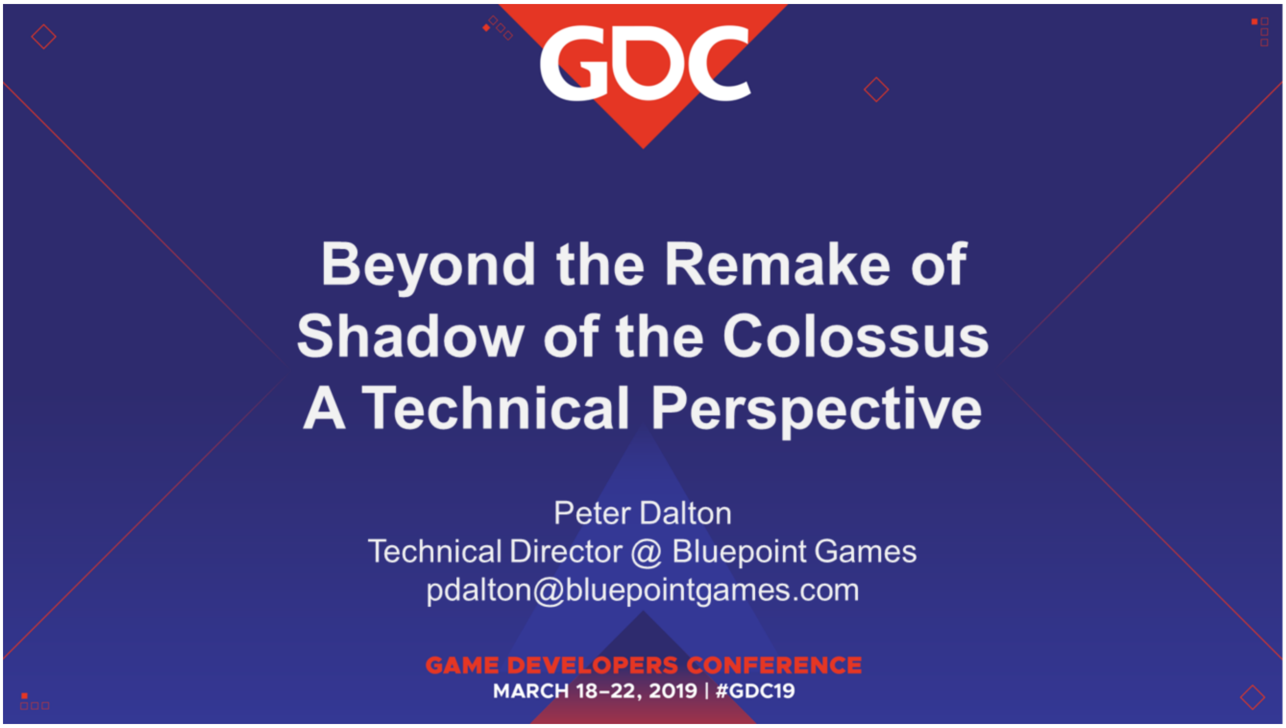


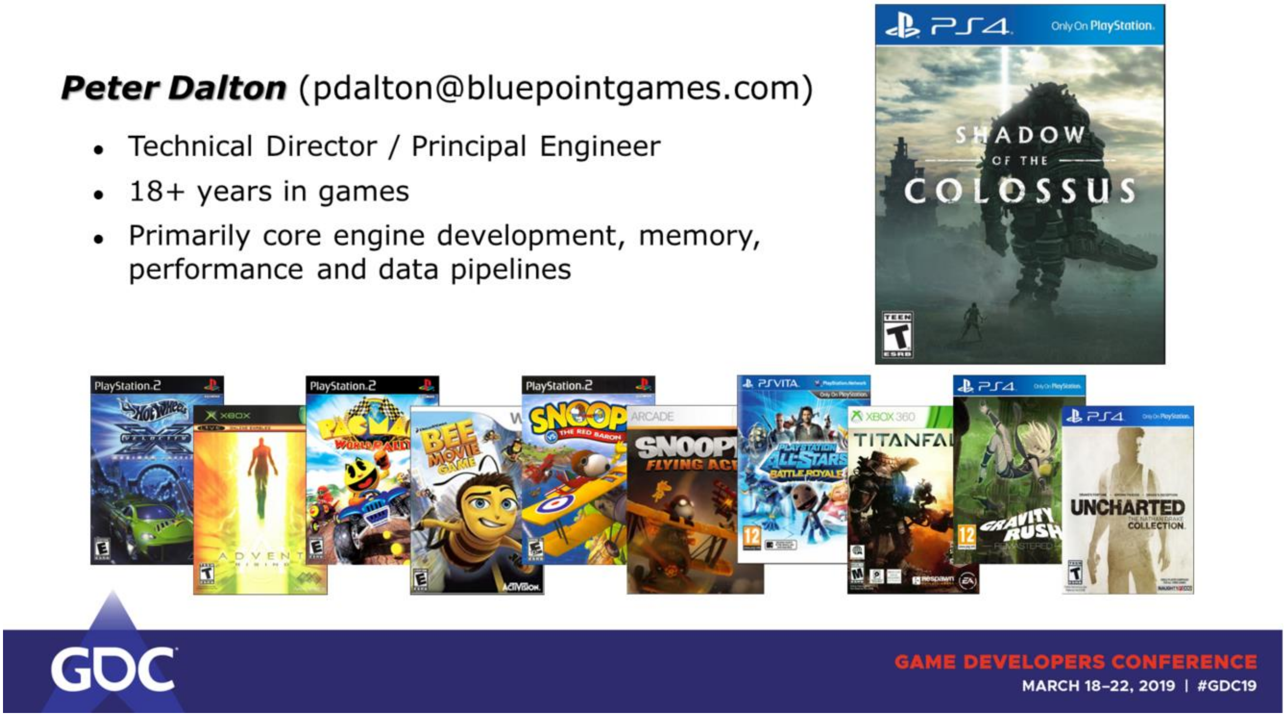
GDC

Beyond the Remake of Shadow of the Colossus A Technical Perspective

Peter Dalton
Technical Director @ Bluepoint Games
pdalton@bluepointgames.com

GAME DEVELOPERS CONFERENCE
MARCH 18–22, 2019 | #GDC19

I would like to start by thanking each of you for attending this talk. There are a lot of great options and I'm flattered that you are here. I will let you know that there are bonus slides at the end of this presentation that I won't have time to cover. However this talk and its slides will be available via the GDC Vault or feel free to reach out to me directly if you are interested. Also please don't forget to fill out your evaluation forms, I need the feedback.



Peter Dalton (pdalton@bluepointgames.com)

- Technical Director / Principal Engineer
- 18+ years in games
- Primarily core engine development, memory, performance and data pipelines

GAME DEVELOPERS CONFERENCE
MARCH 18–22, 2019 | #GDC19

So now that you are here, whether intentional or you just realized you are in the wrong room; you are most likely wondering, who the hell is this guy?

Well, my name is Peter Dalton and I'm the technical director for Bluepoint Games, and as you can tell from the grey in my beard, I've been around for a while.

I've been working in the games industry for over 18 years and I will never leave, I love my job, the insanely hard problems and the talented people I'm surrounded by.

Over the 11+ titles that I have shipped my specialties revolve around core engine development, streaming, memory, performance, I like to believe that my job is to basically, make shit happen.

Bluepoint games

- Bluepoint Games (BP) was founded in 2006
- Started with an original project, quickly moved into remasters, then co-development and now primarily remakes.
- Continually pushing to redefine what a remaster means. From remaster to remake to re-envisioning…
- Quality is the foundation of success.

PS4
Only On PlayStation
SHADOW OF THE COLOSSUS
BLAST FACTOR
GOD OF WAR COLLECTION
ICO THE SHADOW OF THE COLOSSUS COLLECTION
METAL GEAR SOLID HD COLLECTION
PLAYSTATION ALL-STARS BATTLE ROYALE
TITANFALL
flower
GRAVITY RUSH REMASTERED
UNCHARTED THE NATHAN DRAKE COLLECTION

GDC GAME DEVELOPERS CONFERENCE MARCH 18–22, 2019 | #GDC19

So, Bluepoint Games. Perhaps you have heard of us; the Masters of the Remaster, I have to thank Digital Foundry for that title. Seriously the Digital Foundry guys are amazing. Their technical reviews are spot on and Bluepoint is tired of me ranting about how our framerate and frame pacing must be perfect or Digital Foundry will call us out.

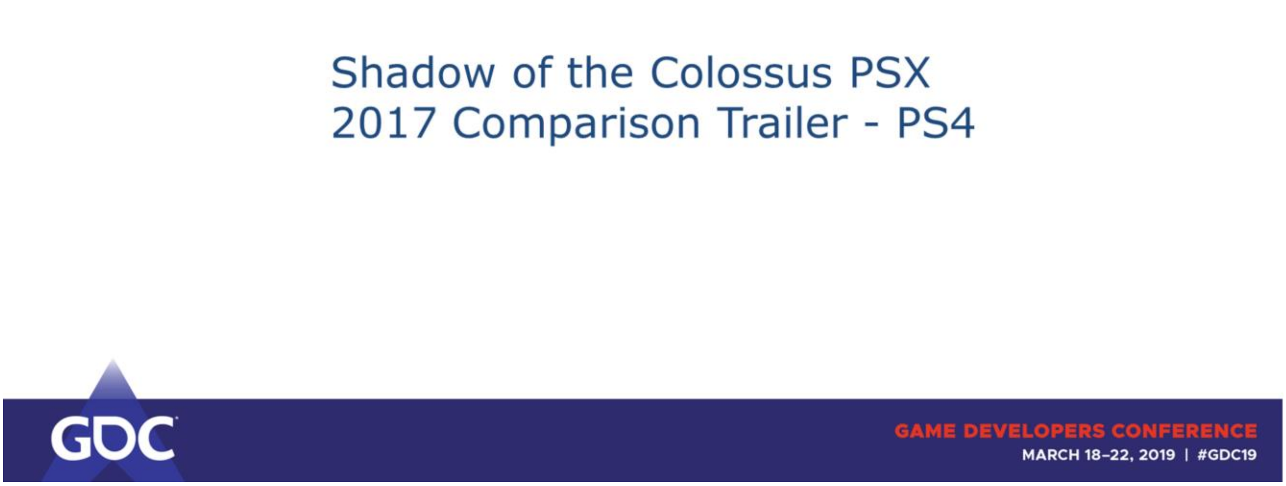
Bluepoint has made a name for itself by remastering games, many of which I'm hoping you have heard of. We have been blessed to work with titles that we absolutely love, and from my perspective it is insanely awesome that we get to see exactly how these games were made.

At Bluepoint there are 2 primary ideals that I try to promote

1. Quality is the foundation of success. I believe that this is absolutely true for our industry. If you want long term sustained success you must have quality to feed it.
2. Every project we release must be better than our previous release. Hopefully you can see this trend by looking at our past projects.

The depth at which we remaster projects has grown exponentially. Starting with the God Of War Collection, which involved texture cleanup, to the Uncharted Collection which involved all aspects of the game being modified. In fact for Shadow we don’t call it a remaster, we coin it a remake given the complexity of the project. And moving forward to our next project, we coin it a re-envisioning given that it goes well beyond what we even thought possible on Shadow.

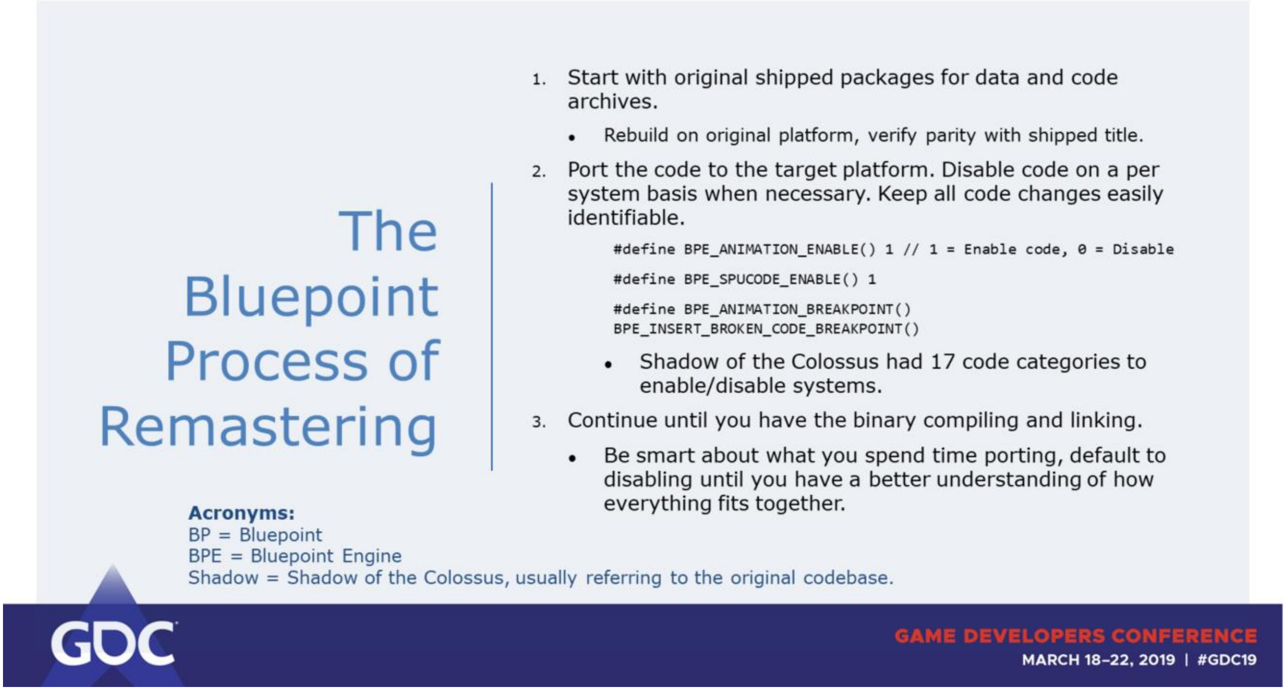
Now I want to take a minute to show you a video from PSX 2017, showing a comparison between the original Shadow and the PS4 remake to give you a better sense of what we do.



Shadow of the Colossus PSX
2017 Comparison Trailer - PS4

GAME DEVELOPERS CONFERENCE
MARCH 18–22, 2019 | #GDC19

Shadow of the Colossus PSX 2017 Comparison Trailer - PS4



The Bluepoint Process of Remastering

1. Start with original shipped packages for data and code archives.
   - Rebuild on original platform, verify parity with shipped title.
2. Port the code to the target platform. Disable code on a per system basis when necessary. Keep all code changes easily identifiable.

```
#define BPE_ANIMATION_ENABLE() 1 // 1 = Enable code, 0 = Disable
#define BPE_SPUCODE_ENABLE() 1
#define BPE_ANIMATION_BREAKPOINT()
BPE_INSERT_BROKEN_CODE_BREAKPOINT()
```

   - Shadow of the Colossus had 17 code categories to enable/disable systems.
3. Continue until you have the binary compiling and linking.
   - Be smart about what you spend time porting, default to disabling until you have a better understanding of how everything fits together.

Acronyms:
BP = Bluepoint
BPE = Bluepoint Engine
Shadow = Shadow of the Colossus, usually referring to the original codebase.

GDC
GAME DEVELOPERS CONFERENCE
MARCH 18–22, 2019 | #GDC19

I feel that it is important that people understand a couple of our over arching goals that we take into each remastering project.

1. We have to be true to the design of the original game.
2. We have to respect the decisions of the original game team.
3. We must not loose the magic of the original game. And if you say to yourself, I can’t find the magic in the original, then what the hell are you doing?
4. We have to acknowledge that there is rarely a black and white answer for what we do and as a result there will be lots of internal debates. These debates are good and health and most importantly show passion.

To help explain our thought process I like to use an analogy.

As a kid growing up, I use to love watching cartoons, who didn’t. I’m sure if you think about it, you had your favorites, certain shows that stand out in your memories. Well in my case I loved Thundercats and have fond memories. However if I go back and watch them now, or try to show them to my kids, they aren’t anything like I remember.

What we are trying to do, is bring those memories back to life and enable others to have similar experiences.

We are trying to recreate the game the original game developers would have release if they had the technology that we have today.

So, let’s take a technical look at the development process that we use at Bluepoint when remastering a title.

Besides determining whether or not it makes business sense to remaster a game, there are several key factors that must be considered.

1. We absolutely must have access to the released package. For Shadow this was the final package that was delivered to Sony for distribution. This should include all patches, back end servers, etc. Basically we need to be able to run the retail project in house.
2. We require all of the final source code required to rebuild the game’s final binaries. Now there are exception.

- We can work around miscellaneous systems. For example, if we are missing an audio library, chances are we are going replacing the entire system so no biggy. However if we are missing gameplay code it quickly becomes cost prohibitive to reverse engineer those libraries.

Other examples include:

- Server code if required is a bonus, however we can work around not having it.
- Tool pipeline code is a bonus, however we usually don’t build our processes around legacy tools and primary only use them for reference.
- And finally any Source assets that were used to build the final distribution package is a bonus, but not mandatory.

So looking at the development process, the goal for the first month is to obtain all of the required pieces and to rebuild the game’s binaries for the original target platform. We then send this build to our internal QA and run a whole suite of parity tests.

I should call out that it is critical to the success of the project that we get these first steps correct. Small errors here will be magnified as we get into full development, leading to waisted time and incorrect decisions being made. Here we feel that taking our time to ensure correctness is preparing for success.

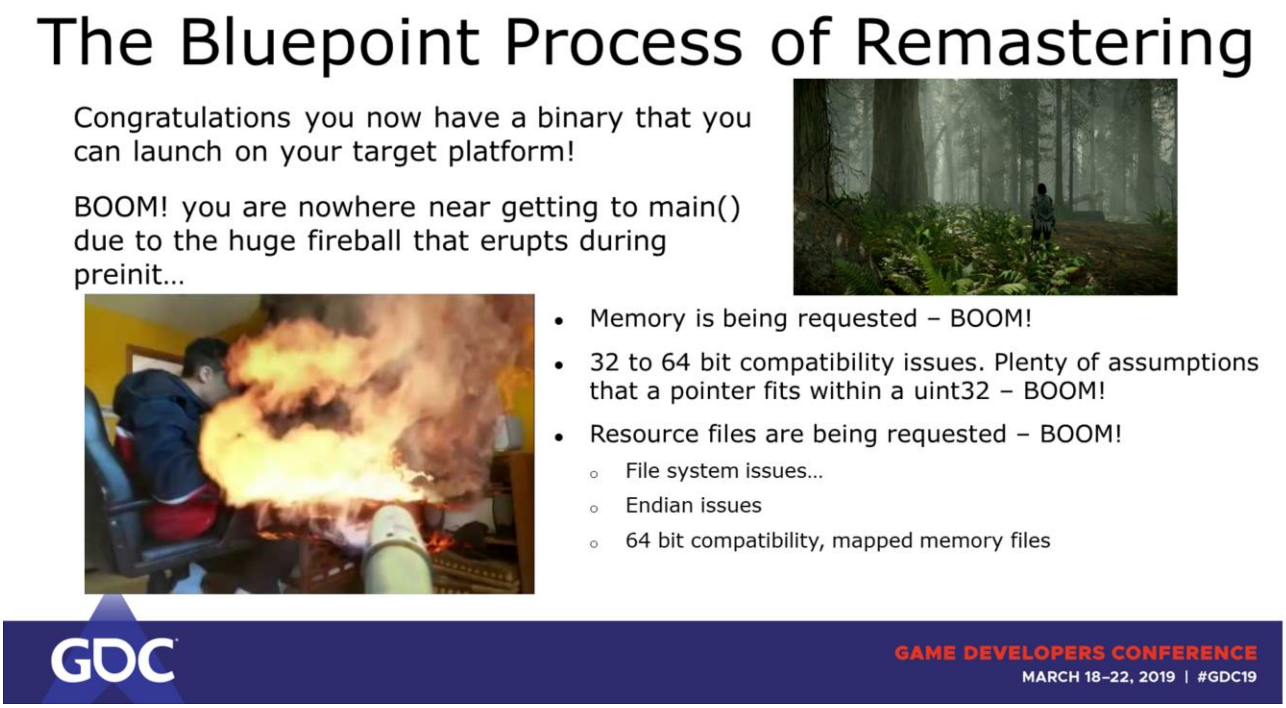
Moving forward, once we have parity we upgrade the original target to its latest SDK to make life easier, then rerun all parity tests. Basically we make incremental changes and verify parity though all steps.

Take Shadow for example, very early on we decided that we were going to use the remastered PS3 version as our base. At this point QA ran extensive parity tests between the PS2 and the PS3 versions, combed old bug databases, search the web for user feedback, etc. Concurrently engineering got the game up and running on the PS3, converted to the latest SDK, localized all Japanese code comments, rearranged code libraries to make use happy and then re-ran all parity tests, and we fixed the issues before moving forward.

At Bluepoint every developer, at their desk has full access to our target platform, which for Shadow was a PS4. And the original game, which in this case was a PS3 dev kit. Programmers are expected to debug and play the game on the PS3 dev kits to determine original code intent and purpose and to diagnose parity issues. We do the same thing for all other departments, Art always has full access to both platforms and are expected to be familiar with both. And in the case of Shadow, QA went the extra mile and threw the PS2 into the mix.

Only after parity is verified do we start porting the code to the target platform. Porting is usually a 1-2 man project that takes approximately 2-3 weeks before we have the code compiling. During this phase we are more interested in just getting everything to compile on the new platform, not porting everything. Basically, we try not to lie to ourselves by pretending we know how everything works and instead are using this time to familiarize ourselves with the code base. We create #defines to disable whole systems, mark up changes we have made to the original code, etc. The most important goal at this point is to:

1. Get the code to compile and link, which is harder than it sounds.
2. Make it trivial to identify where we have modified the original code.



The Bluepoint Process of Remastering

Congratulations you now have a binary that you can launch on your target platform!

BOOM! you are nowhere near getting to main() due to the huge fireball that erupts during preinit…

- Memory is being requested – BOOM!
- 32 to 64 bit compatibility issues. Plenty of assumptions that a pointer fits within a uint32 – BOOM!
- Resource files are being requested – BOOM!
  - File system issues…
  - Endian issues
  - 64 bit compatibility, mapped memory files

So congratulations after about 2 months you have a binary that you can launch on your target platform. Not only did you get the game to compile, you got it to link which was a huge pain in the ass. You can’t help but want to play the game so you hit run and …

(Click) BOOM, everything blows up.

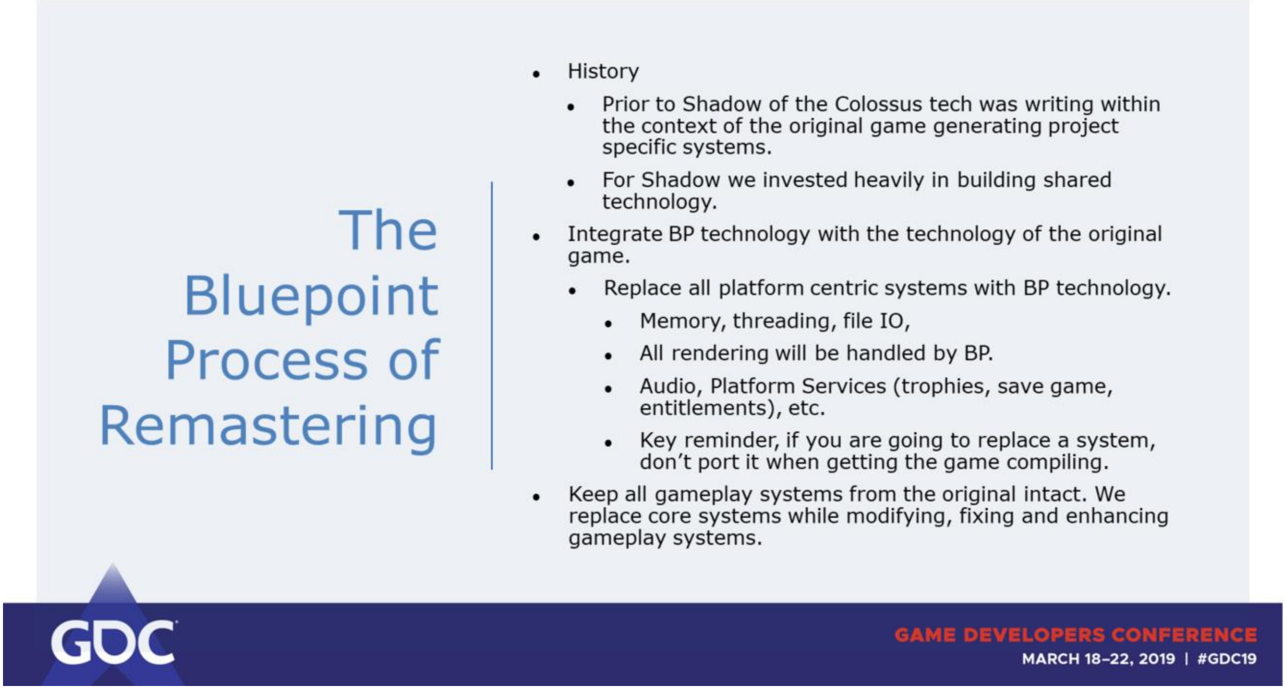
In fact the breakpoint that you put in main() never got hit.

You blew up during preinit, the callstack doesn’t make any sense and everything feels broken.

- (Click) Memory is being requested
- (Click) You have 32 to 64 bit compatibility issues. Plenty of code assumptions that a pointer fits within a uint32.
- (Click) Resource files are being requested
- - Where the hell is the file system?
- - You have Endian issues everywhere.
- - And how do we deal with memory mapped files that assume 32 bit pointers?

At this point you can’t help but wonder: What have I gotten myself into and does it make sense to assign this task to another engineer?

At this point I must tell myself, “stop being a little bitch, and keep going”.



The Bluepoint Process of Remastering

- History
  - Prior to Shadow of the Colossus tech was writing within the context of the original game generating project specific systems.
  - For Shadow we invested heavily in building shared technology.
- Integrate BP technology with the technology of the original game.
  - Replace all platform centric systems with BP technology.
    - Memory, threading, file IO,
    - All rendering will be handled by BP.
    - Audio, Platform Services (trophies, save game, entitlements), etc.
    - Key reminder, if you are going to replace a system, don’t port it when getting the game compiling.
- Keep all gameplay systems from the original intact. We replace core systems while modifying, fixing and enhancing gameplay systems.

GAME DEVELOPERS CONFERENCE
MARCH 18–22, 2019 | #GDC19

Given that the first step was just to get the game to compile on the target platform the next step is to get the game to main() then to the main loop.

The main loop will be pretty much commented out at this point, however our goal is to get the game running displaying nothing but a black screen.

Getting to a stable black screen is a critical milestone for us. It is really the point at which the entire team of engineers can real start piling onto the project and efficiently look at their respective realms.

It is the point at which momentum really starts to pickup.

So I want to stop for a second and share a little bit of history:

With all of the projects that we worked on before Shadow of the Colossus we would work within the realm of the original code base.

- We would rewrite the low level platform code to work on the new target platform.
- We would build a new rendering system to handle the needs of the game.
- Tools and processes would be adapted to closely fit those of the original development team.

The problem was that with every new project it felt like we were completely starting over.

- Code would end up being very tied to a specific title and thus was not moved from project to project.
- The team; art, engineering, including qa, all needed to re-learn processes as we moved from game to game.

Basically it started to conflict with our ideal that each project needs to be better than the previous. So for Shadow of the Colossus we completely revamped how we work. We made a point of building reusable technology and processes that evolve from project to project rather than being recreated. This requires a deeper commitment to processes and long term planning rather than solely focusing on immediate goals.

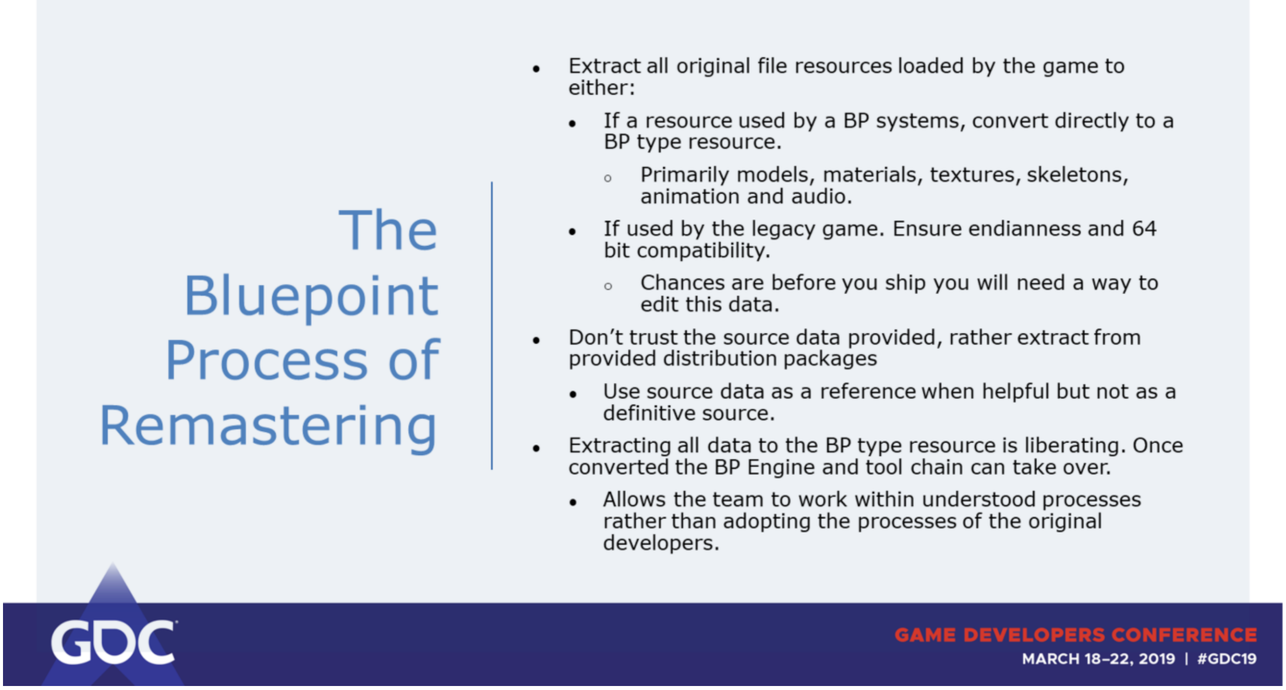
So we took our existing Bluepoint Engine, dusted it off and started to make a major investment. For reference the Bluepoint Engine is proprietary and has been around for quite a while. In fact it has been licensed and used to ship several titles, however it was time for it to evolve.

Now getting back to the goal of getting the game running. We achieve this goal by merging the original code with the Bluepoint Engine. Basically we structure the code libraries so that the original game is built on top of the Bluepoint Engine. This allows the Bluepoint Engine to provide all platform centric system such as

- Memory
- Threading
- File IO
- Rendering
- Etc…

For us it is key to keep in mind which original game systems we want to port to the target platform and which ones will be replaced with Bluepoint systems. Basically don’t waist time porting a system that will be replaced in the future.

With this said I think it is important to make one clarification. When it comes to gameplay systems; AI, character logic, etc… we want to keep all of the original systems intact. While we will fully replace core systems. With gameplay we take a much more surgical approach of fixing and enhancing.



The Bluepoint Process of Remastering

- Extract all original file resources loaded by the game to either:
  - If a resource used by a BP systems, convert directly to a BP type resource.
    - Primarily models, materials, textures, skeletons, animation and audio.
  - If used by the legacy game. Ensure endianness and 64 bit compatibility.
    - Chances are before you ship you will need a way to edit this data.
- Don’t trust the source data provided, rather extract from provided distribution packages
  - Use source data as a reference when helpful but not as a definitive source.
- Extracting all data to the BP type resource is liberating. Once converted the BP Engine and tool chain can take over.
  - Allows the team to work within understood processes rather than adopting the processes of the original developers.

GDC

GAME DEVELOPERS CONFERENCE
MARCH 18–22, 2019 | #GDC19

Next I want to take a minute to discuss the Bluepoint approach to game assets. While it is ideal to get the source assets from the original game team; and we do to varying levels of success; we do not depend upon them. Rather we only use them for reference. The reason we don’t rely upon source assets as a starting point is that they are often wrong or incomplete.

Teams are notorious for not checking everything into source control. Often local file changes or P4 shelved changes are used to build the final package creating very difficult to find parity issues. This is exaggerated when dealing with patches as the likely hood of a one off local change increases.

To eliminate this concern we will spend around 2 months extracting all of the data within the original distribution package to BPE compatible formats. We also assume that before we ship we will have a need to edit every type of data, not just key file formats. While this is time consuming it has several key benefits:

1. We know the data we have is exactly the data that shipped.
2. We learn a lot about how the game is constructed, allowing us to make smarter decisions moving forward. Understanding the content greatly helps us understand the code and enables us to make

decisions that work with the original code rather than fighting it.

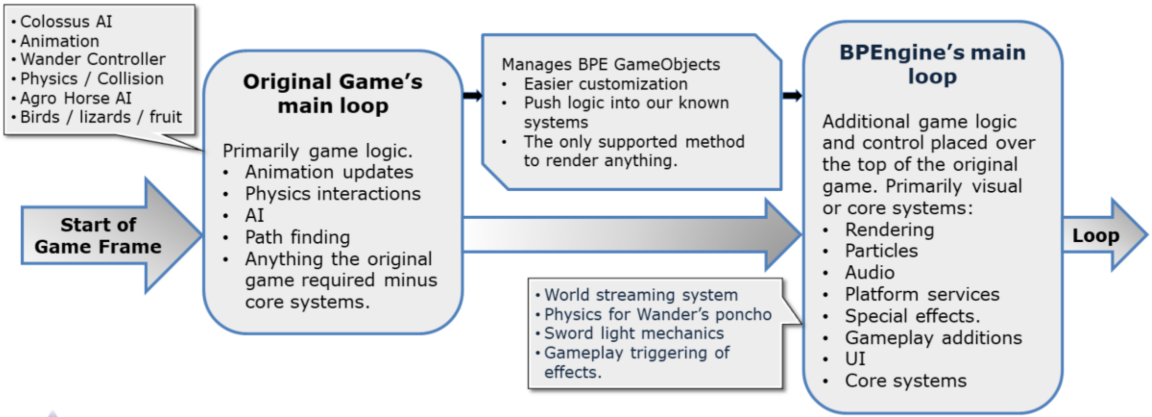
3. Once we have extract key file formats and converted them to BPE formats; such as models, animations, skeletons, textures, collision data, these assets are immediately available within our BPE toolset.

And finally, because we are taking ownership of all data we are no longer dependent upon the content pipeline of the original team. We don’t need their tools and we don’t need to follow unfamiliar practices. It is actually quite liberating and after the extraction is finished we follow Bluepoint processes rather than the processes of a remote team we don’t understand.

Now to be completely honest, there are some game specific file formats that we don’t always extract and we will take shortcuts. We do however regenerate these files addressing endianness and 64 bit compatibility issues and we spend the time to understand and document their purpose. For example in Shadow we did not convert all of the pathing data to BPE formats because we figured that we would never change this data. This came back to bit me in the ass and I’ll tell you more about it later.

The Dance

- When we are finished we want to have a main game loop where we have both the BPEngine running as well as running the original game's main loop.



- Colossus AI
- Animation
- Wander Controller
- Physics / Collision
- Agro Horse AI
- Birds / lizards / fruit

Start of Game Frame

Original Game's main loop

Primarily game logic.
- Animation updates
- Physics interactions
- AI
- Path finding
- Anything the original game required minus core systems.

Manages BPE GameObjects
- Easier customization
- Push logic into our known systems
- The only supported method to render anything.

- World streaming system
- Physics for Wander's poncho
- Sword light mechanics
- Gameplay triggering of effects.

BPEngine's main loop

Additional game logic and control placed over the top of the original game. Primarily visual or core systems:
- Rendering
- Particles
- Audio
- Platform services
- Special effects.
- Gameplay additions
- UI
- Core systems

Loop

So let's take a look at the final results that we achieved in Shadow in relationship to how we integrated the two distinct engines.

I like to call this the dance. Without proper planning and coordination, chaos would take over. However with each engine assuming specific responsibilities we can create a 'harmony of technology'.

What this diagram is trying to illustrate is the responsibilities of each engine. The original Shadow engine maintains sole responsibility for the majority of all gameplay. If there is a gameplay bug or a behavior that we want to modify, chances are we are going to modify the original Shadow codebase.

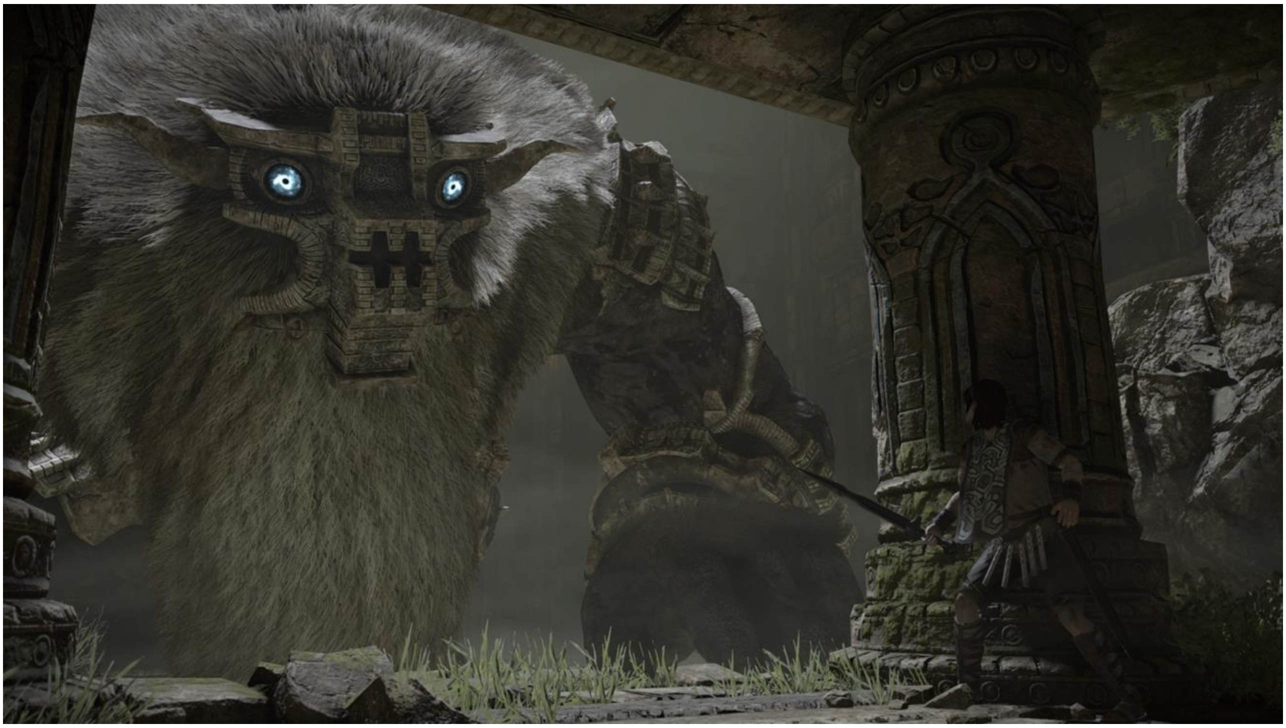
On the other side the Bluepoint Engine handles all core system responsibilities; memory, threading, platform services, etc… along with anything visual. The Bluepoint Engine handles the management of the world:

- The static geometry in the world
- The particle systems placed in the world
- The lighting

- Etc

It remains the responsibility of the Shadow engine to create all dynamic GameObjects such as; Agro, Wander and Colossi within the BP Engine in order to build the completed scene which in turn feeds the renderer.

Take for example Wander, the main character in the game. The original game has a concept; an object representing Wander and also has a link to a BP Engine representation of Wander. The original game code provides all of the simulation logic and pushes matrices and required state information over to the Bluepoint representation. In turn the Bluepoint representation can add additional functionality, such as head tracking and drives the renderer to ensure everything shows up in the final scene.



So let’s take a look at the game and break down a scene staring Colossus 6.

So who is responsible for what.

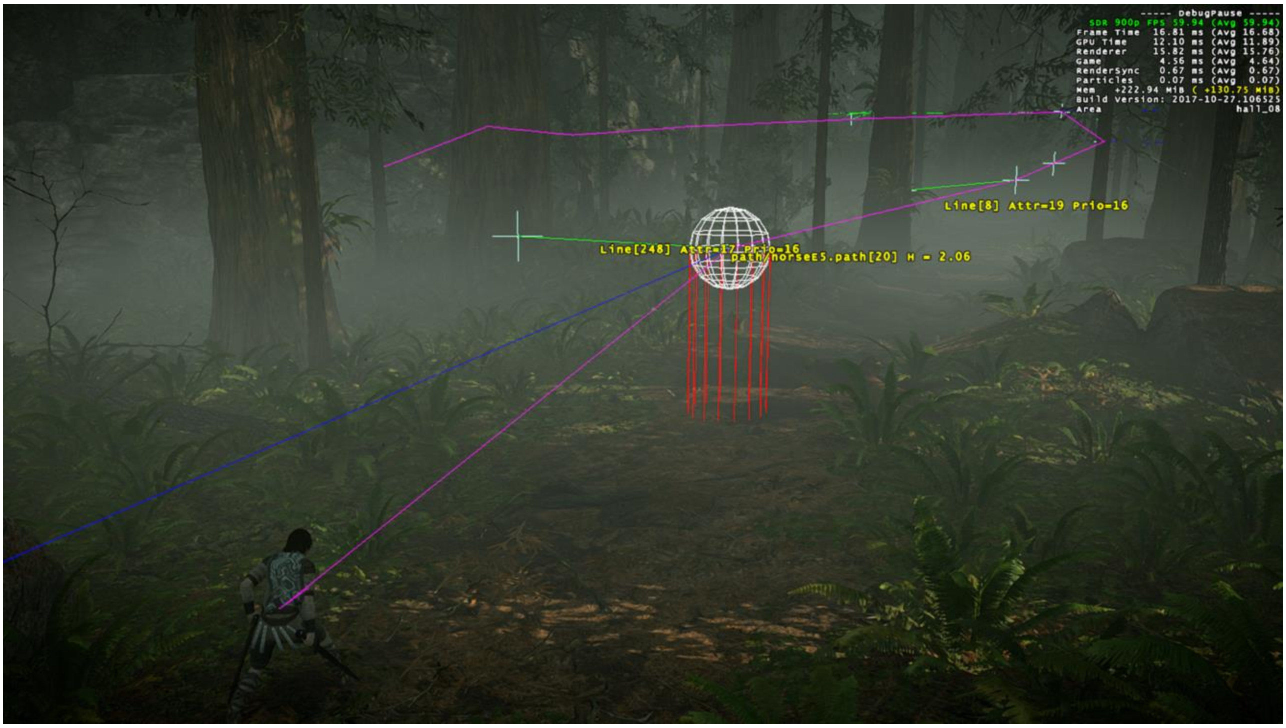
1. Shadow is responsible for all dynamic characters in a scene, in this case Wander and the Colossus.
2. Shadow is responsible for AI behaviors, when QA would report a behavioral problem we addressed it within the original codebase.
3. Shadow is responsible for collision. In retrospect this was perhaps a mistake and we should have brought collision into BPE to make it easier to work with.
4. Shadow is responsible for building the animation blend trees and building the final pose. Shadow has these responsibilities because the simulation is dependent upon the final pose. It would have been better to use BPE systems for animation, however at the time intertwining Shadow and BPE processes was very difficult.
5. The eye state of the Colossus was determined by Shadow code, passed to BPE code where it was managed, used to determine the correct eye color and drive dynamic shader parameters.
6. It is the Shadow code that handles the simulation of the bones that hang around Wander’s waist. These were never lifted into BPE given

that the original coder did such a great job.

So that is about where the original shadow code ends and the BP Engine takes over.

1. BPE handles all rendering and scene management.
2. The entire environment and atmospherics that you see are handled by BPE.
3. The physics simulation of Wander’s poncho is handled by BPE. It is purely visual and needed improvement, thus all aspects were removed from the Shadow code and moved into Bluepoint.
4. BPE is also responsible for the dynamic rings attached to the Colossi. In the original the rings are static and misaligned, they are now dynamic.
5. And finally BPE is responsible for all audio and particles within the scene.

One way to think about our approach is that we are taking the original game and overlaying visuals and enhanced gameplay.



```
----- DebugPause -----
SDR 900p FPS 59.94 (Avg 59.94)
Frame Time  16.81 ms (Avg 16.68)
GPU Time    12.10 ms (Avg 11.89)
Renderer    15.82 ms (Avg 15.76)
Game         4.56 ms (Avg  4.64)
RenderSync   0.67 ms (Avg  0.67)
Particles    0.07 ms (Avg  0.07)
Mem   +222.94 MiB ( +130.75 MiB)
Build Version: 2017-10-27.106525
Area                     hall_08
```

Here is another scene with the path node’s debug display enabled. The blue line simply shows available connections between nodes while the purple line shows the navigation path that Agro, your horse will take to get to you.

When we started the game we figured that we would not change the layout of the environments enough that we would need to modify collision. This became perhaps the biggest lie we told ourselves. Before shipping all collision within the game was completely rebuilt.

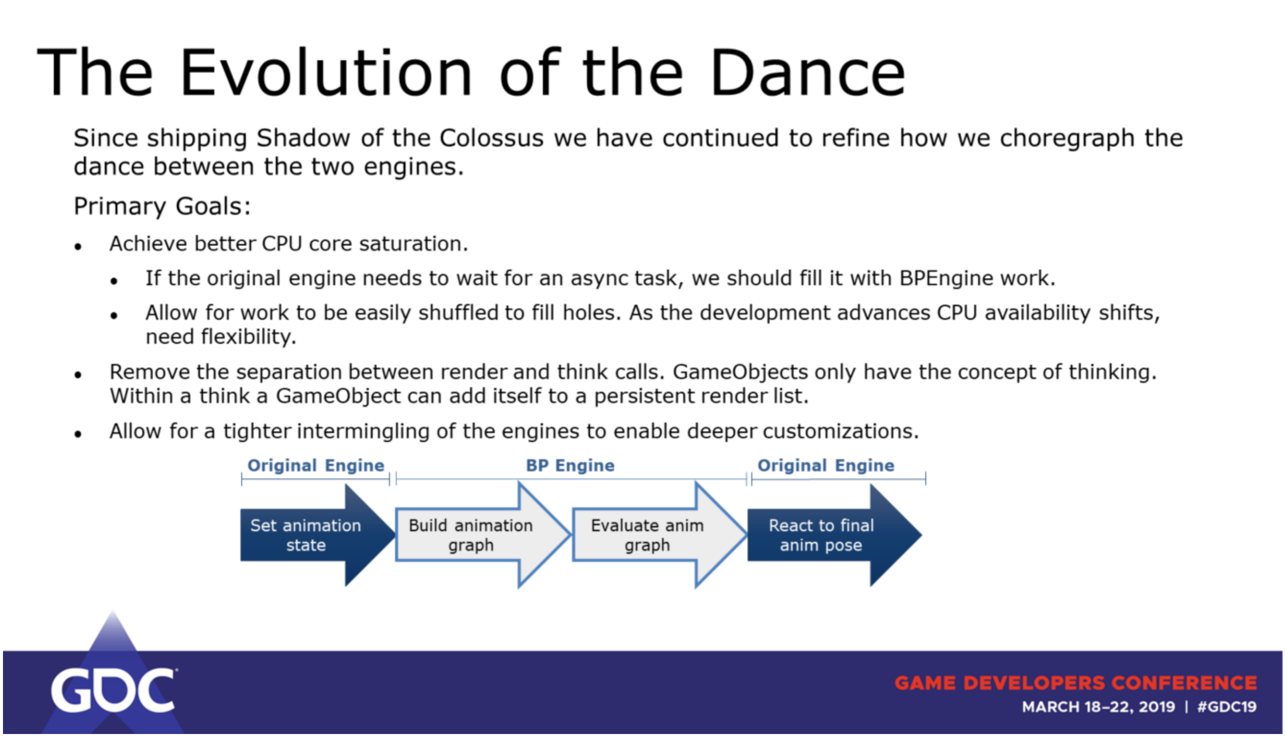
We also told ourselves that because we were not significantly changing the layout of the environment, there would be no need for us to modify path node information. This was also a lie.

As the environments started to be finalized QA started flagging bugs where

- Agro could no longer path to you because someone place a large tree on top of a path node.
- In certain areas Wander could not find lizards because they where now hidden under hills.
- In other areas the hawks and birds in the game would start flying though the sides of mountains that were now in their paths.

After a bit of digging it became obvious that it was not an acceptable solution to require art to go back and fix the geometry to match the constraints of the original. Instead within the last 2 months before shipping we wrote a tool that allowed for path nodes to be visualized and edited directly in game and then serialized back out. Before this issue the pathing logic was the last piece of untouched code and data. While we didn’t necessarily set out to change every single piece of data loaded by the game, it became a necessity.

So I guess the moral of this story is to assume that if something can bite you in the ass, it will. I think this is a safe mantra for anyone in engineering.



The Evolution of the Dance

Since shipping Shadow of the Colossus we have continued to refine how we choregraph the dance between the two engines.

Primary Goals:

- Achieve better CPU core saturation.
  - If the original engine needs to wait for an async task, we should fill it with BPEngine work.
  - Allow for work to be easily shuffled to fill holes. As the development advances CPU availability shifts, need flexibility.
- Remove the separation between render and think calls. GameObjects only have the concept of thinking. Within a think a GameObject can add itself to a persistent render list.
- Allow for a tighter intermingling of the engines to enable deeper customizations.

Original Engine | BP Engine | Original Engine

Set animation state → Build animation graph → Evaluate anim graph → React to final anim pose

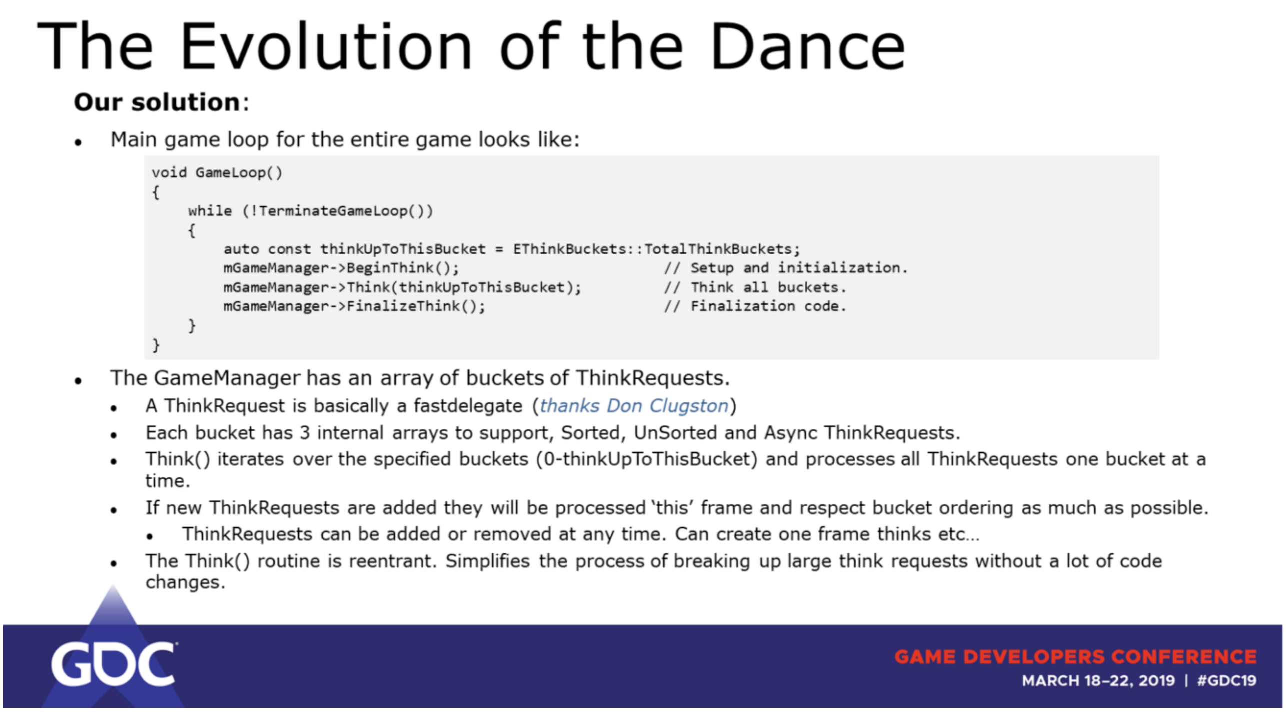
So Shadow was release a year ago this February. I explicitly don't want this talk to feel like a post mortem but more of a sharing of our approach and processes. My goal is to be completely transparent and share what worked and where we fell short.

While on Shadow we had a fairly sophisticated worker job system, we were not able to retro fit enough code to take full advantage of it.

We had this notion of the Shadow code completing its full simulation before moving onto handling BPE gameobjects. The CPU frame was broken up so that the Shadow code took about 70% of the game frame and the BP Engine took the other 30%. We could find areas to optimize the Shadow code by jobifying, however this basically just moved code from one core to another without really increasing CPU saturation. As a result we did not get much parallelization within the game code.

Within the BP Engine we represent a scene as a tree of gameobjects where each gameobject contains components that exposes functionality. To update a scene we would walk the gameobject tree and update all components. The overhead of walking the tree quickly became time consuming. To make things worse, we would actually walk the gameobject tree a second time to build the list of items to render. This happened during the sync point between the game and rendering threads creating a major bottle neck. While towards the very end of Shadow's development we implemented a couple of Hail Marys to address the problem, they were band aids at best.

Moving forward to our next project we knew that we needed to re-architect how we coordinate the game loops between the two engines. Basically it was time to take what we had learned and evolve the dance.



The Evolution of the Dance

Our solution:

- Main game loop for the entire game looks like:

```
void GameLoop()
{
    while (!TerminateGameLoop())
    {
        auto const thinkUpToThisBucket = EThinkBuckets::TotalThinkBuckets;
        mGameManager->BeginThink();                        // Setup and initialization.
        mGameManager->Think(thinkUpToThisBucket);          // Think all buckets.
        mGameManager->FinalizeThink();                     // Finalization code.
    }
}
```

- The GameManager has an array of buckets of ThinkRequests.
  - A ThinkRequest is basically a fastdelegate (*thanks Don Clugston*)
  - Each bucket has 3 internal arrays to support, Sorted, UnSorted and Async ThinkRequests.
  - Think() iterates over the specified buckets (0-thinkUpToThisBucket) and processes all ThinkRequests one bucket at a time.
  - If new ThinkRequests are added they will be processed 'this' frame and respect bucket ordering as much as possible.
    - ThinkRequests can be added or removed at any time. Can create one frame thinks etc…
  - The Think() routine is reentrant. Simplifies the process of breaking up large think requests without a lot of code changes.

The major breakthrough came when we decided to stop treating the original code base as special, but rather as just another think process that happens at a specific point in the frame. You could think of the entire original Shadow game code as a component on a GameObject exposing functionality. Unfortunately this is easier said than done.

To evolve to where we are today we had to make the following changes.

1. The first thing we did was to eliminate the need to walk gameobjects to determine what needs to be rendered. Rather we make components responsible for adding and removing persistent render items. We support a commandbuffer style interface for changing render item parameters to keep everything threadsafe.

2. We no longer walk the gameobject tree to update components. Rather components register ThinkRequest delegates.

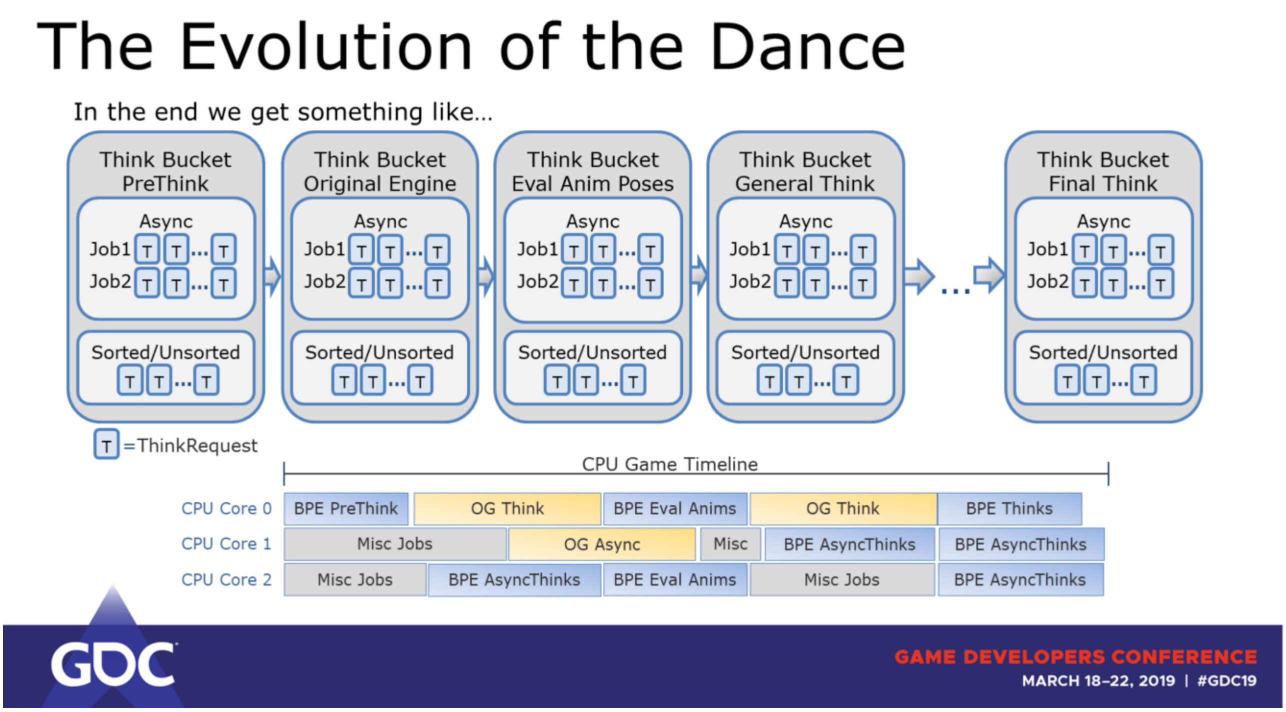
3. ThinkRequests can be added or removed at any point from any thread. A single object can submit as many ThinkRequests as desired based upon demand. This makes it very easy to create an object that only needs to think for one frame or under certain conditions.

4. ThinkRequests can be explicitly invoked at any point. If not explicitly invoked they will be invoked when their bucket is processed. However it is trivial to be in the middle of a

ThinkRequest, start an async job that you need to wait for and invoke other ThinkRequests ensuring we never stall.

With these changes and looking back at Shadow, now rather than merging the Shadow game loop with the BPE game loop we would simply have the Shadow code register a ThinkRequest within the correct bucket eliminating the distinction between the Shadow update and the BPE update. When the original Shadow code needed to wait for async jobs, it could simply start processing other ThinkRequests rather than stalling.

This type of behavior is proving to be critical on our next project. In fact this last week, I spent a day rearranging ThinkRequest dependencies to fix stalls saving approximately 2ms within in the game loop. I didn't optimizing any code rather I simply fixed scheduling issues to increase CPU saturation. Having the flexibility to easily rearrange a frame's work to fill dead areas and remove contention is proving to be awesome.



The Evolution of the Dance

In the end we get something like…

Think Bucket PreThink → Think Bucket Original Engine → Think Bucket Eval Anim Poses → Think Bucket General Think → … → Think Bucket Final Think

Each Think Bucket:

- Async
  - Job1 T T … T
  - Job2 T T … T
- Sorted/Unsorted
  - T T … T

T =ThinkRequest

CPU Game Timeline

CPU Core 0: BPE PreThink | OG Think | BPE Eval Anims | OG Think | BPE Thinks

CPU Core 1: Misc Jobs | OG Async | Misc | BPE AsyncThinks | BPE AsyncThinks

CPU Core 2: Misc Jobs | BPE AsyncThinks | BPE Eval Anims | Misc Jobs | BPE AsyncThinks

So, here is a diagram showing our current running layout. It is much easier to use and a lot more flexible.

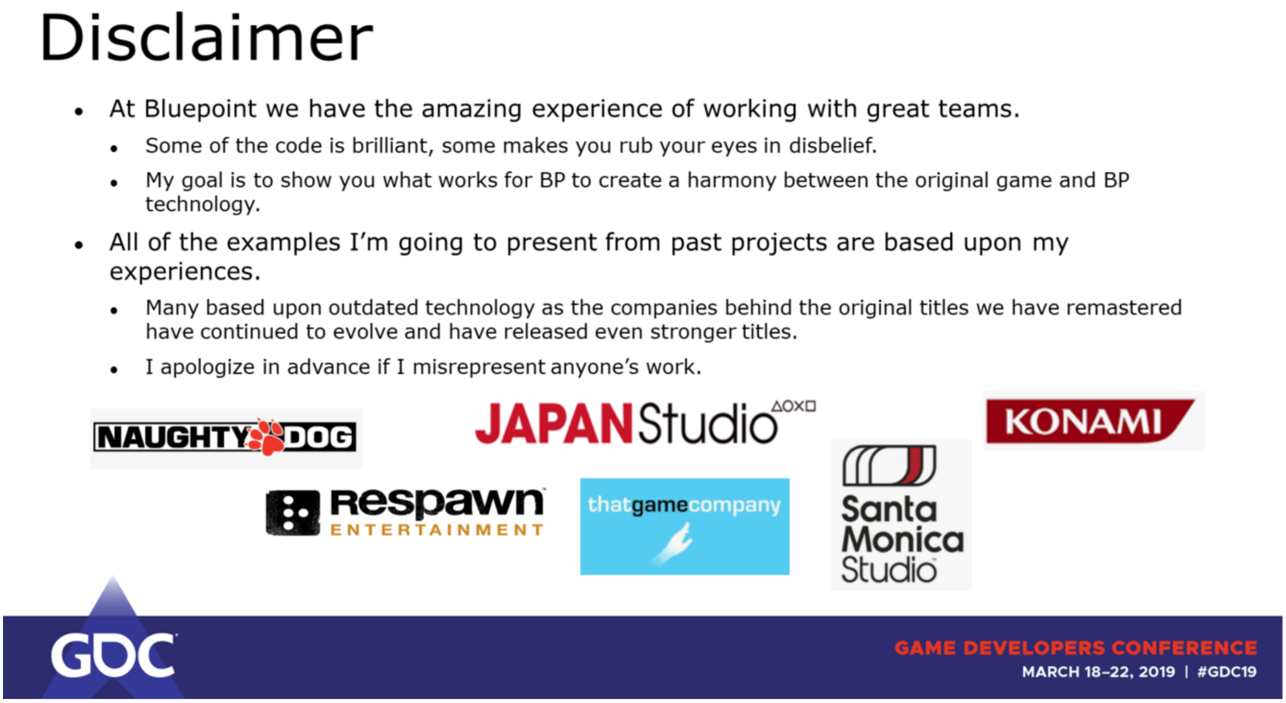
At this point, my only regret is that we didn’t implement this system earlier.

So rather than looking at Shadow, let’s take look at how we have evolved and what is next.

This Photo by Unknown Author is licensed under CC BY-NC-ND

Sorry I hope you didn’t think I was going to expose our next project, I know this isn’t funny, well maybe a little bit to me.

While I’m super excited about our next project and the tech that we are building, I’m pretty sure Bluepoint would make me walk home, all the way to Texas, just to shoot me if I screwed this up.



Disclaimer

- At Bluepoint we have the amazing experience of working with great teams.
  - Some of the code is brilliant, some makes you rub your eyes in disbelief.
  - My goal is to show you what works for BP to create a harmony between the original game and BP technology.
- All of the examples I'm going to present from past projects are based upon my experiences.
  - Many based upon outdated technology as the companies behind the original titles we have remastered have continued to evolve and have released even stronger titles.
  - I apologize in advance if I misrepresent anyone's work.

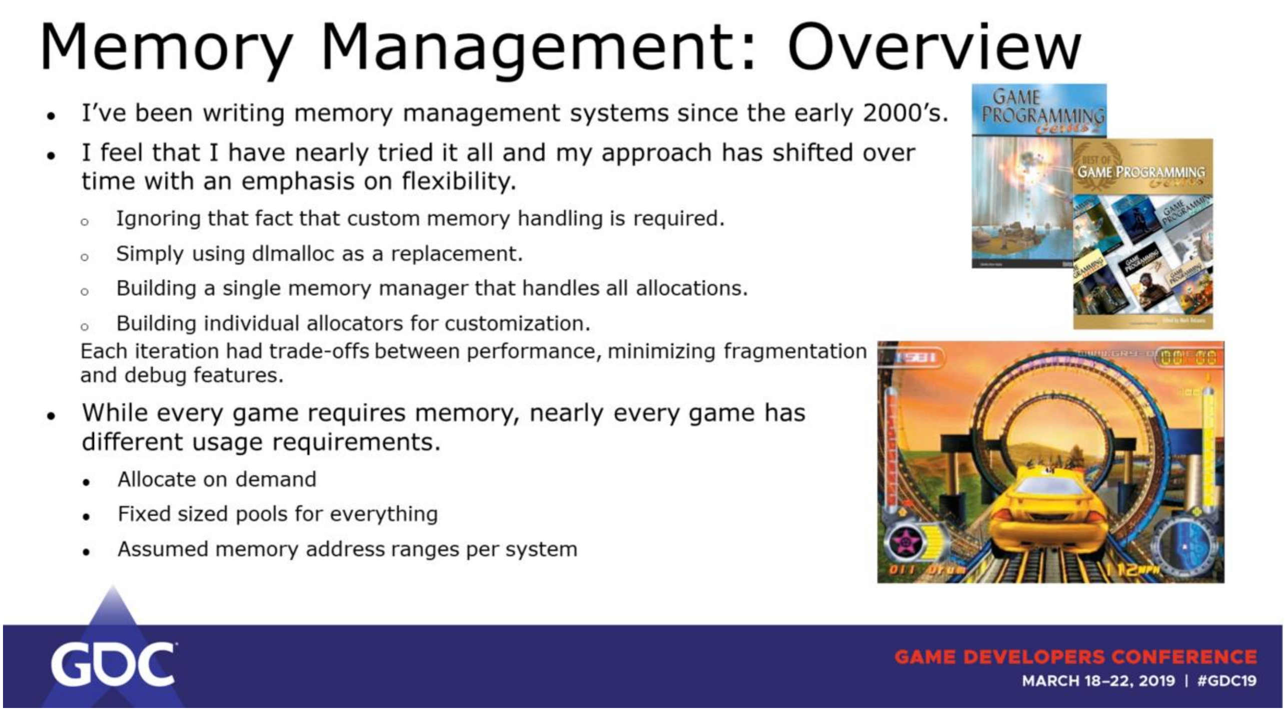
So next, I want to talk about what we have learned from past projects and how it affects our systems.

How we go about creating a harmony of technology by doing a deep dive into our memory system. How it is designed with flexibility and what steers those decisions.

Before I do so, here is a quick disclaimer.

- All of the examples that I'm going to share are based upon my personal experiences and memories.
- My examples are based upon outdated technology from the companies in question and don't represent their current technology or processes.

And finally I apologize in advance if I misrepresent anyone's work.



Memory Management: Overview

- I’ve been writing memory management systems since the early 2000’s.
- I feel that I have nearly tried it all and my approach has shifted over time with an emphasis on flexibility.
  - Ignoring that fact that custom memory handling is required.
  - Simply using dlmalloc as a replacement.
  - Building a single memory manager that handles all allocations.
  - Building individual allocators for customization.

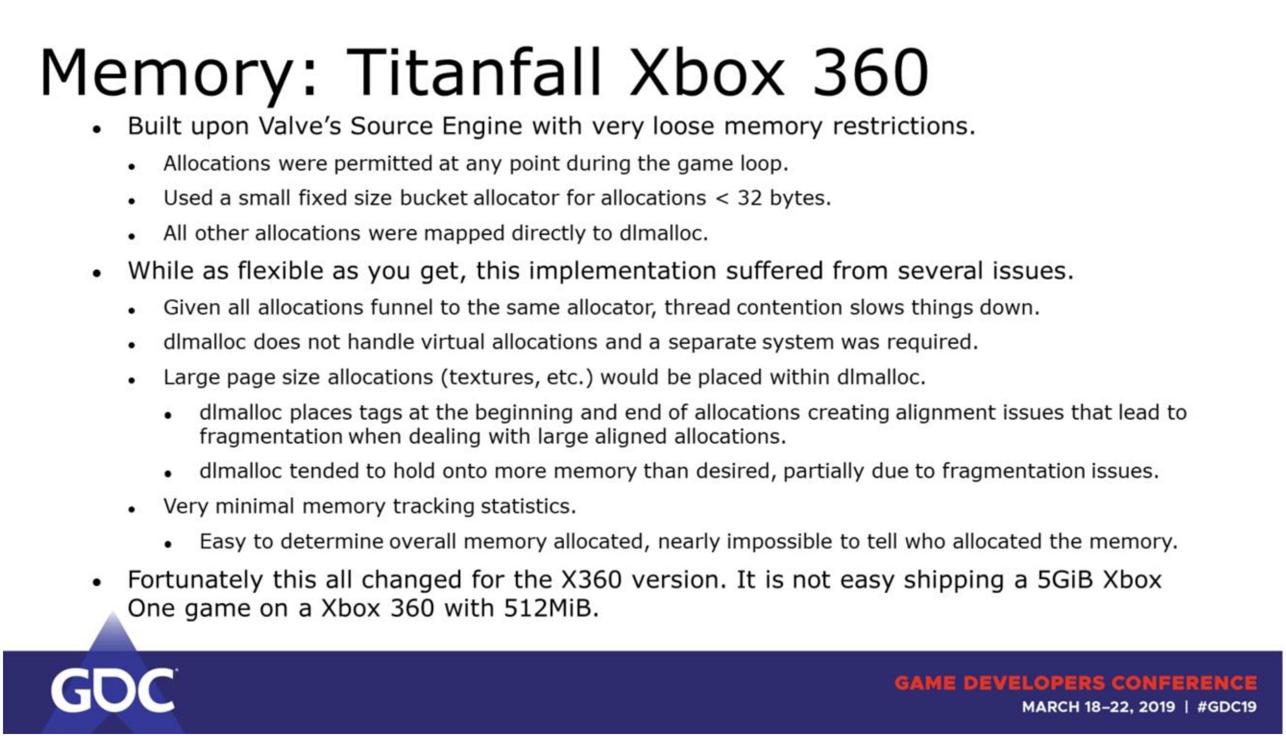
  Each iteration had trade-offs between performance, minimizing fragmentation and debug features.
- While every game requires memory, nearly every game has different usage requirements.
  - Allocate on demand
  - Fixed sized pools for everything
  - Assumed memory address ranges per system

So memory, who needs it, I’m pretty sure I do.

I’ve been dealing with memory systems since I started my career. Over this time I have tried numerous approaches and have shifted the way I view memory biased by the games I’ve shipped. In fact I wrote an article for Game Programming Gems years ago. As I look back at this article, I still embrace the ideas presented but am embarrassed that the implementation was so short sighted.

I actually got my first job in the game’s industry working for Beyond Games creating a HotWheels game. The key to landing this job was a BSP collision system that I had written that required less than half of the memory footprint of their current system; cutting memory requirements from 6 to 2Megs on a PS2.

If there is one lesson that I learned then, that is just as true today, it is that proper memory management is critical to achieving performance and ship titles.



Memory: Titanfall Xbox 360

- Built upon Valve's Source Engine with very loose memory restrictions.
  - Allocations were permitted at any point during the game loop.
  - Used a small fixed size bucket allocator for allocations < 32 bytes.
  - All other allocations were mapped directly to dlmalloc.
- While as flexible as you get, this implementation suffered from several issues.
  - Given all allocations funnel to the same allocator, thread contention slows things down.
  - dlmalloc does not handle virtual allocations and a separate system was required.
  - Large page size allocations (textures, etc.) would be placed within dlmalloc.
    - dlmalloc places tags at the beginning and end of allocations creating alignment issues that lead to fragmentation when dealing with large aligned allocations.
    - dlmalloc tended to hold onto more memory than desired, partially due to fragmentation issues.
  - Very minimal memory tracking statistics.
    - Easy to determine overall memory allocated, nearly impossible to tell who allocated the memory.
- Fortunately this all changed for the X360 version. It is not easy shipping a 5GiB Xbox One game on a Xbox 360 with 512MiB.

So let's look at Bluepoint's hardest title to date. Shipping Titanfall on the Xbox 360. What an amazing game Respawn created and if you buy me a drink I have a lot of stories. If you don't recall, the original Titanfall was developed by Respawn and released on the PC and Xbox One. Bluepoint released the Xbox 360 version approximately 2 months later.

Simple summary, if I ignore the performance aspect: It is not easy to ship a 5GiB Xbox One game on an Xbox 360 with 512MiB. So how did we do it, a lot of hard work, or as we like to call it Bluepoint Magic.

The original title used a fixed sized bucket allocator for allocations less than 32 bytes and dlmalloc for everything else.

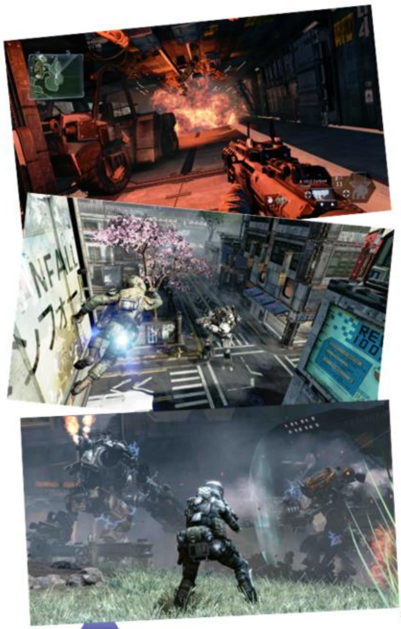
- Where all memory allocations were funneled to a single allocator and dispatched from there, creating a chock point where thread contention became a major issue.

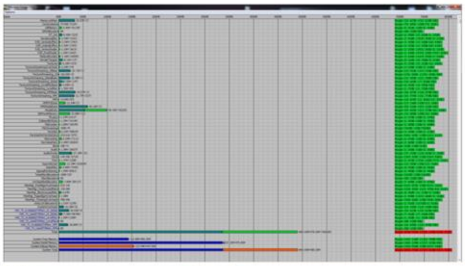
I know that a lot of people love dlmalloc and argue that it is sufficient. While it has a lot of great features, in practice on Titanfall we found that:

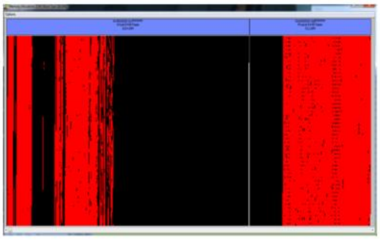
1. The version of dlmalloc we used did not have a concept of virtual allocations, thus a custom system was required.
2. dlmalloc was not ideal for large page aligned allocations. dlmalloc

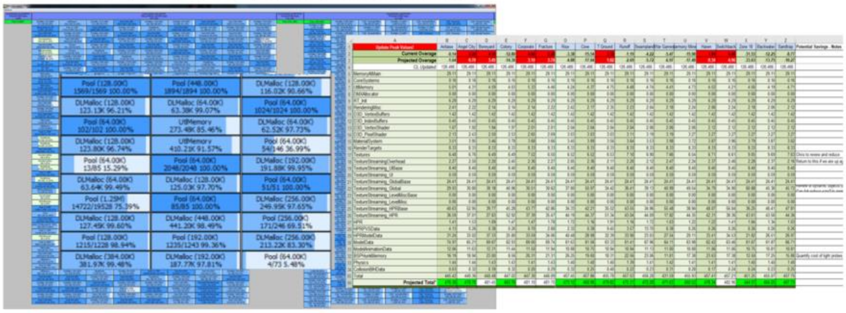
works by placing tags at the beginning of an allocation, if you need a texture with a 128 byte alignment, the tag can create memory waste.

3. Due to dlmalloc not being ideal for small allocations, virtual allocs or page sized allocations we needed dlmalloc to release unused memory back to other memory systems. In practice we found that dlmalloc held onto more memory that we thought was ideal.









Pool (128.00K) 1569/1569 100.00%	Pool (448.00K) 1894/1894 100.00%	DLMalloc (128.00K) 116.02K 90.66%
DLMalloc (128.00K) 123.13K 96.21%	DLMalloc (64.00K) 63.38K 99.07%	Pool (64.00K) 1024/1024 100.00%
Pool (64.00K) 102/102 100.00%	UtlMemory 273.49K 85.46%	DLMalloc (64.00K) 62.52K 97.73%
DLMalloc (128.00K) 123.80K 96.74%	UtlMemory 410.21K 91.57%	Pool (64.00K) 54/146 36.99%
Pool (64.00K) 13/85 15.29%	Pool (64.00K) 2048/2048 100.00%	DLMalloc (192.00K) 191.88K 99.95%
DLMalloc (64.00K) 63.64K 99.49%	DLMalloc (128.00K) 125.03K 97.70%	Pool (64.00K) 51/51 100.00%
Pool (1.25M) 14722/19528 75.39%	Pool (64.00K) 85/85 100.00%	DLMalloc (256.00K) 249.95K 97.65%
DLMalloc (128.00K) 127.49K 99.60%	DLMalloc (448.00K) 441.20K 98.49%	Pool (256.00K) 171/246 69.51%
Pool (128.00K) 1215/1228 98.94%	Pool (192.00K) 1235/1243 99.36%	DLMalloc (256.00K) 213.22K 83.30%
DLMalloc (384.00K) 381.97K 99.48%	DLMalloc (192.00K) 187.77K 97.81%	Pool (64.00K) 4/73 5.48%

* Images taken from the Xbox 360 build.

The key to us shipping Titanfall, from a memory point of view, was memory tracking.

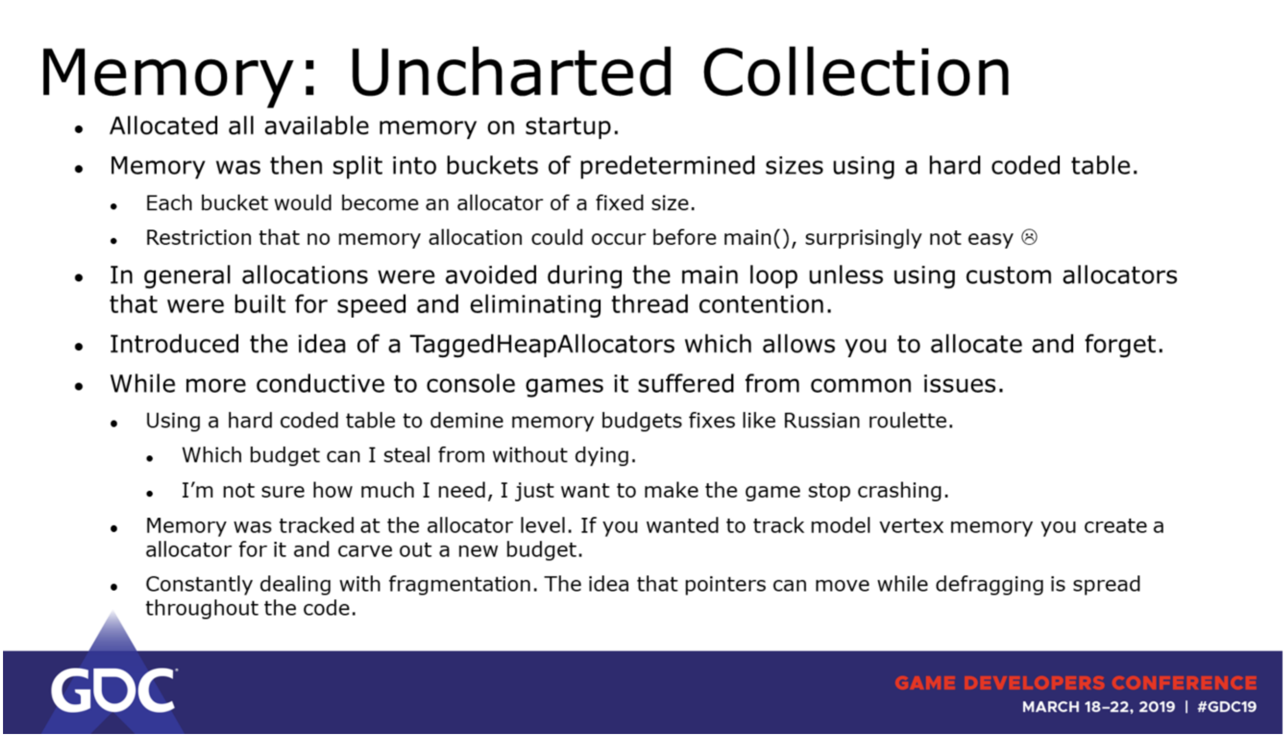
- Knowing exactly where all memory was at all times.
- Tracking fragmentation issues
- And tracking memory by category so that peaks could be compared from build to build to determine trends.

From there we were able to optimize data formats and nearly every system to cut memory usage. We reworked the memory system to:

- Use a fixed block size allocator for small allocations less than 512 bytes.
- We used dlmalloc for medium sized allocations, and several changes where made to dlmalloc to get it to aggressively release memory.
- We used a large page based allocator, used primarily by textures and large vertex buffers.
- And we added the concept of a single frame allocation, where small allocations would be placed within unused DXT mipmap memory. This memory was only valid for a single frame and coordinated with the texture streaming system.

Basically every byte was used.

By the end we had spread sheets backed by more spread sheets for every level showing max memory usage, how much memory was available for texture streaming etc. While I’m still in shock that we pulled it off, looking back, the whole system was perhaps more involved than it needed to be.



Memory: Uncharted Collection

- Allocated all available memory on startup.
- Memory was then split into buckets of predetermined sizes using a hard coded table.
    - Each bucket would become an allocator of a fixed size.
    - Restriction that no memory allocation could occur before main(), surprisingly not easy ☹
- In general allocations were avoided during the main loop unless using custom allocators that were built for speed and eliminating thread contention.
- Introduced the idea of a TaggedHeapAllocators which allows you to allocate and forget.
- While more conductive to console games it suffered from common issues.
    - Using a hard coded table to demine memory budgets fixes like Russian roulette.
        - Which budget can I steal from without dying.
        - I'm not sure how much I need, I just want to make the game stop crashing.
    - Memory was tracked at the allocator level. If you wanted to track model vertex memory you create a allocator for it and carve out a new budget.
    - Constantly dealing with fragmentation. The idea that pointers can move while defragging is spread throughout the code.

So Uncharted. What a great project. Going from a PS3 target to the PS4, how could there be any memory considerations?

Well there weren't any real memory considerations provided we kept the streaming textures reasonable however, there are a couple of key things to learn.

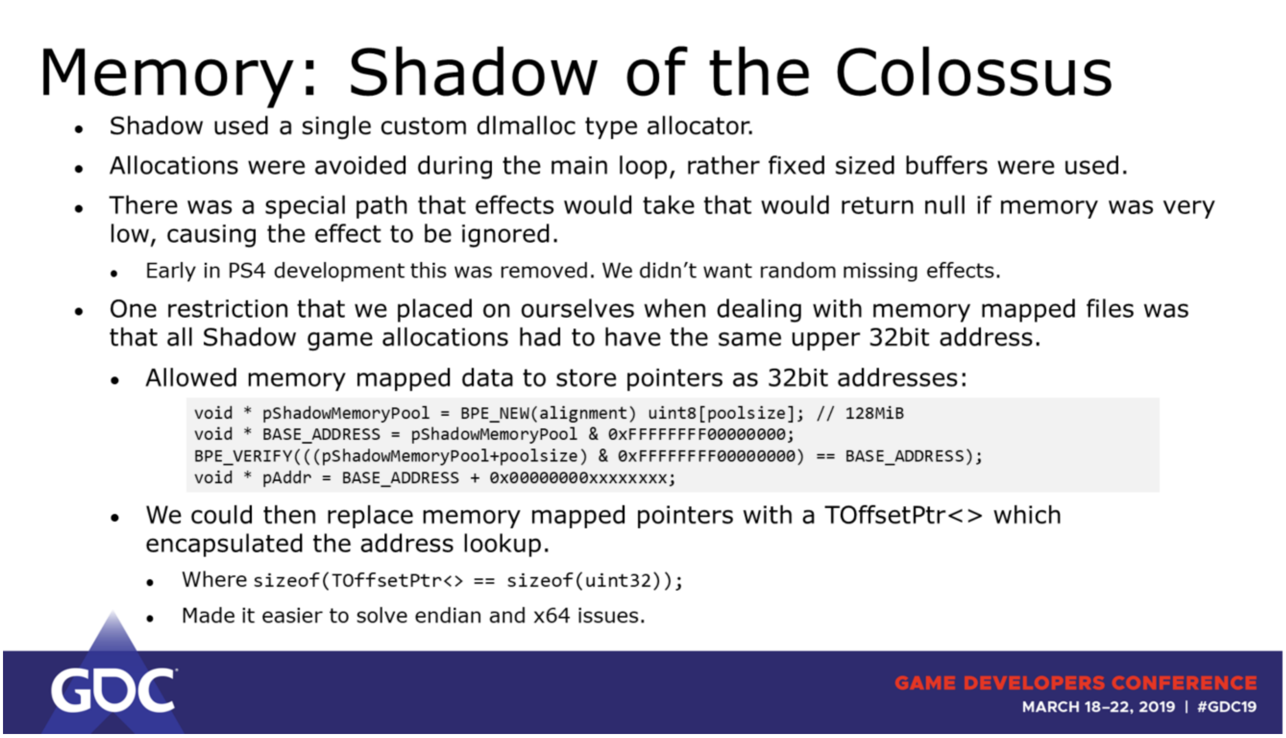
1. All memory was allocated at startup by the core memory system. From here all memory was assigned to specific allocators. There was a hard coded table within the code dictating how much memory each allocator was allotted.

- The down side is that when a allocator ran out of memory you would play Russian roulette to push memory around until the problem went away, often over allocating.

2. There was no virtual memory support and fragmentation issues were evident. There was code to handle the shuffling of memory and pointers when defragging, however it came across as error prone and touched numerous systems.

3. The later Uncharted games adopted a rule that memory could not be allocated before hitting main(). I love this ideal given that it is strait forward when to create your memory allocator and at the end of main you can easily check to ensure there are no memory leaks.

And finally Uncharted introduced the idea of a TaggedHeapAllocator. Which is very similar to the single frame allocations that we implemented for Titanfall, just evolved.



Memory: Shadow of the Colossus

- Shadow used a single custom dlmalloc type allocator.
- Allocations were avoided during the main loop, rather fixed sized buffers were used.
- There was a special path that effects would take that would return null if memory was very low, causing the effect to be ignored.
  - Early in PS4 development this was removed. We didn't want random missing effects.
- One restriction that we placed on ourselves when dealing with memory mapped files was that all Shadow game allocations had to have the same upper 32bit address.
  - Allowed memory mapped data to store pointers as 32bit addresses:

```
void * pShadowMemoryPool = BPE_NEW(alignment) uint8[poolsize]; // 128MiB
void * BASE_ADDRESS = pShadowMemoryPool & 0xFFFFFFFF00000000;
BPE_VERIFY(((pShadowMemoryPool+poolsize) & 0xFFFFFFFF00000000) == BASE_ADDRESS);
void * pAddr = BASE_ADDRESS + 0x00000000xxxxxxxx;
```

  - We could then replace memory mapped pointers with a TOffsetPtr<> which encapsulated the address lookup.
    - Where `sizeof(TOffsetPtr<> == sizeof(uint32));`
    - Made it easier to solve endian and x64 issues.

GDC

GAME DEVELOPERS CONFERENCE
MARCH 18–22, 2019 | #GDC19

[NOTE] Originally after: Memory: Uncharted Collection

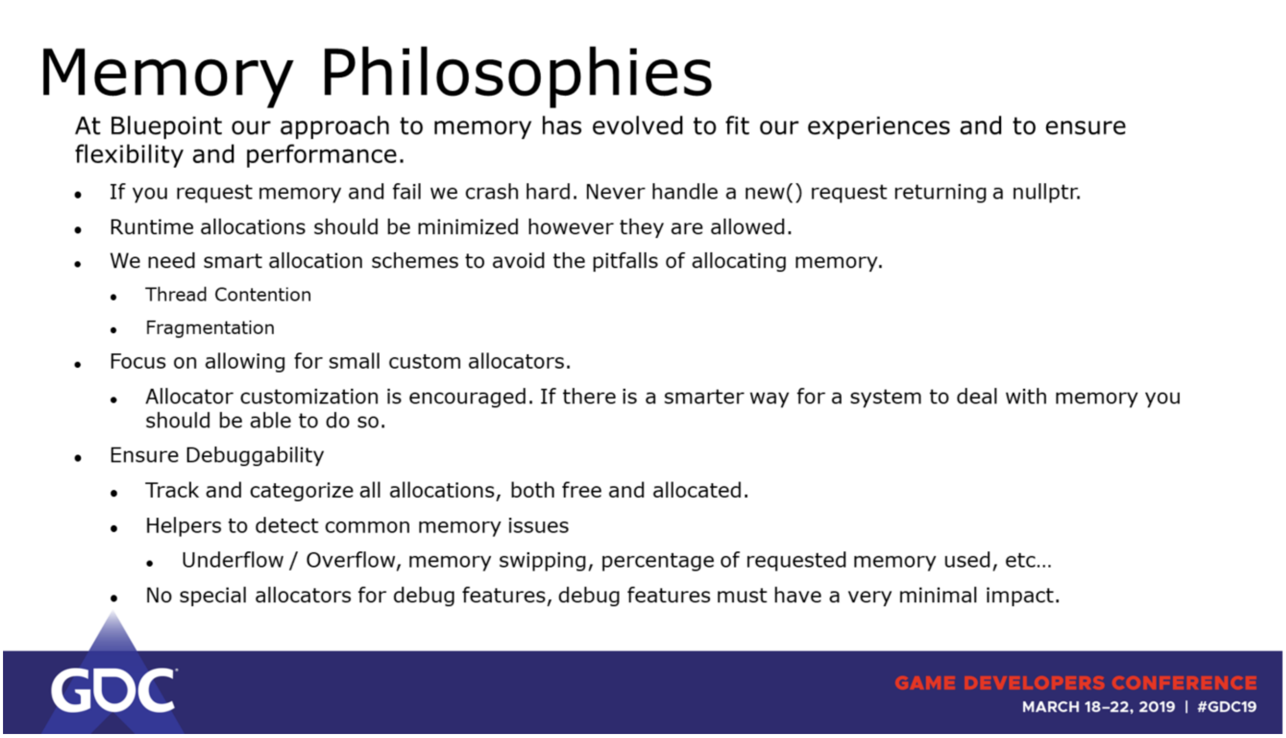
Next, what did the original Shadow of the Colossus code look like?

- It used a single custom dlmalloc type allocator.
- And in general allocations were avoided during the runtime loop.

There was one interesting code path that particle effects would take when allocating memory. If memory was exhausted the memory system would return NULL and the particle spawning would be skipped. Everywhere else, if you ran out of memory the game would halt. On the PS4 we removed this path as we could never think of a case were it was acceptable for us to run out of memory.

One restriction that we placed on our selves that affected the memory system design is that we wanted to ensure that all memory allocations requested by original Shadow code came out of a fixed memory block. We did this to make it easier to support memory mapped files that contained 32 bit pointers.

To resolve a pointer we simple added the 32 bit value along with a base address for the starting block. Provided the top 32 bits of the 64 bit address were always the same, we never run into issues.



Memory Philosophies

At Bluepoint our approach to memory has evolved to fit our experiences and to ensure flexibility and performance.

- If you request memory and fail we crash hard. Never handle a new() request returning a nullptr.
- Runtime allocations should be minimized however they are allowed.
- We need smart allocation schemes to avoid the pitfalls of allocating memory.
  - Thread Contention
  - Fragmentation
- Focus on allowing for small custom allocators.
  - Allocator customization is encouraged. If there is a smarter way for a system to deal with memory you should be able to do so.
- Ensure Debuggability
  - Track and categorize all allocations, both free and allocated.
  - Helpers to detect common memory issues
    - Underflow / Overflow, memory swipping, percentage of requested memory used, etc…
  - No special allocators for debug features, debug features must have a very minimal impact.

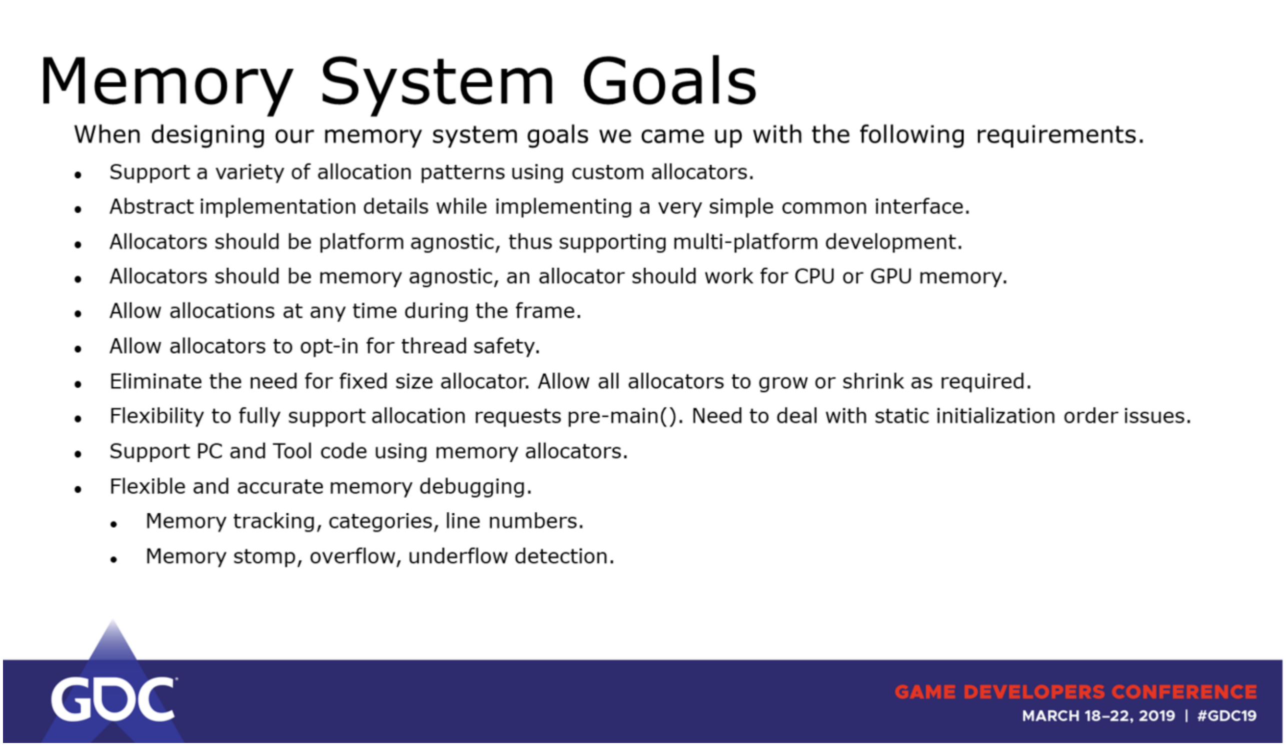
GDC

GAME DEVELOPERS CONFERENCE
MARCH 18–22, 2019 | #GDC19

So what were some of the philosophies that we felt were important when designing the memory system for the Bluepoint Engine.

1. You should never run out of memory, never expect null from an allocation request. In fact we explicitly halt the game if this happens.
2. We need smart allocation schemes to allow for a wide variety of allocation patterns. Basically encourage small custom allocators rather than an uber allocator.
3. Allocators should help to eliminate thread contention.
4. Ensure Debuggability across all allocators without a lot of custom work. To us debuggability in this case means

- The ability to track and categorize all memory allocations.
- And tools to help detect and diagnose common memory issues; with the most common being memory stomps.



Memory System Goals

When designing our memory system goals we came up with the following requirements.

- Support a variety of allocation patterns using custom allocators.
- Abstract implementation details while implementing a very simple common interface.
- Allocators should be platform agnostic, thus supporting multi-platform development.
- Allocators should be memory agnostic, an allocator should work for CPU or GPU memory.
- Allow allocations at any time during the frame.
- Allow allocators to opt-in for thread safety.
- Eliminate the need for fixed size allocator. Allow all allocators to grow or shrink as required.
- Flexibility to fully support allocation requests pre-main(). Need to deal with static initialization order issues.
- Support PC and Tool code using memory allocators.
- Flexible and accurate memory debugging.
  - Memory tracking, categories, line numbers.
  - Memory stomp, overflow, underflow detection.

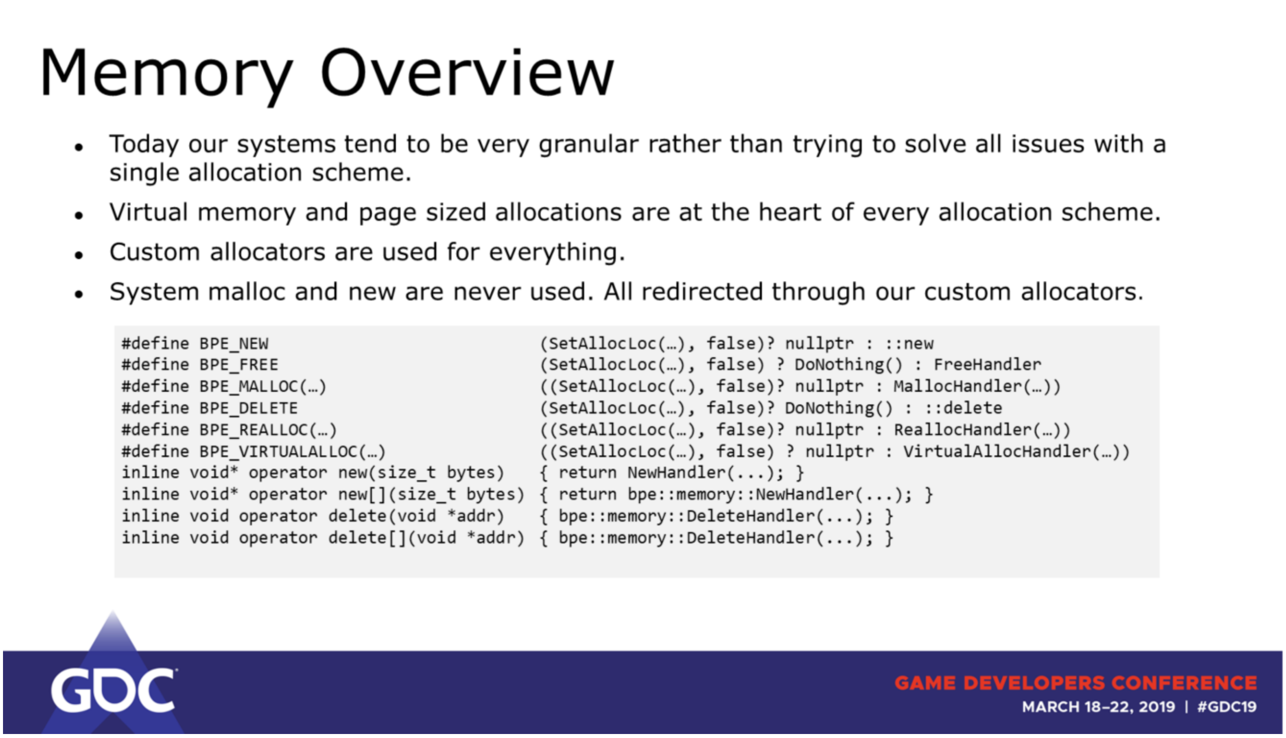
With these basic philosophies in place, we built a list of our goals. I'm only going to focus on a couple of key points:

1. All allocators should be platform agnostic. We didn't want to write custom allocators for each platform. An allocator should deal with memory patterns not the specifics of where the memory came from.

2. Allocators should be memory agnostic. Basically every allocator should work with both CPU or GPU memory. While this seems trivial it does have implications. During Shadow our per-frame GPU memory allocators benefited greatly by removing tags and markers that were being written directly in the memory blocks for tracking purposes.

3. Assume that memory will be allocated at any point. Programmers need power and flexibility to create great systems.

4. Eliminate the need for fixed sized allocators. Basically allocators should support working with a fixed memory block but should also support growing and shrinking as required. We want to avoid over allocating and the memory Russian roulette game.



Memory Overview

- Today our systems tend to be very granular rather than trying to solve all issues with a single allocation scheme.
- Virtual memory and page sized allocations are at the heart of every allocation scheme.
- Custom allocators are used for everything.
- System malloc and new are never used. All redirected through our custom allocators.

```
#define BPE_NEW                           (SetAllocLoc(…), false)? nullptr : ::new
#define BPE_FREE                          (SetAllocLoc(…), false) ? DoNothing() : FreeHandler
#define BPE_MALLOC(…)                     ((SetAllocLoc(…), false)? nullptr : MallocHandler(…))
#define BPE_DELETE                        (SetAllocLoc(…), false)? DoNothing() : ::delete
#define BPE_REALLOC(…)                    ((SetAllocLoc(…), false)? nullptr : ReallocHandler(…))
#define BPE_VIRTUALALLOC(…)               ((SetAllocLoc(…), false) ? nullptr : VirtualAllocHandler(…))
inline void* operator new(size_t bytes)   { return NewHandler(...); }
inline void* operator new[](size_t bytes) { return bpe::memory::NewHandler(...); }
inline void operator delete(void *addr)   { bpe::memory::DeleteHandler(...); }
inline void operator delete[](void *addr) { bpe::memory::DeleteHandler(...); }
```

To get started, let's take a look at how we ensure all memory allocations are routed through our memory routines.

We start by override system memory routines, which is pretty strait forward.

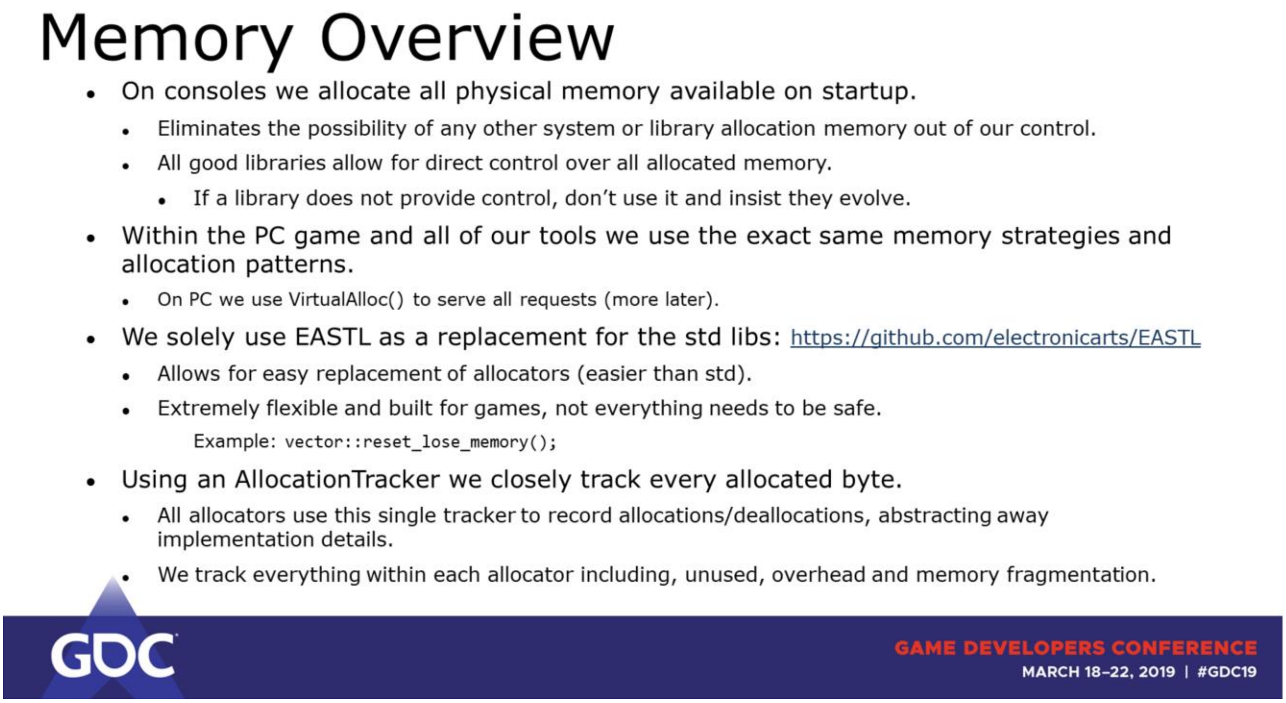
And second we create BPE macros that wrap all memory requests.

We chose to use macros for a couple of reasons:

1. This solution is simple, strait forward and requires no additional steps.
2. Some sort of redirection is required to support malloc and free. VirtualAlloc will also require custom wrappers to be platform agnostic, so it is nice to standardize everything.
3. Macros allow us to provide additional features such as location tracking and passing additional parameters directly to the memory requests such as; alignment requirements and requesting specific allocators.

Converting to using macros is not a big undertaking. I've done this about 3 times in my career and each time it takes about a week. If you don't redirect and control all allocation routines in your code I would highly recommend you start and using macros is the cleanest method I’ve found.

And even if we forget to wrap a new or delete call, it will still get picked up due to overriding the system memory routines, so you can’t miss anything.



Memory Overview

- On consoles we allocate all physical memory available on startup.
  - Eliminates the possibility of any other system or library allocation memory out of our control.
  - All good libraries allow for direct control over all allocated memory.
    - If a library does not provide control, don't use it and insist they evolve.
- Within the PC game and all of our tools we use the exact same memory strategies and allocation patterns.
  - On PC we use VirtualAlloc() to serve all requests (more later).
- We solely use EASTL as a replacement for the std libs: https://github.com/electronicarts/EASTL
  - Allows for easy replacement of allocators (easier than std).
  - Extremely flexible and built for games, not everything needs to be safe.

    Example: `vector::reset_lose_memory();`
- Using an AllocationTracker we closely track every allocated byte.
  - All allocators use this single tracker to record allocations/deallocations, abstracting away implementation details.
  - We track everything within each allocator including, unused, overhead and memory fragmentation.

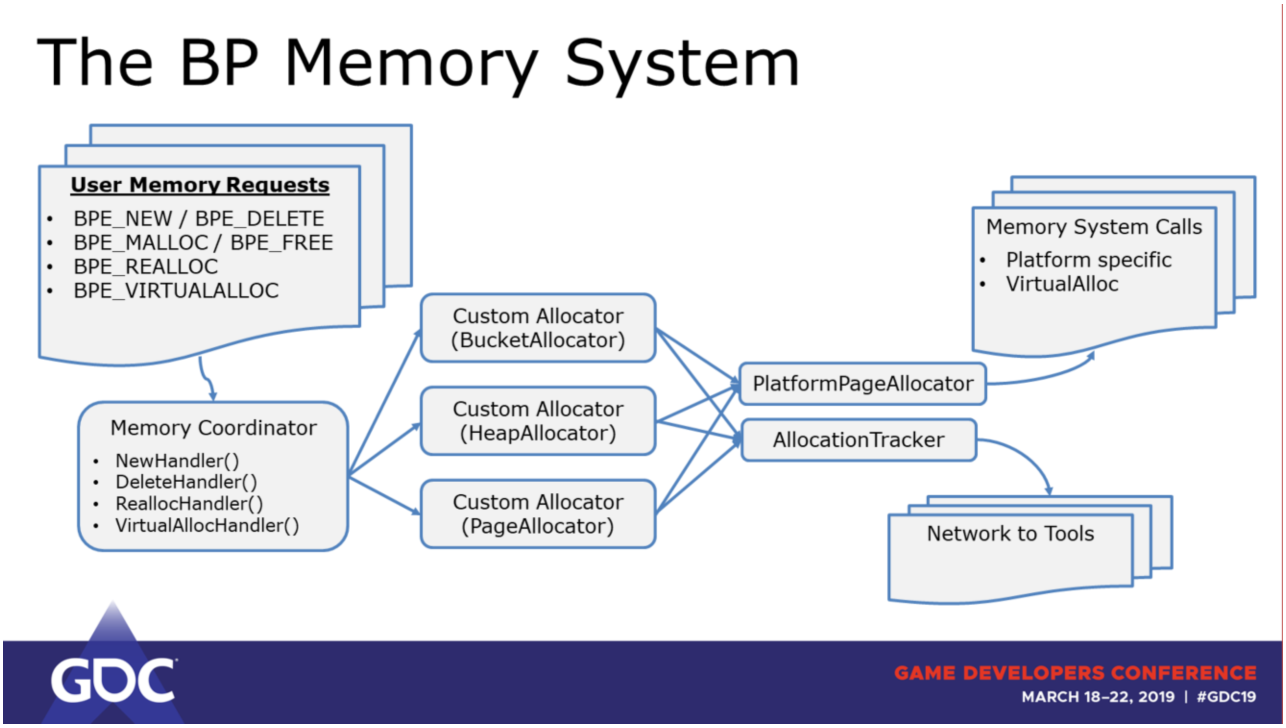
GAME DEVELOPERS CONFERENCE
MARCH 18–22, 2019 | #GDC19

And finally, to ensure we redirect all memory routines into our system, we simply control all memory. On consoles we allocate all physical memory upon initialization. As a result, if we miss an allocation due to a 3rd party library allocation, there simply isn't any memory for it to request.

Talking about 3rd party libraries, whether audio, physics, video playback or anything else there are a couple of mandates that I believe everyone should insist upon.

1. If a library does not provide control for how it gets its memory, don't use it.
2. If you can't control the threading behavior of a library, don't use it.
3. If you can't control how it loads files, don't use it.

There is one 3rd party library I would like to call out that we use extensively, the EASTL which is a replacement for the std containers. I hope anyone that has been in game development for a while at least has an opinion on the debate of whether or not to use the std containers. For us, we don't. However we have found an awesome replacement that provides a lot of really cool features that while not always safe, allows for smarter code. If you have not checked out EASTL I strongly suggest you do.



The BP Memory System

User Memory Requests

- BPE_NEW / BPE_DELETE
- BPE_MALLOC / BPE_FREE
- BPE_REALLOC
- BPE_VIRTUALALLOC

Memory Coordinator

- NewHandler()
- DeleteHandler()
- ReallocHandler()
- VirtualAllocHandler()

Custom Allocator (BucketAllocator)

Custom Allocator (HeapAllocator)

Custom Allocator (PageAllocator)

PlatformPageAllocator

AllocationTracker

Memory System Calls

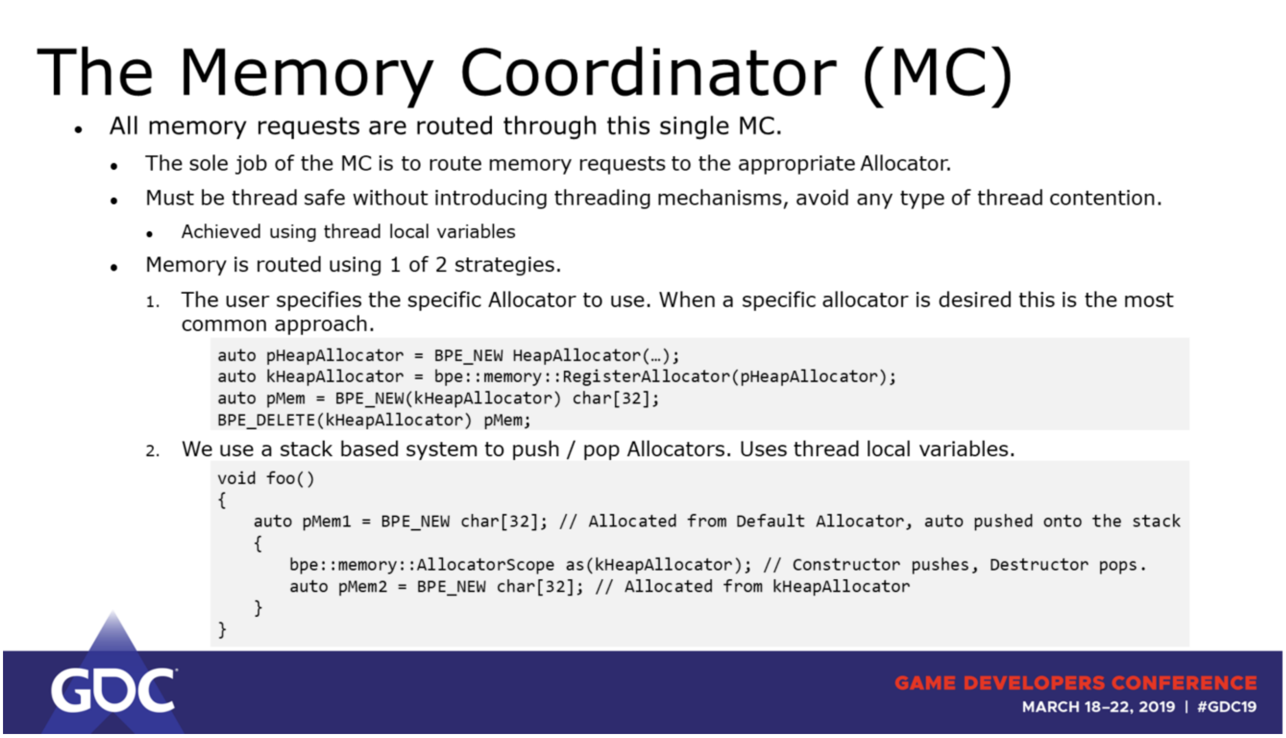
- Platform specific
- VirtualAlloc

Network to Tools

So let’s take a look at a diagram illustrating the flow of memory requests.

- As you can see it all starts with overriding system allocation routines which are redirected into our memory coordinator.
- From there memory requests are funneled into the appropriate allocator.

These allocators manage their own memory pools to service memory requests and fall back to requesting memory pages from the PlatformPageAllocator. Note that the only system that is platform specific is the PlatformPageAllocator. This helps to ensure allocators only deal with memory usage patterns and ensures that we get consistency across all platforms.



The Memory Coordinator (MC)

- All memory requests are routed through this single MC.
  - The sole job of the MC is to route memory requests to the appropriate Allocator.
  - Must be thread safe without introducing threading mechanisms, avoid any type of thread contention.
    - Achieved using thread local variables
  - Memory is routed using 1 of 2 strategies.
    1. The user specifies the specific Allocator to use. When a specific allocator is desired this is the most common approach.

```
auto pHeapAllocator = BPE_NEW HeapAllocator(…);
auto kHeapAllocator = bpe::memory::RegisterAllocator(pHeapAllocator);
auto pMem = BPE_NEW(kHeapAllocator) char[32];
BPE_DELETE(kHeapAllocator) pMem;
```

    2. We use a stack based system to push / pop Allocators. Uses thread local variables.

```
void foo()
{
    auto pMem1 = BPE_NEW char[32]; // Allocated from Default Allocator, auto pushed onto the stack
    {
        bpe::memory::AllocatorScope as(kHeapAllocator); // Constructor pushes, Destructor pops.
        auto pMem2 = BPE_NEW char[32]; // Allocated from kHeapAllocator
    }
}
```

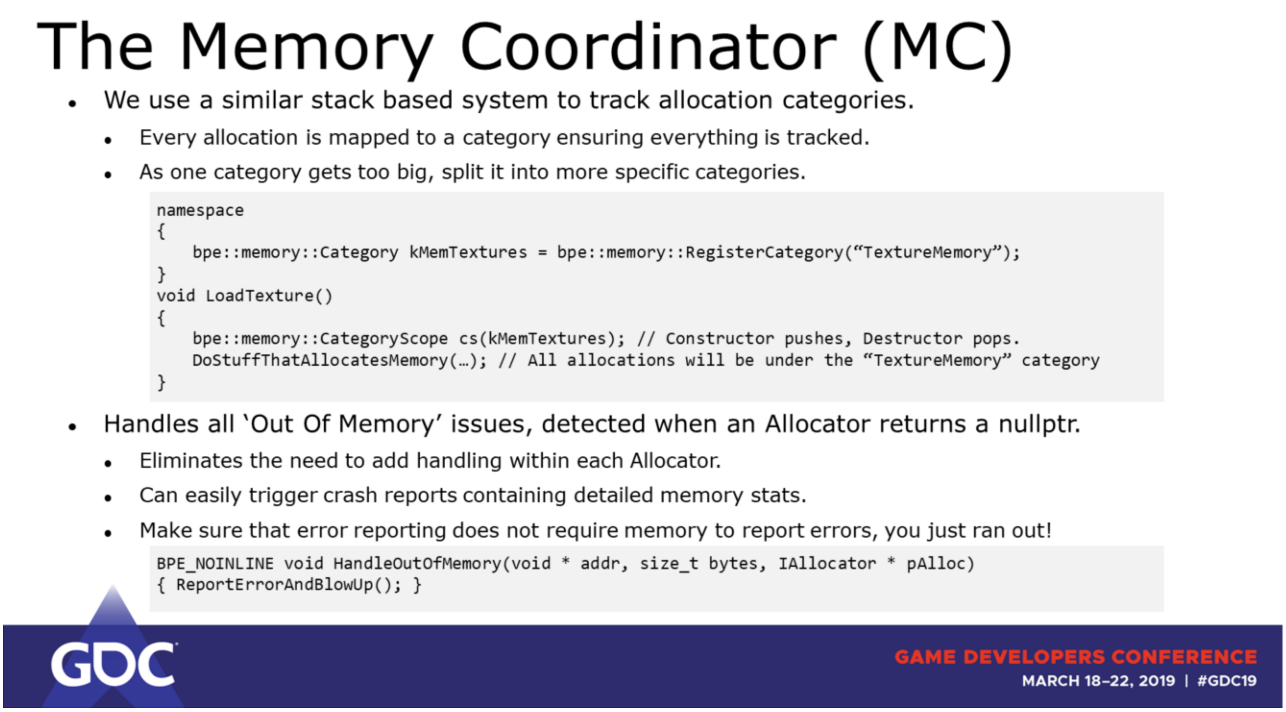
So the MC contains some of the hardest hit code in the entire codebase. It’s primary purpose is to redirect memory requests to the appropriate allocator, all without creating thread contention, thus it must avoiding locking mechanism.

We are able to achieve this goal by ensuring that the MC is basically stateless and the few stack based variables that it forwards into the allocators are all stored using thread local storage.

There is one caveat to keep in mind. Our threaded job system is constructed using fibers that we switch in and out.

Because we are using thread local storage we need to prevent fibers from picking up the wrong set of variables when switched.

So within our job system, if we perform a context switch and switch out a fiber we also create a copy of the memory local thread variables. Then when the job system switches back to the previously suspended fiber we restore the local thread variables. This keeps everyone happy and consistent.



The Memory Coordinator (MC)

- We use a similar stack based system to track allocation categories.
  - Every allocation is mapped to a category ensuring everything is tracked.
  - As one category gets too big, split it into more specific categories.

```
namespace
{
    bpe::memory::Category kMemTextures = bpe::memory::RegisterCategory("TextureMemory");
}
void LoadTexture()
{
    bpe::memory::CategoryScope cs(kMemTextures); // Constructor pushes, Destructor pops.
    DoStuffThatAllocatesMemory(…); // All allocations will be under the "TextureMemory" category
}
```

- Handles all 'Out Of Memory' issues, detected when an Allocator returns a nullptr.
  - Eliminates the need to add handling within each Allocator.
  - Can easily trigger crash reports containing detailed memory stats.
  - Make sure that error reporting does not require memory to report errors, you just ran out!

```
BPE_NOINLINE void HandleOutOfMemory(void * addr, size_t bytes, IAllocator * pAlloc)
{ ReportErrorAndBlowUp(); }
```

GAME DEVELOPERS CONFERENCE
MARCH 18–22, 2019 | #GDC19

By using thread local variables, not only do we allow memory requests to be directed to any specific allocator, but it is also at the heart of how we categorize memory allocations. In our engine every single allocation is categorized, even if it is just categorized as a 'General Allocation'. Our strategy is that once a category starts to account for too much memory, we simply start splitting the category and refining our tracking.

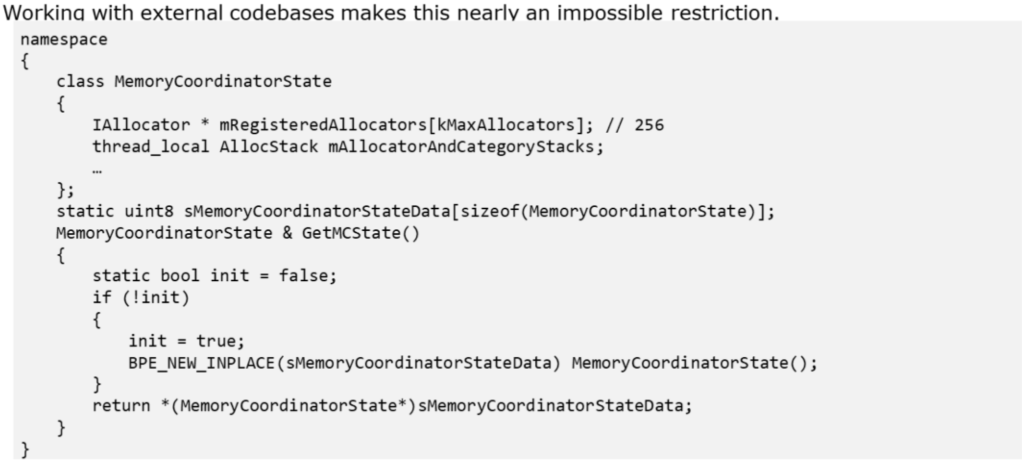
And finally the MC gives us a great single location to track all OutOfMemory issues, eliminating the need to spread the code between individual allocators.

One tip that you can see from the code snippet is that we create a specific function that is never inlined to handle OutOfMemory issues.

This has been great. If QA ever encounters a crash, it is trivial to determine if it was an OutOfMemory issue by directly looking at the extracted callstack, no need to look at the log or try to interpret the line number.

The Memory Coordinator (MC)

- Need to support Pre-Main() allocations and deal with static initialization order issues.
  - In ideal setup there would be zero allocators pre-main() and zero deallocations post-main().
  - Working with external codebases makes this nearly an impossible restriction.

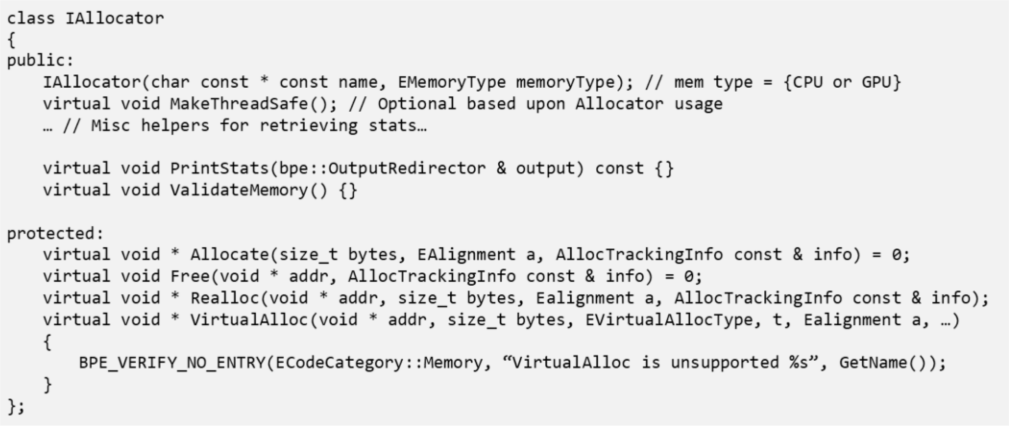


```
namespace
{
    class MemoryCoordinatorState
    {
        IAllocator * mRegisteredAllocators[kMaxAllocators]; // 256
        thread_local AllocStack mAllocatorAndCategoryStacks;
        …
    };
    static uint8 sMemoryCoordinatorStateData[sizeof(MemoryCoordinatorState)];
    MemoryCoordinatorState & GetMCState()
    {
        static bool init = false;
        if (!init)
        {
            init = true;
            BPE_NEW_INPLACE(sMemoryCoordinatorStateData) MemoryCoordinatorState();
        }
        return *(MemoryCoordinatorState*)sMemoryCoordinatorStateData;
    }
}
```

One last trick that I would like to share, while pretty strait forward and really not special, but took a bit of iteration to get correct is how we deal with static initialization order issues. This technic allows for the memory system to be initialized even before static initialization has occurred for this .cpp file. You can check out the snippet if you are interested.

The Allocator (The heart)

- Each Allocator implements a very simple interface that standardizes all requests.
  - We use virtual methods here for dispatching rather than something like FastDelegates to minimize complexity.

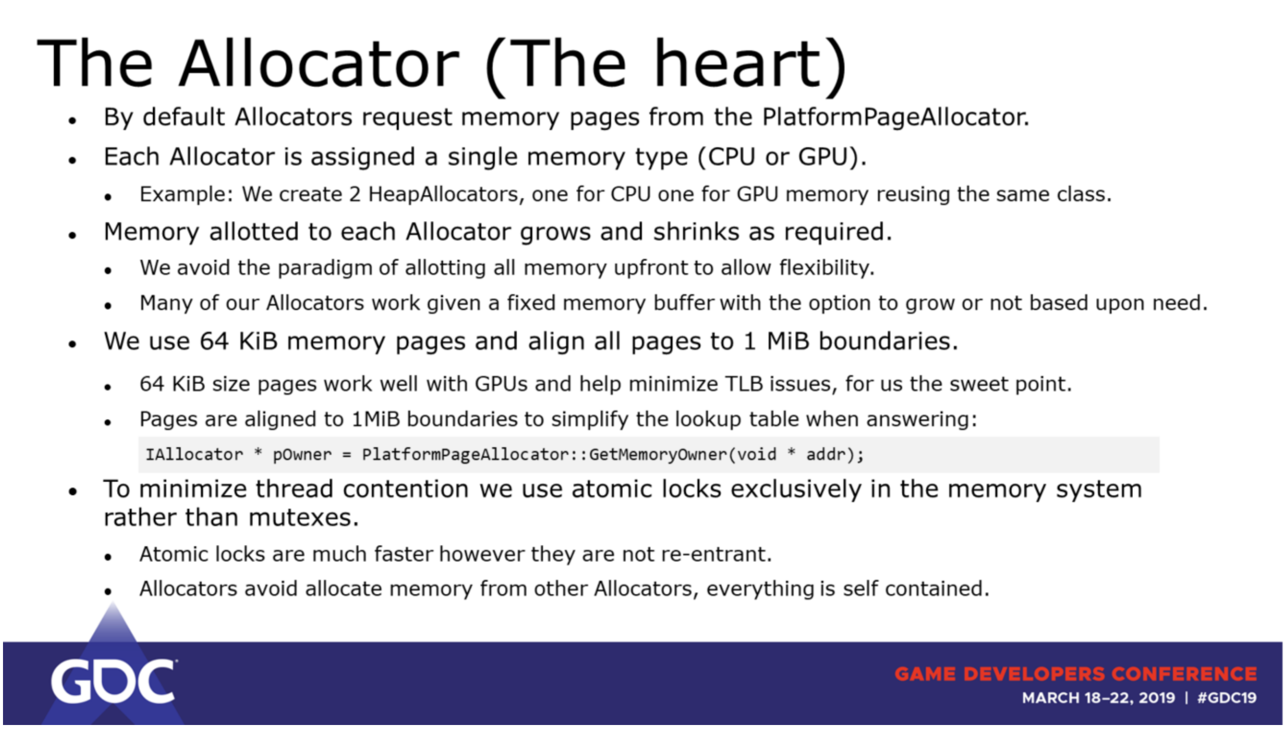


```
class IAllocator
{
public:
    IAllocator(char const * const name, EMemoryType memoryType); // mem type = {CPU or GPU}
    virtual void MakeThreadSafe(); // Optional based upon Allocator usage
    … // Misc helpers for retrieving stats…

    virtual void PrintStats(bpe::OutputRedirector & output) const {}
    virtual void ValidateMemory() {}

protected:
    virtual void * Allocate(size_t bytes, EAlignment a, AllocTrackingInfo const & info) = 0;
    virtual void Free(void * addr, AllocTrackingInfo const & info) = 0;
    virtual void * Realloc(void * addr, size_t bytes, Ealignment a, AllocTrackingInfo const & info);
    virtual void * VirtualAlloc(void * addr, size_t bytes, EVirtualAllocType, t, Ealignment a, …)
    {
        BPE_VERIFY_NO_ENTRY(ECodeCategory::Memory, “VirtualAlloc is unsupported %s”, GetName());
    }
};
```

The allocator, or as I like to think about it, the heart of the system.

This snippet shows portions of the interface that that all allocators must provide. The primary idea here that I want to call your attention to is the fact that we declare all memory routines as protected.

Basically we want to prevent code from directly bypassing the Memory Coordinator. This helps to ensure that all allocators follow consistent conventions and are properly registered with the Memory Coordinator.



The Allocator (The heart)

- By default Allocators request memory pages from the PlatformPageAllocator.
- Each Allocator is assigned a single memory type (CPU or GPU).
  - Example: We create 2 HeapAllocators, one for CPU one for GPU memory reusing the same class.
- Memory allotted to each Allocator grows and shrinks as required.
  - We avoid the paradigm of allotting all memory upfront to allow flexibility.
  - Many of our Allocators work given a fixed memory buffer with the option to grow or not based upon need.
- We use 64 KiB memory pages and align all pages to 1 MiB boundaries.
  - 64 KiB size pages work well with GPUs and help minimize TLB issues, for us the sweet point.
  - Pages are aligned to 1MiB boundaries to simplify the lookup table when answering:

```
IAllocator * pOwner = PlatformPageAllocator::GetMemoryOwner(void * addr);
```

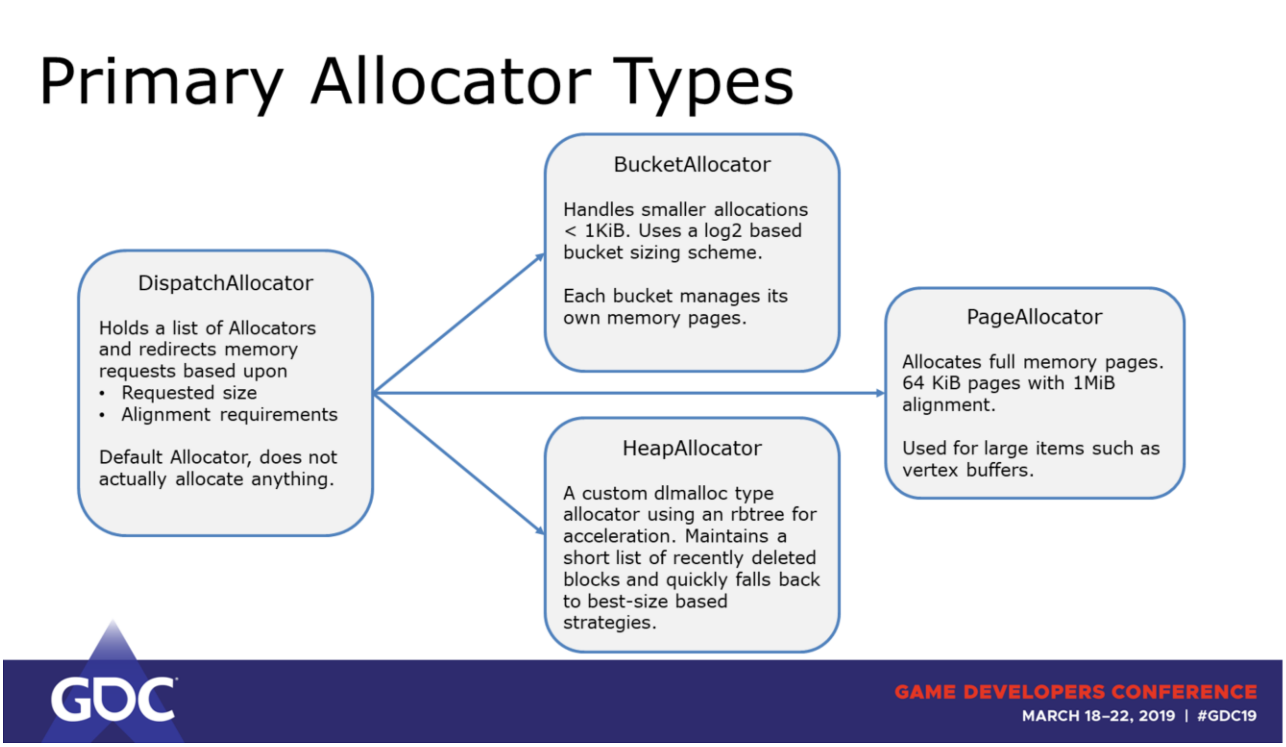
- To minimize thread contention we use atomic locks exclusively in the memory system rather than mutexes.
  - Atomic locks are much faster however they are not re-entrant.
  - Allocators avoid allocate memory from other Allocators, everything is self contained.

GAME DEVELOPERS CONFERENCE
MARCH 18–22, 2019 | #GDC19

I mentioned earlier anytime an allocator needs to grow its memory pool to service a request, it requests new memory pages from the PlatformPageAllocator.

For Shadow, on the PS4, we determined that 64KiB pages were the ideal size. We also align all requested pages to 1MiB boundaries. We wanted smaller page sizes to minimize waist and fragmentation, while also minimizing TLB issues. For us, 64KiB pages is the sweet spot.

I also want to point out that when coding allocators, we strictly avoid the use of mutexes. While allocators need to deal with threading issues we stick to using atomic locks. These are significantly faster and the restriction of not being re-entrant is easy to work around.



Primary Allocator Types

DispatchAllocator

Holds a list of Allocators and redirects memory requests based upon
- Requested size
- Alignment requirements

Default Allocator, does not actually allocate anything.

BucketAllocator

Handles smaller allocations < 1KiB. Uses a log2 based bucket sizing scheme.

Each bucket manages its own memory pages.

HeapAllocator

A custom dlmalloc type allocator using an rbtree for acceleration. Maintains a short list of recently deleted blocks and quickly falls back to best-size based strategies.

PageAllocator

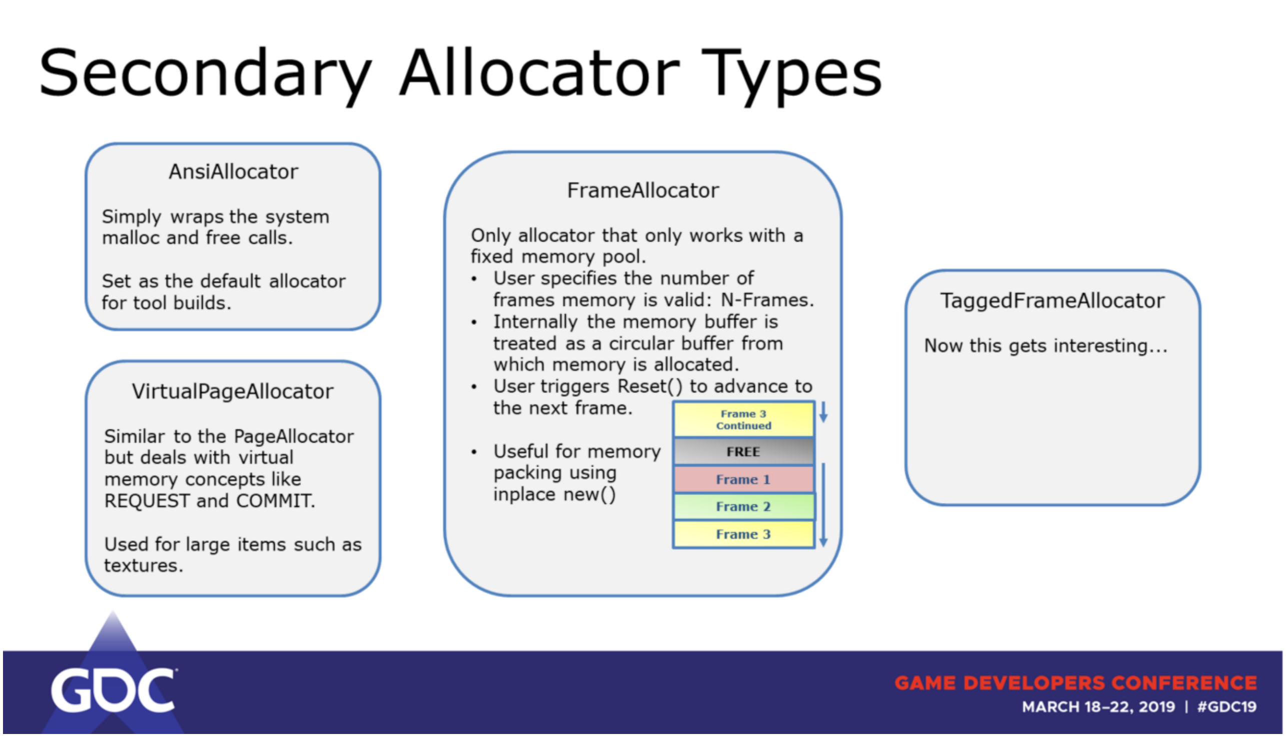
Allocates full memory pages. 64 KiB pages with 1MiB alignment.

Used for large items such as vertex buffers.

So let's take a look at a few of the different types of allocators that we support.

The DispatchAllocator is the default allocator that the majority of all allocations get redirect to. It is a very simple redirector which basically looks at the alignment and size requirements of the request and forwards them to the appropriate allocator.

- We use a BucketAllocator for small allocations.
- We use a HeapAllocator which is similar to dlmalloc. It uses a intrusive red black tree to store free blocks and uses a heuristic of preferring recently freed memory followed by best fit. We then add headers and post-headers to the allocations to provide tracking information.
- And to round it off we use a PageAllocator for large memory requests that fit nicely within our 64KiB pages.



Secondary Allocator Types

AnsiAllocator

Simply wraps the system malloc and free calls.

Set as the default allocator for tool builds.

VirtualPageAllocator

Similar to the PageAllocator but deals with virtual memory concepts like REQUEST and COMMIT.

Used for large items such as textures.

FrameAllocator

Only allocator that only works with a fixed memory pool.

- User specifies the number of frames memory is valid: N-Frames.
- Internally the memory buffer is treated as a circular buffer from which memory is allocated.
- User triggers Reset() to advance to the next frame.
- Useful for memory packing using inplace new()

Frame 3 Continued

FREE

Frame 1

Frame 2

Frame 3

TaggedFrameAllocator

Now this gets interesting…

GDC

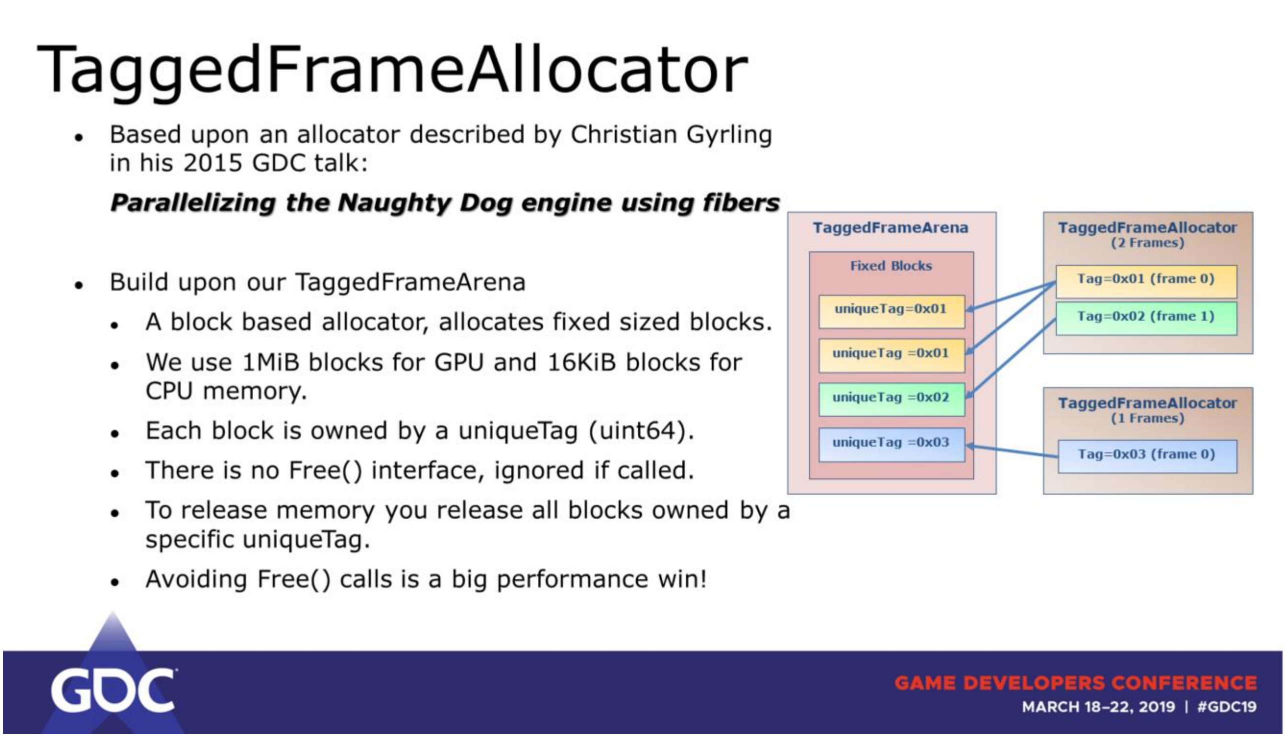
GAME DEVELOPERS CONFERENCE

MARCH 18–22, 2019 | #GDC19

We also have several special use allocators.

- A simple AnsiAllocator that is only used by tool code.
- There is a VirtualPageAllocator to handle virtual memory requests.
- And there is a FrameAllocator which is great for:
- Packing multiple request together to ensure memory coherency.
- And creating small block of easily reusable memory.

And let’s not forget about the TaggedFrameAllocator which is awesome.



TaggedFrameAllocator

- Based upon an allocator described by Christian Gyrling in his 2015 GDC talk:

  Parallelizing the Naughty Dog engine using fibers

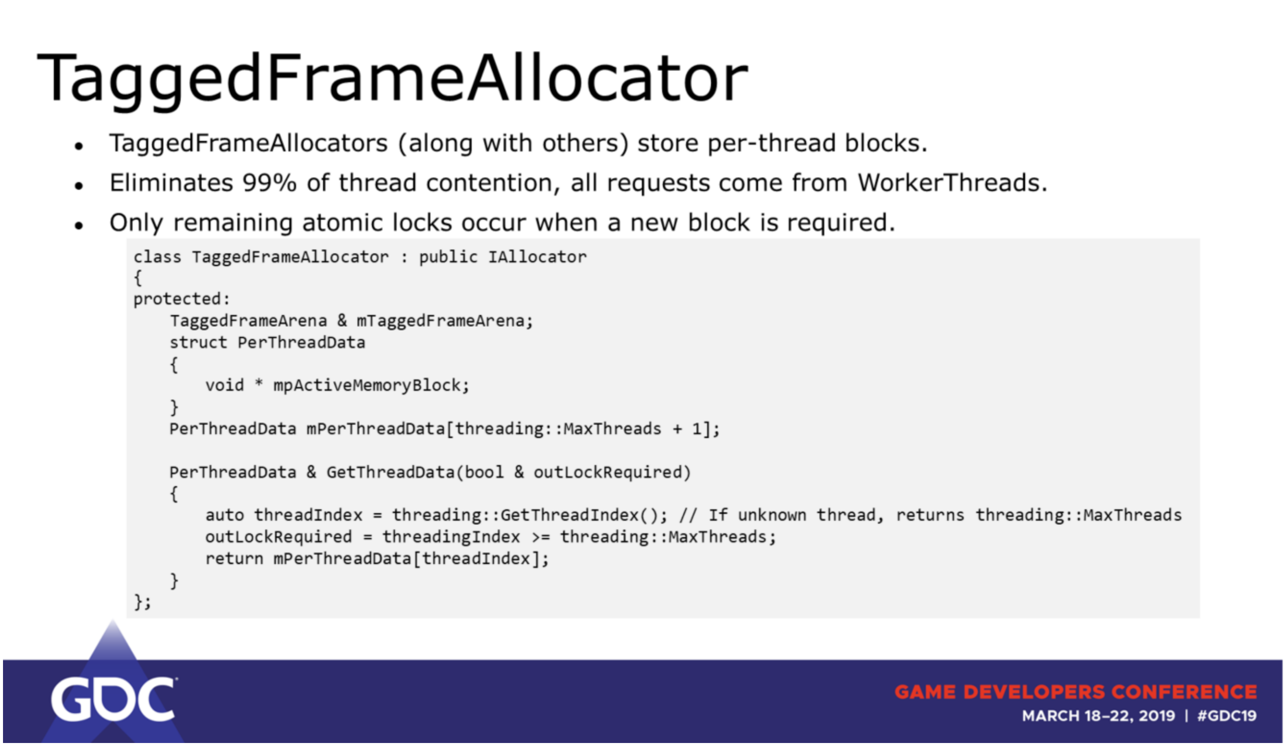
- Build upon our TaggedFrameArena
  - A block based allocator, allocates fixed sized blocks.
  - We use 1MiB blocks for GPU and 16KiB blocks for CPU memory.
  - Each block is owned by a uniqueTag (uint64).
  - There is no Free() interface, ignored if called.
  - To release memory you release all blocks owned by a specific uniqueTag.
  - Avoiding Free() calls is a big performance win!

Our TaggedFrameAllocator is based upon an allocator described by Christian Gyrling in his 2015 GDC talk.

The basic idea behind the TaggedFrameAllocator is that we are creating an extremely fast allocator where allocations are only valid for a short period of time. These are basically temporary memory allocations that are valid for N number of frames, where a frame is arbitrarily defined for each allocator.

The fact that users don’t need to track the memory or call free() to release memory is a huge win. Not only is the code requesting the memory easier to write, the TaggedFrameAllocator can take a bunch of shortcuts knowing that individual free() calls do not need to be supported.

It is also impossible to create memory leaks when using this allocator.



TaggedFrameAllocator

- TaggedFrameAllocators (along with others) store per-thread blocks.
- Eliminates 99% of thread contention, all requests come from WorkerThreads.
- Only remaining atomic locks occur when a new block is required.

```
class TaggedFrameAllocator : public IAllocator
{
protected:
    TaggedFrameArena & mTaggedFrameArena;
    struct PerThreadData
    {
        void * mpActiveMemoryBlock;
    }
    PerThreadData mPerThreadData[threading::MaxThreads + 1];

    PerThreadData & GetThreadData(bool & outLockRequired)
    {
        auto threadIndex = threading::GetThreadIndex(); // If unknown thread, returns threading::MaxThreads
        outLockRequired = threadingIndex >= threading::MaxThreads;
        return mPerThreadData[threadIndex];
    }
};
```

GDC

GAME DEVELOPERS CONFERENCE
MARCH 18–22, 2019 | #GDC19

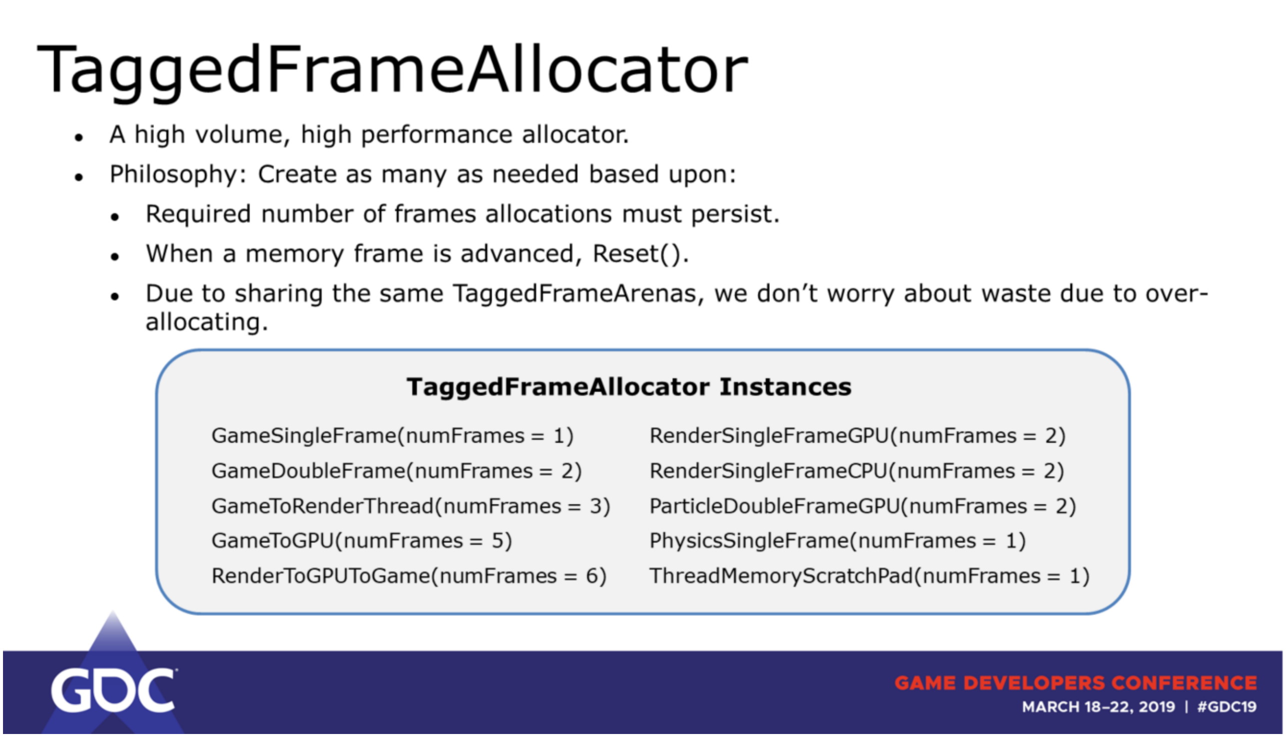
We discussed earlier that one of the goals behind allocators, is that they should help to minimize thread contention.

This is achieved using two strategies.

First we use atomic locks rather than mutexes.

And secondly by using thread local storage.

Several of our allocators use the pattern illustrated in the code snippet. For the TaggedFrameAllocator, this allows it to service memory request for multiple threads simultaneously without ever locking. If you have ever had to deal with multi-threaded performance this should immediately jump out as a huge win.



TaggedFrameAllocator

- A high volume, high performance allocator.
- Philosophy: Create as many as needed based upon:
  - Required number of frames allocations must persist.
  - When a memory frame is advanced, Reset().
  - Due to sharing the same TaggedFrameArenas, we don't worry about waste due to over-allocating.

TaggedFrameAllocator Instances

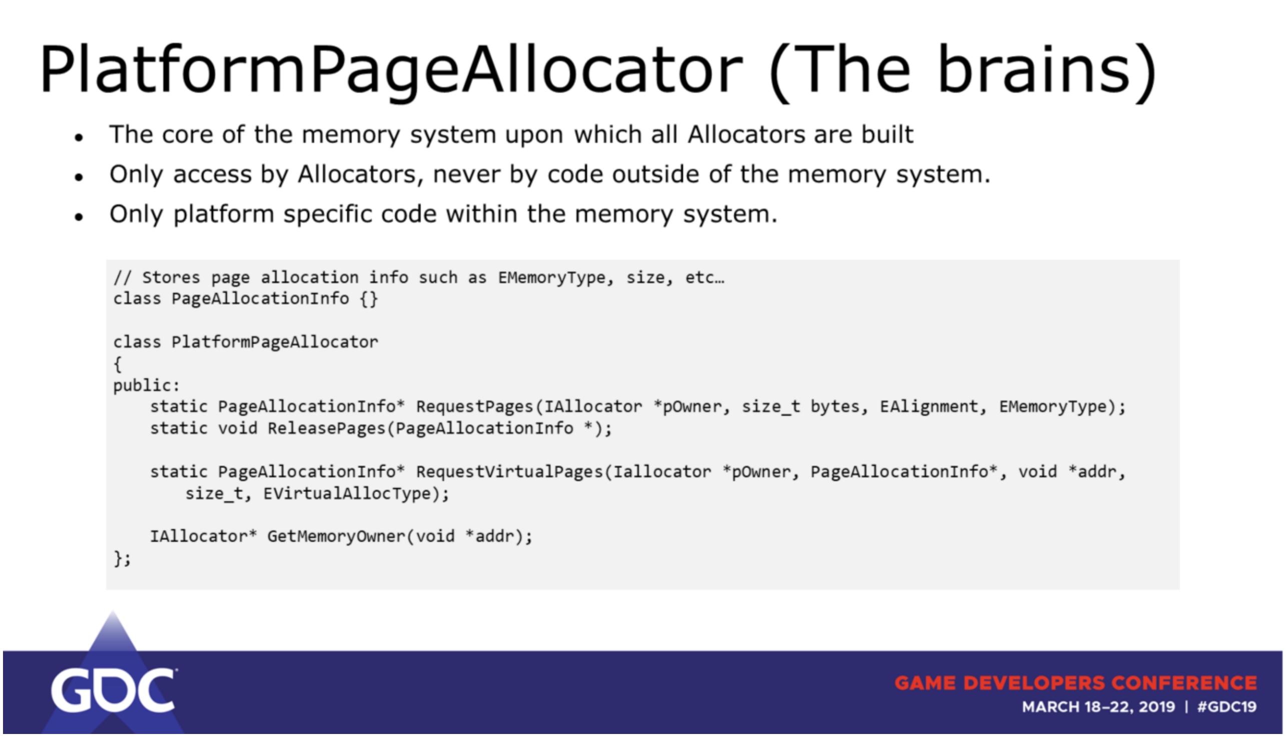
GameSingleFrame(numFrames = 1)
GameDoubleFrame(numFrames = 2)
GameToRenderThread(numFrames = 3)
GameToGPU(numFrames = 5)
RenderToGPUToGame(numFrames = 6)
RenderSingleFrameGPU(numFrames = 2)
RenderSingleFrameCPU(numFrames = 2)
ParticleDoubleFrameGPU(numFrames = 2)
PhysicsSingleFrame(numFrames = 1)
ThreadMemoryScratchPad(numFrames = 1)

Within the Bluepoint Engine, there are more TaggedFrameAllocators that any other type of allocator.

Each TaggedFrameAllocator is built ontop of a shared TaggedFrameArena that allows all TaggedFrameAllocators to share a common memory pool, eliminating over allocation issues.

The TaggedFrameArena will also grow and shrink upon demand, thus all pool sizes are dynamic.

If you don't have something similar to this in your code base I would highly recommend you make it happen.



PlatformPageAllocator (The brains)

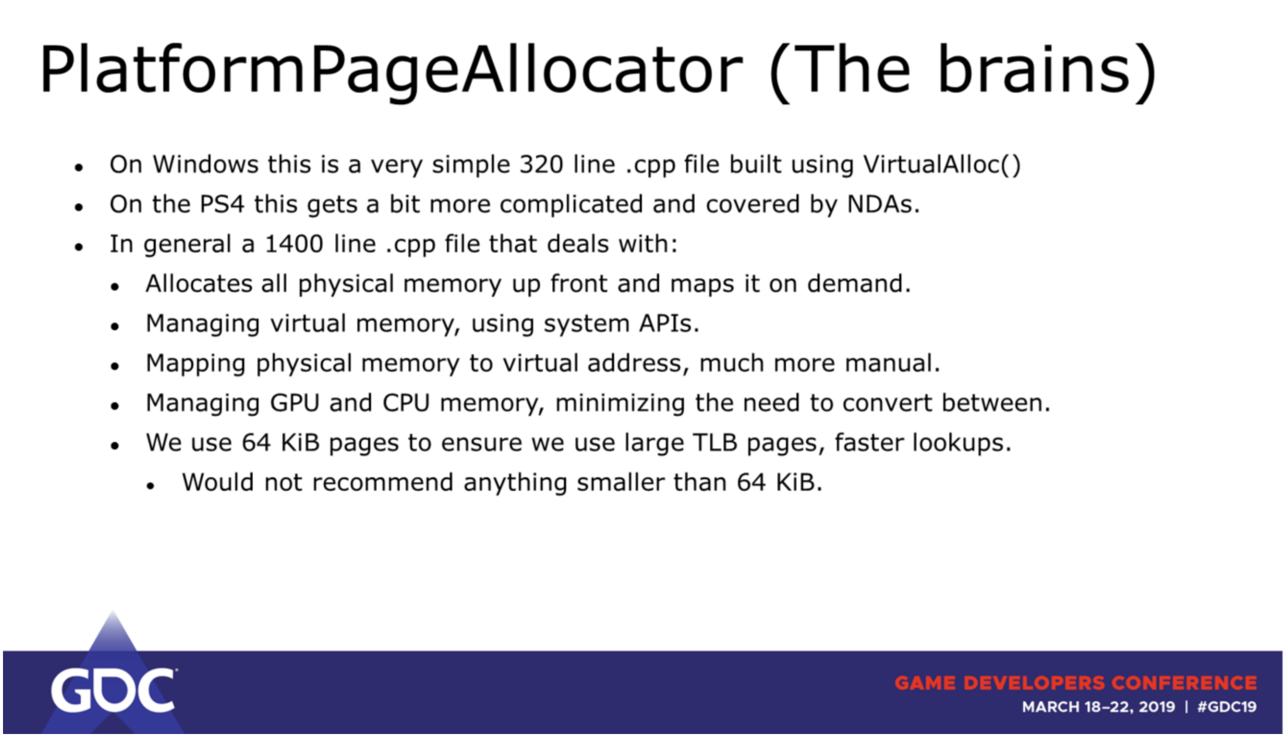
- The core of the memory system upon which all Allocators are built
- Only access by Allocators, never by code outside of the memory system.
- Only platform specific code within the memory system.

```
// Stores page allocation info such as EMemoryType, size, etc…
class PageAllocationInfo {}

class PlatformPageAllocator
{
public:
    static PageAllocationInfo* RequestPages(IAllocator *pOwner, size_t bytes, EAlignment, EMemoryType);
    static void ReleasePages(PageAllocationInfo *);

    static PageAllocationInfo* RequestVirtualPages(Iallocator *pOwner, PageAllocationInfo*, void *addr,
        size_t, EVirtualAllocType);

    IAllocator* GetMemoryOwner(void *addr);
};
```

And finally the PlatformPageAllocator, or the brains of the operation.

As we have discuss this class is responsible for abstracting away the details of how 64KiB pages are managed per platform.

In practice, we restrict anyone from accessing this class directly. The only customers of this class, are the memory allocators.



PlatformPageAllocator (The brains)

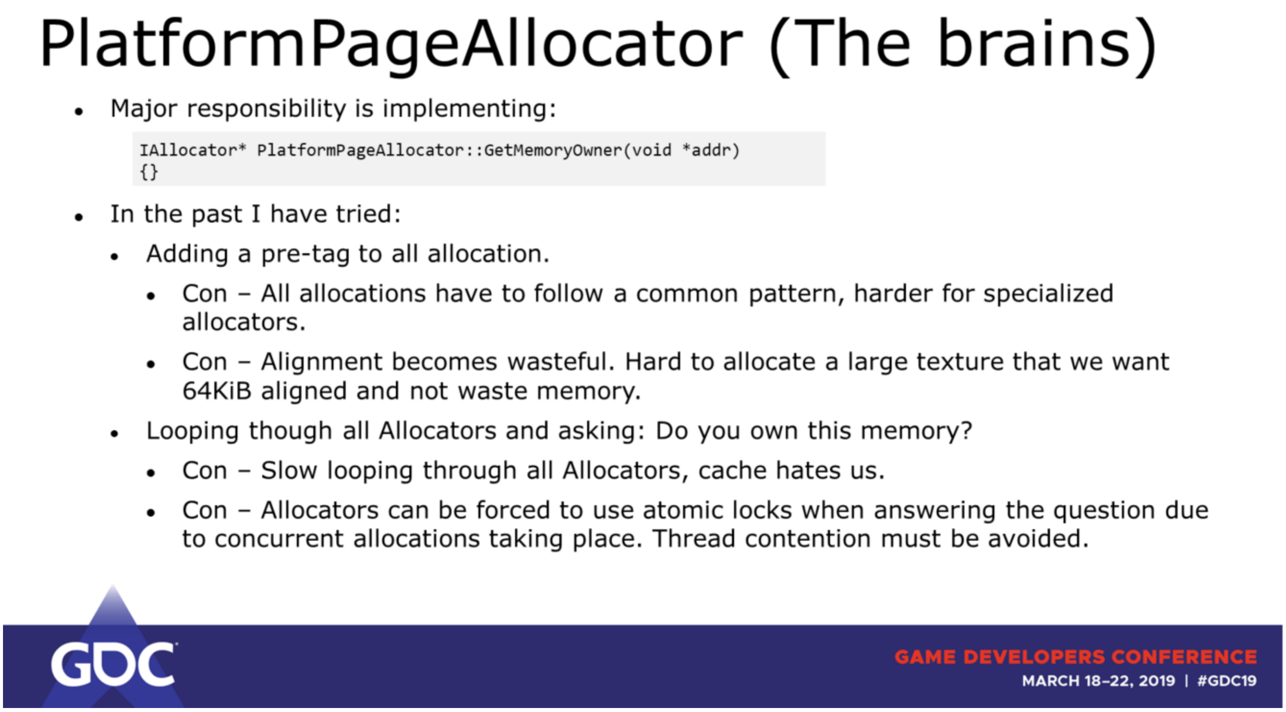
- On Windows this is a very simple 320 line .cpp file built using VirtualAlloc()
- On the PS4 this gets a bit more complicated and covered by NDAs.
- In general a 1400 line .cpp file that deals with:
  - Allocates all physical memory up front and maps it on demand.
  - Managing virtual memory, using system APIs.
  - Mapping physical memory to virtual address, much more manual.
  - Managing GPU and CPU memory, minimizing the need to convert between.
  - We use 64 KiB pages to ensure we use large TLB pages, faster lookups.
    - Would not recommend anything smaller than 64 KiB.

Our windows version is very simple and is built using VirtualAlloc().

The PS4 version is more complex due to platform considerations.

At the heart of both implementations is the concept of virtual memory and mapping physical memory to virtual addresses on demand.

If you are not familiar with the differences between virtual and physical memory you might want to investigate.



PlatformPageAllocator (The brains)

- Major responsibility is implementing:

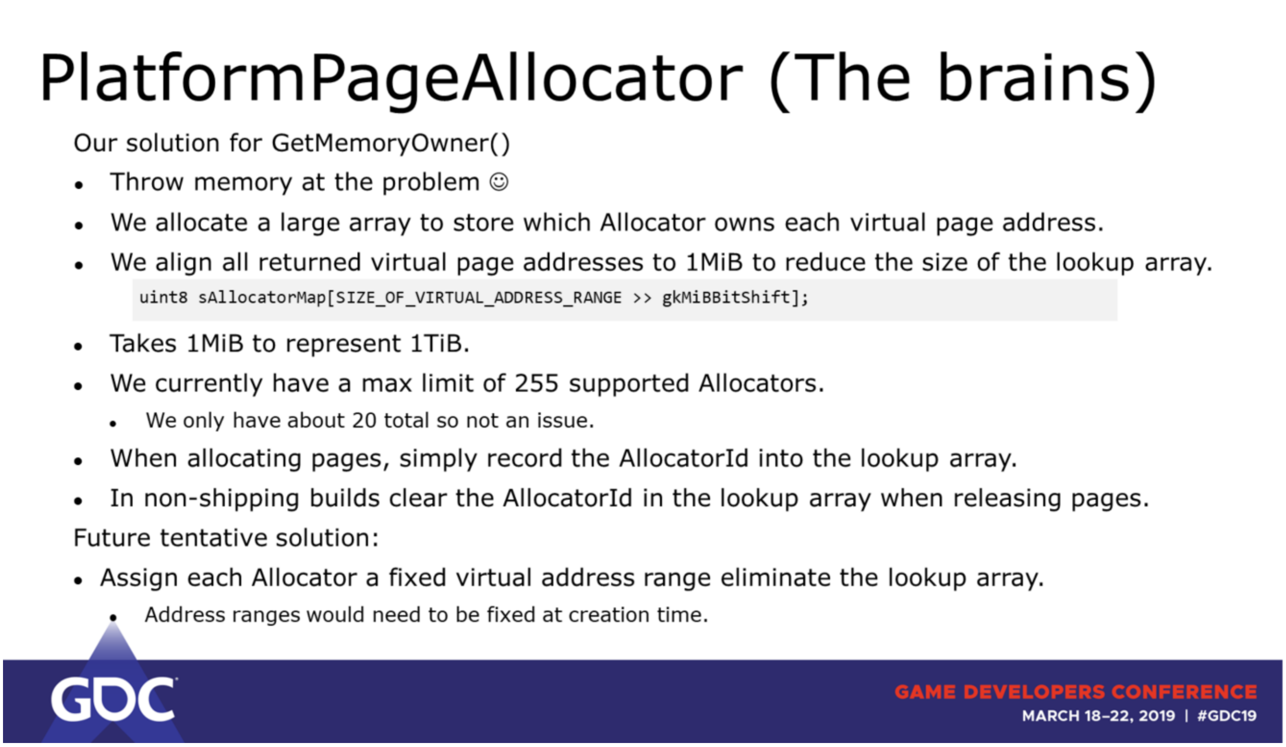
```
IAllocator* PlatformPageAllocator::GetMemoryOwner(void *addr)
{}
```

- In the past I have tried:
  - Adding a pre-tag to all allocation.
    - Con – All allocations have to follow a common pattern, harder for specialized allocators.
    - Con – Alignment becomes wasteful. Hard to allocate a large texture that we want 64KiB aligned and not waste memory.
  - Looping though all Allocators and asking: Do you own this memory?
    - Con – Slow looping through all Allocators, cache hates us.
    - Con – Allocators can be forced to use atomic locks when answering the question due to concurrent allocations taking place. Thread contention must be avoided.

Looking at the PlatformPageAllocator there is one last problem that I would like to discuss which has been a problem in every non uber memory scheme I have worked with.

Given that we are avoiding an uber allocator and embracing numerous allocation patterns, we need to be able to take any random memory address and determine which allocator it belongs to. This is required to properly direct free() calls.

While BPE macros allow you to optionally specify which allocator owns memory when calling BPE_FREE(), this is not ideal. Instead we need to support the GetMemoryOwner() routine.



PlatformPageAllocator (The brains)

Our solution for GetMemoryOwner()

- Throw memory at the problem ☺
- We allocate a large array to store which Allocator owns each virtual page address.
- We align all returned virtual page addresses to 1MiB to reduce the size of the lookup array.

```
uint8 sAllocatorMap[SIZE_OF_VIRTUAL_ADDRESS_RANGE >> gkMiBBitShift];
```

- Takes 1MiB to represent 1TiB.
- We currently have a max limit of 255 supported Allocators.
  - We only have about 20 total so not an issue.
- When allocating pages, simply record the AllocatorId into the lookup array.
- In non-shipping builds clear the AllocatorId in the lookup array when releasing pages.

Future tentative solution:

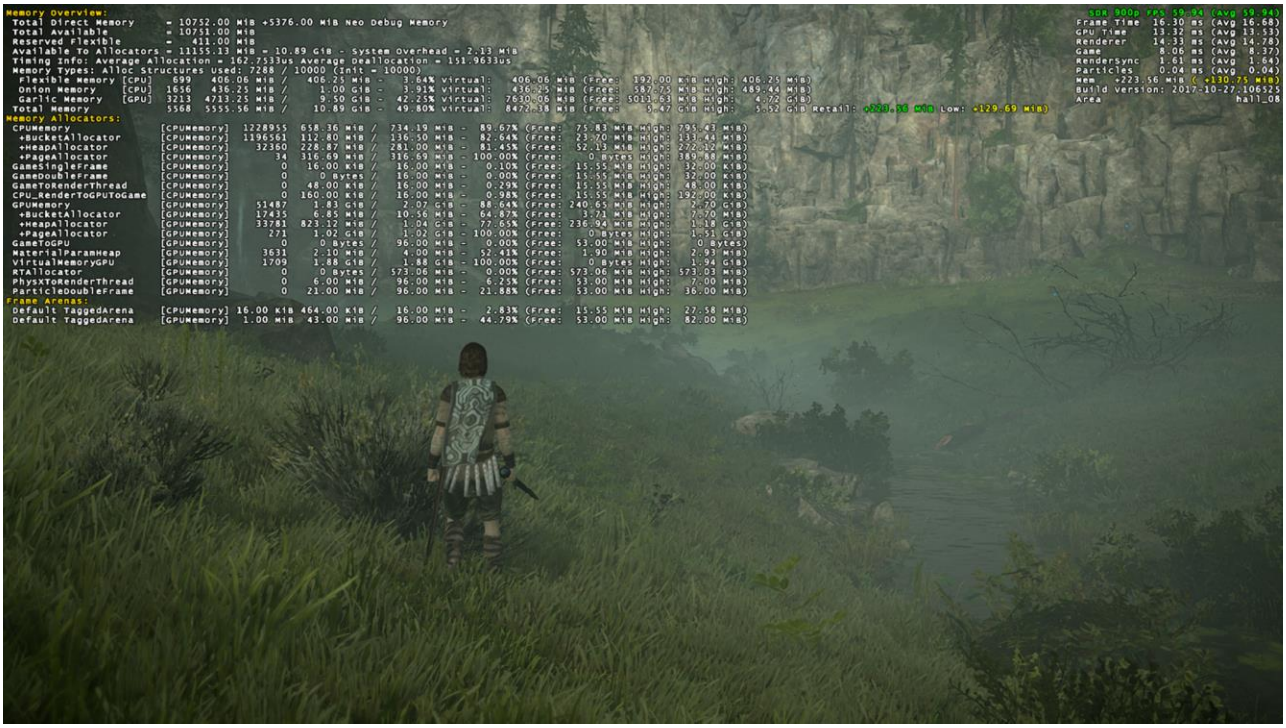
- Assign each Allocator a fixed virtual address range eliminate the lookup array.
  - Address ranges would need to be fixed at creation time.

So how did we solve this problem. We threw memory at the problem.

We break up the problem by realizing that we only need to determine which allocator owns the memory page that contains the memory address in question.

Within the PlatformPageAllocator we use a 1MiB block of memory to store a direct lookup table which maps 1TiB of virtual memory to its owning allocator.

If you have a better solution I would love to hear about it.



```
Memory Overview:
 Total Direct Memory      = 10752.00 MiB +5376.00 MiB Neo Debug Memory
 Total Available          = 10751.00 MiB
 Reserved Flexible        =   411.00 MiB
 Available To Allocators = 11155.13 MiB = 10.89 GiB - System Overhead = 2.13 MiB
 Timing Info: Average Allocation = 162.7533us Average Deallocation = 151.9633us
 Memory Types: Alloc Structures Used: 7288 / 10000 (Init = 10000)
  Flexible Memory [CPU]   699   406.06 MiB /   406.25 MiB -   3.64% Virtual:   406.06 MiB (Free:  192.00 KiB High: 406.25 MiB)
  Onion Memory    [CPU]  1656   436.25 MiB /     1.00 GiB -   3.91% Virtual:   436.25 MiB (Free:  587.75 MiB High: 489.44 MiB)
  Garlic Memory   [GPU]  3213  4713.25 MiB /     9.50 GiB -  42.25% Virtual:  7630.06 MiB (Free: 5011.63 MiB High:   4.72 GiB)
 Total Memory            5568  5555.56 MiB /    10.89 GiB -  49.80% Virtual:  8472.38 MiB (Free:     5.47 GiB High:   5.52 GiB Retail: +223.56 MiB Low: +129.69 MiB)
Memory Allocators:
 CPUMemory               [CPUMemory]  1228955  658.36 MiB /  734.19 MiB -  89.67% (Free:  75.83 MiB High: 795.43 MiB)
  +BucketAllocator       [CPUMemory]  1196561  112.80 MiB /  136.50 MiB -  82.64% (Free:  23.70 MiB High: 133.44 MiB)
  +HeapAllocator         [CPUMemory]    32360  228.87 MiB /  281.00 MiB -  81.45% (Free:  52.13 MiB High: 272.12 MiB)
  +PageAllocator         [CPUMemory]       34  316.69 MiB /  316.69 MiB - 100.00% (Free:    0 Bytes High: 389.88 MiB)
 GameSingleFrame         [CPUMemory]        0   16.00 KiB /   16.00 MiB -   0.10% (Free:  15.55 MiB High:  32.00 KiB)
 GameDoubleFrame         [CPUMemory]        0     0 Bytes /   16.00 MiB -   0.00% (Free:  15.55 MiB High:  32.00 KiB)
 GameToRenderThread      [CPUMemory]        0   48.00 KiB /   16.00 MiB -   0.29% (Free:  15.55 MiB High:  48.00 KiB)
 CPU_RenderToGPUToGame   [CPUMemory]        0  160.00 KiB /   16.00 MiB -   0.98% (Free:  15.55 MiB High: 192.00 KiB)
 GPUMemory               [GPUMemory]    51487    1.83 GiB /    2.07 GiB -  88.64% (Free: 240.65 MiB High:   2.70 GiB)
  +BucketAllocator       [GPUMemory]    17435    6.85 MiB /   10.56 MiB -  64.87% (Free:   3.71 MiB High:   7.70 MiB)
  +HeapAllocator         [GPUMemory]    33781  823.12 MiB /    1.04 GiB -  77.65% (Free: 236.94 MiB High:   1.18 GiB)
  +PageAllocator         [GPUMemory]      271    1.02 GiB /    1.02 GiB - 100.00% (Free:    0 Bytes High:   1.51 GiB)
 GameToGPU               [GPUMemory]        0     0 Bytes /   96.00 MiB -   0.00% (Free:  53.00 MiB High:    0 Bytes)
 MaterialParamHeap       [GPUMemory]     3631    2.10 MiB /    4.00 MiB -  52.41% (Free:   1.90 MiB High:   2.93 MiB)
 VirtualMemoryGPU        [GPUMemory]     1709    1.88 GiB /    1.88 GiB - 100.00% (Free:    0 Bytes High:   1.94 GiB)
 RTAllocator             [GPUMemory]        0     0 Bytes /  573.06 MiB -   0.00% (Free: 573.06 MiB High: 573.03 MiB)
 PhysXToRenderThread     [GPUMemory]        0    6.00 MiB /   96.00 MiB -   6.25% (Free:  53.00 MiB High:   7.00 MiB)
 ParticleDoubleFrame     [GPUMemory]        0   21.00 MiB /   96.00 MiB -  21.88% (Free:  53.00 MiB High:  36.00 MiB)
Frame Arenas:
 Default TaggedArena     [CPUMemory] 16.00 KiB 464.00 KiB /   16.00 MiB -   2.83% (Free:  15.55 MiB High:  27.58 MiB)
 Default TaggedArena     [GPUMemory]  1.00 MiB  43.00 MiB /   96.00 MiB -  44.79% (Free:  53.00 MiB High:  82.00 MiB)
```

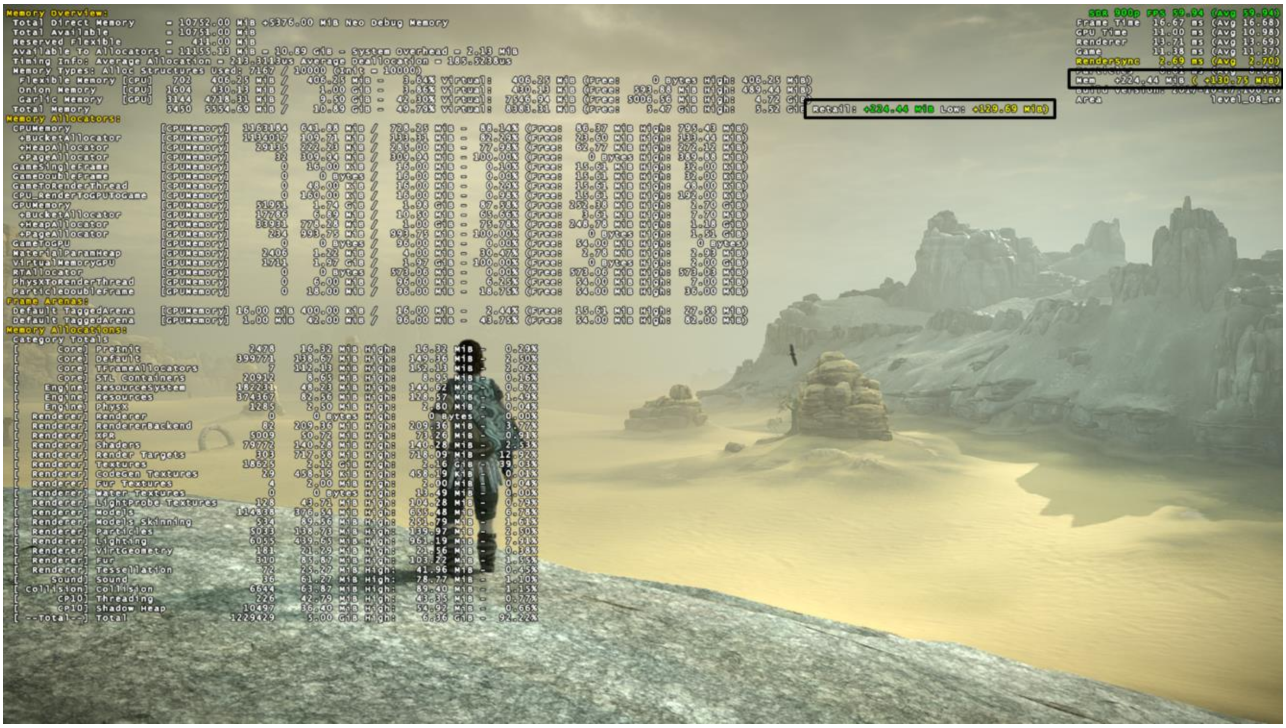
```
   SDR 900p FPS 59.94 (Avg 59.94)
Frame Time   16.30 ms (Avg 16.68)
GPU Time     13.32 ms (Avg 13.53)
Renderer     14.33 ms (Avg 14.78)
Game          8.06 ms (Avg  8.37)
RenderSync    1.61 ms (Avg  1.64)
Particles     0.04 ms (Avg  0.04)
Mem    +223.56 MiB ( +130.75 MiB)
Build Version: 2017-10-27.106525
Area                       hall_08
```

So what do you get when you put it all together? Hopefully a game without performance issues that runs within retail memory.

Ok, that isn't going to happen.

But hopefully you have the flexibility and tools necessary to get there from a memory perspective.

Here is a screenshot showing our memory statistics. You can see the PlatformPageAllocator at the top, and each of the various allocators below.



```
Memory Overview:
 Total Direct Memory      = 10752.00 MiB +5376.00 MiB Neo Debug Memory
 Total Available          = 10751.00 MiB
 Reserved Flexible        =   411.00 MiB
 Available To Allocators = 11155.13 MiB = 10.89 GiB - System Overhead = 2.13 MiB
 Timing Info: Average Allocation = 213.3113us Average Deallocation = 185.5238us
 Memory Types: Alloc Structures Used: 7167 / 10000 (Init = 10000)
  Flexible Memory [CPU]   702   406.25 MiB /   406.25 MiB =   3.64% Virtual:   406.25 MiB (Free:      0 Bytes High: 406.25 MiB)
  Onion Memory    [CPU]  1604   430.13 MiB /     1.00 GiB =   3.86% Virtual:   430.13 MiB (Free:  593.88 MiB High: 489.44 MiB)
  Garlic Memory   [GPU]  3144  4718.31 MiB /     9.50 GiB =  42.30% Virtual:  7546.94 MiB (Free: 5006.56 MiB High:   4.72 GiB)
 Total Memory            5450  5554.69 MiB /    10.89 GiB =  49.79% Virtual:  8383.31 MiB (Free:    5.47 GiB High:   5.52 GiB  Retail: +224.44 MiB Low: +129.69 MiB)
Memory Allocators:
 CPUMemory                  [CPUMemory] 1163184  641.88 MiB /  728.25 MiB =  88.14% (Free:  86.37 MiB High: 795.43 MiB)
  +BucketAllocator          [CPUMemory] 1134017  109.71 MiB /  133.31 MiB =  82.29% (Free:  23.60 MiB High: 133.44 MiB)
  +HeapAllocator            [CPUMemory]   29135  222.23 MiB /  285.00 MiB =  77.98% (Free:  62.77 MiB High: 272.12 MiB)
  +PageAllocator            [CPUMemory]      32  309.94 MiB /  309.94 MiB = 100.00% (Free:    0 Bytes High: 389.88 MiB)
 GameSingleFrame            [CPUMemory]       0   16.00 KiB /   16.00 MiB =   0.10% (Free:  15.61 MiB High:  32.00 KiB)
 GameDoubleFrame            [CPUMemory]       0     0 Bytes /   16.00 MiB =   0.00% (Free:  15.61 MiB High:  32.00 KiB)
 GameToRenderThread         [CPUMemory]       0   48.00 KiB /   16.00 MiB =   0.29% (Free:  15.61 MiB High:  48.00 KiB)
 CPU_RenderToGPUToGame      [CPUMemory]       0  160.00 KiB /   16.00 MiB =   0.98% (Free:  15.61 MiB High: 192.00 KiB)
 GPUMemory                  [GPUMemory]   51951    1.74 GiB /    1.98 GiB =  87.58% (Free: 252.38 MiB High:   2.70 GiB)
  +BucketAllocator          [GPUMemory]   17786    6.89 MiB /   10.50 MiB =  65.66% (Free:   3.61 MiB High:   7.70 MiB)
  +HeapAllocator            [GPUMemory]   33931  778.28 MiB /    1.00 GiB =  75.78% (Free: 248.78 MiB High:   1.18 GiB)
  +PageAllocator            [GPUMemory]     234  993.75 MiB /  993.75 MiB = 100.00% (Free:    0 Bytes High:   1.51 GiB)
 GameToGPU                  [GPUMemory]       0     0 Bytes /   96.00 MiB =   0.00% (Free:  54.00 MiB High:    0 Bytes)
 MaterialParamHeap          [GPUMemory]    2406    1.22 MiB /    4.00 MiB =  30.47% (Free:   2.78 MiB High:   2.93 MiB)
 VirtualMemoryGPU           [GPUMemory]    1711    1.97 GiB /    1.97 GiB = 100.00% (Free:    0 Bytes High:   2.00 GiB)
 RTAllocator                [GPUMemory]       0     0 Bytes /  573.06 MiB =   0.00% (Free: 573.06 MiB High: 573.03 MiB)
 PhysXToRenderThread        [GPUMemory]       0    6.00 MiB /   96.00 MiB =   6.25% (Free:  54.00 MiB High:   7.00 MiB)
 ParticleDoubleFrame        [GPUMemory]       0   18.00 MiB /   96.00 MiB =  18.75% (Free:  54.00 MiB High:  36.00 MiB)
Frame Arenas:
 Default TaggedArena        [CPUMemory] 16.00 KiB 400.00 KiB /   16.00 MiB =   2.44% (Free:  15.61 MiB High:  27.58 MiB)
 Default TaggedArena        [GPUMemory]  1.00 MiB  42.00 MiB /   96.00 MiB =  43.75% (Free:  54.00 MiB High:  82.00 MiB)
Memory Allocations:
 Category Totals
 [        Core] PreInit                 2478   16.32 MiB High:  16.32 MiB -  0.29%
 [        Core] Default               399771  138.67 MiB High: 149.36 MiB -  2.50%
 [        Core] TFrameAllocators           7  112.13 MiB High: 152.13 MiB -  2.02%
 [        Core] STL Containers         20912    8.65 MiB High:   8.95 MiB -  0.16%
 [      Engine] ResourceSystem        182231   48.23 MiB High: 144.62 MiB -  0.87%
 [      Engine] Resources             374367   82.56 MiB High: 128.57 MiB -  1.49%
 [      Engine] PhysX                   1285    2.50 MiB High:   2.80 MiB -  0.04%
 [    Renderer] Renderer                   0     0 Bytes High:    0 Bytes -  0.00%
 [    Renderer] RendererBackend           82  209.36 MiB High: 209.36 MiB -  3.77%
 [    Renderer] XPR                     5009   50.72 MiB High:  73.26 MiB -  0.91%
 [    Renderer] Shaders                79772  140.28 MiB High: 140.28 MiB -  2.53%
 [    Renderer] Render Targets           303  717.58 MiB High: 718.09 MiB - 12.92%
 [    Renderer] Textures               18625    2.12 GiB High:   2.16 GiB - 39.03%
 [    Renderer] CodeGen Textures          29  458.19 KiB High: 458.19 KiB -  0.01%
 [    Renderer] Fur Textures               4    2.00 MiB High:   2.00 MiB -  0.04%
 [    Renderer] Water Textures             0     0 Bytes High:  13.49 MiB -  0.00%
 [    Renderer] LightProbe Textures      128   43.71 MiB High: 104.28 MiB -  0.79%
 [    Renderer] Models                114838  376.54 MiB High: 655.48 MiB -  6.78%
 [    Renderer] Models Skinning          534   89.56 MiB High: 251.79 MiB -  1.61%
 [    Renderer] Particles               5033  138.73 MiB High: 139.97 MiB -  2.50%
 [    Renderer] Lighting                6055  439.65 MiB High: 961.19 MiB -  7.91%
 [    Renderer] VirtGeometry             181   21.29 MiB High:  21.56 MiB -  0.38%
 [    Renderer] Fur                      310   85.87 MiB High: 103.22 MiB -  1.55%
 [    Renderer] Tessellation              72   25.27 MiB High:  41.96 MiB -  0.45%
 [       Sound] Sound                     36   61.27 MiB High:  78.77 MiB -  1.10%
 [   Collision] Collision               6644   63.87 MiB High:  89.40 MiB -  1.15%
 [        CP10] Threading                226   42.79 MiB High:  43.35 MiB -  0.77%
 [        CP10] Shadow Heap            10497   36.40 MiB High:  54.92 MiB -  0.66%
 [ --Total--] Total                  1229429    5.00 GiB High:   6.36 GiB - 92.22%
```

```
     SDR 900p FPS 59.94 (Avg 59.94)
Frame Time   16.67 ms (Avg 16.68)
GPU Time     11.00 ms (Avg 10.98)
Renderer     13.71 ms (Avg 13.69)
Game         11.38 ms (Avg 11.37)
RenderSync    2.69 ms (Avg  2.70)
Particles     0.01 ms (Avg  0.01)
Mem    +224.44 MiB ( +130.75 MiB)
Build Version: 2017-10-27.100325
Area                  level_08_ne
```

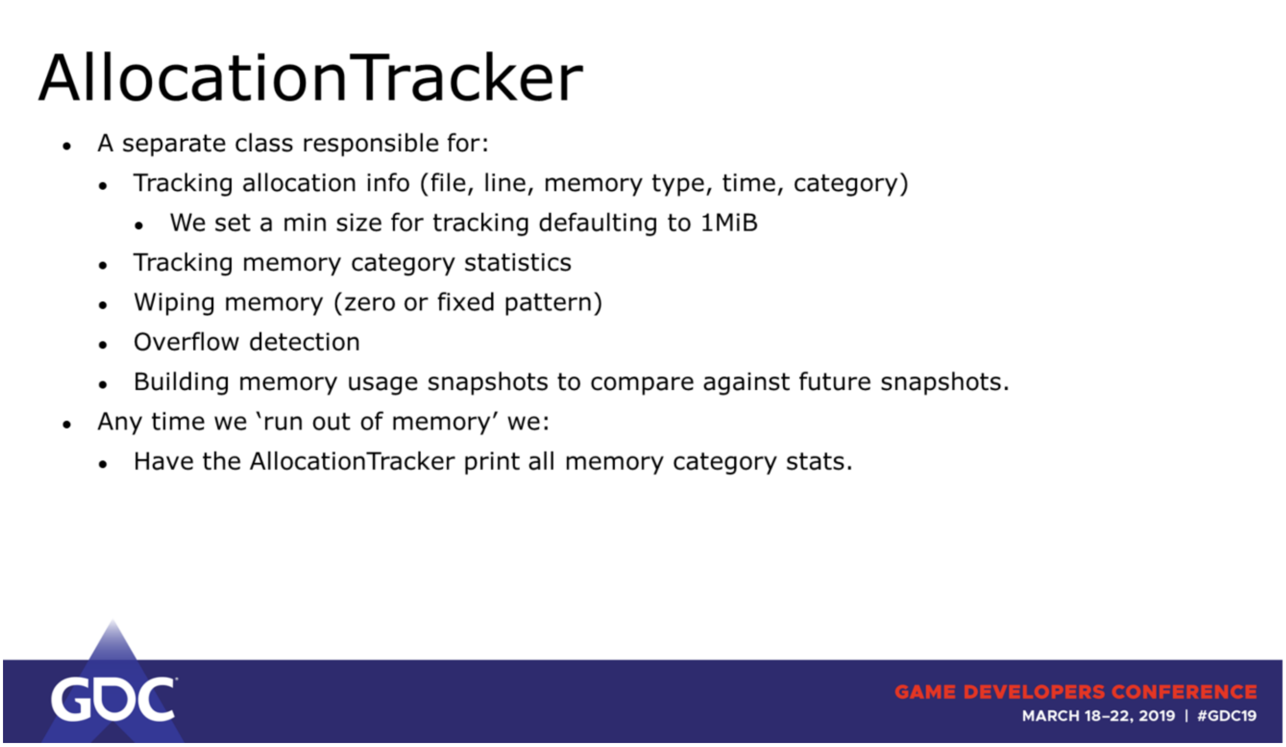
Here is another screenshot that shows memory category tracking.

I want to call your attention to the outlined black boxes that I have marked up.

Most consoles often have the concept of development memory, basically the development kits have more memory than the retail kits.

We make it very easy, at all times to see where memory usage is in regards to a retail kit.

From this screenshot you can see that the closest we came to running out of memory on a retail kit was 130MiBs; or the equivalent of the max memory of 4, PS2 games, there is no way we could run out, right?

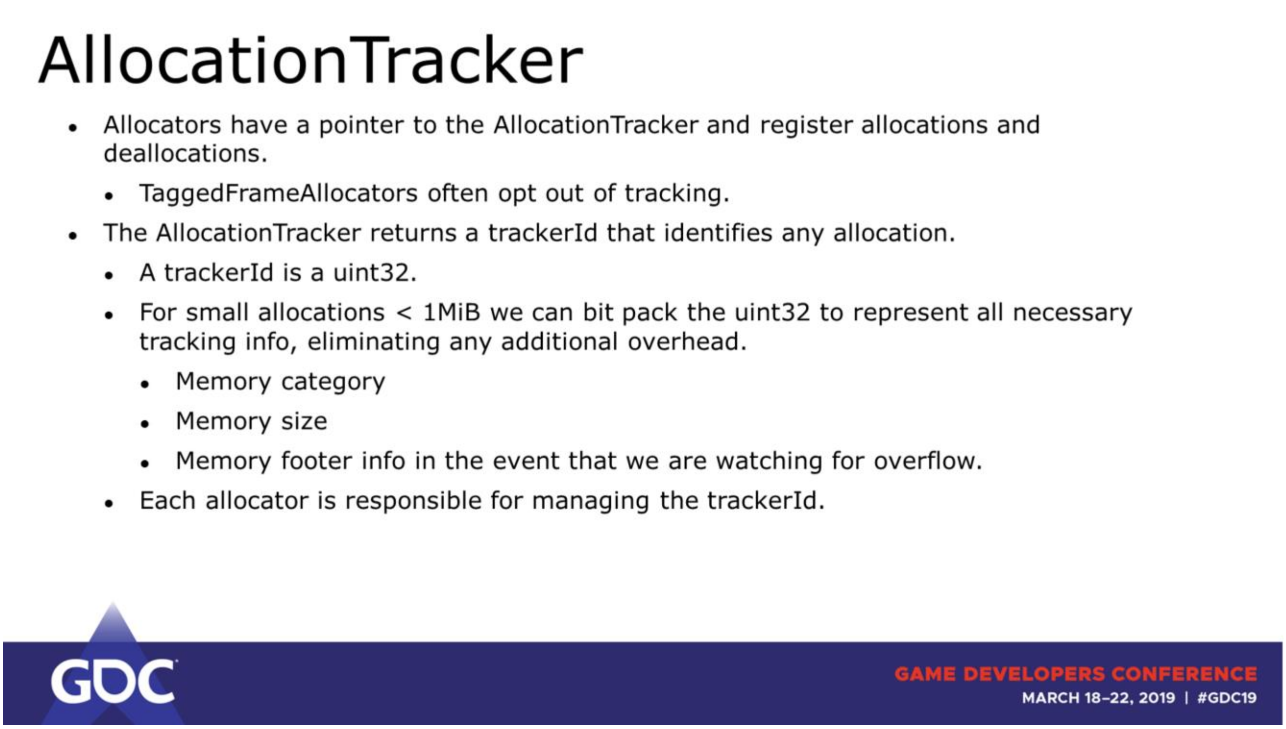


AllocationTracker

- A separate class responsible for:
  - Tracking allocation info (file, line, memory type, time, category)
    - We set a min size for tracking defaulting to 1MiB
  - Tracking memory category statistics
  - Wiping memory (zero or fixed pattern)
  - Overflow detection
  - Building memory usage snapshots to compare against future snapshots.
- Any time we ‘run out of memory’ we:
  - Have the AllocationTracker print all memory category stats.

[NOTE] Originally right before: Game Benefit: Texture Streaming

Bonus Slide: Out of Time

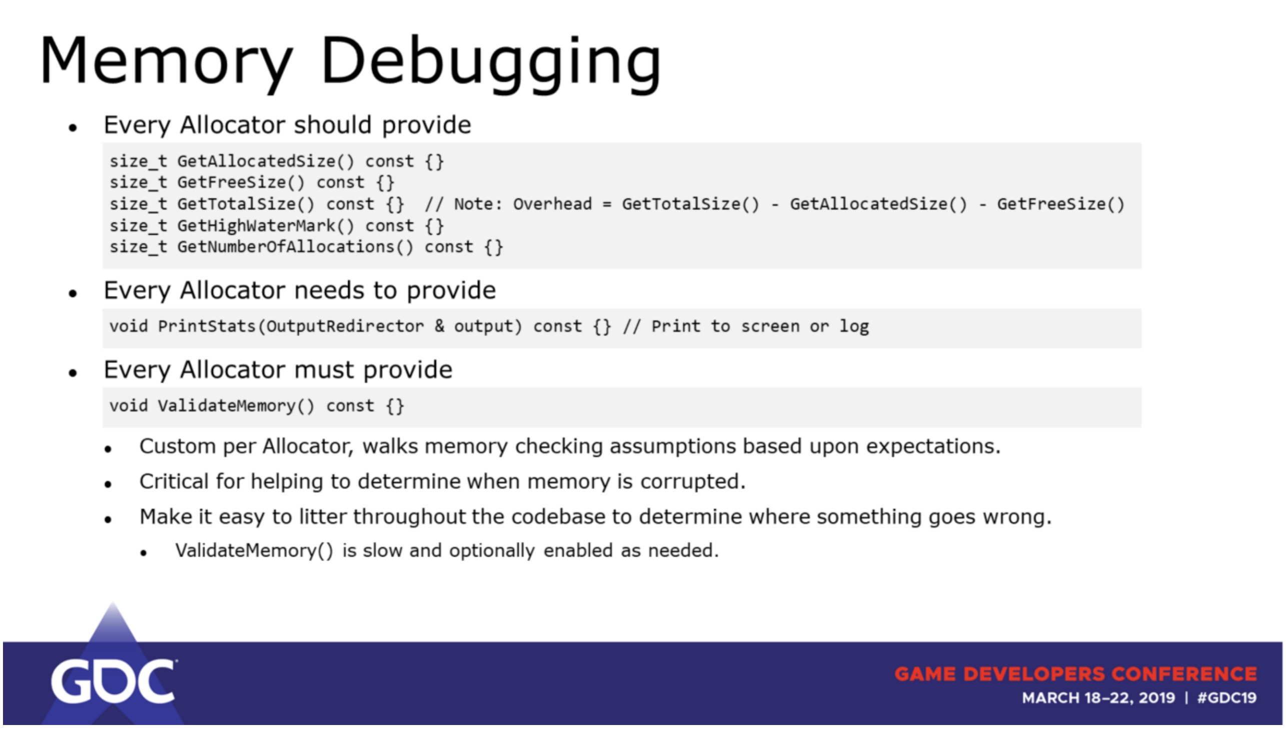


AllocationTracker

- Allocators have a pointer to the AllocationTracker and register allocations and deallocations.
  - TaggedFrameAllocators often opt out of tracking.
- The AllocationTracker returns a trackerId that identifies any allocation.
  - A trackerId is a uint32.
  - For small allocations < 1MiB we can bit pack the uint32 to represent all necessary tracking info, eliminating any additional overhead.
    - Memory category
    - Memory size
    - Memory footer info in the event that we are watching for overflow.
  - Each allocator is responsible for managing the trackerId.

[NOTE] Originally right after: AllocationTracker

Bonus Slide: Out of Time



Memory Debugging

- Every Allocator should provide

```
size_t GetAllocatedSize() const {}
size_t GetFreeSize() const {}
size_t GetTotalSize() const {}  // Note: Overhead = GetTotalSize() - GetAllocatedSize() - GetFreeSize()
size_t GetHighWaterMark() const {}
size_t GetNumberOfAllocations() const {}
```

- Every Allocator needs to provide

```
void PrintStats(OutputRedirector & output) const {} // Print to screen or log
```

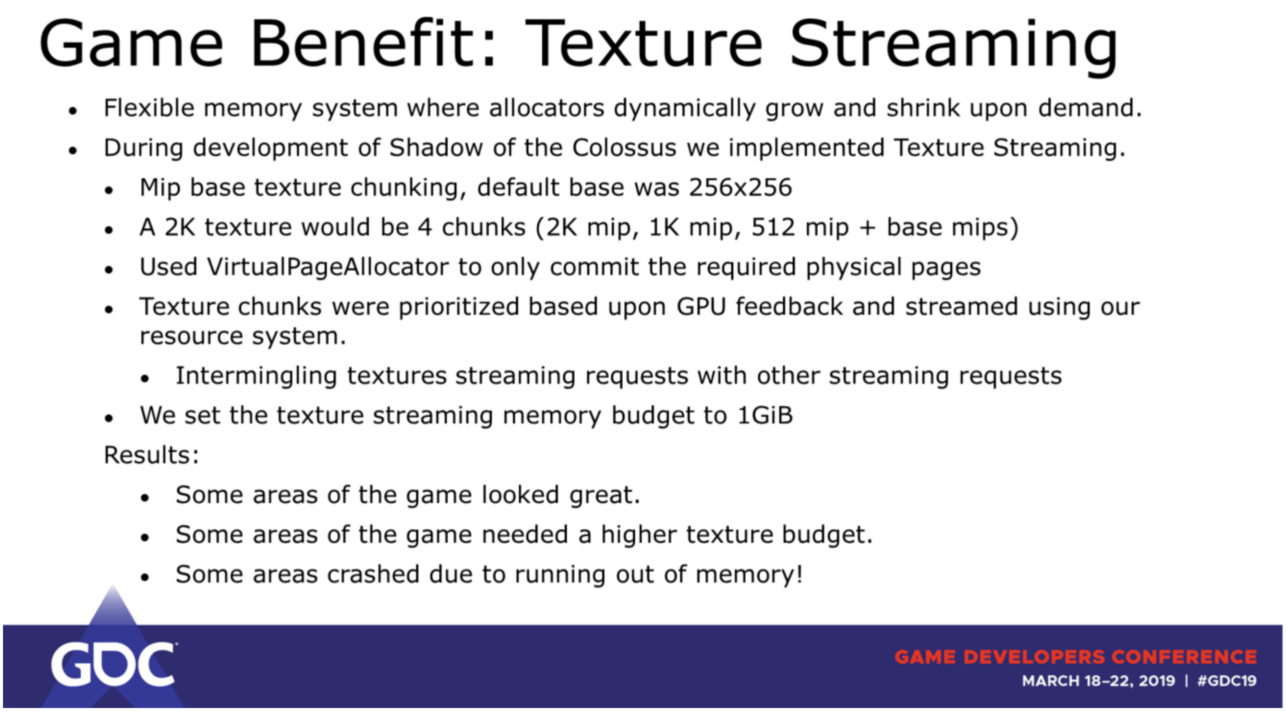
- Every Allocator must provide

```
void ValidateMemory() const {}
```

  - Custom per Allocator, walks memory checking assumptions based upon expectations.
  - Critical for helping to determine when memory is corrupted.
  - Make it easy to litter throughout the codebase to determine where something goes wrong.
    - ValidateMemory() is slow and optionally enabled as needed.

[NOTE] Originally right after: AllocationTracker

Bonus Slide: Out of Time



Game Benefit: Texture Streaming

- Flexible memory system where allocators dynamically grow and shrink upon demand.
- During development of Shadow of the Colossus we implemented Texture Streaming.
  - Mip base texture chunking, default base was 256x256
  - A 2K texture would be 4 chunks (2K mip, 1K mip, 512 mip + base mips)
  - Used VirtualPageAllocator to only commit the required physical pages
  - Texture chunks were prioritized based upon GPU feedback and streamed using our resource system.
    - Intermingling textures streaming requests with other streaming requests
  - We set the texture streaming memory budget to 1GiB

Results:

- Some areas of the game looked great.
- Some areas of the game needed a higher texture budget.
- Some areas crashed due to running out of memory!

So to finish up I want to share one additional feature in Shadow of the Colossus.

It is related to memory and made a significant impact on development.

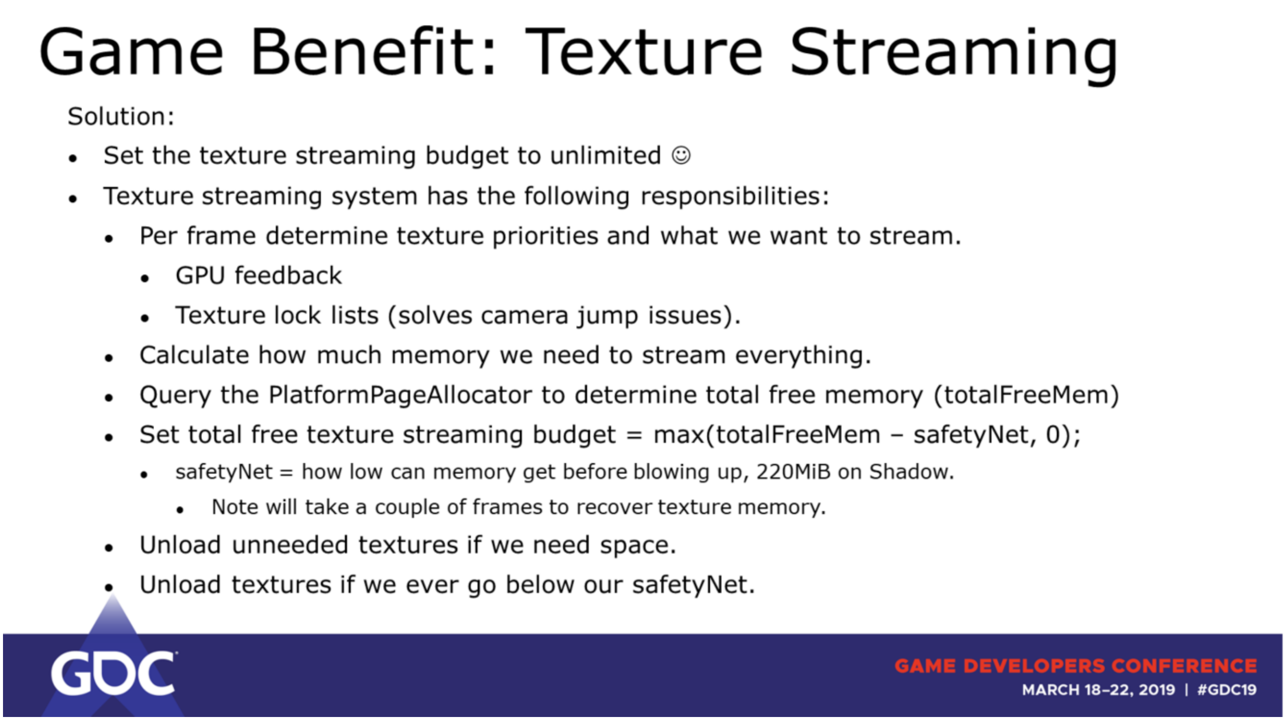
Texture streaming. I’m sure many of you have texture streaming solutions.

We use GPU based feedback to determine what to stream and specify a fixed streaming texture memory budget.

For Shadow we initially set this budged to 1GiB and after some tweaking:

- Most of the game looked great.
- Some areas required a higher texture budget to facilitate its needs.
- And some areas would simply crash due to running OutOfMemory before even hitting the 1GiB budget.

It was about half way through development that the team began to stress about memory and started get flash backs of previous projects.



Game Benefit: Texture Streaming

Solution:

- Set the texture streaming budget to unlimited ☺
- Texture streaming system has the following responsibilities:
  - Per frame determine texture priorities and what we want to stream.
    - GPU feedback
    - Texture lock lists (solves camera jump issues).
  - Calculate how much memory we need to stream everything.
  - Query the PlatformPageAllocator to determine total free memory (totalFreeMem)
  - Set total free texture streaming budget = max(totalFreeMem – safetyNet, 0);
    - safetyNet = how low can memory get before blowing up, 220MiB on Shadow.
      - Note will take a couple of frames to recover texture memory.
  - Unload unneeded textures if we need space.
  - Unload textures if we ever go below our safetyNet.

So how did we solve this problem. We set the texture streaming budget to unlimited. Problem solved and now everyone is happy.

Instead of relying upon fixed memory budgets, the texture manager monitors the total available memory by querying the PlatformPageAllocator and acting appropriately.

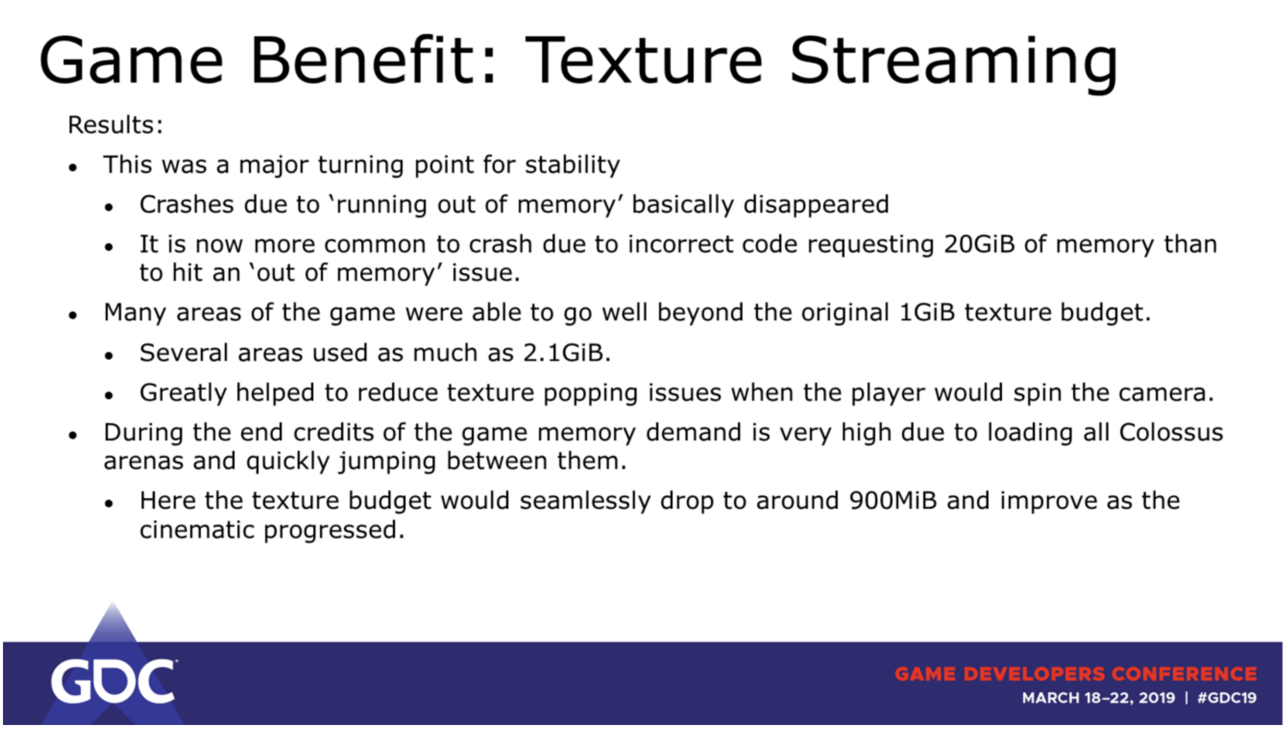
If there is extra memory, stream in more textures.

If no more textures are required, do nothing.

If memory is low, start releasing textures.

I should note that this does require balancing to avoid ping pong effects and to determine proper memory thresholds.

However these thresholds are game specific, not area specific so they only need to be calculated once.



Game Benefit: Texture Streaming

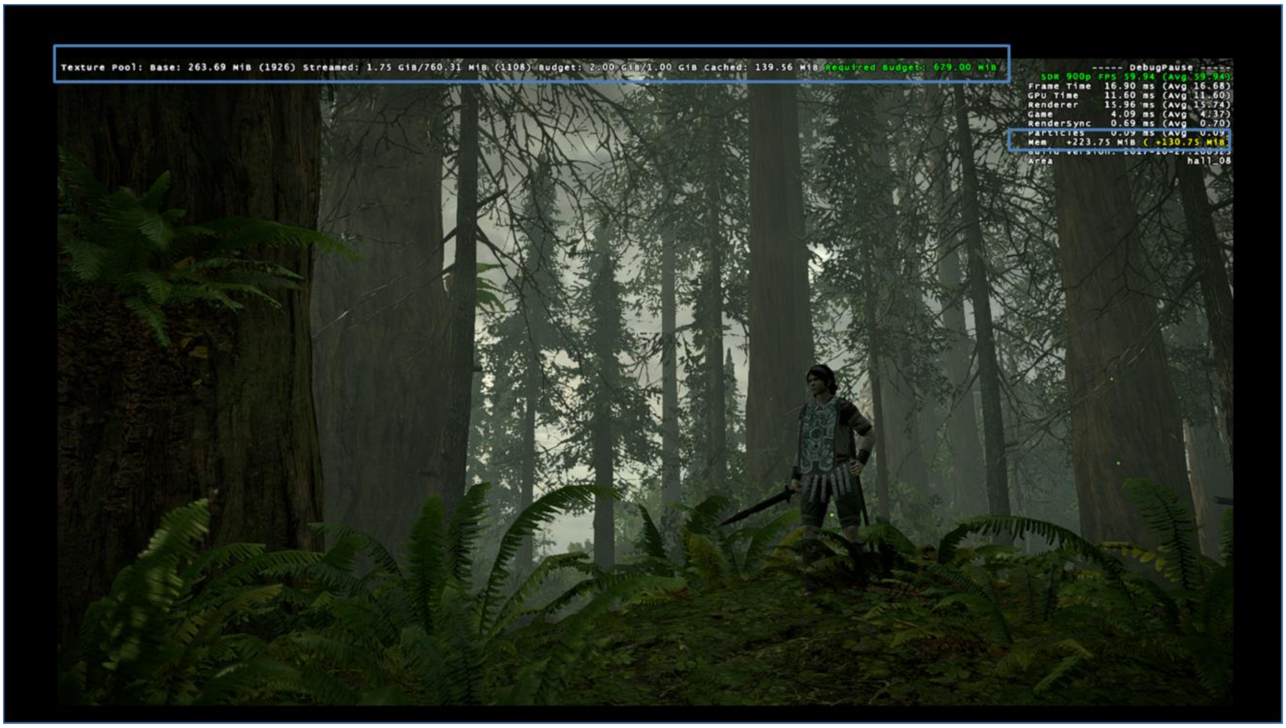
Results:

- This was a major turning point for stability
  - Crashes due to 'running out of memory' basically disappeared
  - It is now more common to crash due to incorrect code requesting 20GiB of memory than to hit an 'out of memory' issue.
- Many areas of the game were able to go well beyond the original 1GiB texture budget.
  - Several areas used as much as 2.1GiB.
  - Greatly helped to reduce texture popping issues when the player would spin the camera.
- During the end credits of the game memory demand is very high due to loading all Colossus arenas and quickly jumping between them.
  - Here the texture budget would seamlessly drop to around 900MiB and improve as the cinematic progressed.

This was really a major turning point for stability as OutOfMemory issues basically disappeared.

Of course we still had to deal with memory usage issues where memory was scarce. However these areas only needed to be pushed to acceptable levels and the texture manager would auto calibrate.

Moving forward we are pushing our texture streaming solution beyond what was achievable in Shadow by prioritizing and loading even higher resolution textures when memory allows.

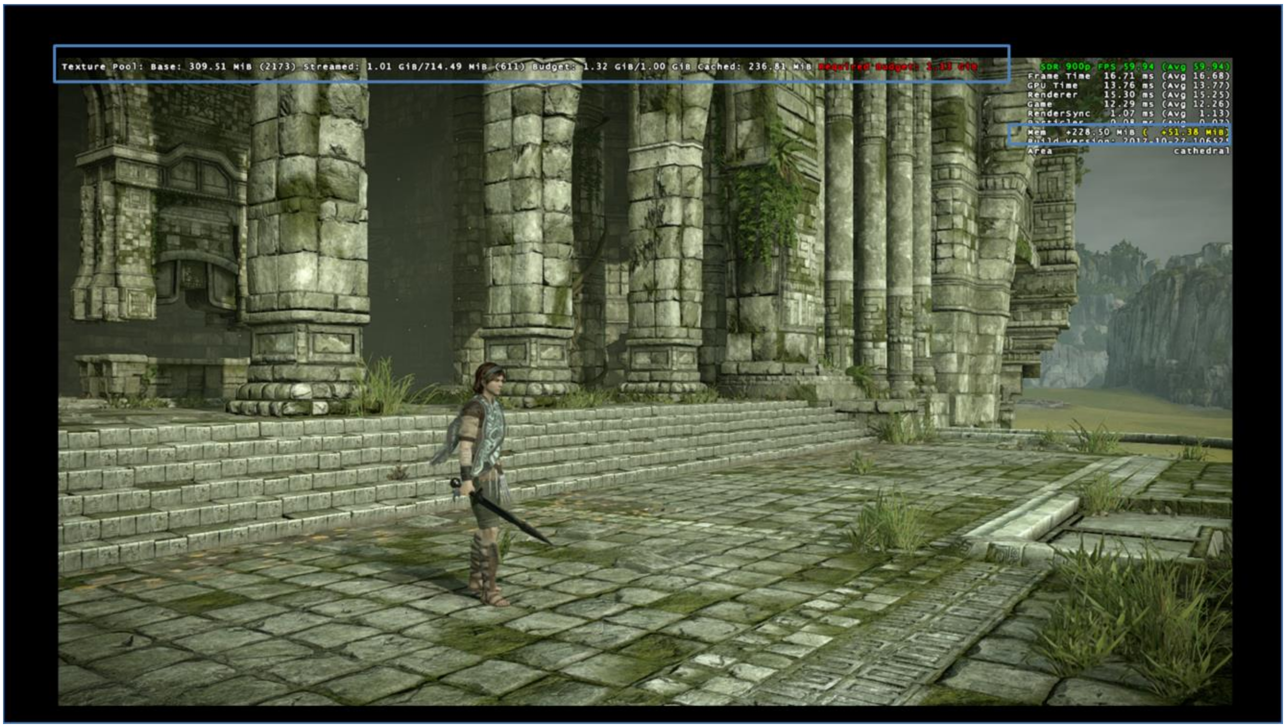


```
Texture Pool: Base: 263.69 MiB (1926) Streamed: 1.75 GiB/760.31 MiB (1108) Budget: 2.00 GiB/1.00 GiB Cached: 139.56 MiB Required Budget: 679.00 MiB
```

```
----- DebugPause -----
  SDR 900p FPS 59.94 (Avg 59.94)
Frame Time  16.90 ms (Avg 16.68)
GPU Time    11.60 ms (Avg 11.60)
Renderer    15.96 ms (Avg 15.74)
Game         4.09 ms (Avg  4.37)
RenderSync   0.69 ms (Avg  0.70)
Particles    0.09 ms (Avg  0.09
Mem   +223.75 MiB ( +130.75 MiB
Build Version: 2017-10-27.100[illegible]
Area                   hall_08
```

So here is a screenshot showing the texture streaming stats at the top of the screen.

Here you can see that we have over 2GiBs of textures loaded however the scene actually only requires about 680 MiB of texture data as the texture manager maintains a buffer of 220MiBs free.



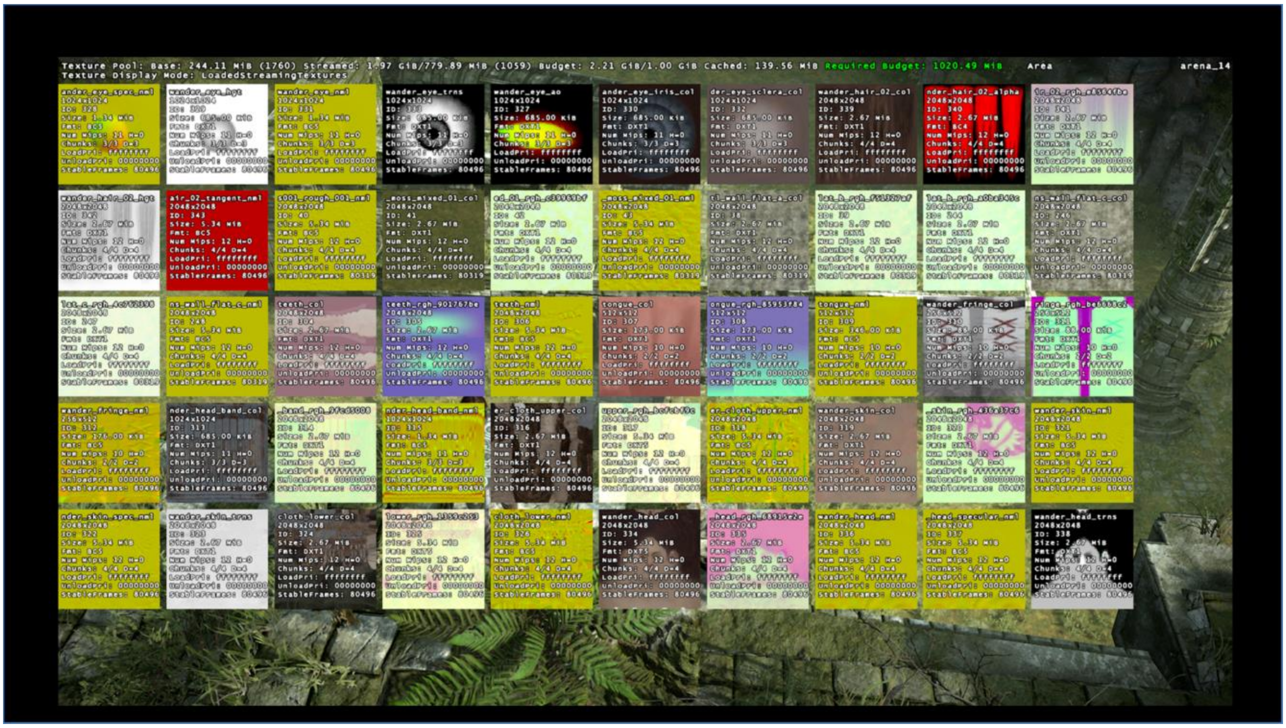
```
Texture Pool: Base: 309.51 MiB (2173) Streamed: 1.01 GiB/714.49 MiB (611) Budget: 1.32 GiB/1.00 GiB Cached: 236.81 MiB Required Budget: 1.13 GiB
```

```
SDR 900p FPS 59.94 (Avg 59.94)
Frame Time   16.71 ms (Avg 16.68)
GPU Time     13.76 ms (Avg 13.77)
Renderer     15.30 ms (Avg 15.25)
Game         12.29 ms (Avg 12.26)
RenderSync    1.07 ms (Avg  1.13)
Particles     0.08 ms (Avg  0.07)
Mem    +228.50 MiB (  +51.38 MiB)
Build version: 2017-10-27 10652
Area                    cathedral
```

Here is a more demanding area. There is 1.3 GiB of textures loaded, however only 1.13 GiB are required.

During development the art department would play the game watching these statistics to verify budgets were satisfied.

By the end the only questionable location in the game was the final cutscene where the secret garden is exposed.



```
Texture Pool: Base: 244.11 MiB (1760) Streamed: 1.97 GiB/779.89 MiB (1059) Budget: 2.21 GiB/1.00 GiB Cached: 139.56 MiB Required Budget: 1020.49 MiB     Area     arena_14
Texture Display Mode: LoadedStreamingTextures

ander_eye_spec_nml
1024x1024
ID: 328
Size: 1.34 MiB
Fmt: BC5
Num Mips: 11 M=0
Chunks: 3/3 D=3
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

wander_eye_hgt
1024x1024
ID: 329
Size: 685.00 KiB
Fmt: DXT1
Num Mips: 11 M=0
Chunks: 3/3 D=3
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

wander_eye_nml
1024x1024
ID: 331
Size: 1.34 MiB
Fmt: BC5
Num Mips: 11 M=0
Chunks: 3/3 D=3
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

wander_eye_trns
1024x1024
ID: 333
Size: 685.00 KiB
Fmt: DXT1
Num Mips: 11 M=0
Chunks: 3/3 D=3
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

wander_eye_ao
1024x1024
ID: 327
Size: 685.00 KiB
Fmt: DXT1
Num Mips: 11 M=0
Chunks: 3/3 D=3
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

ander_eye_iris_col
1024x1024
ID: 330
Size: 685.00 KiB
Fmt: DXT1
Num Mips: 11 M=0
Chunks: 3/3 D=3
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

der_eye_sclera_col
1024x1024
ID: 332
Size: 685.00 KiB
Fmt: DXT1
Num Mips: 11 M=0
Chunks: 3/3 D=3
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

wander_hair_02_col
2048x2048
ID: 339
Size: 2.67 MiB
Fmt: DXT1
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

nder_hair_02_alpha
2048x2048
ID: 340
Size: 2.67 MiB
Fmt: BC4
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

ir_02_rgh_e8564fbe
2048x2048
ID: 341
Size: 2.67 MiB
Fmt: DXT1
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

wander_hair_02_hgt
2048x2048
ID: 342
Size: 2.67 MiB
Fmt: DXT1
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

air_02_tangent_nml
2048x2048
ID: 343
Size: 5.34 MiB
Fmt: BC5
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

s001_rough_001_nml
2048x2048
ID: 40
Size: 5.34 MiB
Fmt: BC5
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80319

_moss_mixed_01_col
2048x2048
ID: 41
Size: 2.67 MiB
Fmt: DXT1
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80319

ed_01_rgh_c39969bf
2048x2048
ID: 42
Size: 2.67 MiB
Fmt: DXT1
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80319

_moss_mixed_01_nml
2048x2048
ID: 43
Size: 5.34 MiB
Fmt: BC5
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80319

tl_wall_flat_a_col
2048x2048
ID: 38
Size: 2.67 MiB
Fmt: DXT1
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80319

lat_b_rgh_f51327af
2048x2048
ID: 39
Size: 2.67 MiB
Fmt: DXT1
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80319

lat_b_rgh_a0ba345c
2048x2048
ID: 244
Size: 2.67 MiB
Fmt: DXT1
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80319

ns_wall_flat_c_col
2048x2048
ID: 246
Size: 2.67 MiB
Fmt: DXT1
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80319

lat_c_rgh_4c762398
2048x2048
ID: 247
Size: 2.67 MiB
Fmt: DXT1
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80319

ns_wall_flat_c_nml
2048x2048
ID: 248
Size: 5.34 MiB
Fmt: BC5
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80319

teeth_col
2048x2048
ID: 304
Size: 2.67 MiB
Fmt: DXT1
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

teeth_rgh_901767be
2048x2048
ID: 305
Size: 2.67 MiB
Fmt: DXT1
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

teeth_nml
2048x2048
ID: 306
Size: 5.34 MiB
Fmt: BC5
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

tongue_col
512x512
ID: 307
Size: 173.00 KiB
Fmt: DXT1
Num Mips: 10 M=0
Chunks: 2/2 D=2
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

ongue_rgh_85953f84
512x512
ID: 308
Size: 173.00 KiB
Fmt: DXT1
Num Mips: 10 M=0
Chunks: 2/2 D=2
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

tongue_nml
512x512
ID: 309
Size: 346.00 KiB
Fmt: BC5
Num Mips: 10 M=0
Chunks: 2/2 D=2
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

wander_fringe_col
256x512
ID: 310
Size: 88.00 KiB
Fmt: DXT1
Num Mips: 10 M=0
Chunks: 2/2 D=2
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

ringe_rgh_beb668c2
256x512
ID: 311
Size: 88.00 KiB
Fmt: DXT1
Num Mips: 10 M=0
Chunks: 2/2 D=2
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

wander_fringe_nml
256x512
ID: 312
Size: 176.00 KiB
Fmt: BC5
Num Mips: 10 M=0
Chunks: 2/2 D=2
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

nder_head_band_col
1024x1024
ID: 313
Size: 685.00 KiB
Fmt: DXT1
Num Mips: 11 M=0
Chunks: 3/3 D=3
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

_band_rgh_9fcd5008
2048x2048
ID: 314
Size: 2.67 MiB
Fmt: DXT1
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

nder_head_band_nml
1024x1024
ID: 315
Size: 1.34 MiB
Fmt: BC5
Num Mips: 11 M=0
Chunks: 3/3 D=3
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

er_cloth_upper_col
2048x2048
ID: 316
Size: 2.67 MiB
Fmt: DXT1
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

upper_rgh_bcfcbf9c
2048x2048
ID: 317
Size: 5.34 MiB
Fmt: DXT5
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

er_cloth_upper_nml
2048x2048
ID: 318
Size: 5.34 MiB
Fmt: BC5
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

wander_skin_col
2048x2048
ID: 319
Size: 2.67 MiB
Fmt: DXT1
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

_skin_rgh_436a37c6
2048x2048
ID: 320
Size: 2.67 MiB
Fmt: DXT1
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

wander_skin_nml
2048x2048
ID: 321
Size: 5.34 MiB
Fmt: BC5
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

nder_skin_spec_nml
2048x2048
ID: 322
Size: 5.34 MiB
Fmt: BC5
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

wander_skin_trns
2048x2048
ID: 323
Size: 2.67 MiB
Fmt: DXT1
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

cloth_lower_col
2048x2048
ID: 324
Size: 2.67 MiB
Fmt: DXT1
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

lower_rgh_1359c253
2048x2048
ID: 325
Size: 5.34 MiB
Fmt: DXT5
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

cloth_lower_nml
2048x2048
ID: 326
Size: 5.34 MiB
Fmt: BC5
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

wander_head_col
2048x2048
ID: 334
Size: 5.34 MiB
Fmt: DXT5
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

_head_rgh_68915e2e
2048x2048
ID: 335
Size: 2.67 MiB
Fmt: DXT1
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

wander_head_nml
2048x2048
ID: 336
Size: 5.34 MiB
Fmt: BC5
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

_head_specular_nml
2048x2048
ID: 337
Size: 5.34 MiB
Fmt: BC5
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496

wander_head_trns
2048x2048
ID: 338
Size: 2.67 MiB
Fmt: DXT1
Num Mips: 12 M=0
Chunks: 4/4 D=4
LoadPri: ffffffff
UnloadPri: 00000000
StableFrames: 80496
```

And of course, the minute you tell any one that the texture budget is exceeded you need to be able to show them why and what is loaded.

Only then can properly debate "how big must the eye texture be".

Bluepoint games

Bluepoint would like to thank everyone for the amazing reception Shadow of the Colossus has received and hope you enjoy it.

Also we are hiring:

- Artists
- Programmers
- Animators
- VFX Artists

Basically if you are a bad-ass at your craft, we want to work with you.

Checkout: ***www.bluepointgames.com***

jobs@bluepointgames.com

So once again I would like to thank you for your participation. I’m flattered that I had the opportunity to ramble on.

And yes Bluepoint is hiring. Basically if you are a bad-ass and you know how to get shit done, we want to work with you.

Use the email address on the slide and mention my name, I’ll let everyone know that we are best friends so that I can get the recruiting bonus and then perhaps it will be time for me to buy you a beer.

Also please remember to fill out the speaker evaluation forms so I know whether or not I should ever do this again,

Thank you!

Questions?

Peter Dalton
pdalton@bluepointgames.com

www.bluepointgames.com
jobs@bluepointgames.com