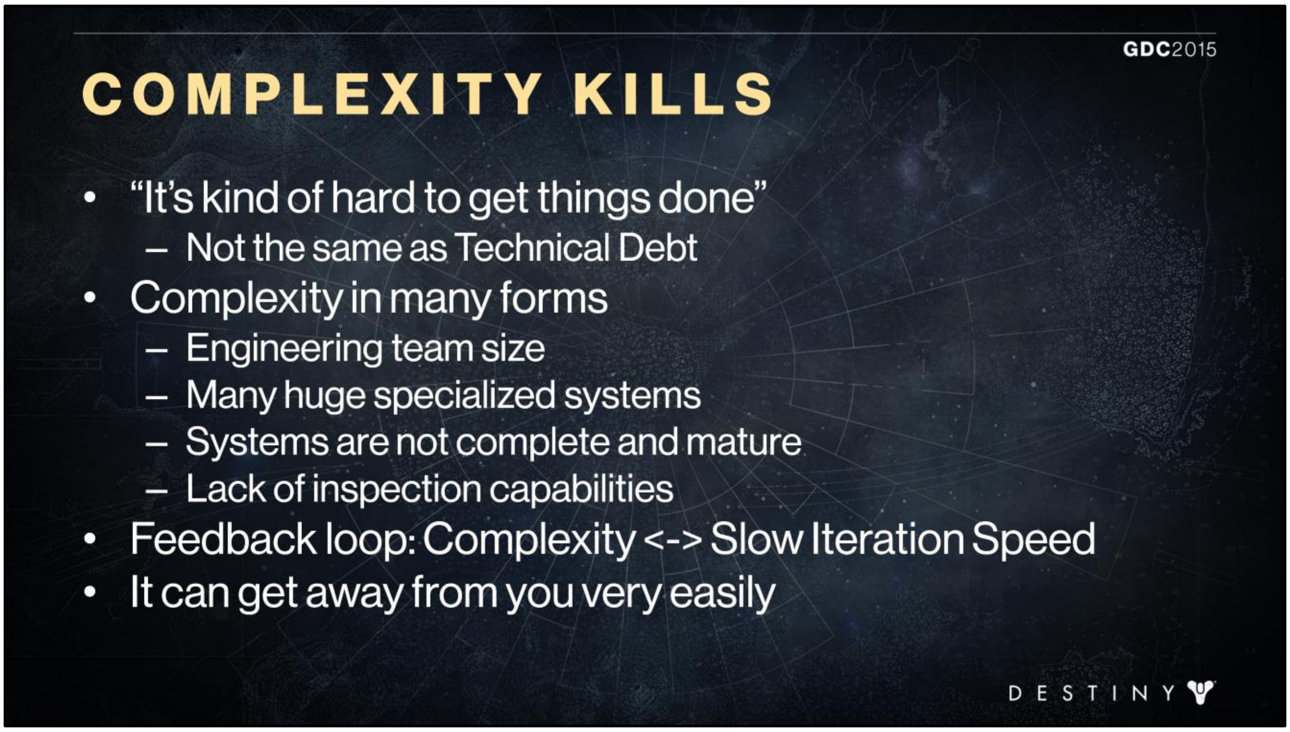
** Many of these principles are actually the same ones as server scalability. You need to take your content processing pipeline and make sure it can scale **OUT**, instead of needing to scale **UP**. If you can do that, you can stay closer to linear processing capacity, and spend more of your time in the sweet spot underneath the content curve.

THE GREAT STRUGGLE

GDC2015

DESTINY

Now, let’s finish up by talking about the epic struggle against software complexity. So far I’ve just touched on this in passing, but really it’s the entire theme of this presentation.


GDC2015

COMPLEXITY KILLS

- "It's kind of hard to get things done"
  - Not the same as Technical Debt
- Complexity in many forms
  - Engineering team size
  - Many huge specialized systems
  - Systems are not complete and mature
  - Lack of inspection capabilities
- Feedback loop: Complexity <-> Slow Iteration Speed
- It can get away from you very easily

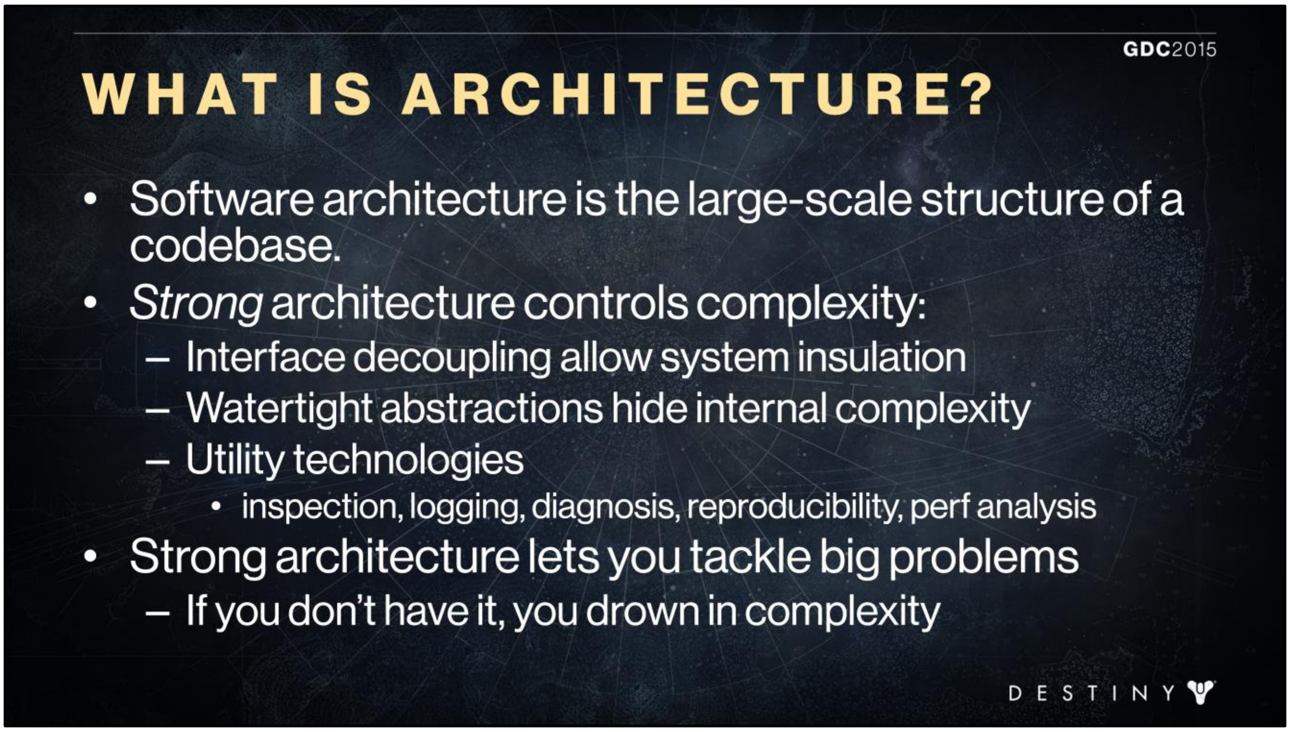
DESTINY

Software complexity manifests itself as a feeling that "it's kind of hard to get things done". Not talking about Technical Debt here, but structural complexity.

** Adding engineers increases complexity. Butcher's Law - "Lines of code = No. of engineers x No. of months, which may have no relation to the complexity of the problem they were asked to solve"
** Modern engines have many huge specialized systems. The Destiny shader system is 100 times more complex than the original Halo shader system that shipped in 2001.
** Another source of complexity is systems that are not finished, which incur a tax on everything around them due to edge cases, error handling and clumsy APIs.
** Lack of inspection means when something goes wrong you have to dive deep into all this complexity, which acts to slow you down.

** All of this complexity leads to a positive feedback loop.
- It's hard to get things done, so you just need to get your code written as quickly as possible, you don't have time for the right way. This leads to more complexity.
- Your iteration speed is slow, so you make smaller numbers of large checkins. You have fewer chances to incrementally refine your code. This leads to more complexity.

** And so the cycle continues. Unless you are continuously paying attention to this, you can get stuck in a very inefficient state. So how can we deal with this?



GDC2015

WHAT IS ARCHITECTURE?

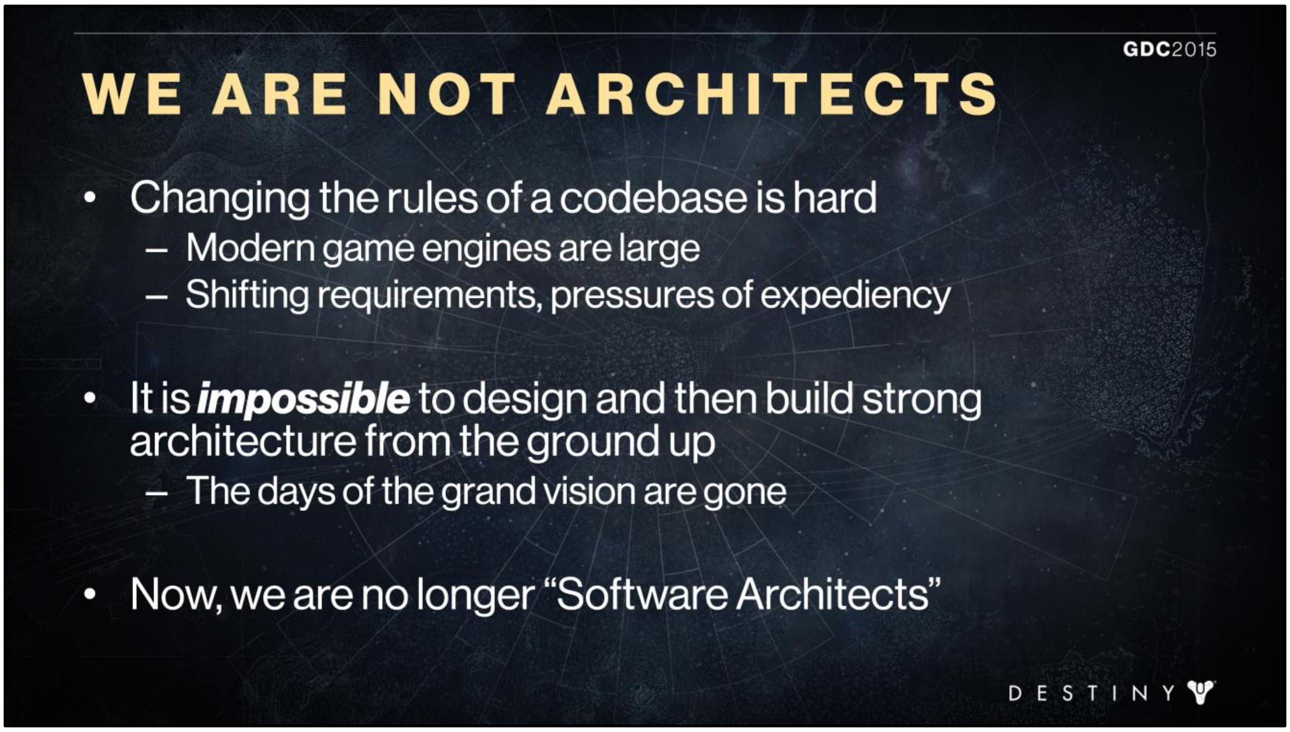
- Software architecture is the large-scale structure of a codebase.
- *Strong* architecture controls complexity:
  - Interface decoupling allow system insulation
  - Watertight abstractions hide internal complexity
  - Utility technologies
    - inspection, logging, diagnosis, reproducibility, perf analysis
- Strong architecture lets you tackle big problems
  - If you don't have it, you drown in complexity

DESTINY

We haven't really defined software architecture yet, but I believe we can now. Software architecture is the large-scale structure of a codebase.

** **Strong** architecture is structure that controls complexity within your engine. It's the set of interfaces that allow you to insulate systems from each other's implementation details. It's the watertight abstractions (as opposed to leaky abstractions) that hide internal complexity and present a simple outward wrapper. It's the set of utility technologies that lets you quickly understand what's going on within your engine. I could go on and on, but we all know what strong architecture looks like.

** So what's the benefit of architecture? It doesn't let you build features faster. It doesn't increase the quality of your game. But it does allow you to tackle big problems that would otherwise be out of reach. If you tackle big problems without strong architecture, you wind up drowning in complexity and unable to complete your project.


GDC2015

WE ARE NOT ARCHITECTS

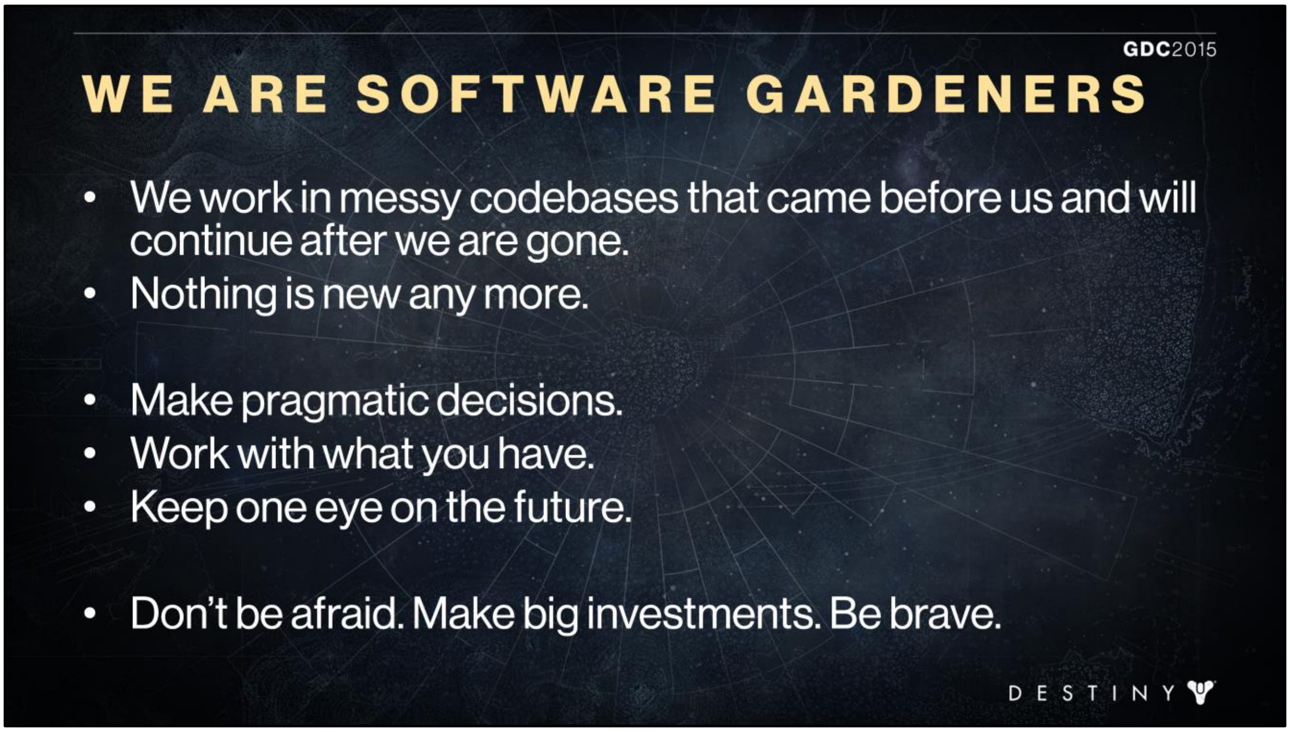
- Changing the rules of a codebase is hard
  - Modern game engines are large
  - Shifting requirements, pressures of expediency
- It is ***impossible*** to design and then build strong architecture from the ground up
  - The days of the grand vision are gone
- Now, we are no longer "Software Architects"

DESTINY

There's just one problem with this. As we've already established, changing the rules of a codebase is hard. You're working with gigantic codebases, under shifting requirements, and constant pressures of expediency. But you don't get to press the reset button. These investments are too large to throw away.

** I believe this means it is literally impossible to design and then build large-scale strong architecture from the ground up. Not any more. Those days are behind us now.

** Now, we are no longer software architects.



GDC2015

WE ARE SOFTWARE GARDENERS

- We work in messy codebases that came before us and will continue after we are gone.
- Nothing is new any more.

- Make pragmatic decisions.
- Work with what you have.
- Keep one eye on the future.

- Don't be afraid. Make big investments. Be brave.

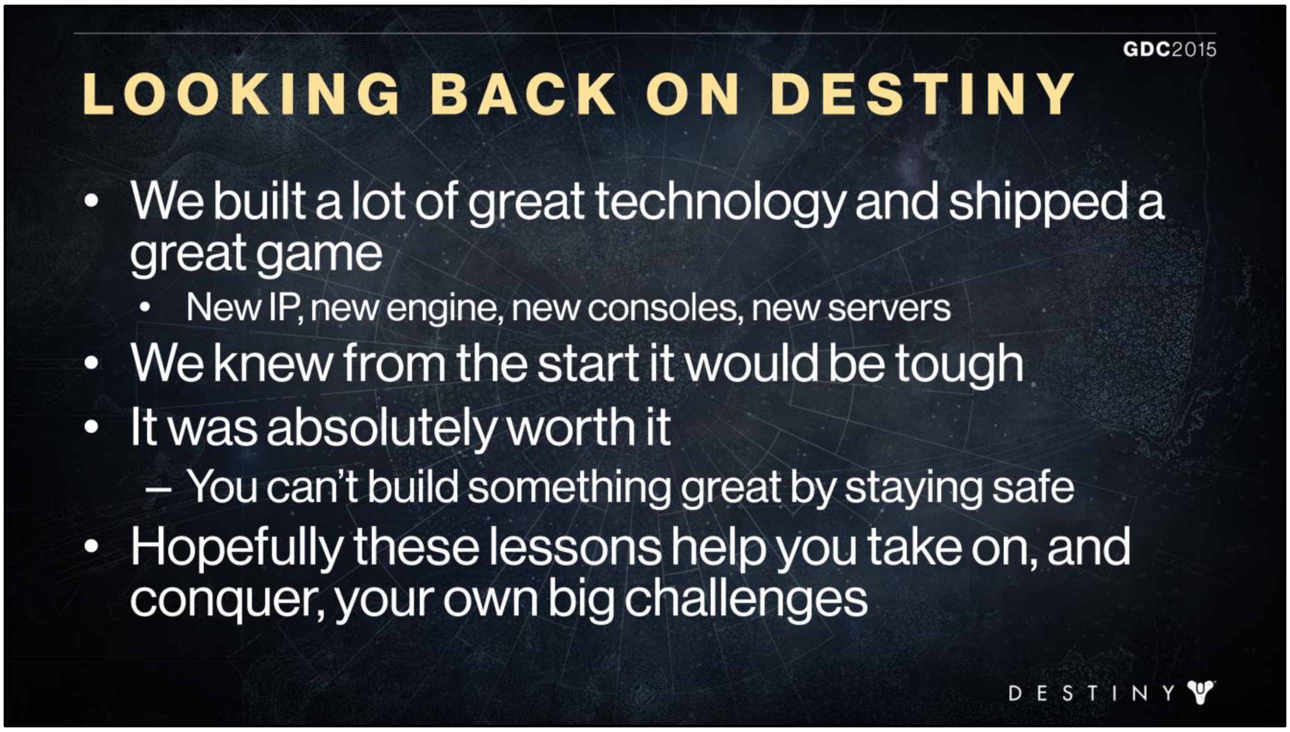
DESTINY

Instead, we should think of ourselves as **software gardeners**.

** We work in environments that are messy, full of sprawling life and complexity.
We spend our days grappling with the limits of our ability to effect change.
** We don't really get to build new things any more. But… there is a joy to be found in the work still.

** Make pragmatic decisions.
Work with what you have.
Stay in the moment but keep one eye on the future.

** And don't be afraid to make big investments. Be brave.


GDC2015

LOOKING BACK ON DESTINY

- We built a lot of great technology and shipped a great game
  - New IP, new engine, new consoles, new servers
- We knew from the start it would be tough
- It was absolutely worth it
  – You can't build something great by staying safe
- Hopefully these lessons help you take on, and conquer, your own big challenges

DESTINY

So when we look back on Destiny, we built a lot of great technology and shipped a great game.

** We knew from the start it would be tough, and it was, but it was absolutely worth it. You can't build something great by staying safe.

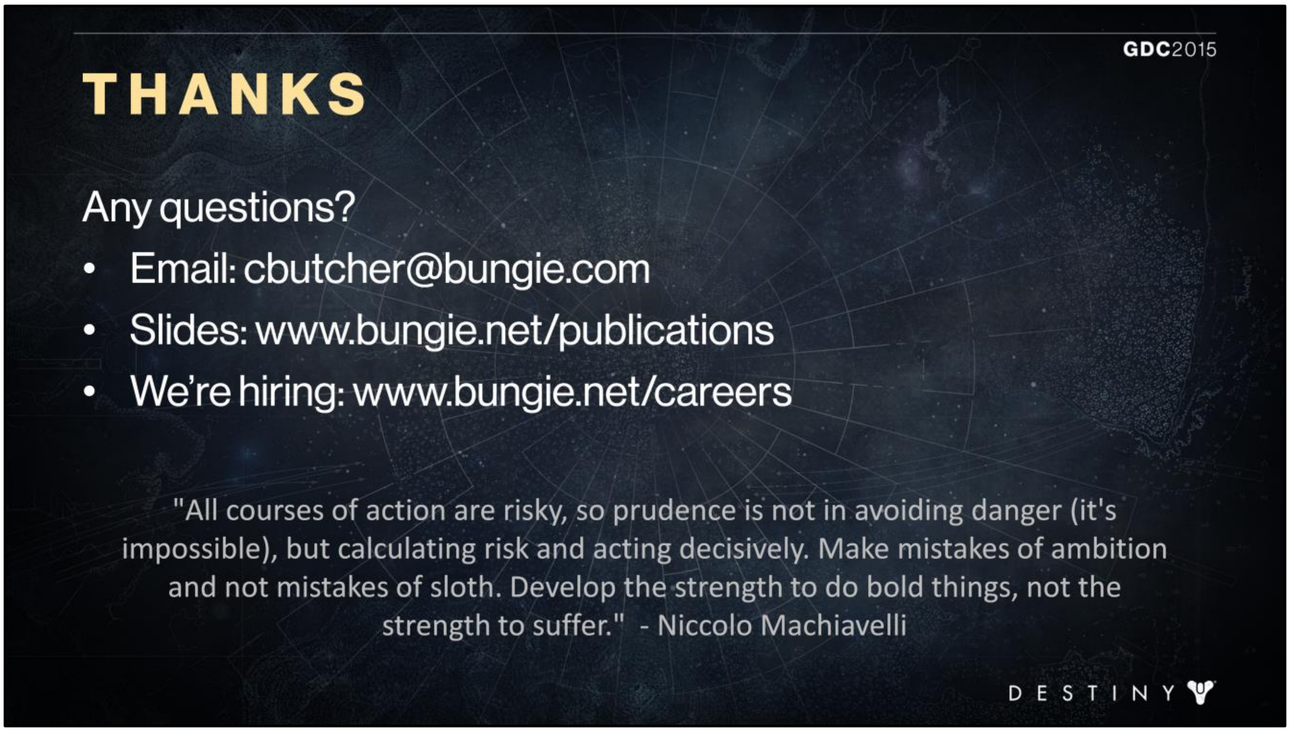
** I hope our experiences can help all of you in taking on your own big challenges, and completing them successfully.

Thanks very much for listening.

WE'RE HIRING

WWW.BUNGIE.NET/CAREERS

CAREERS@BUNGIE.COM


GDC2015

THANKS

Any questions?

- Email: cbutcher@bungie.com
- Slides: www.bungie.net/publications
- We're hiring: www.bungie.net/careers

"All courses of action are risky, so prudence is not in avoiding danger (it's impossible), but calculating risk and acting decisively. Make mistakes of ambition and not mistakes of sloth. Develop the strength to do bold things, not the strength to suffer." - Niccolo Machiavelli

DESTINY

So, that's the end of the talk. If you have any questions or you just want to chat about architecture please drop me a line. Also, if you find these kinds of problems fascinating, and you love working with great people, please consider coming to work with us.

Hopefully we have time for a few questions in person (lol) … please come up to a microphone.