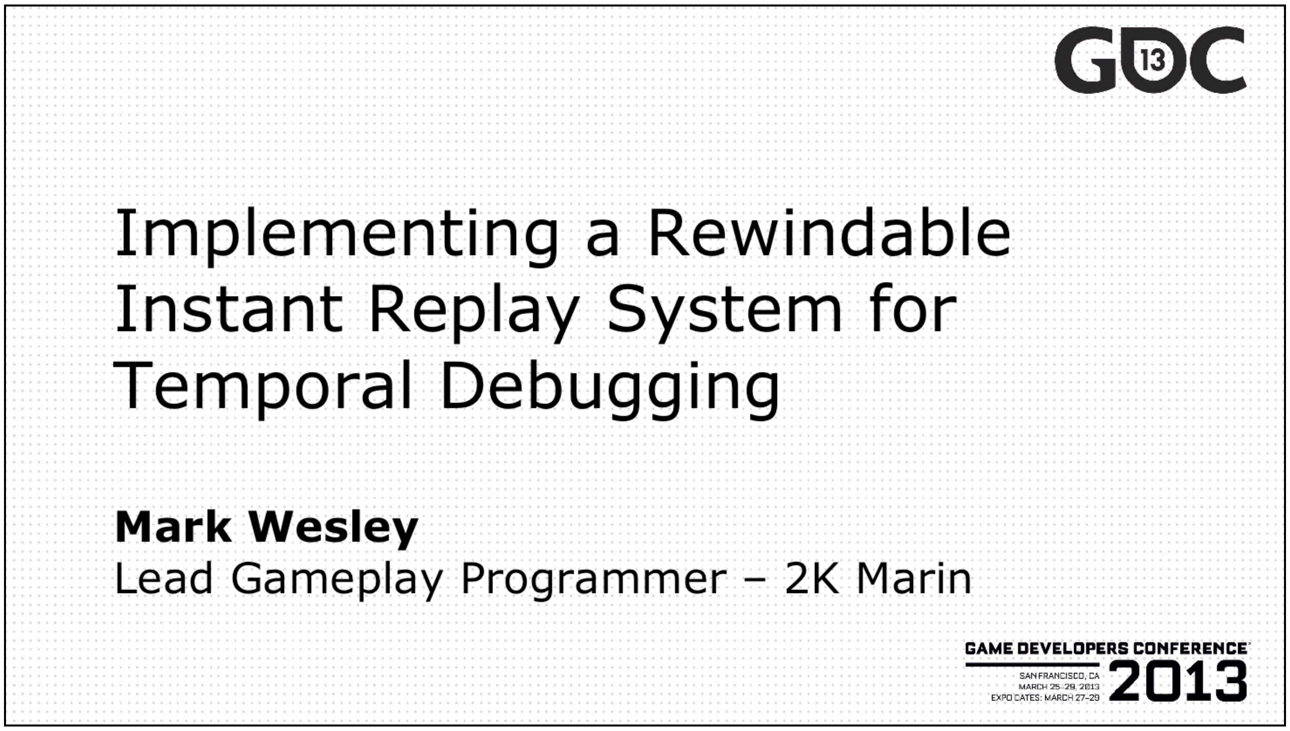
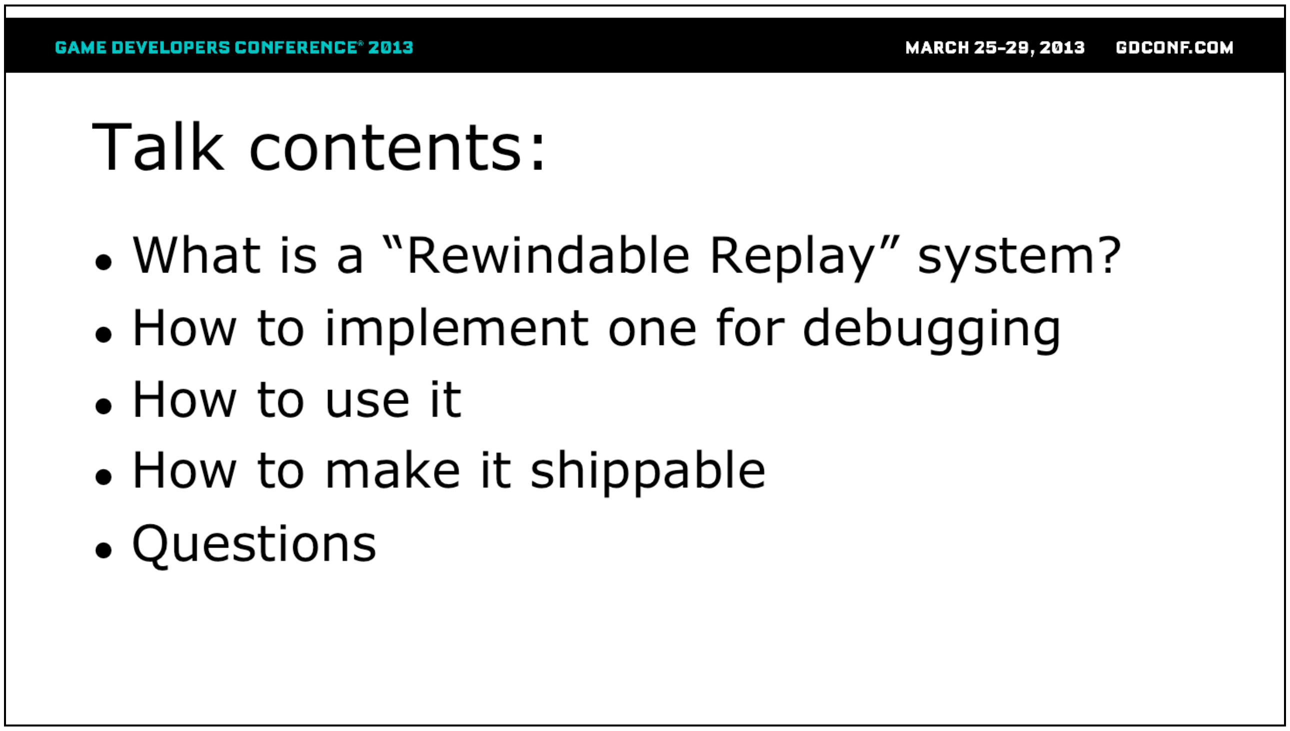


Implementing a Rewindable Instant Replay System for Temporal Debugging

Mark Wesley
Lead Gameplay Programmer – 2K Marin

GAME DEVELOPERS CONFERENCE 2013
SAN FRANCISCO, CA
MARCH 25-29, 2013
EXPO DATES: MARCH 27-29

- Hi everyone, I'm Mark Wesley and I'm the Lead Gameplay Programmer at 2K Marin, located just over the Golden Gate Bridge in beautiful Marin County.
- My talk today is on "Implementing a Rewindable Instant Replay System for Temporal Debugging"

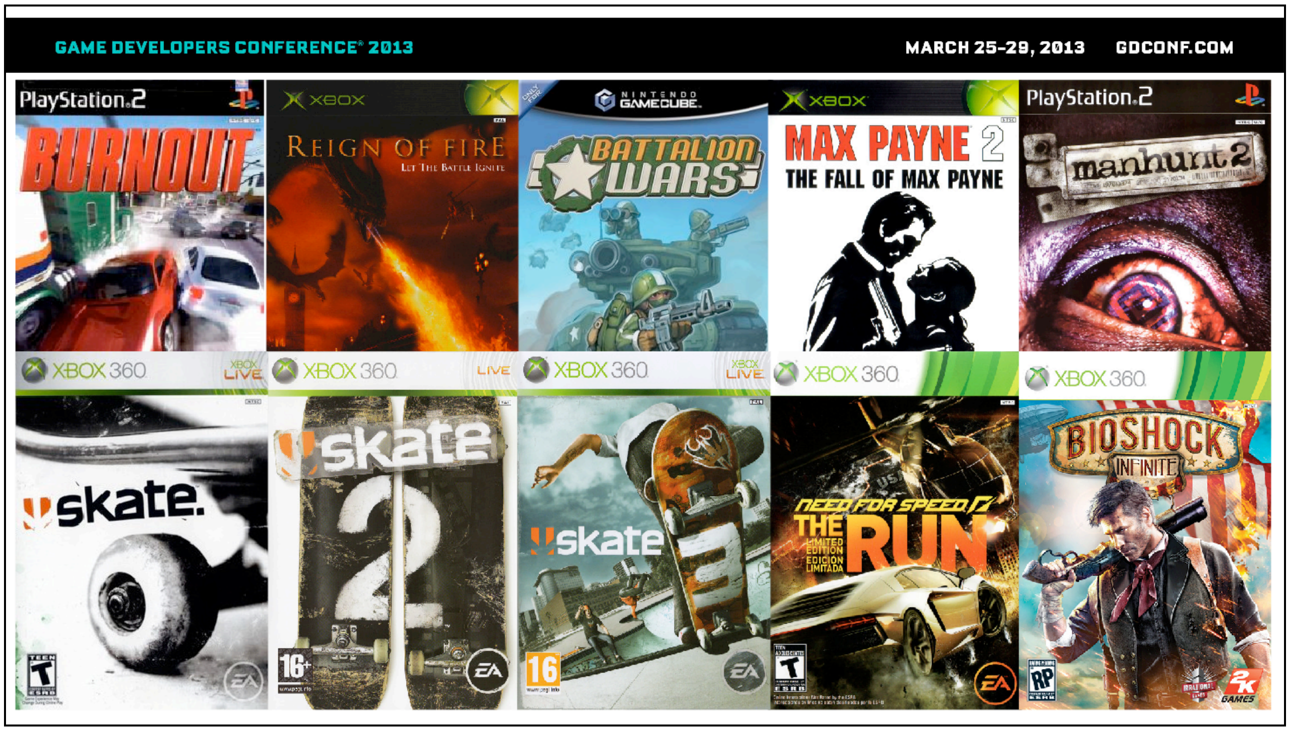


Talk contents:

- What is a “Rewindable Replay” system?
- How to implement one for debugging
- How to use it
- How to make it shippable
- Questions

So today I’m going to:

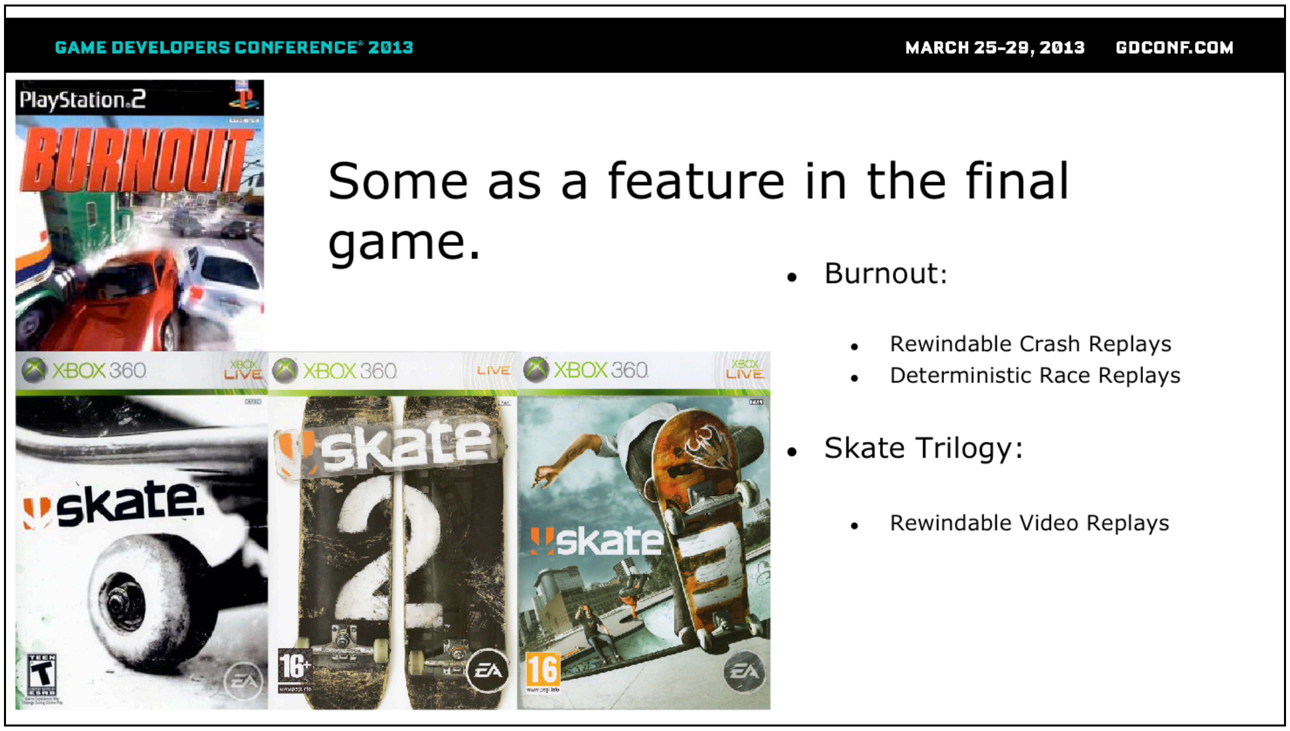
- Tell you what a Rewindable Replay system is.
- How to implement one for debugging and development
- How to then use it to debug your own games
- How you could then make it shippable if you chose to
- And there will be a little bit of time at the end for some questions



- First a quick intro to me…
- I’ve worked on a bunch of games…
- And many of those have included replay of some form…

Several of those included Replay of some form

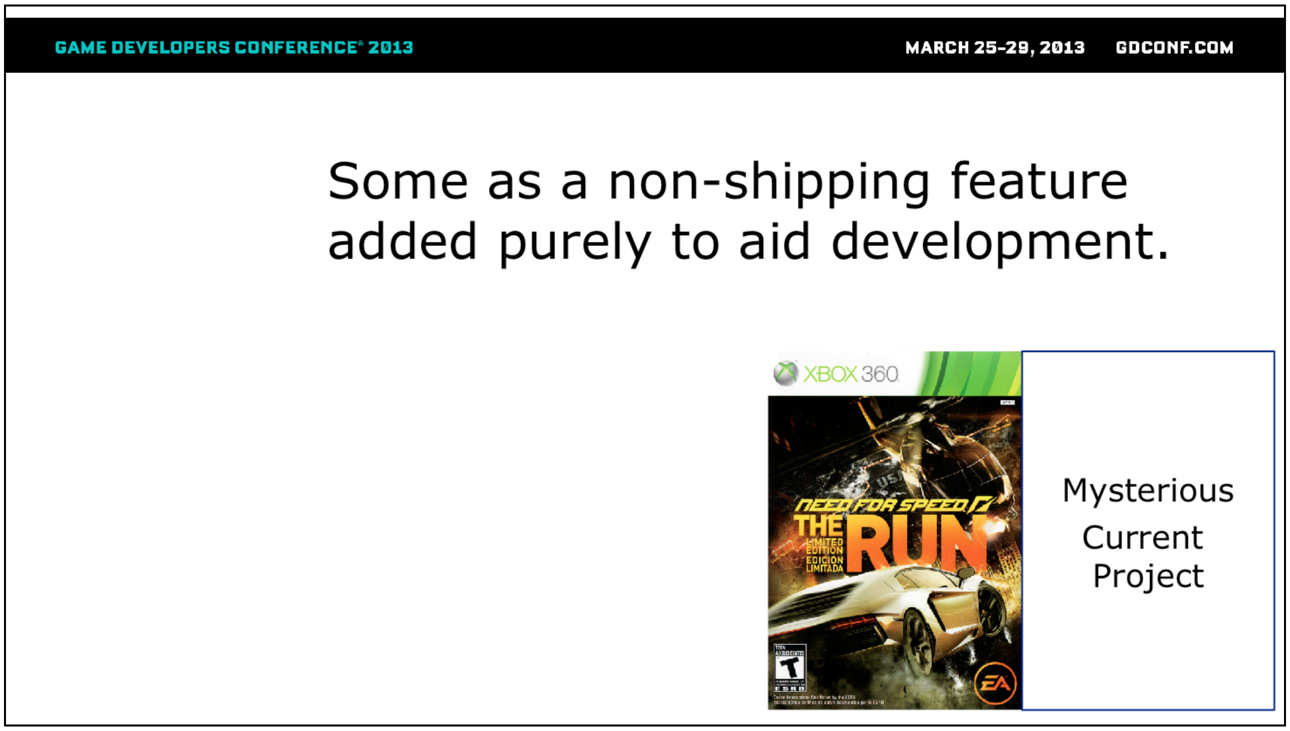
Mysterious Current Project


GAME DEVELOPERS CONFERENCE® 2013 MARCH 25-29, 2013 GDCONF.COM

Some as a feature in the final game.

- Burnout:
  - Rewindable Crash Replays
  - Deterministic Race Replays
- Skate Trilogy:
  - Rewindable Video Replays

- Some as a feature in the final game.
- Burnout for example had a rewindable Crash Replay System which I wrote, and also deterministic forward-only race replays which I helped to maintain.
- And the skate series had an awesome video suite which I didn't write, but I did help to extend it for debugging and we used it extensively when developing and debugging the AI, Physics and other Gameplay elements.

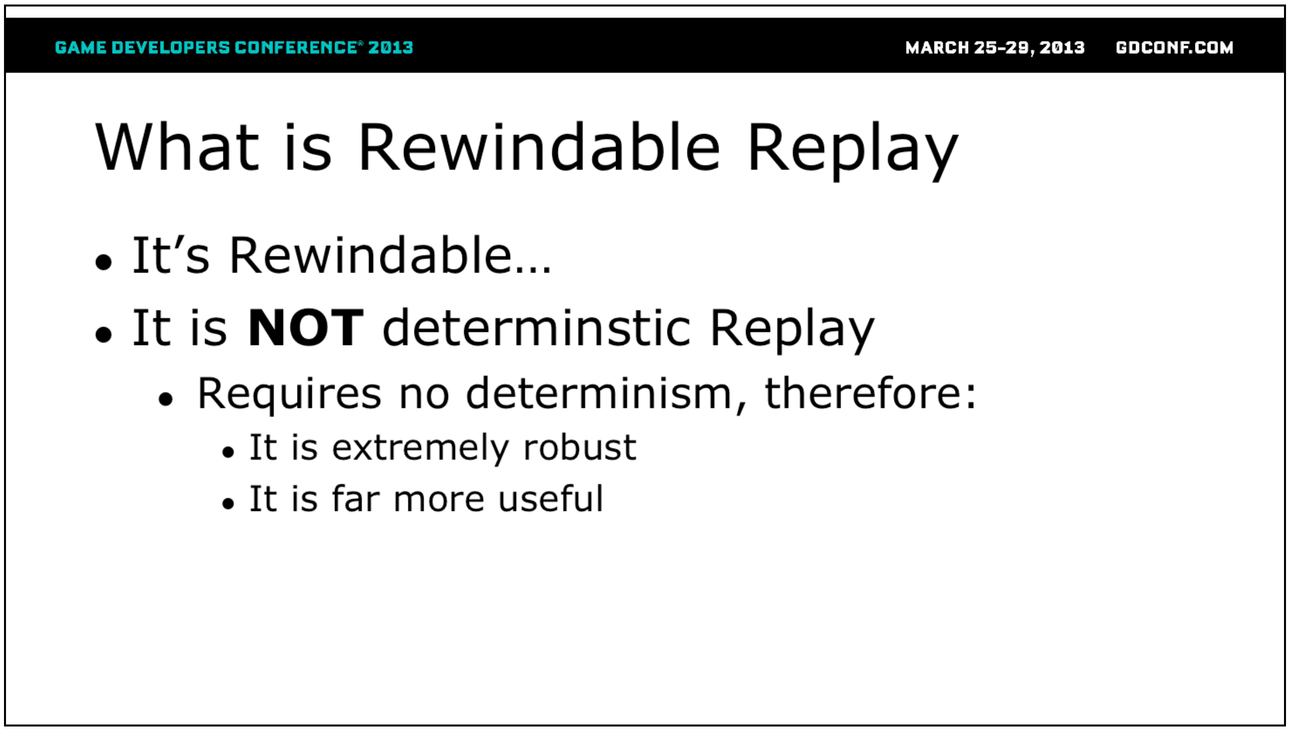


GAME DEVELOPERS CONFERENCE® 2013 MARCH 25-29, 2013 GDCONF.COM

Some as a non-shipping feature added purely to aid development.

Mysterious Current Project

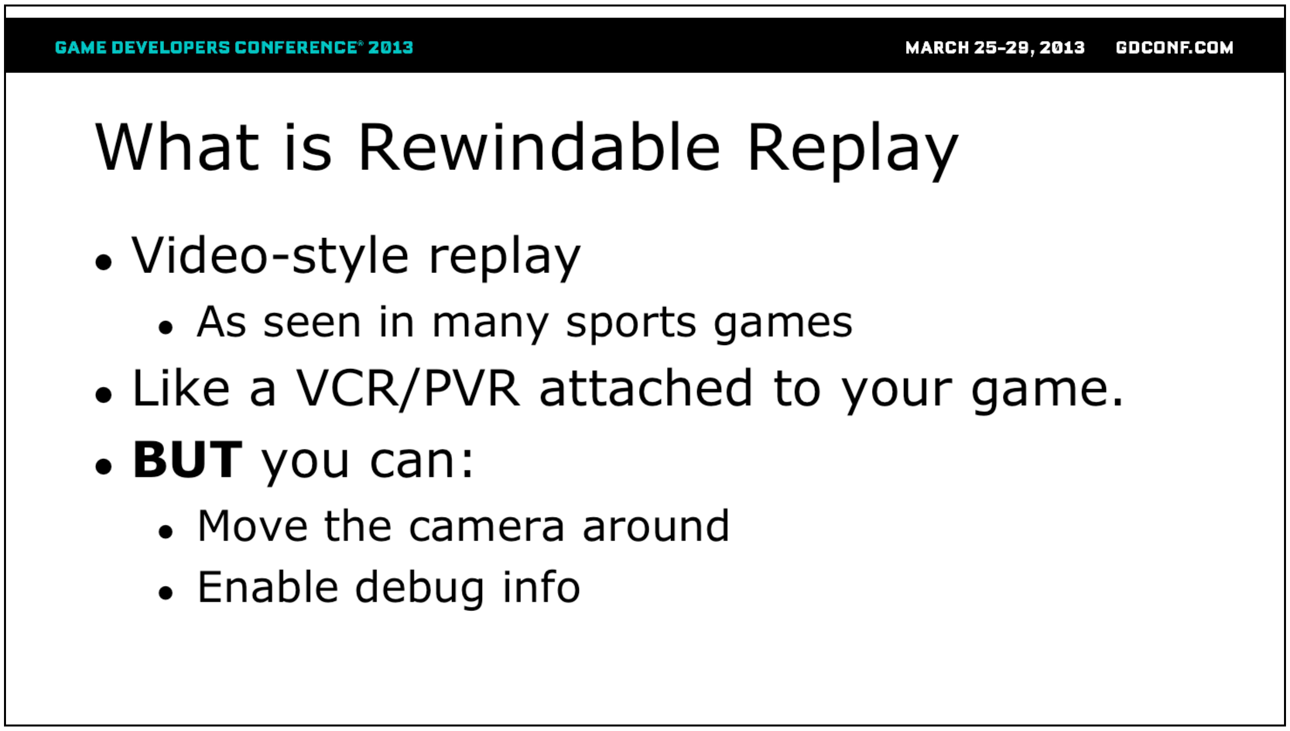
And for pretty much anything I've worked on since, the first thing I do is add a development-only replay system for debugging as I find it makes myself and the rest of the team work so much more efficiently.


GAME DEVELOPERS CONFERENCE® 2013 MARCH 25-29, 2013 GDCONF.COM

What is Rewindable Replay

- It’s Rewindable…
- It is **NOT** determinstic Replay
  - Requires no determinism, therefore:
    - It is extremely robust
    - It is far more useful

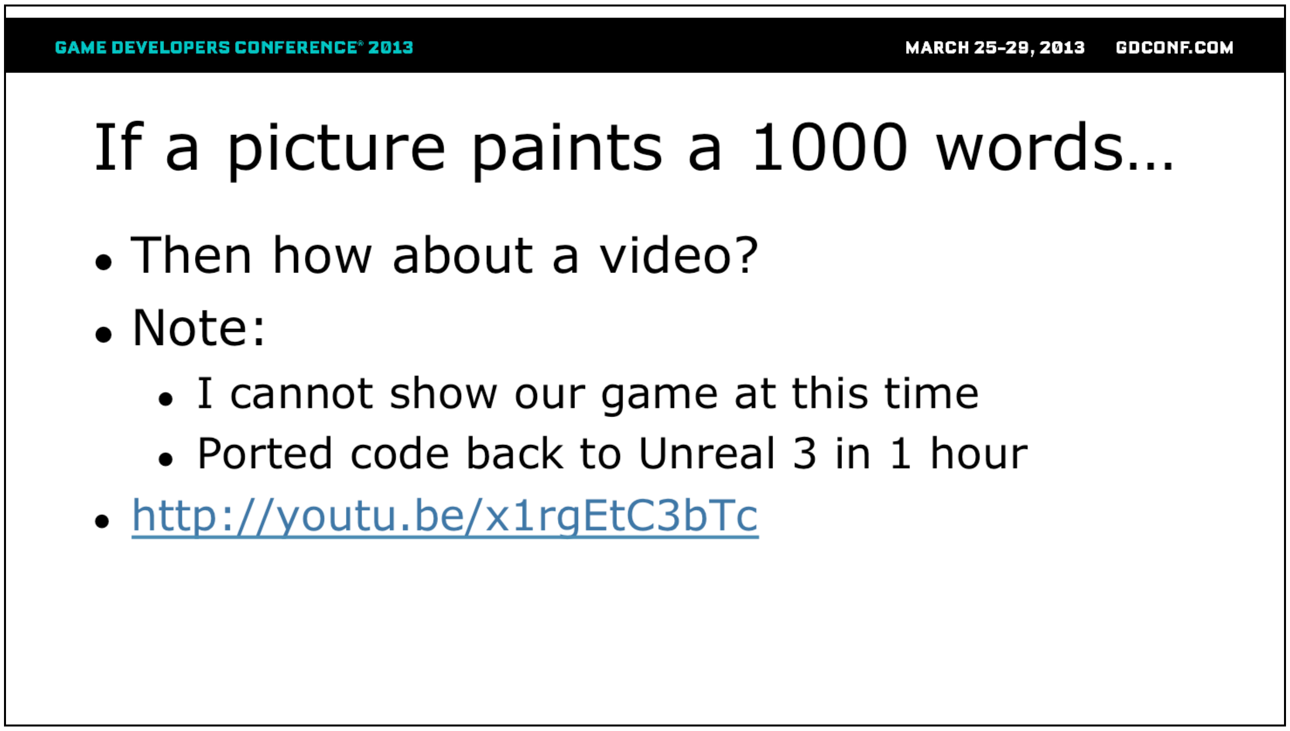
- So… what is a rewindable replay system?
- Well, it’s rewindable…
- And it is NOT determinstic replay – this is an important distinction.
- This requires absolutely no determinism in your game whatsoever
  - This makes it incredibly robust
  - And in my opinion far far more useful.



What is Rewindable Replay

- Video-style replay
  - As seen in many sports games
- Like a VCR/PVR attached to your game.
- **BUT** you can:
  - Move the camera around
  - Enable debug info

- So in essence it’s video-style replay
  - As you might have seen in some sports games.
- It’s kind-of like having a VCR or PVR attached to your game.
- But far better as you can
  - move the camera around
    - view things from any position or angle
  - turn on things like debug info on the fly
    - change rendering modes
    - switch to wireframe
    - and so on…

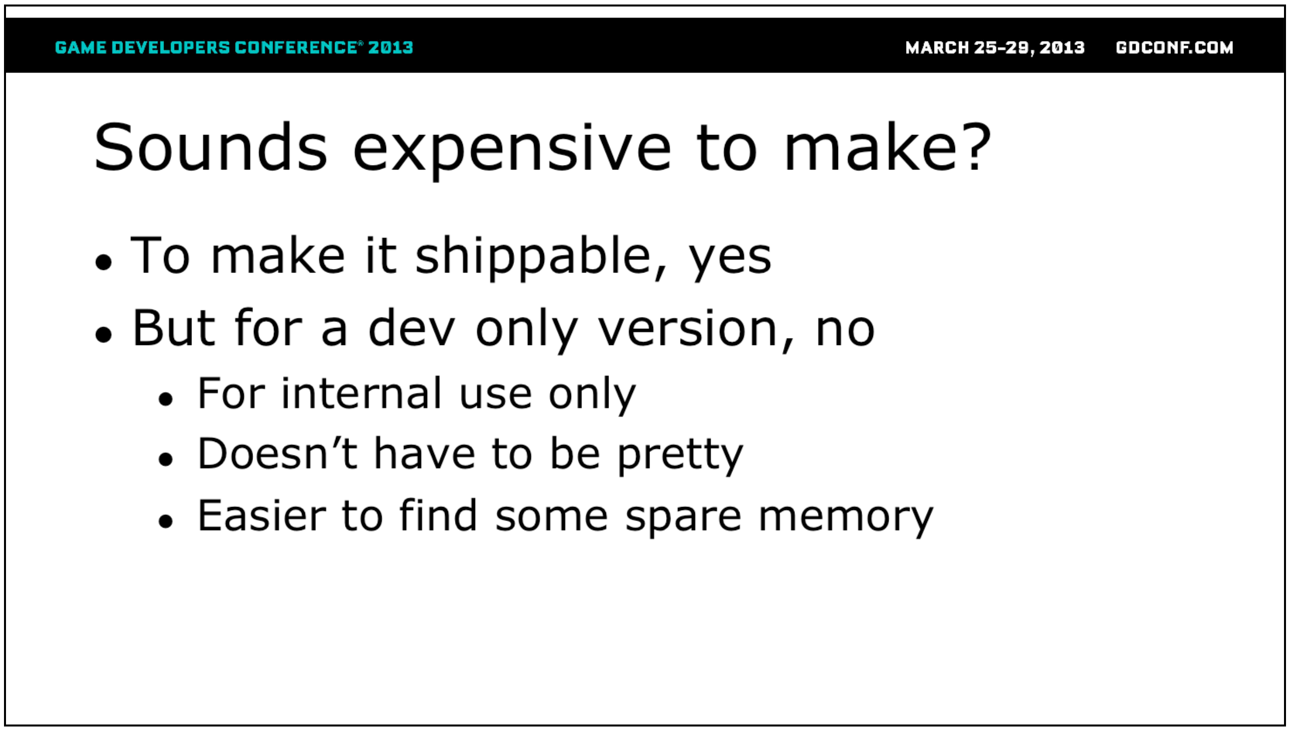


GAME DEVELOPERS CONFERENCE® 2013 MARCH 25-29, 2013 GDCONF.COM

If a picture paints a 1000 words…

- Then how about a video?
- Note:
  - I cannot show our game at this time
  - Ported code back to Unreal 3 in 1 hour
- http://youtu.be/x1rgEtC3bTc

- So it's probably easier to explain if I just show you a video.
- Unfortunately I'm not allowed to show our game at this time…
- But I was able to port the code back into the base version of Unreal 3 in just 1 hour, and I think this is a testament to not only how re-usable such a system is but also how applicable it is to most game genres.
- Note that you can jump in and out of replay at any point and continue execution of the game.
- http://youtu.be/x1rgEtC3bTc

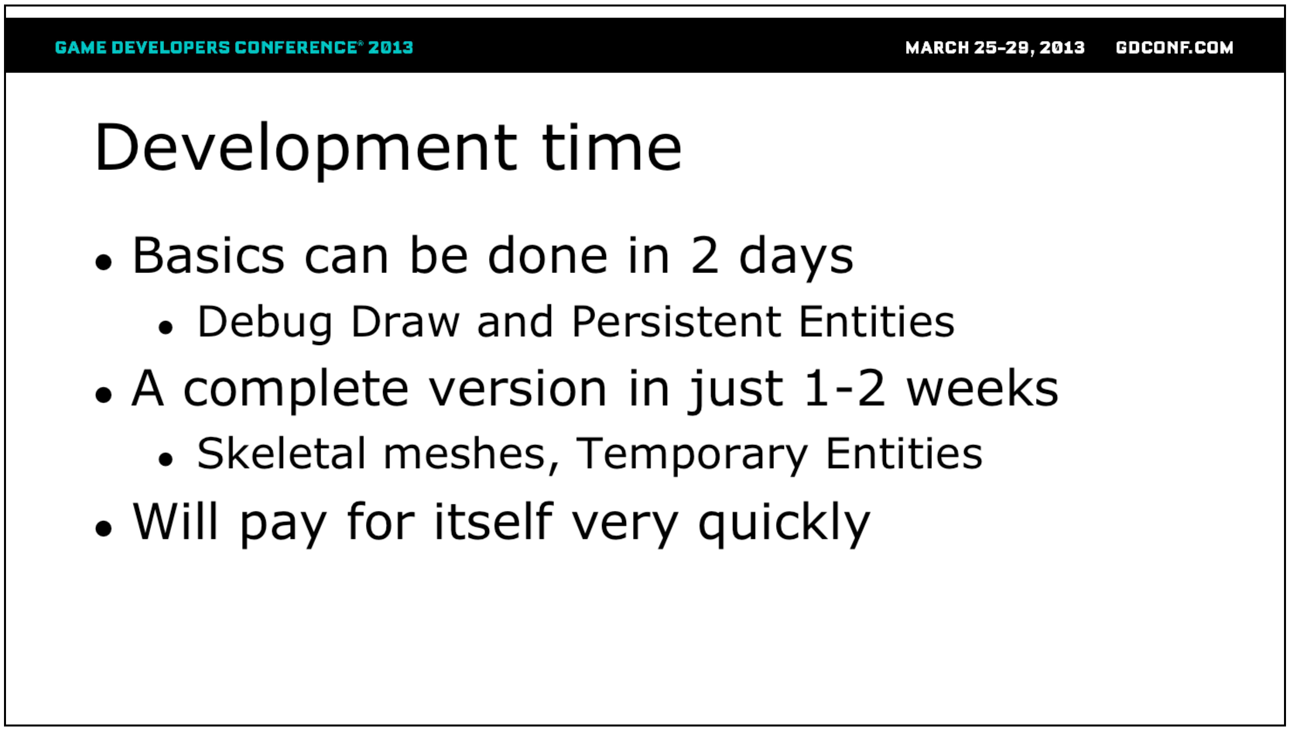

GAME DEVELOPERS CONFERENCE® 2013 MARCH 25-29, 2013 GDCONF.COM

Sounds expensive to make?

- To make it shippable, yes
- But for a dev only version, no
  - For internal use only
  - Doesn't have to be pretty
  - Easier to find some spare memory

So you might be thinking… "well that's great, but isn't that a lot of work to implement"

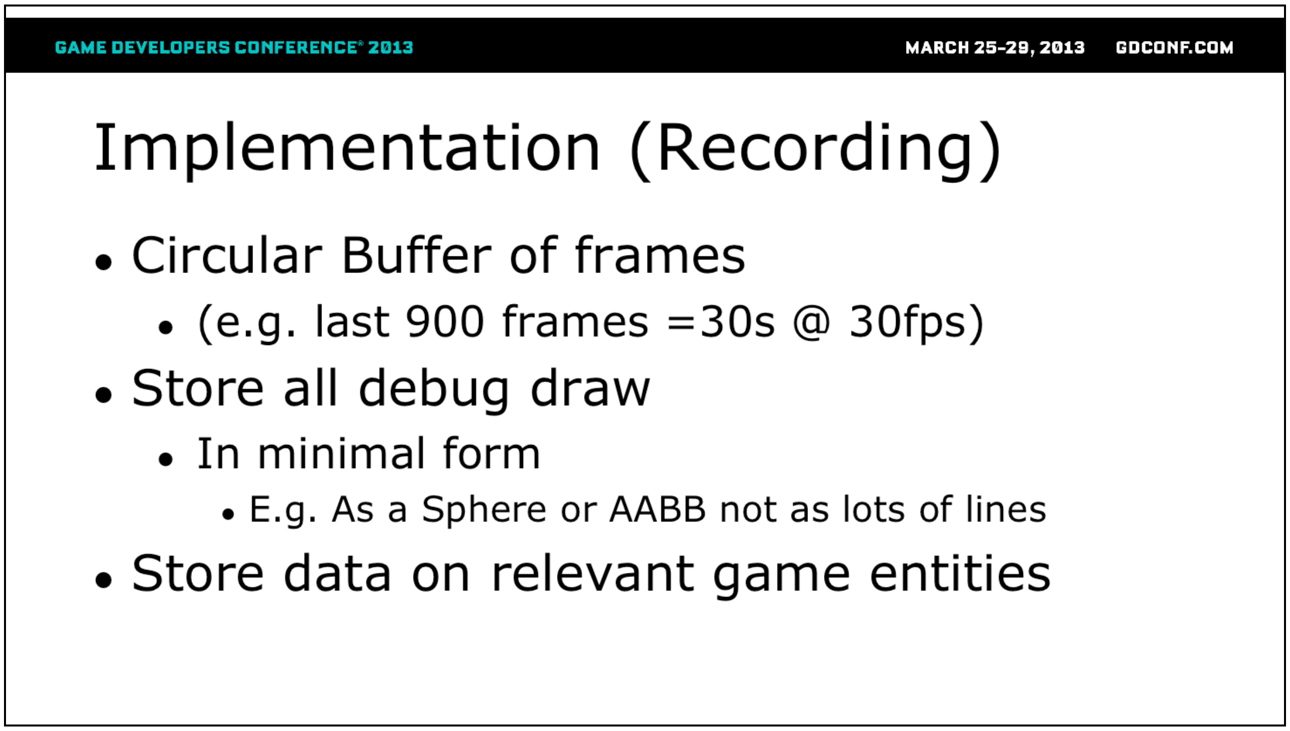
- And to make it shippable, the answer is, well, yes…
- But for a development only version then no, it really doesn't have to be. Because it's:
  - For internal use only
  - Doesn't have to be pretty
  - And it's a lot easier to find some spare memory – devkits usually have more memory than retail machines, your dev PCs are probably better than your minimum spec, etc.



Development time

- Basics can be done in 2 days
  - Debug Draw and Persistent Entities
- A complete version in just 1-2 weeks
  - Skeletal meshes, Temporary Entities
- Will pay for itself very quickly

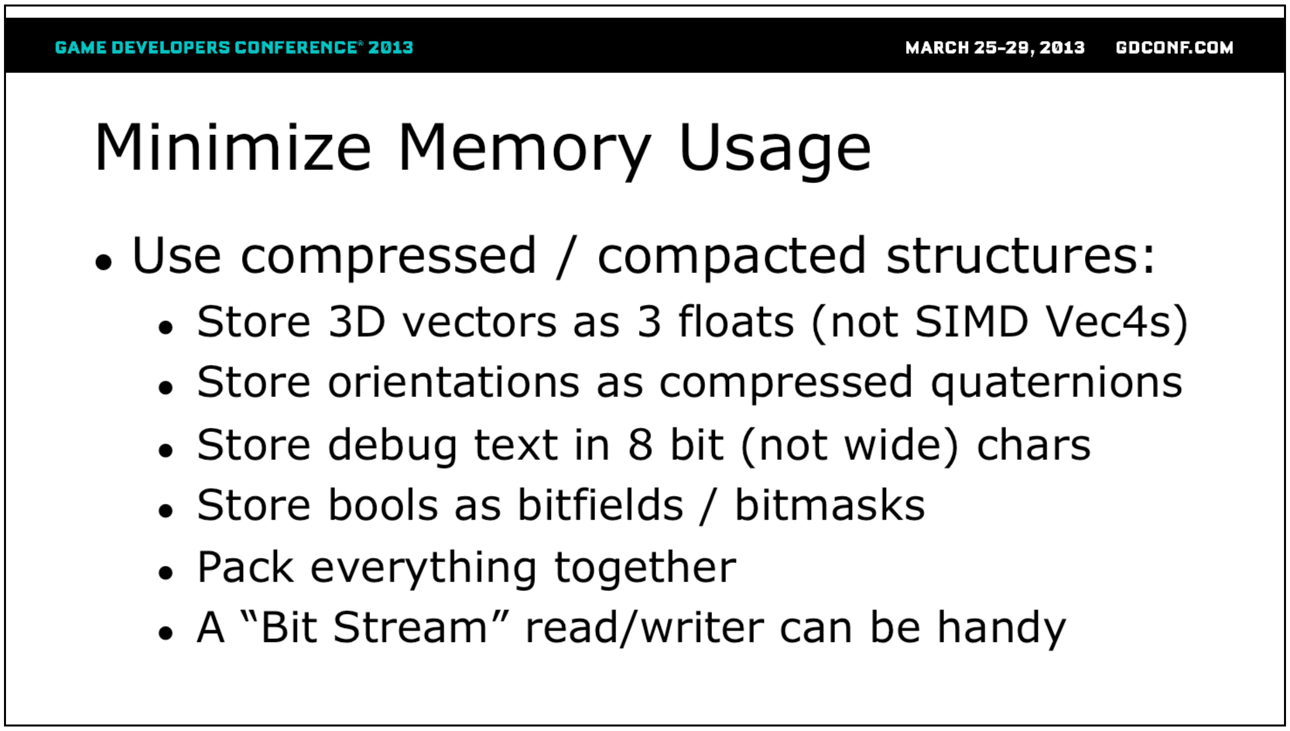
- So actually you can get a basic version working in just a couple of days.
- And if you can devote 1-2 weeks to it then you'll probably have everything you'll really want.
- And in my experience it pays for itself incredibly quickly.



Implementation (Recording)

- Circular Buffer of frames
  - (e.g. last 900 frames =30s @ 30fps)
- Store all debug draw
  - In minimal form
    - E.g. As a Sphere or AABB not as lots of lines
- Store data on relevant game entities

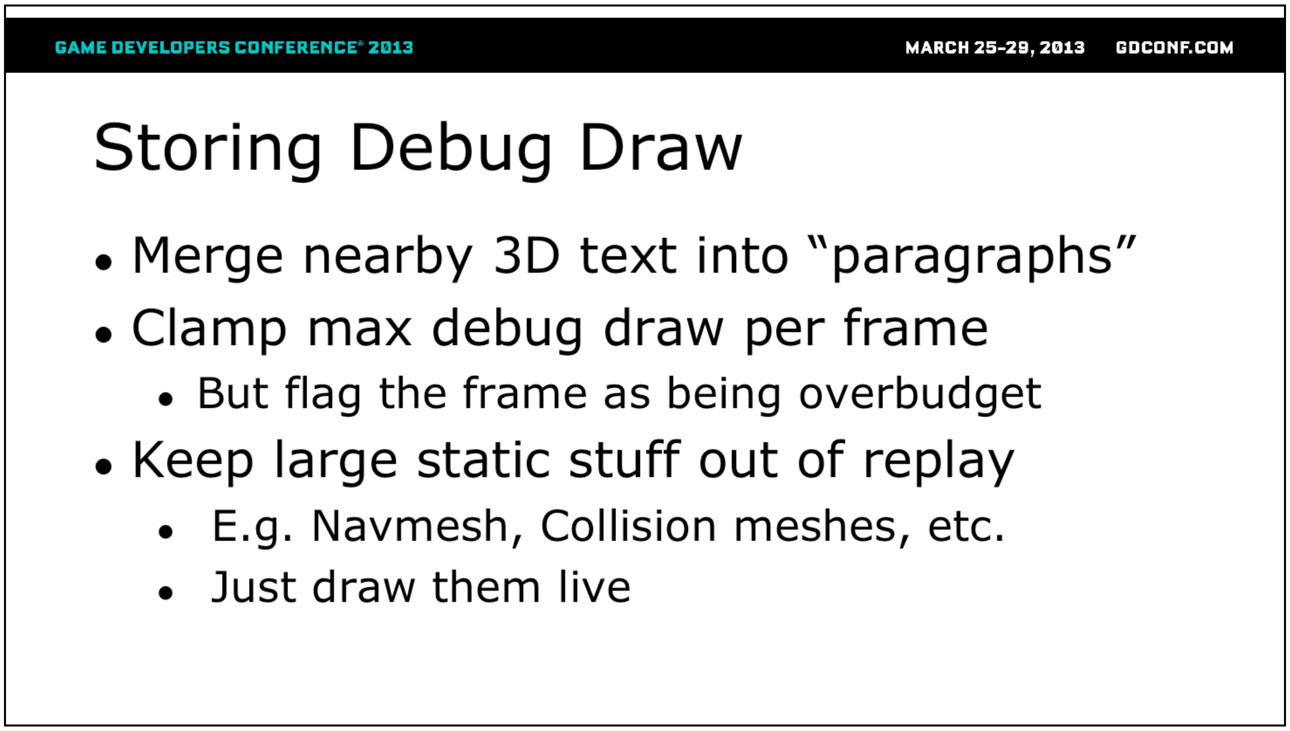
- So, to implement one, lets start with how to record the replay.
- In essence it's just a circular buffer of frames.
- And then for each frame store all of the debug draw for that frame
  - In a minimal form, so if you're storing a sphere just store the center and the radius, not as the hundreds of lines which you then render for it.
- Also store just enough data on any relevant entities (e.g. player, AI, dynamic physics objects, etc.)



Minimize Memory Usage

- Use compressed / compacted structures:
  - Store 3D vectors as 3 floats (not SIMD Vec4s)
  - Store orientations as compressed quaternions
  - Store debug text in 8 bit (not wide) chars
  - Store bools as bitfields / bitmasks
  - Pack everything together
  - A “Bit Stream” read/writer can be handy

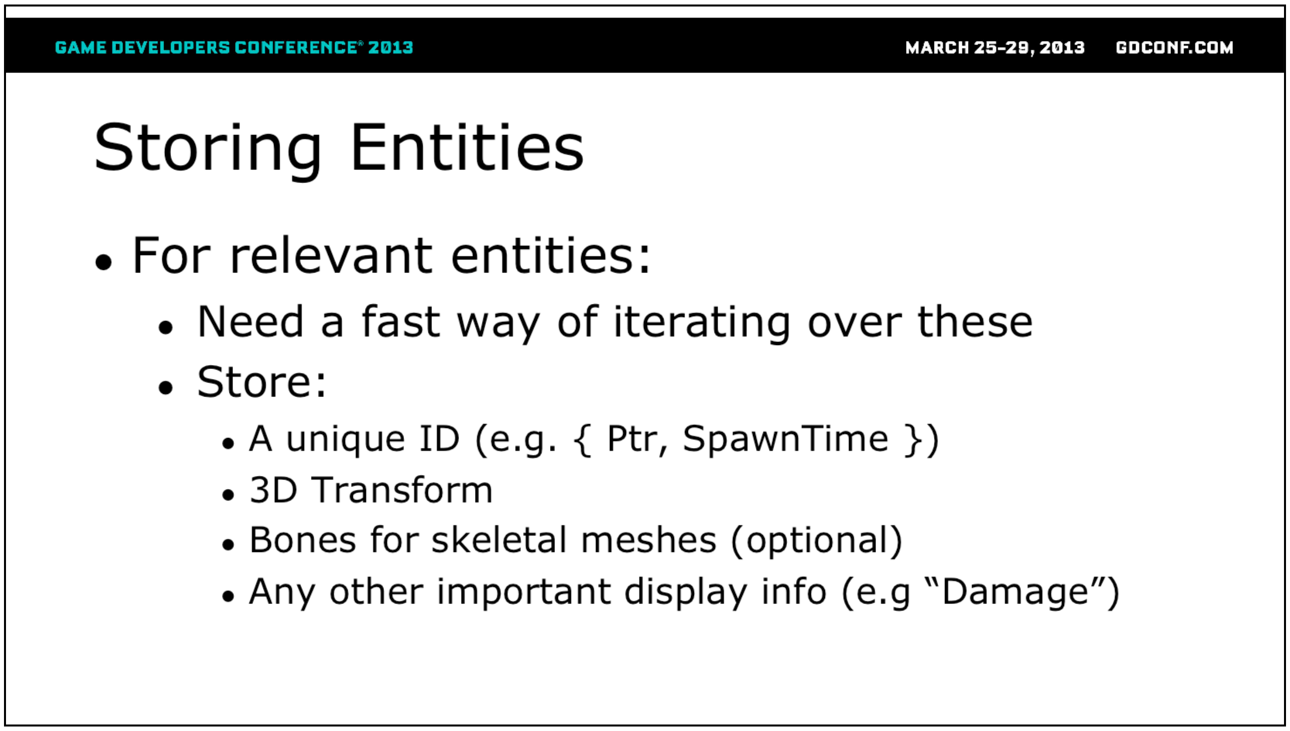
- Whilst you’re doing this, think about how to use as little memory as possible
  - Then you’ll have more room to store longer replays and the system will be more useful.
- So don’t store your vectors in your fancy 16 byte aligned SIMD-style vectors, instead just use 3 normally-aligned floats
- I’d recommend storing orientations as compressed quaternions
  - You might already have something like this for your animation system
  - Otherwise even something as simple as quantizing each component into a UInt8 works fine for this kind-of thing.
- Bools only need a single bit, so put them as bitfields, or bitmasks if you want it more cross-platform
- And generally try and pack everything together
- A “Bit Stream” style read/writer can be really handy here, making it easy to store floats on arbitrary bit boundaries and generally removing all alignment padding wastage automatically.



Storing Debug Draw

- Merge nearby 3D text into “paragraphs”
- Clamp max debug draw per frame
  - But flag the frame as being overbudget
- Keep large static stuff out of replay
  - E.g. Navmesh, Collision meshes, etc.
  - Just draw them live

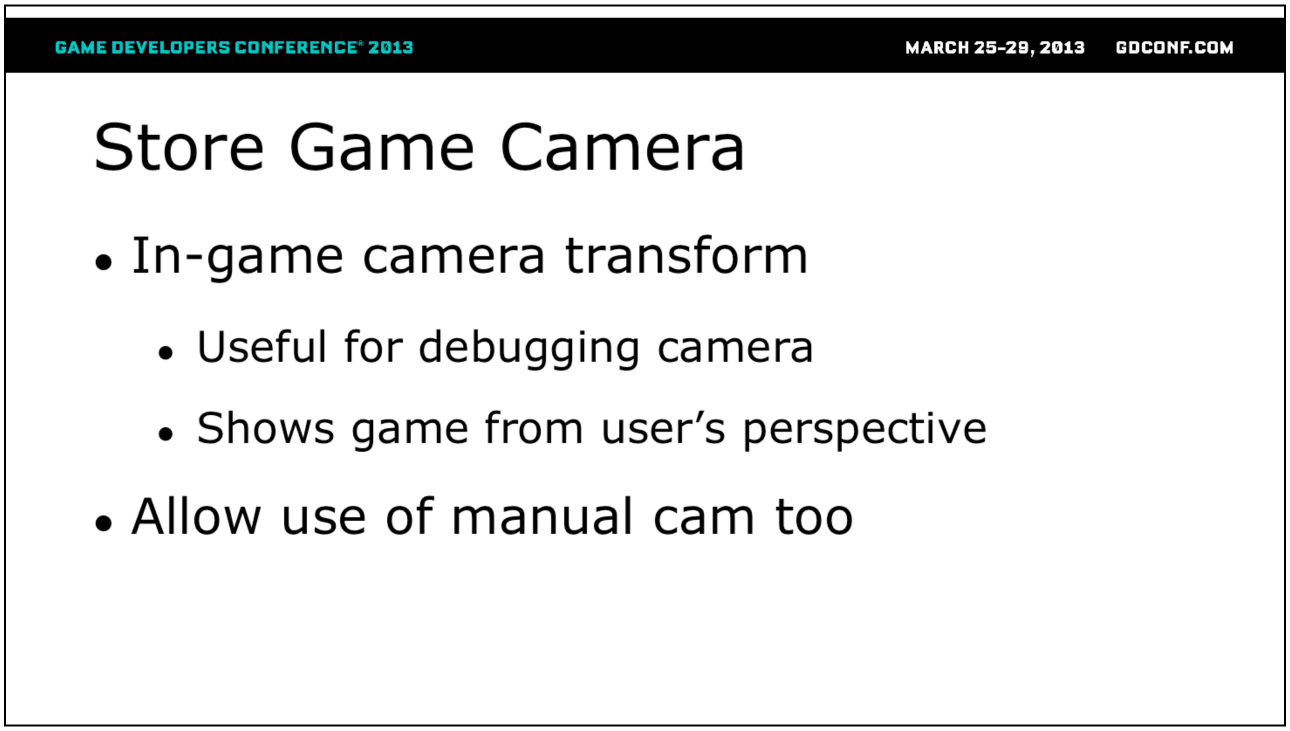
- For replay, storing text in 3D (next to each entity) can be really useful, and in general I’d suggest that for any 3D text you merge nearby text into paragraphs.
  - This has 2 big benefits:
    - It makes it much easier to read than 2 bits of text drawn on top of each other
    - And it saves memory as you only need to store 1 vec3 per paragraph
- Clamp maximum debug draw data stored per frame - so if someone turns on an insane amount of debug draw the game still runs correctly.
  - I recommend that you store 1 bit in each frame to tag whether that frame ran out of room, then when viewing a replay you’ll know if/why not all of your debug draw is showing up.
- Keep large static debug draw out of replay
  - E.g. better to render things like Navmesh or static collision meshes live rather than store the same hundreds of lines in every frame



Storing Entities

- For relevant entities:
  - Need a fast way of iterating over these
  - Store:
    - A unique ID (e.g. { Ptr, SpawnTime })
    - 3D Transform
    - Bones for skeletal meshes (optional)
    - Any other important display info (e.g "Damage")

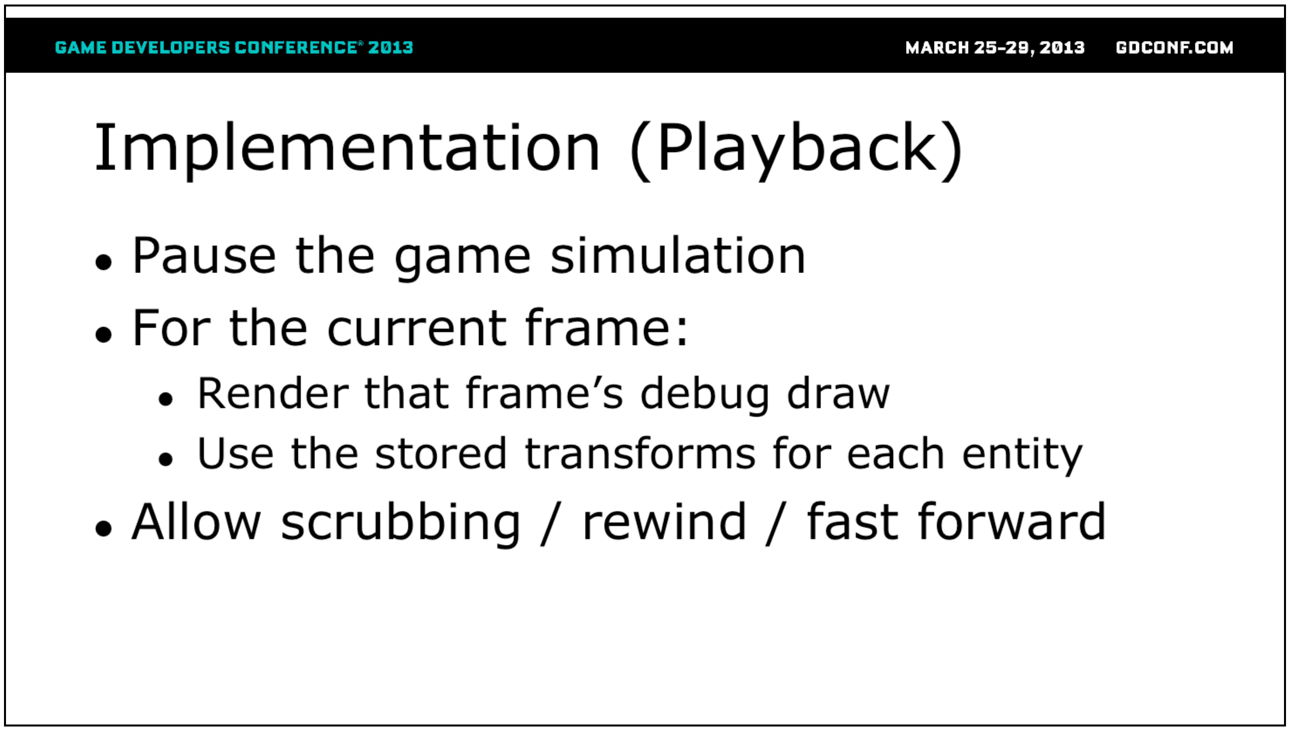
- For storting entities:
  - Need a fast way of iterating over the interesting ones. If you have thousands of entities and only want to store a few then you might want to keep a list of the interesting ones to speed this up – because we want the CPU impact of recording a replay to be completely negligible
  - A unique ID – this helps to associate an entity between 2 frames of replay, and is also useful for mapping back to additional information on that entity
  - The 3D Transform
  - Bones (transforms) for skeletal meshes: make this optional, toggle-able at runtime – these can use a lot of memory, and although are vital for debugging animation and making a good-looking replay, they're not necessary for debugging most gameplay things so if memory is scarce some people might prefer to have longer replays without storing the animation bones.
  - Any other important display info. This really depends on your game and your entities. e.g maybe you have a damage or dirt value, or some flags that effect if something "IsGlowing", blurring, etc.


GAME DEVELOPERS CONFERENCE® 2013 MARCH 25-29, 2013 GDCONF.COM

Store Game Camera

- In-game camera transform
  - Useful for debugging camera
  - Shows game from user's perspective
- Allow use of manual cam too

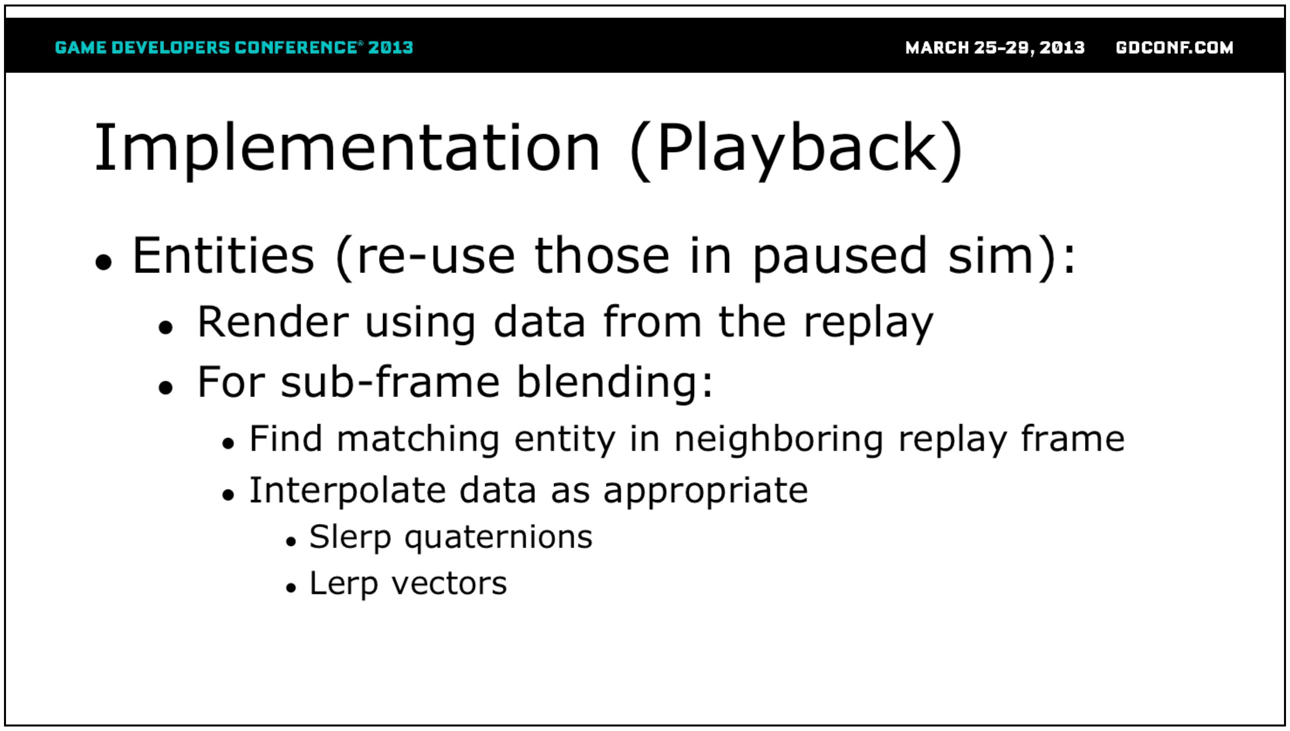
- I highly recommend that you also store the in-game camera transform
  - This is not only really useful for debugging the camera system
  - But it's also great to jump back to the player's perspective, and it's a useful starting point to position the manual camera at.
- You'll definitely want a manual camera mode too so that you can view everything from anywhere


GAME DEVELOPERS CONFERENCE® 2013 MARCH 25-29, 2013 GDCONF.COM

Implementation (Playback)

- Pause the game simulation
- For the current frame:
  - Render that frame's debug draw
  - Use the stored transforms for each entity
- Allow scrubbing / rewind / fast forward

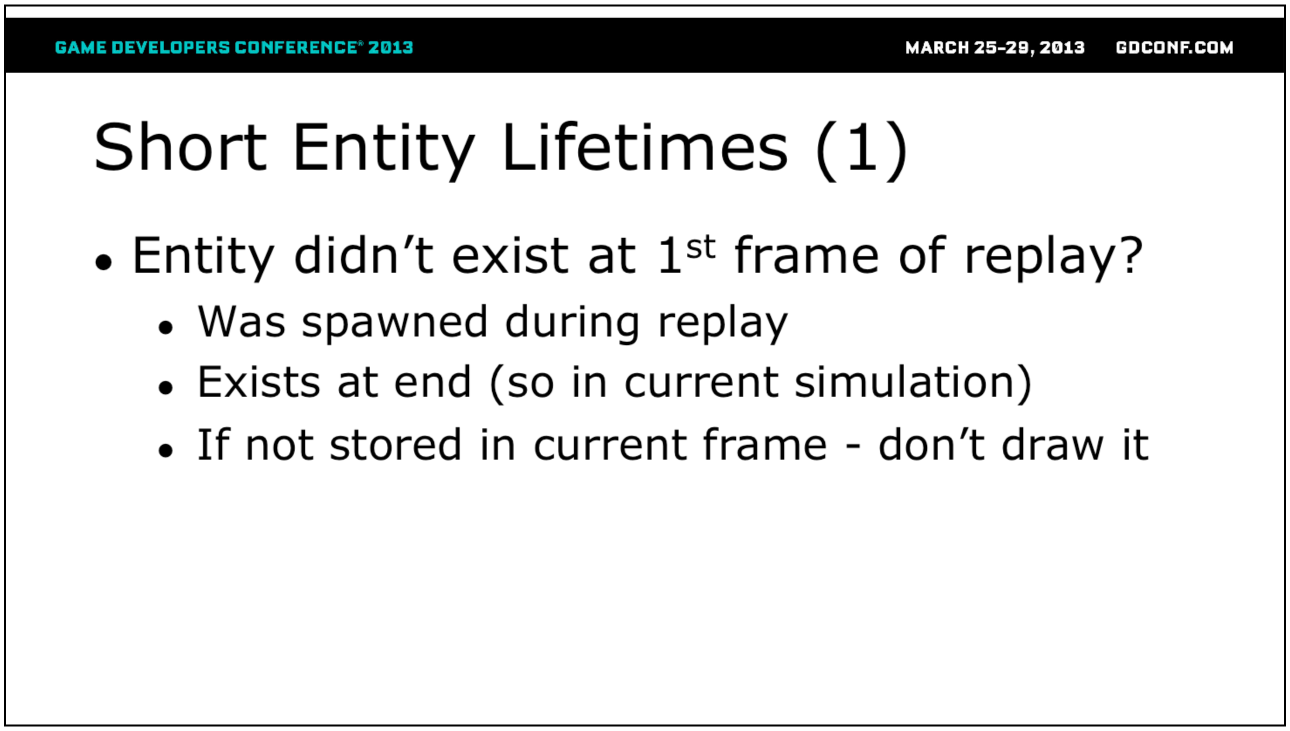
- So, you've stored a replay, now to play it back
- Just pause the game simulation (and stop submitting frames to the replay)
- For the currently viewed frame
  - Render that frame's debug draw
  - Show the entities in the world (the ones that still exist in the currently paused simulation) using the stored data at that frame in the replay
- Allow user to scrub / rewind / fast forward through the replay.



Implementation (Playback)

- Entities (re-use those in paused sim):
  - Render using data from the replay
  - For sub-frame blending:
    - Find matching entity in neighboring replay frame
    - Interpolate data as appropriate
      - Slerp quaternions
      - Lerp vectors

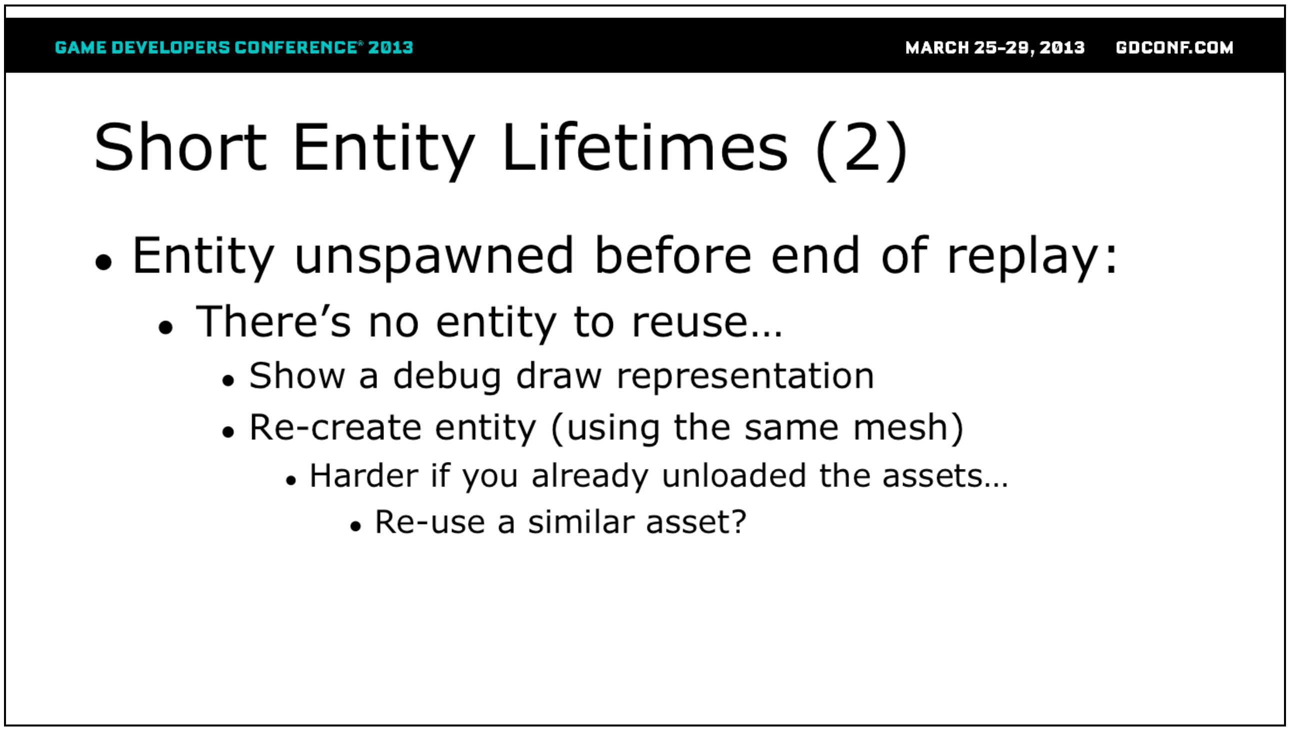
- So, you've stored a replay, now to play it back
- For entities, we re-use the ones that still exist in the currently paused simulation, but:
  - Render entities using the transforms (and other settings) from the replayed frame
  - For sub-frame blending:
    - Note it's still handy to be able to step through and look at "concrete" recorded frames, but it's great to be able to see what the change between 2 frames was, especially in the case of large changes in position or orientation.
    - To implement, just:
      - Find the matching entity in the neighboring replay frame (using the Unique ID)
      - Interpolate data as appropriate (e.g. Slerp quaternions, Lerp vectors, etc.)



Short Entity Lifetimes (1)

- Entity didn't exist at 1st frame of replay?
  - Was spawned during replay
  - Exists at end (so in current simulation)
  - If not stored in current frame - don't draw it

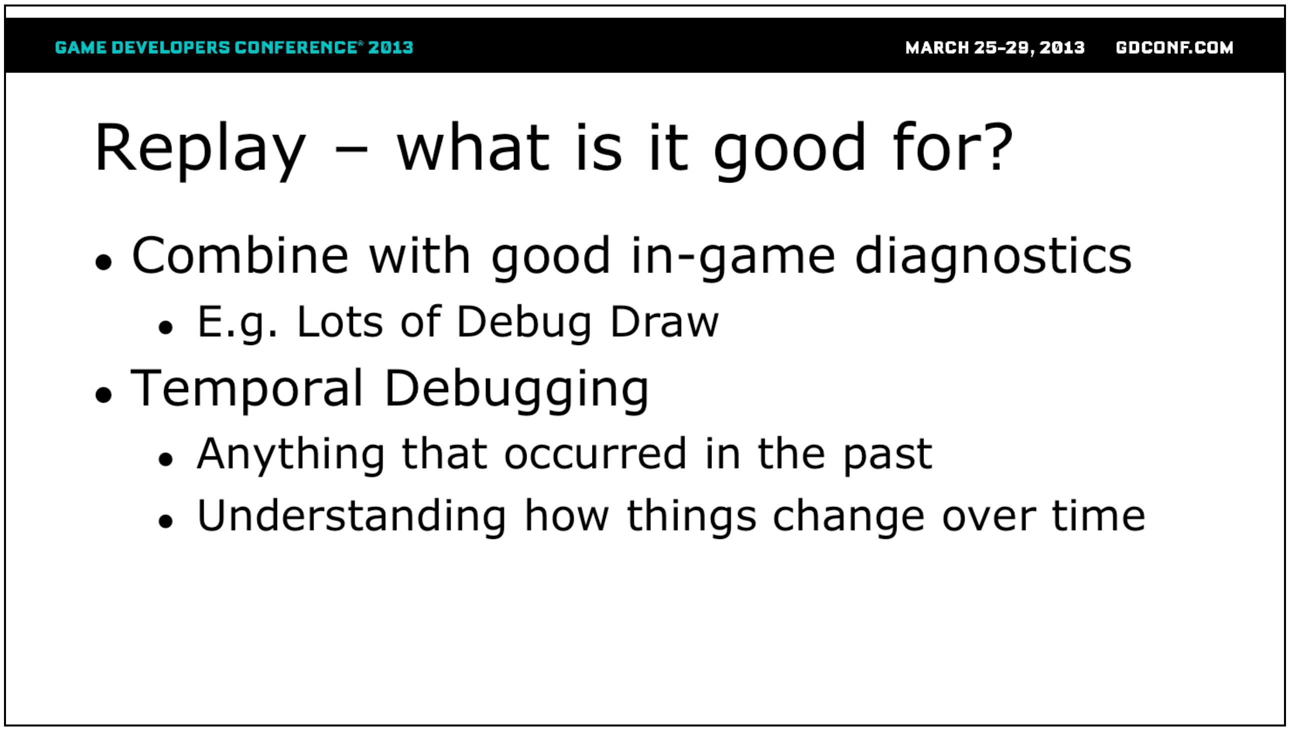
- Entities with shorter lifetimes are a little bit trickier than anything that's persistent
- First the easier case, where an entity was spawned during the replay:
  - If they weren't in the currently replayed frame, just don't draw them.


GAME DEVELOPERS CONFERENCE® 2013 MARCH 25-29, 2013 GDCONF.COM

Short Entity Lifetimes (2)

- Entity unspawned before end of replay:
  - There's no entity to reuse…
    - Show a debug draw representation
    - Re-create entity (using the same mesh)
      - Harder if you already unloaded the assets…
        - Re-use a similar asset?

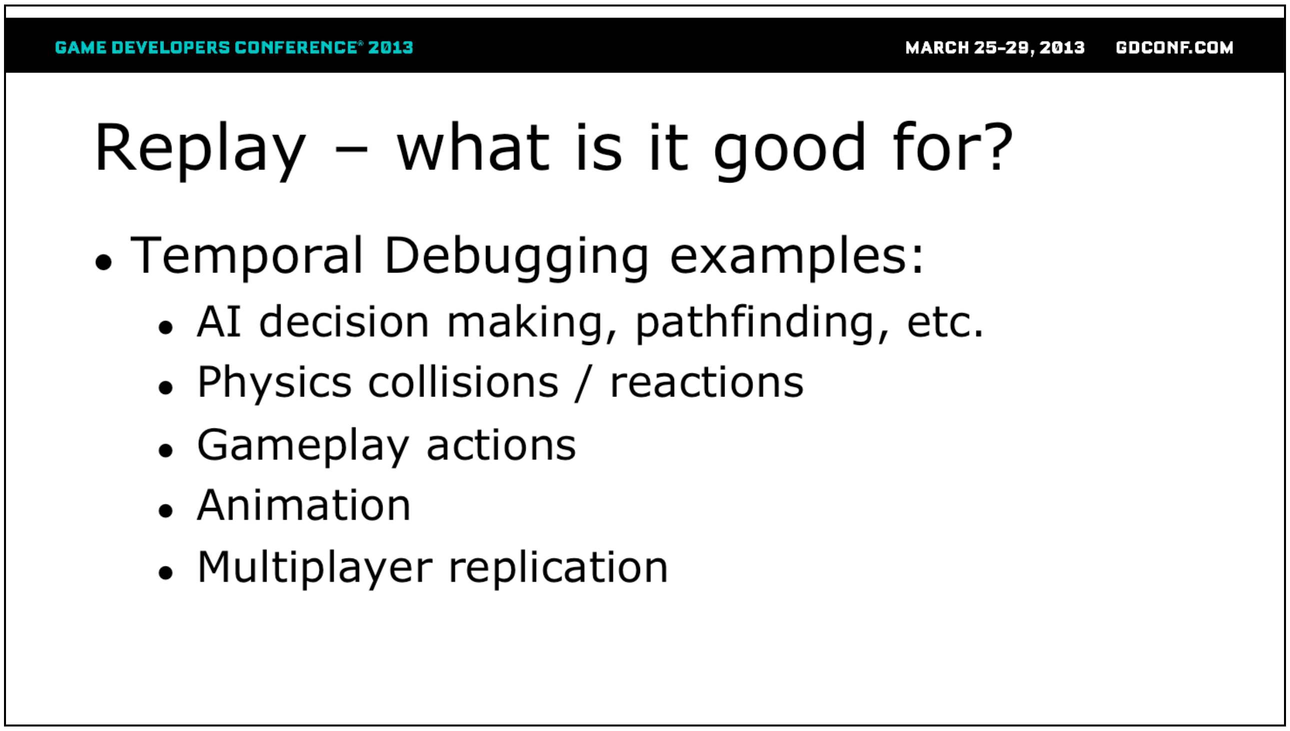
- Now the harder case, where an entity was unspawned during the replay:
  - So it's not present in the currently paused simulation, so we cannot reuse it
    - You could just draw a debug representation – e.g. a stick figure (using the stored bones), or a debug sphere
    - Better is to re-create the entity using the original mesh – by storing a mapping from the UID of each recently spawned entity and how to recreate them.
    - This gets even harder if you've already unloaded the assets…
      - You probably don't want to try and fix this unless you want to ship this as a feature
      - But for debugging, just re-use a similar asset (e.g. another biped with the same rig / skeleton)


GAME DEVELOPERS CONFERENCE® 2013 MARCH 25-29, 2013 GDCONF.COM

Replay – what is it good for?

- Combine with good in-game diagnostics
  - E.g. Lots of Debug Draw
- Temporal Debugging
  - Anything that occurred in the past
  - Understanding how things change over time

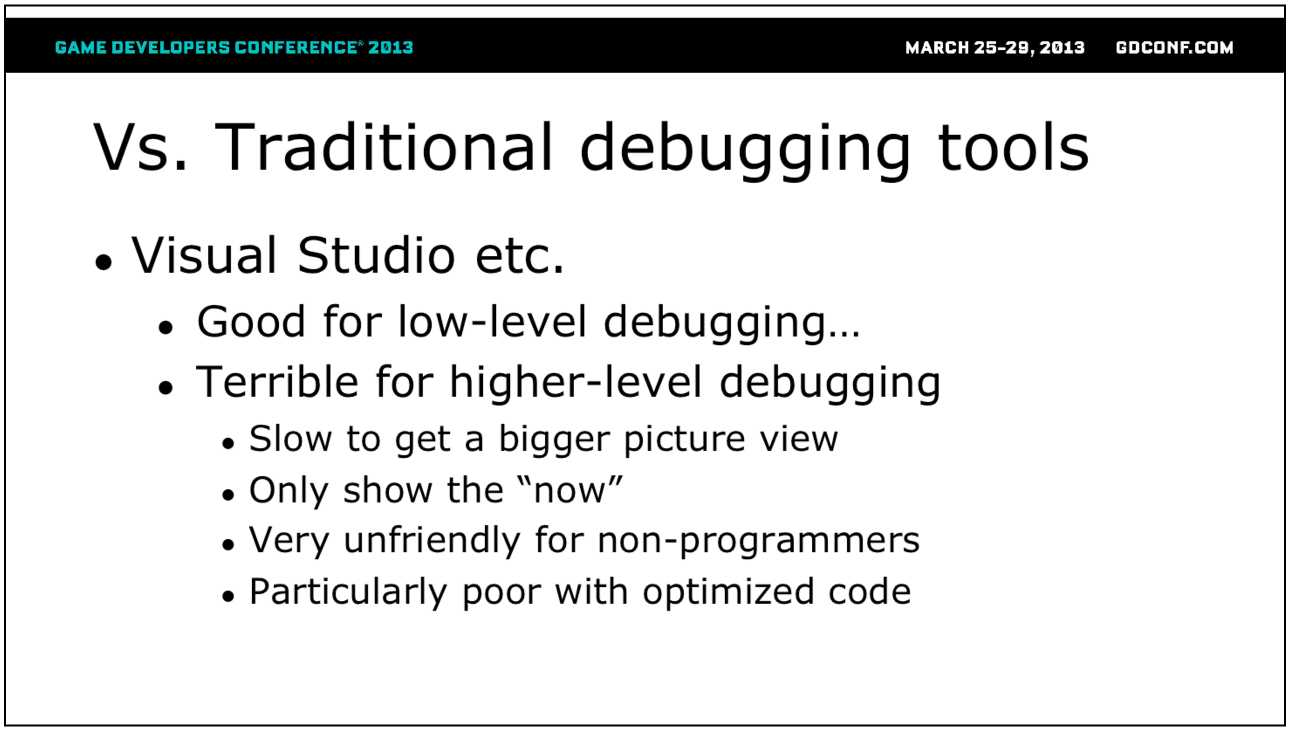
- So what can you actually use a rewindable replay system for?
- Well, combined with some good in-game diagnostics (e.g. lots of good debug draw)
- Then it helps provide what I'd call "temporal debugging", e.g:
  - Debugging anything that occurred in the past (and how/why it happened)
  - Understanding how things changed over time



Replay – what is it good for?

- Temporal Debugging examples:
  - AI decision making, pathfinding, etc.
  - Physics collisions / reactions
  - Gameplay actions
  - Animation
  - Multiplayer replication

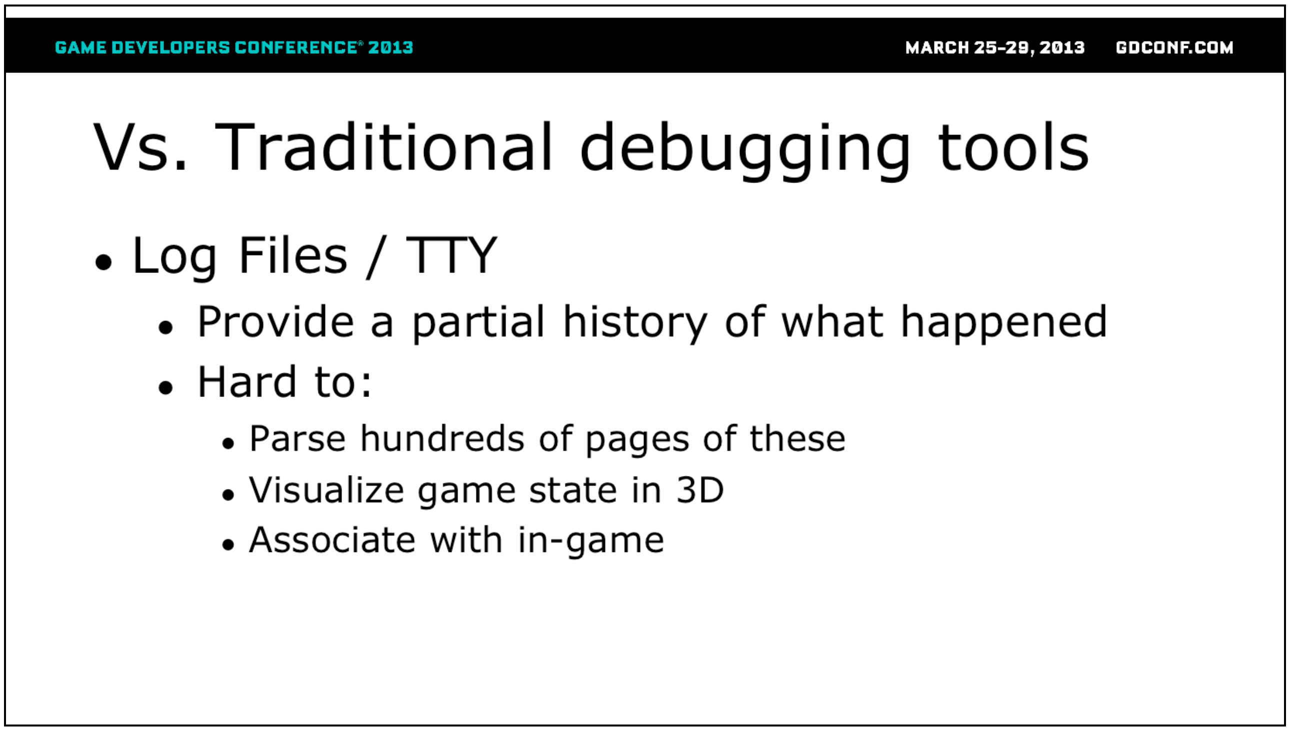
- Some temporal debugging examples:
  - Great for AI decision making, pathfinding, etc. You can see how the AI state's changed, why they made a decision on any given frame, how their states and decisions changed over tine.
  - Physics collisions, reactions – e.g. understanding and reviewing collisions, impulses, motion etc.
  - Pretty much any Gameplay action - see when and why they happened
  - Animations. Our animators love it. They can review the animations in game, at any angle, in slow-motion, rewind, see sub-frame interpolations, etc.
  - Multiplayer replication – you can show the replication information on the frame it was sent, on the frame it arrived, and how it relates to your current forward prediction and help debug issues in lag, jitter and generally make for a smoother online experience.


GAME DEVELOPERS CONFERENCE® 2013 MARCH 25-29, 2013 GDCONF.COM

Vs. Traditional debugging tools

- Visual Studio etc.
  - Good for low-level debugging…
  - Terrible for higher-level debugging
    - Slow to get a bigger picture view
    - Only show the "now"
    - Very unfriendly for non-programmers
    - Particularly poor with optimized code

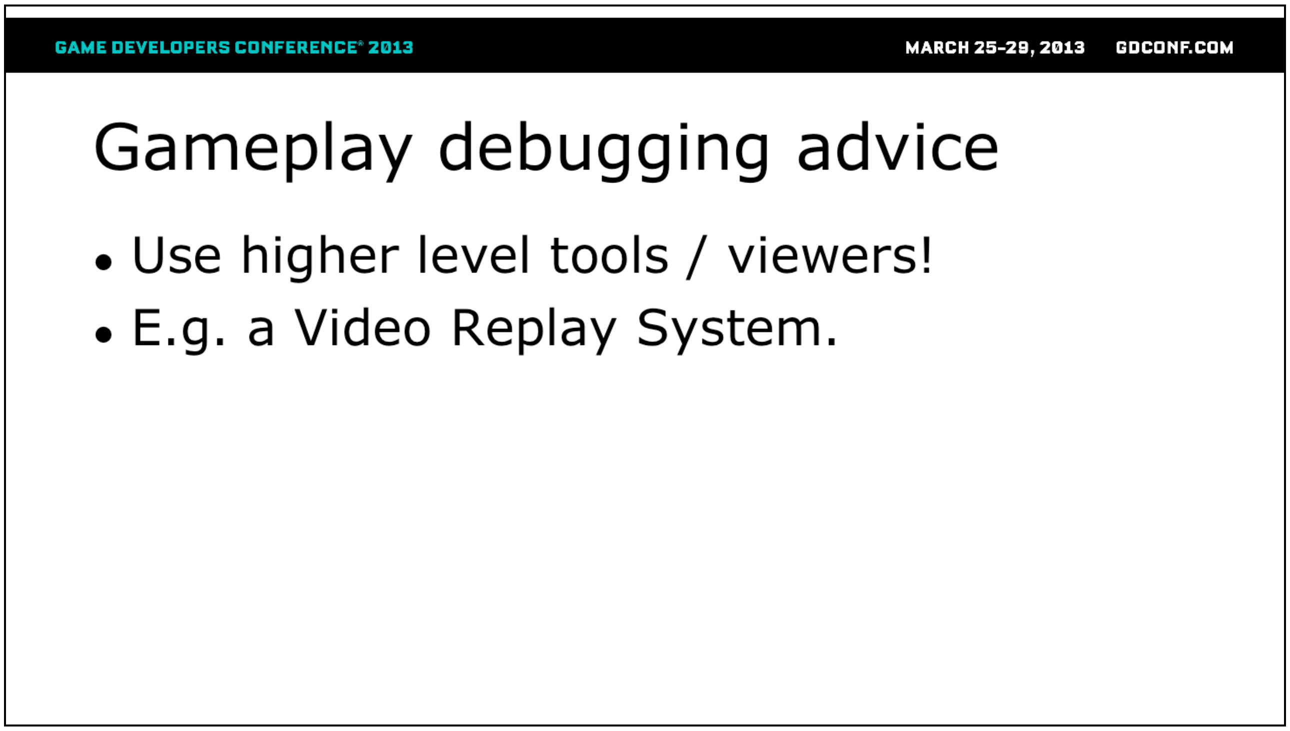
- A quick comparison with traditional debugging tools
- So there are traditional debuggers, e.g. Visual Studio
  - And sure they're great for low-level debugging,
  - But they're terrible for higher-level debugging
    - Slow to get a higher-level view on things – hard to visualize what's happening
    - Only show the current state – no clue to what happened before this point to lead to that state
    - Very unfriendly for non-programmers (and often unavailable on their machines anyway).
    - Even for programmers they're particularly poor with optimized code – so you have to manually go through the assembly, look in registers and memory, cast them manually in a watch window… Or run in debug but then that's too slow, or have some halfway-house where you only build the subsystem you're interested in in debug and the rest is in release, but it's still far from ideal…
- For "Higher-level" debugging = e.g. gameplay, AI, animation, physics and multiplayer, you can work far more efficiently outside of the debugger.


GAME DEVELOPERS CONFERENCE® 2013 MARCH 25-29, 2013 GDCONF.COM

Vs. Traditional debugging tools

- Log Files / TTY
  - Provide a partial history of what happened
  - Hard to:
    - Parse hundreds of pages of these
    - Visualize game state in 3D
    - Associate with in-game

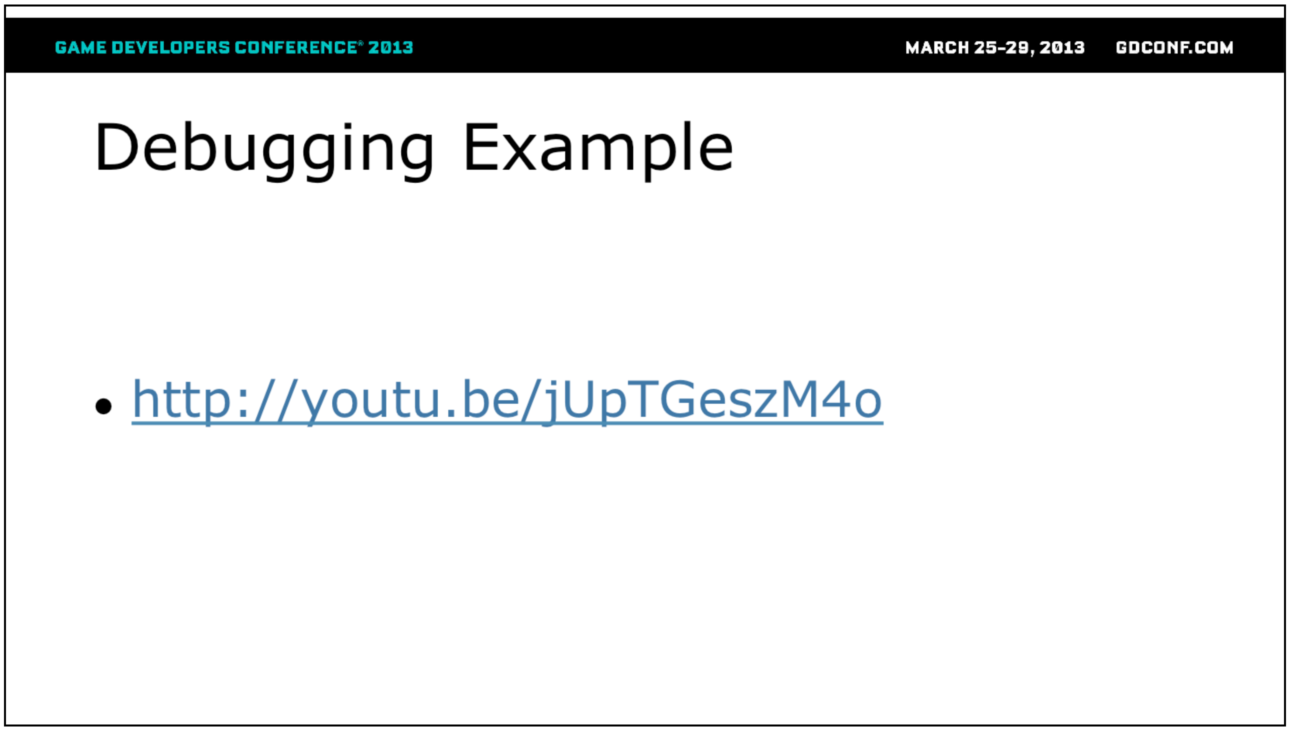
- The other common debugging tool is a Log File / TTY / “printf debugging”
- These do provide a partial history of what happened
- But it’s:
  - Very hard to parse hundreds of pages of these
  - Hard to relate to game state in 3D – what does a position or direction of <1.23, 0.11, 5.2> even mean?
  - Hard to associate with what else was happening on a given frame


GAME DEVELOPERS CONFERENCE® 2013 MARCH 25-29, 2013 GDCONF.COM

Gameplay debugging advice

- Use higher level tools / viewers!
- E.g. a Video Replay System.

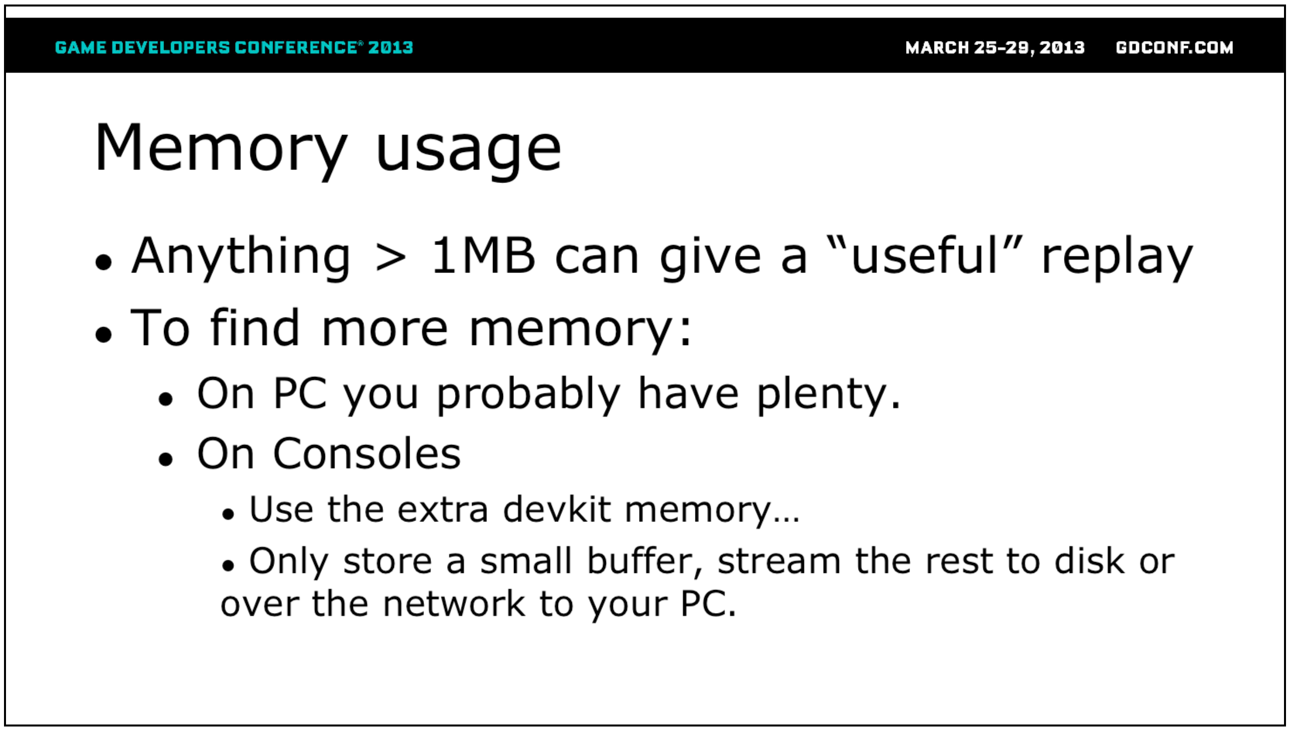
- So my general advice is:
  - Use higher level tools and viewers whenever possible!
  - And I think a replay system is a great tool to have as part of this


GAME DEVELOPERS CONFERENCE® 2013 MARCH 25-29, 2013 GDCONF.COM

Debugging Example

- http://youtu.be/jUpTGeszM4o

- So lets look at an example of using replay to debug something
- Again, sadly, I’m demoing this on a sample project instead of on a project that I’ve worked on, but hopefully you’ll be able to see the approaches you can use and some of the benefits from using a rewindable replay system.
- http://youtu.be/jUpTGeszM4o

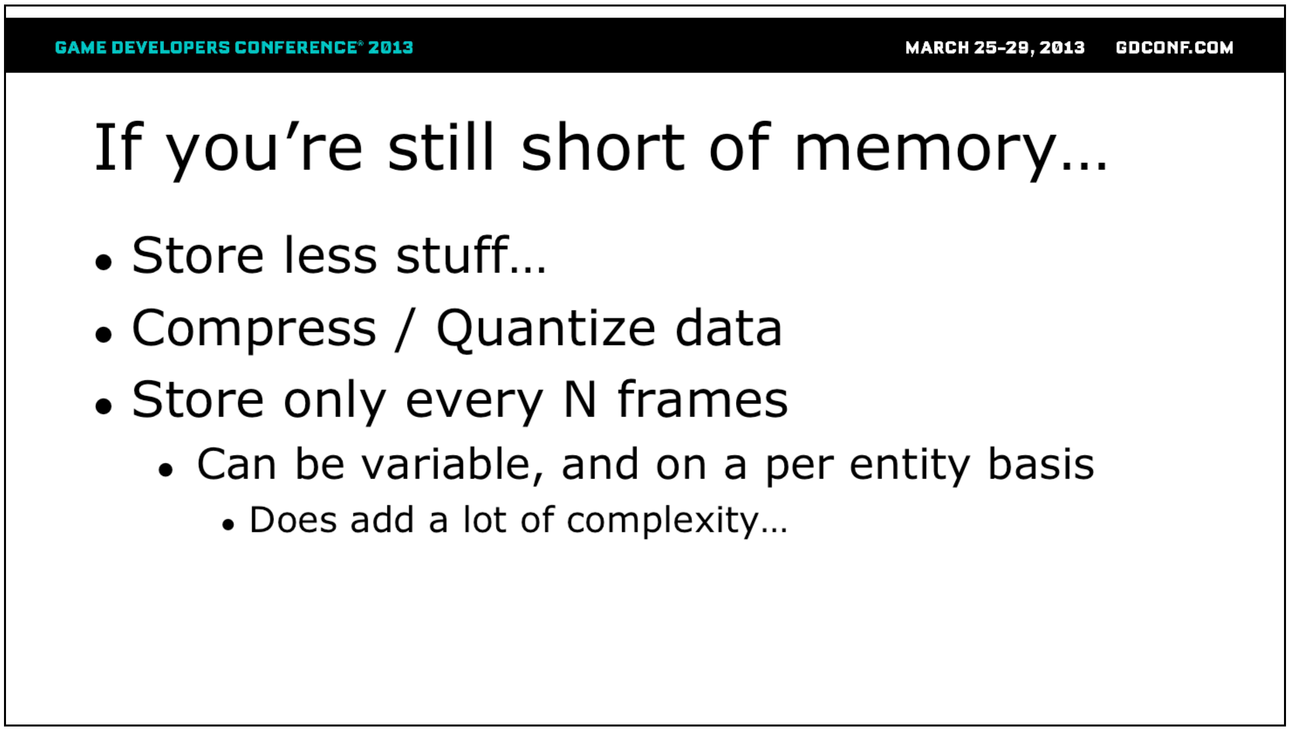


GAME DEVELOPERS CONFERENCE® 2013 MARCH 25-29, 2013 GDCONF.COM

Memory usage

- Anything > 1MB can give a “useful” replay
- To find more memory:
  - On PC you probably have plenty.
  - On Consoles
    - Use the extra devkit memory…
    - Only store a small buffer, stream the rest to disk or over the network to your PC.

- So how much moery does this need?
- Well, if you can provide anything over 1MB then you can store a “useful” replay (e.g. skeletal meshes disabled, but everything else enabled, debug draw limited to a few primitives per-frame)
- To find some more memory…
  - Well on PC you probably already have plenty, especially if you have a 64 bit build.
  - On consoles…
    - You could use some of the extra memory that’s available in most devkits
    - Or only store a small buffer in memory, and stream the rest to disk or over the network to your PC in the background. (This can be re-used for other debug systems too…)

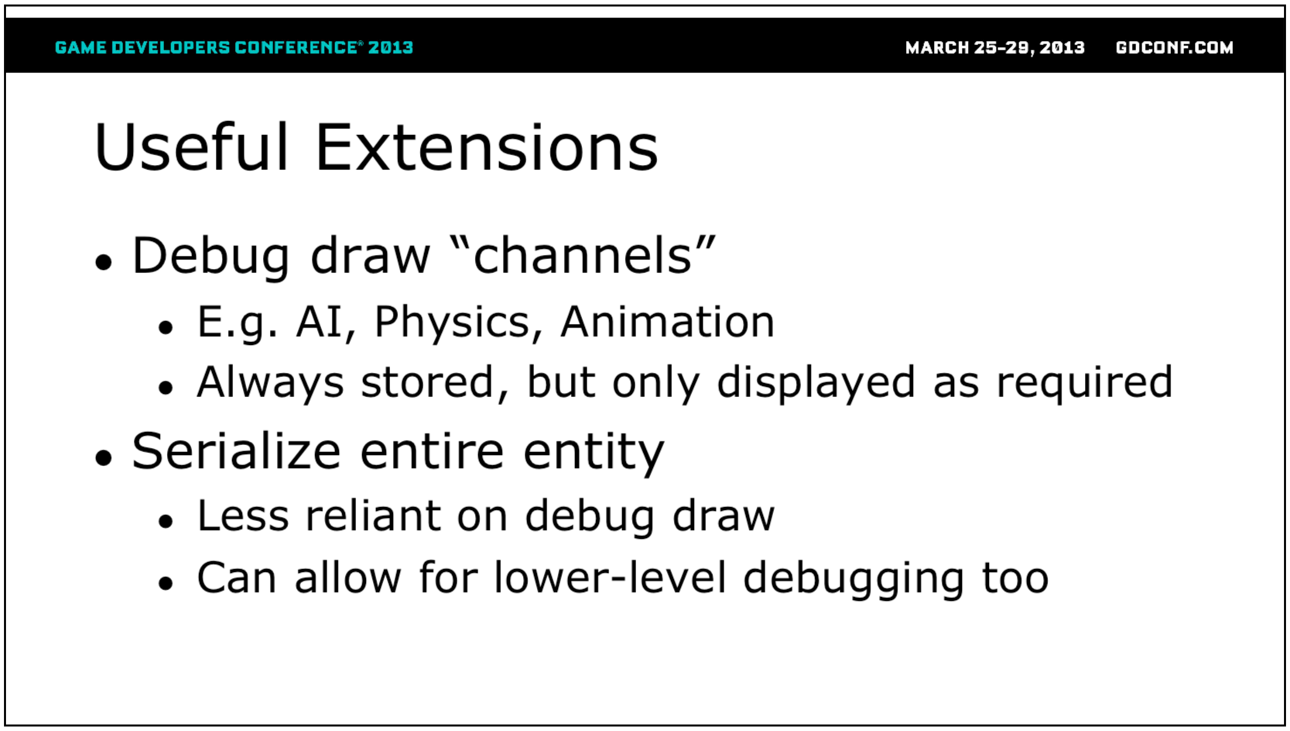


If you're still short of memory…

- Store less stuff…
- Compress / Quantize data
- Store only every N frames
  - Can be variable, and on a per entity basis
    - Does add a lot of complexity…

If you're still short of memory – e.g. you're on a very small device, or want to get this in the retail version:

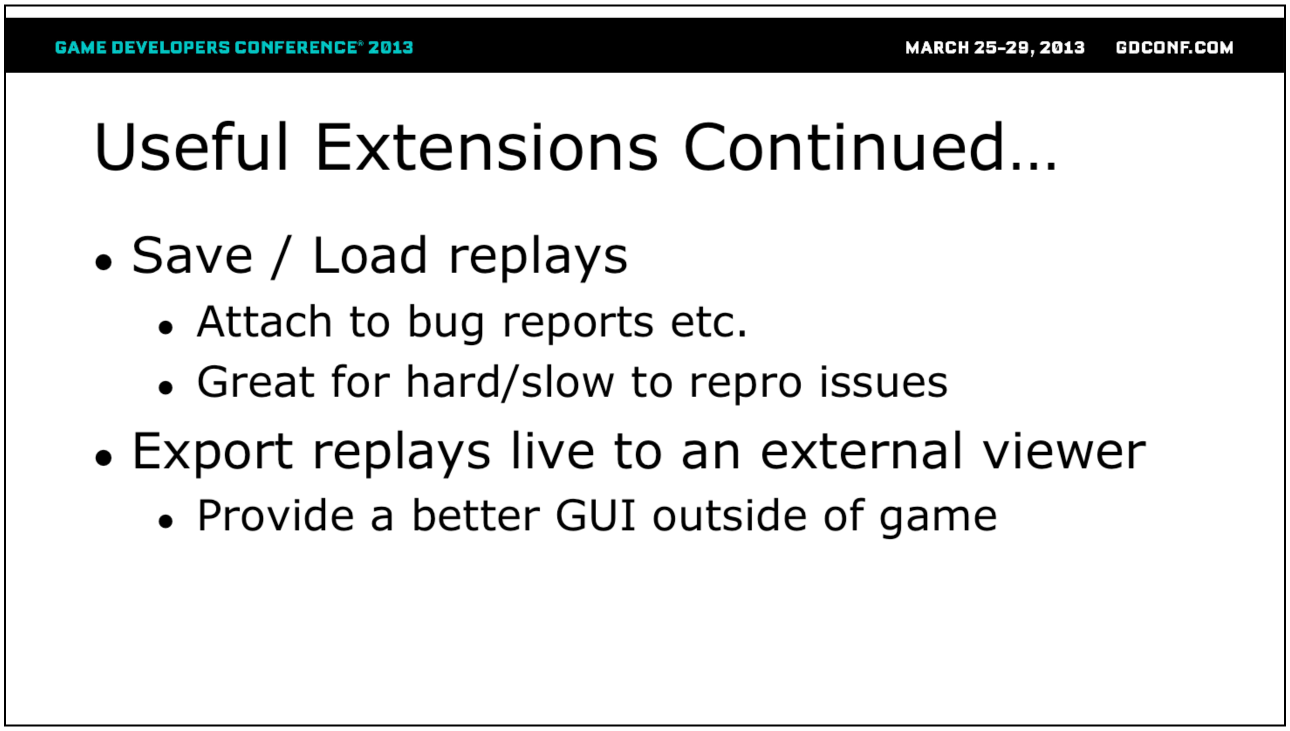
- Store less stuff… (e.g. turn off bones, unimportant entities, unimportant state info, etc.)
- Compress / Quantize data – e.g. pack floats with small ranges into int16s (or smaller)
- Store only every N frames
  - This can be variable and on a per entity basis
  - You can for example keep a small buffer of the last few exact frames, compare them against the interpolated results, and put down a new frame whenever you would otherwise exceed some error tolerance.



Useful Extensions

- Debug draw "channels"
  - E.g. AI, Physics, Animation
  - Always stored, but only displayed as required
- Serialize entire entity
  - Less reliant on debug draw
  - Can allow for lower-level debugging too

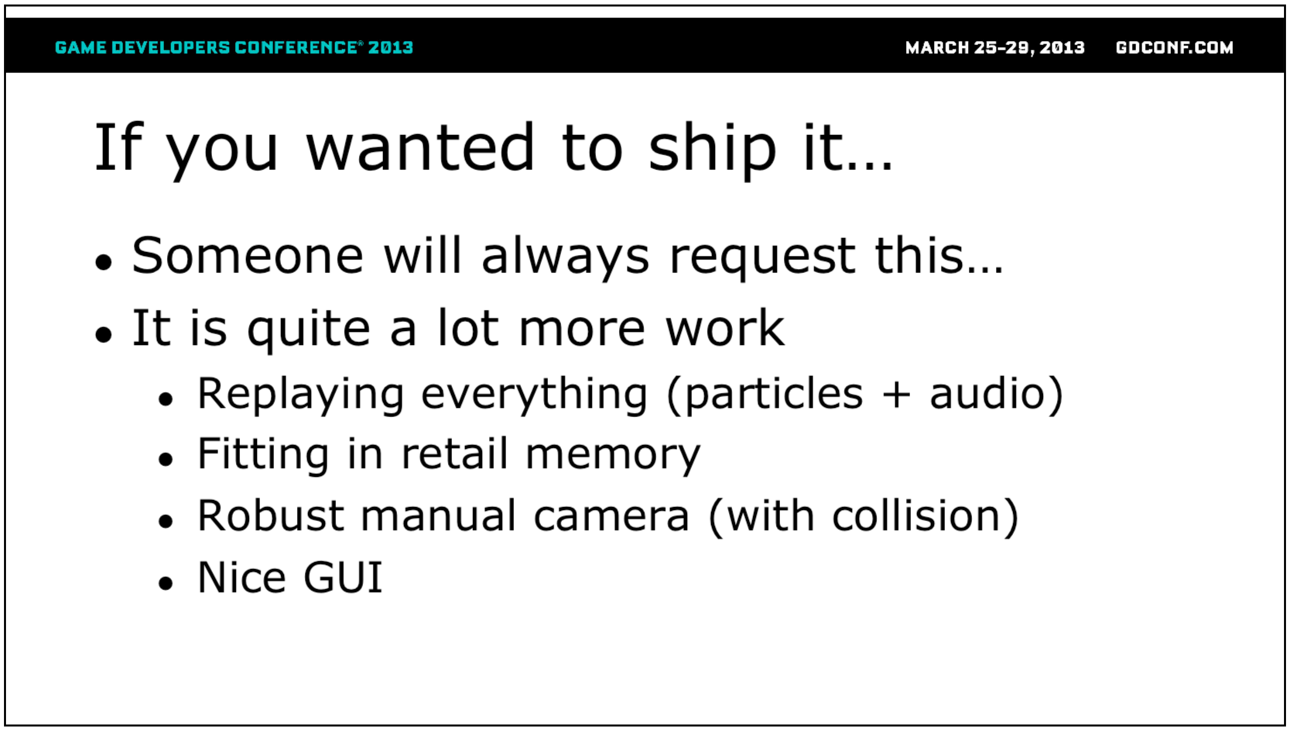
- Some useful extensions you can build on top of this:
- Debug draw "channels" – so you're always submitting AI, Physics and other debug info, but it's not necessarily displayed during the game and in replay you can optionally choose the channels you wish to display.
  - This is particularly great for when another team-member encounters an issue on their machine, where they are unlikely to have had specific debug draw on, because you're still able to rewind and enable the channels that you're interested in.
- Serialize more of the entity data, or entire entity, so that you can show the entire state of everything at any frame.
  - Can make you less reliant on adding debug draw for everything, and also potentially allow you to dig into some lower-level details too.


GAME DEVELOPERS CONFERENCE® 2013 MARCH 25-29, 2013 GDCONF.COM

Useful Extensions Continued…

- Save / Load replays
  - Attach to bug reports etc.
  - Great for hard/slow to repro issues
- Export replays live to an external viewer
  - Provide a better GUI outside of game

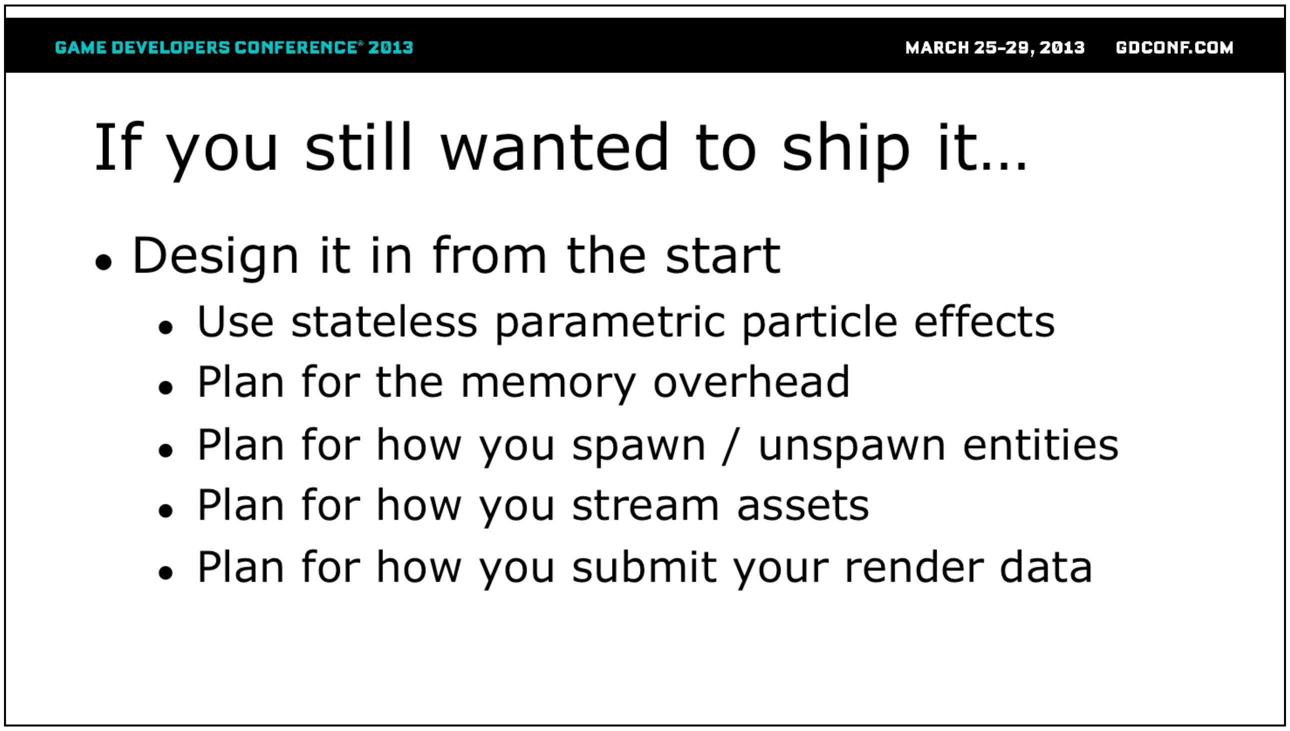
- Save / load replays
  - Have your QA automatically attach them to bug reports etc.
  - Replay has effectively already serialized everything anyway…
  - Can be great for hard/slow to repro issues – just watch the replay, enable any relevant debug channels, and you can often understand and fix the issue without even having to reproduce the problem.
- Similarly, export serialized data to an external viewer / visual debugger
  - Where you might be able to provide a better GUI outside of the game (or on your PC whilst debugging on console)



If you wanted to ship it…

- Someone will always request this…
- It is quite a lot more work
  - Replaying everything (particles + audio)
  - Fitting in retail memory
  - Robust manual camera (with collision)
  - Nice GUI

- I guarantee that when you first implement this in your game, someone will ask how much work it would take to put this in your retail game for the end-user to play with
- And it really is quite a lot more work
  - Replaying everything (particles + audio), ensuring required assets are in memory
  - Fitting in retail memory (or on your min spec PC)
  - Robust manual camera (no flying through objects) – for debugging you don't want any camera collision, but for your end user you do so that they don't see things like how all of your levels are just like Hollywood sets.
  - Nice GUI

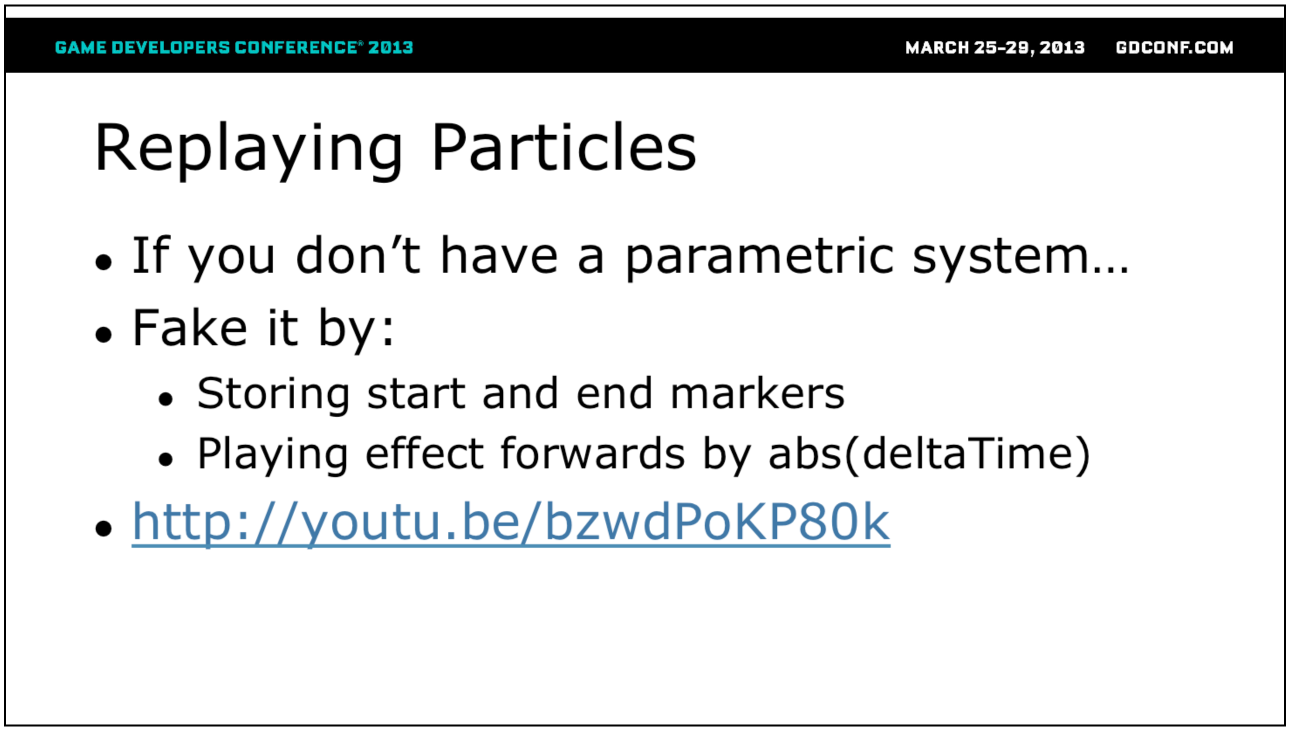


GAME DEVELOPERS CONFERENCE® 2013 MARCH 25-29, 2013 GDCONF.COM

If you still wanted to ship it…

- Design it in from the start
  - Use stateless parametric particle effects
  - Plan for the memory overhead
  - Plan for how you spawn / unspawn entities
  - Plan for how you stream assets
  - Plan for how you submit your render data

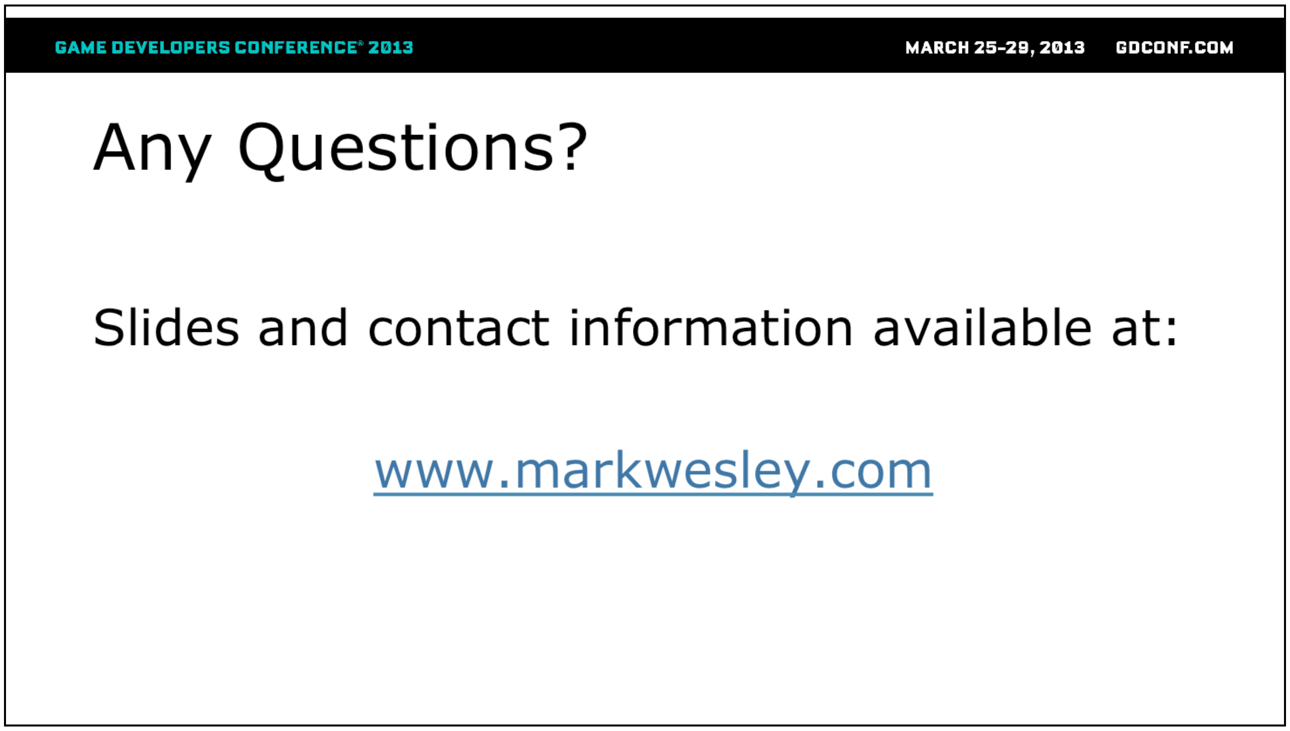
- If you still really wanted to ship it, then I'd highly recommend that you design it in from the start – there are a number of things which are particularly hard to do otherwise:
- Limit yourself to stateless, parametric particle effects - then you only need to store start/end markers in your replay and you're cheaply and easily replay particle
- Plan for the memory overhead and reserve that space
- Plan for how you spawn / unspawn entities to make it easier
- Plan for how you stream assets to make it easier
- Plan for how you submit your render data to make it easier to record or replay the data.



Replaying Particles

- If you don’t have a parametric system…
- Fake it by:
  - Storing start and end markers
  - Playing effect forwards by abs(deltaTime)
- http://youtu.be/bzwdPoKP80k

- If you don’t have a parametric particle system then you can still try and fake it
- Store markers to say when an effect started and ended
- Then just always play the effect forwards (by the absolute value of the time delta) when you play forward and rewind between those markers
- I’ll show you an example video of this in practice…
- http://youtu.be/bzwdPoKP80k



Any Questions?

Slides and contact information available at:

www.markwesley.com

So… any questions?

You can also contact me via the website URL shown.

Thanks everyone!