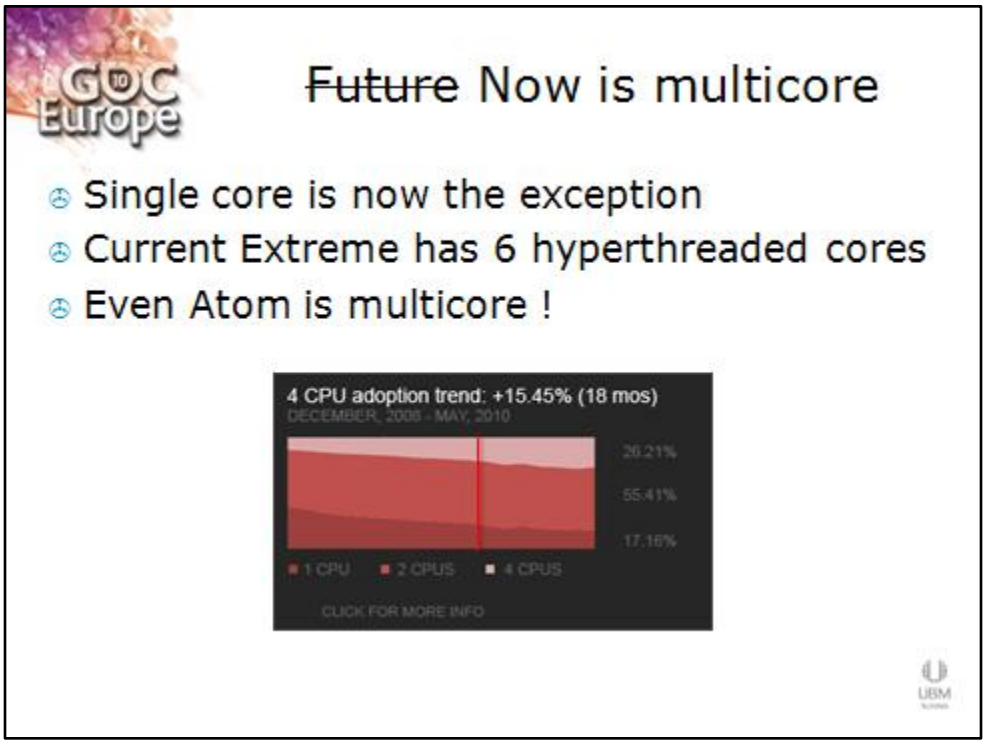
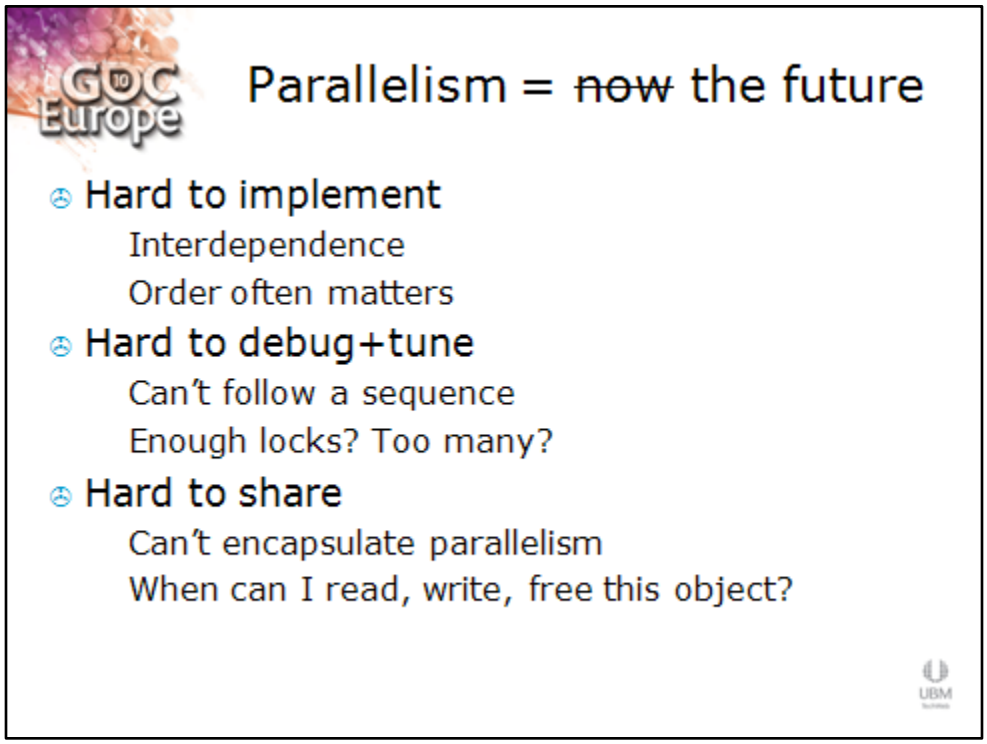

Supported by
gamescom
BIU
UBM
UFO invasion
DX11 & multicore
to the rescue
Jérôme Muffat-Méridol
senior application engineer
Intel corporation
Game Developers Conference™ Europe
August 16-18, 2010
Cologne Congress Center East | Cologne, Germany
Visit www.GDCEurope.com for more information
GDC 10
Europe



Today, we're going to talk about writing a game engine for multicore processors, and to start with some perspective of what's going on in the real world, I'll take just the one stat that I hope we can all easily agree on: the Steam Hardware Survey.

The trend is a very steady migration to more cores, with 4C and 1C exchanging 2nd and 3rd place somewhere around end of last year.

That's a lot of CPU power people have, let's tap into that, plenty of good ideas to have !



Parallelism is difficult, but maybe even more so for games.

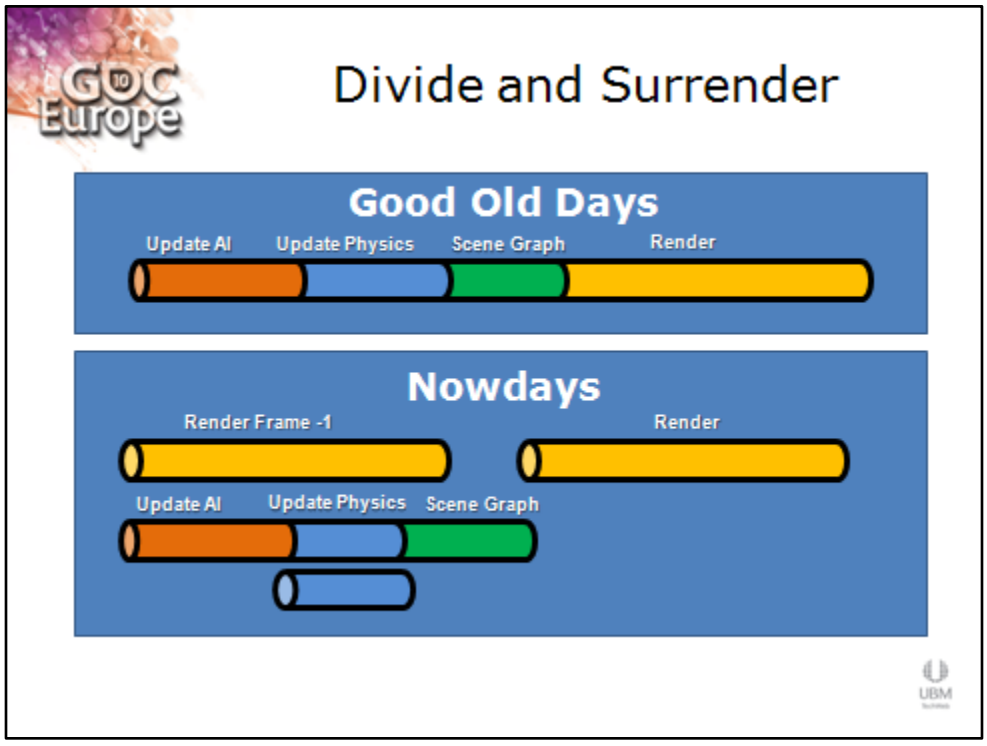
In a game everything is interconnected, or rather is susceptible to be interconnected. I've had my hardest time with a bugs involving the player trying to jump while inside an elevator, and it was a single threaded engine back then… I'm sure everybody here has some horror story like that…

In a game, order often matters and one place where it really does is when drawing: do that in the wrong order and best case you'll lose performance, worst case you'll get the wrong picture…

Debugging is harder, do I really need to go there ?

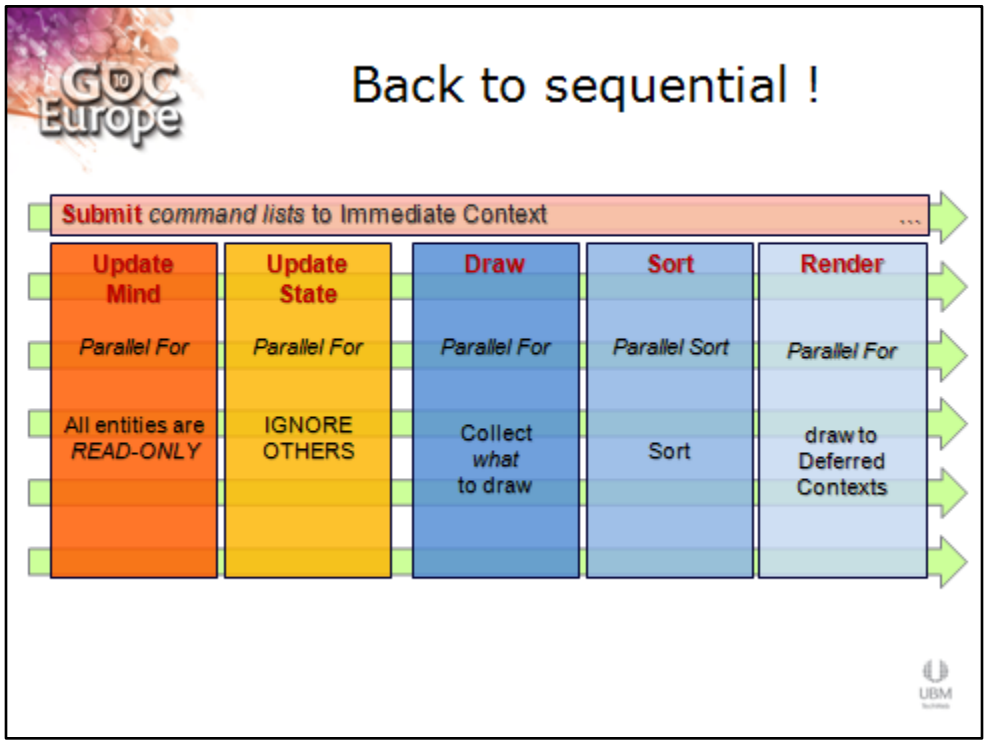
But fundamentally, the main issue I see is that more or less everybody in the team needs to be parallelism aware. You don't want people to start adding mutexes all over the place, you don't want data to change and pull the rug from below your feet, the most important thing to tackle when writing a parallel engine is keeping the concept simple enough that the team is confident they know what they can do (and what they can't)

The architecture I'm presenting today aims at that: use simple rules, be simple to explain.
As it turns out, these rules ease implementation and provide better context while debugging.



In the good old days, things would happen sequentially, one thing at a time, and a lot of a game's complexity could be dismissed by simply reordering things (like, if you update Player last, you've got your hands free to change anything to make the player's (unpredictable) action work out all right.

Nowdays, for all the reasons mentionned previously, we divide the work where possible with a mic of functionnal decomposition and breaking down into sub tasks. Often, you start in a Divide and Conquer mood and end up with many more mutually exclusive things than we'd want. Things end up scaling only to a given number of cores…

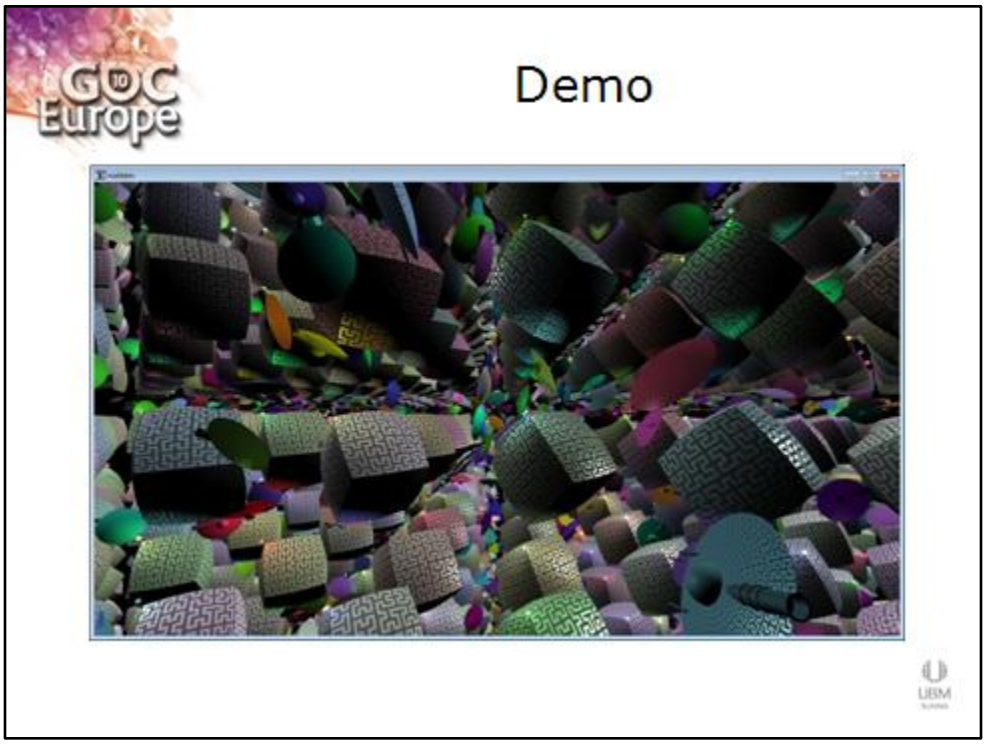


I want you to have a general idea of where we're going today, in an effort to make it easier to follow me.
So, like in an episode of Columbo, you get to see what really happened and then we'll discover it again, step by step.

This is the architecture I am going to describe in this talk
- it is designed to scale with the number of cores while adding little high-level constraints.
- The general philosophy is to split the work in phases that each can run in parallel
- This follows the traditional "update and draw" model, ie lock-step

- Rules are simple enough that they can be understood and enforced by any coder on the team (including scripts)

-But let's rewind and look at how this all came about



Show demo, explain that everything is being updated and drawn in parallel, with each entity potentially on a different thread.

We have 4096 cubes and as many lights.
The engine uses basic deferred shading, with a first pass storing normals and colors per pixel, lighting being applied in a second pass, I initially wasn't sure DX11 multithreaded rendering would cope with that sort of thing.

The lights are attached to a cube only temporarily and will change owner from time to time, these are all independent entities and can all be updated/rendered from any thread.
Let's add some UFOs !

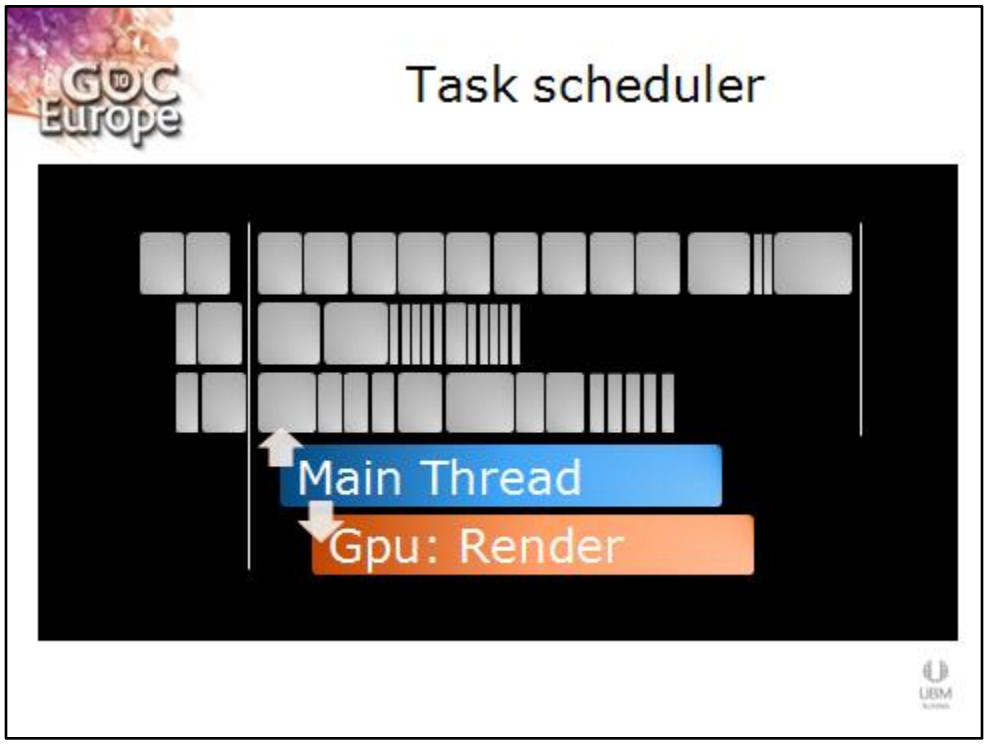
Cool ☺

A bit of background:

This all started two years ago, here in Cologne, at a demoparty
I was presenting on Intel's Threading Building Blocks and someone came up with that question…
This led to a sample that has the nice property of being "as simple as possible but not simpler", this makes it easy to dive into the code, it also forces focus on the essential
http://www.gamasutra.com/view/feature/4287/sponsored_feature_doityourself_.php

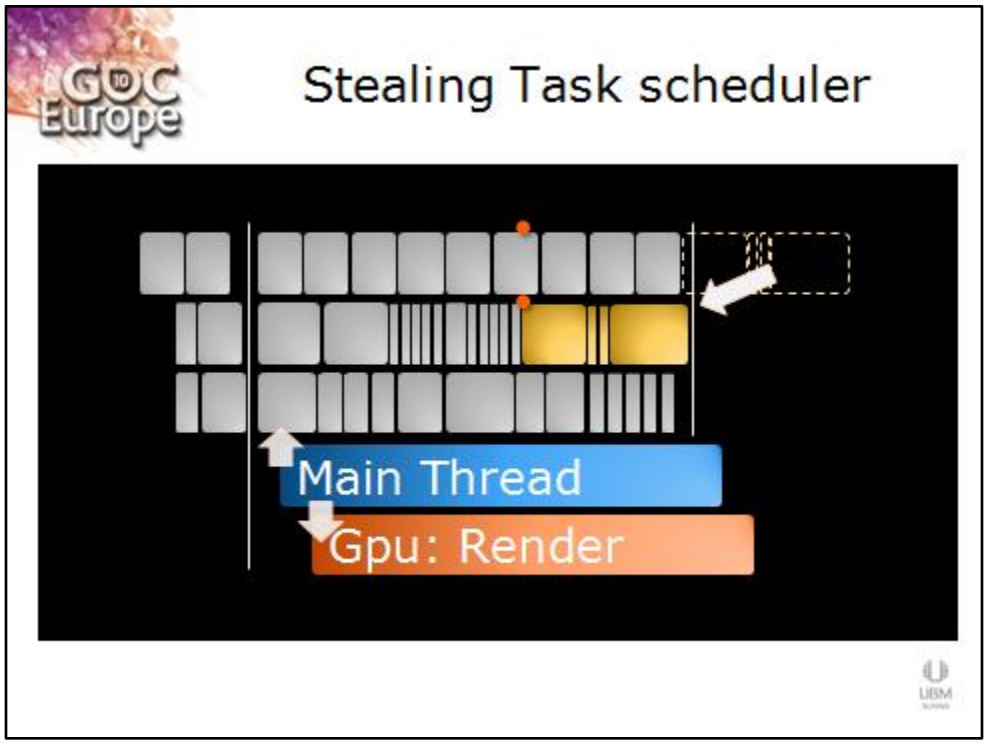
This first version implemented the core concepts of the parallel game loop and demonstrated good speedup by parallelising most of the game loop.
Submission of draw calls was the stumbling block, as it needs to be done in order…

Version 2 solves this problem, but before we get there, let's look at how all the rest manages to run in parallel.



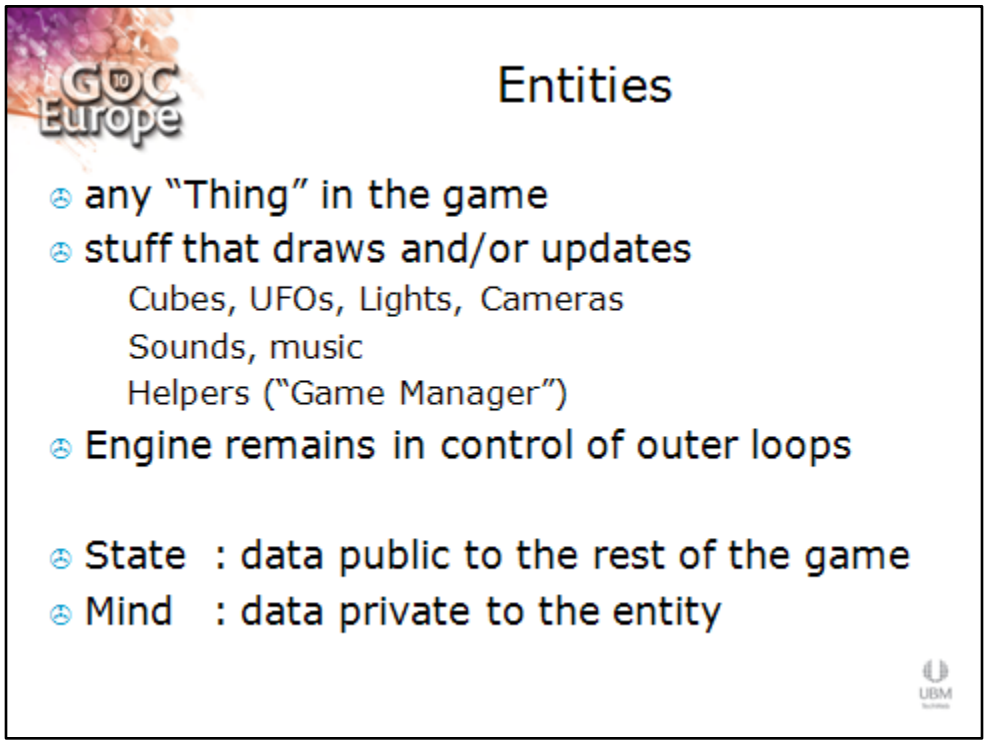
SHOW OF HANDS: How many people are familiar with task schedulers ?

The concept is reasonably simple:
-Here, you can imagine a quad core system
-Application is written from the point of view of one main thread
-Main thread submits work to GPU
-In a similar fashion, it submits work to the task scheduler
-A task is a small, *independent* chunk of work, typically a subset of a loop
-Scheduler has one queue per worker thread and splits work somewhat arbitrarily (here, 13 tasks per worker)
-There is no way to know in advance how long a task will take to execute…



When Thread2 comes to starve, it attempts to steal work from other threads, here from Thread1
-Overhead of stealing taken by thread that would otherwise become idle == cheap
-Synchronisation only needs to happen between the two threads == low contention
-Tasks can spawn more tasks with very low contention (adding to their own queue)
-When main thread waits for workers to finish, it becomes a worker itself

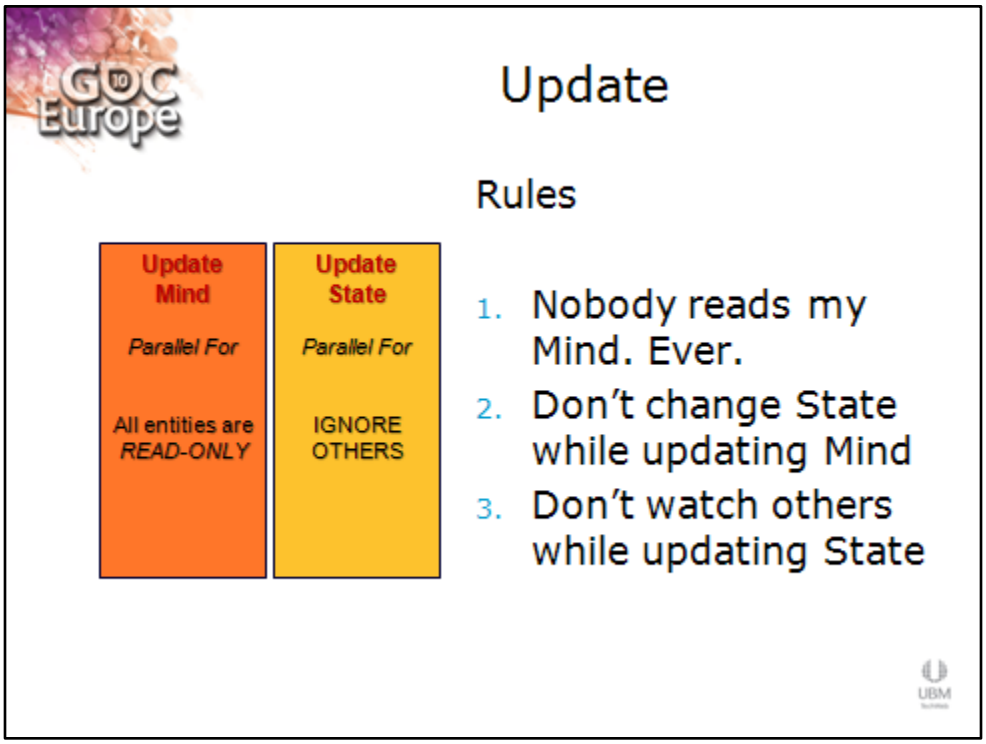
-Effectively load balances the system automatically (given task granularity is small)
-Makes it possible to think in terms of phases as I presented on the first slide



Game manager = the entity responsible for orchestrating the world.

The point of having everything as entities and of having nothing else is that the engine can be in control of the outer loops, pretty much the way you let the GPU walk vertices and pixels the way it wants.

An entity in nulstein has two parts.
-Its state is the usual amount of data necessary to represent it.
-Its mind is data that is never shared with any other entity and, thus, does not require locking of any kind

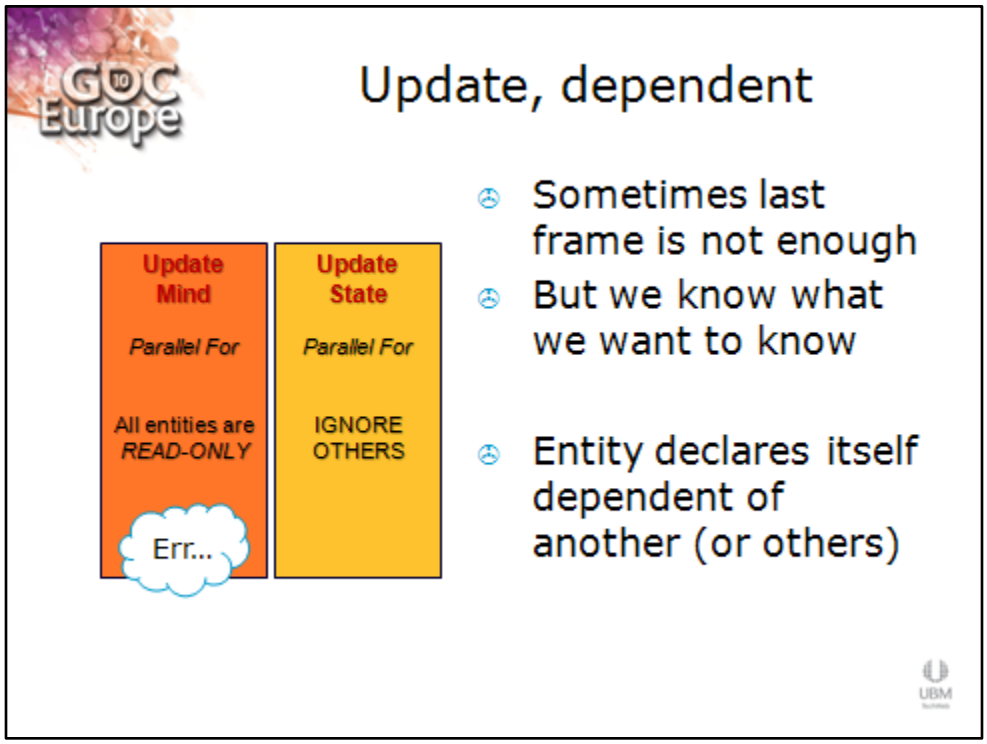


These rules work like in a wargame, the "turn" is split in two phases, one for observation, one for action/resolution

Every entity can update their mind in parallel: they're only reading data
Every entity can update their state in parallel: they're not interacting

No deadlocks : there are no locks
No race conditions : there are NO locks

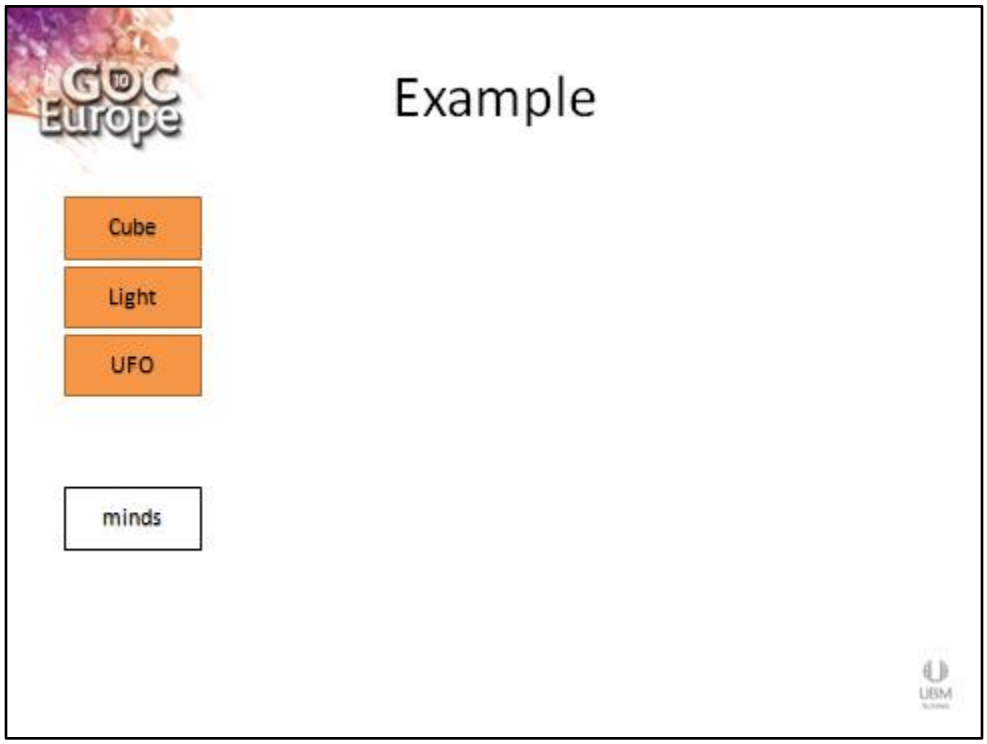
This is called ***implicit*** synchronisation



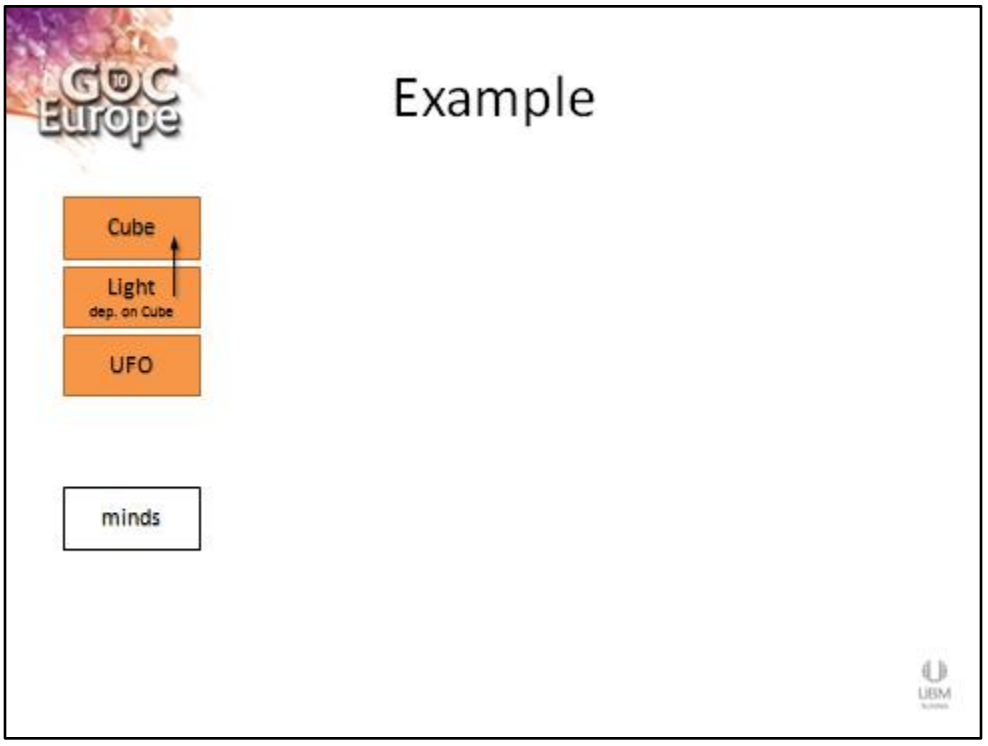
"camera flies in and attaches to a car" example

Every frame that the camera is attached:
- During Update Mind, it tells the engine that it wants to Update State after the car.
- When car has finished updating its state, it will fire all its dependent tasks (UpdateState's for our camera, the driver, headlights, you name it)
- When these update, they can safely read the car's state: it is done updating

Unless used pathologically (ie every entity dependent on the previous), this remains parallel thanks to stealing task scheduler

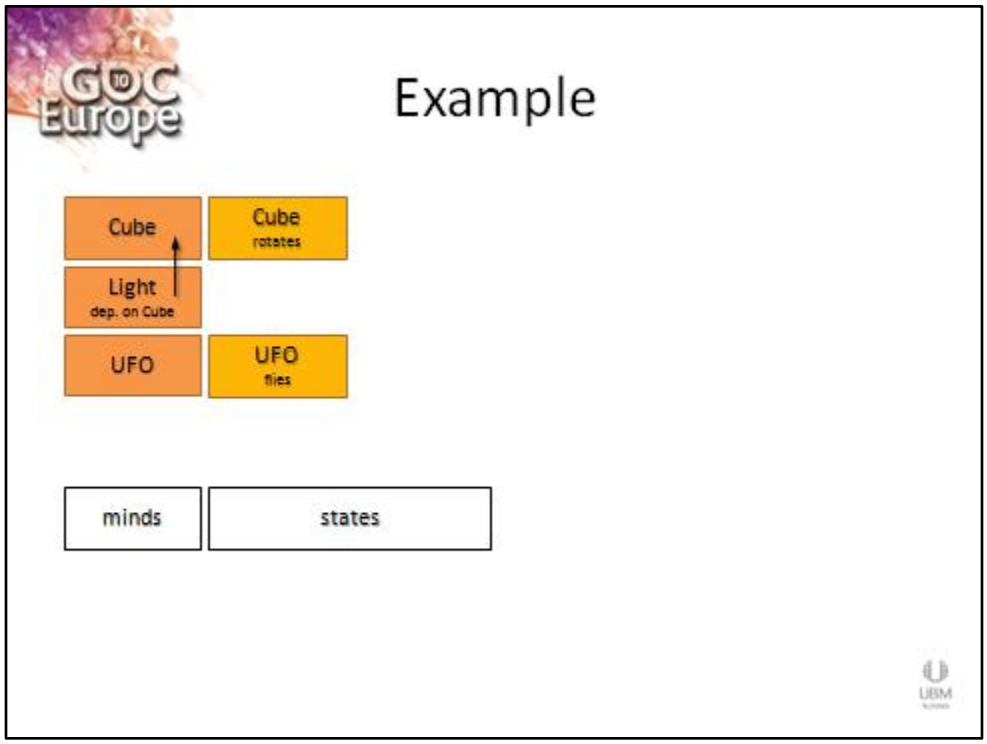


Starts like that…

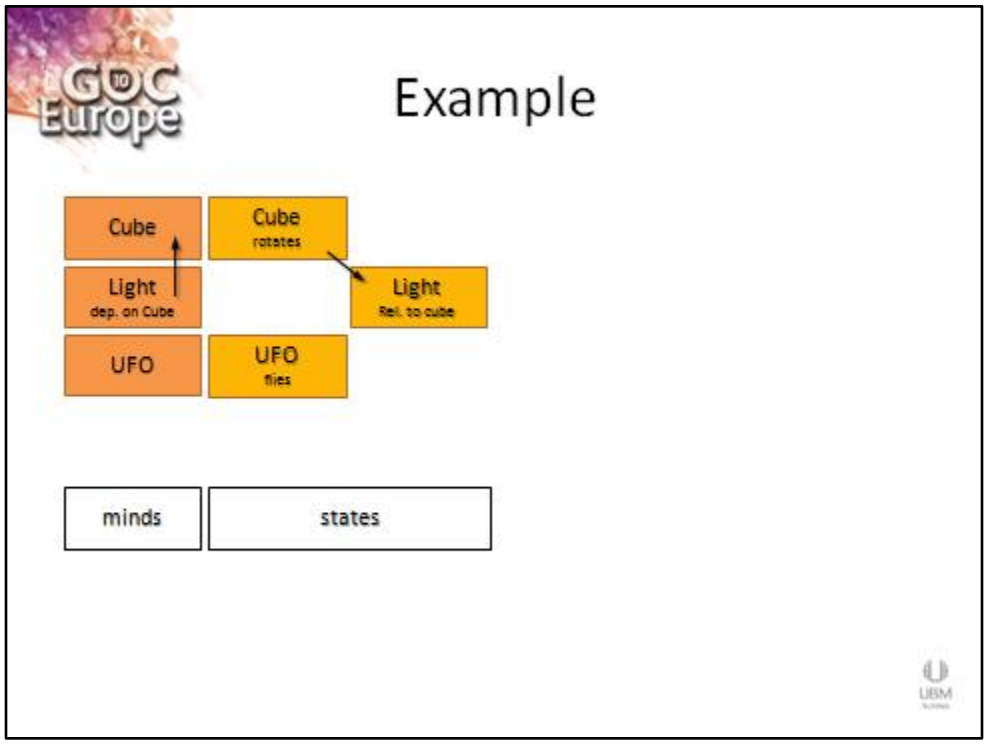


All entities collect information.

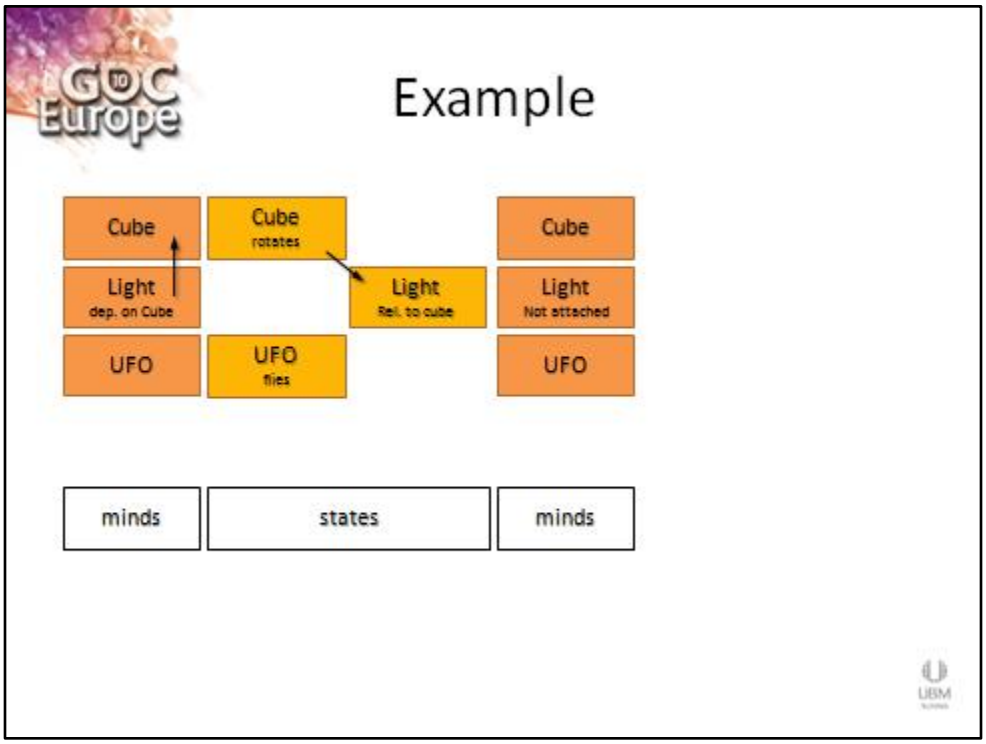
Light decides it remains attached to cube and tells the engine “I need the cube to update before me”



Non dependent entities update.

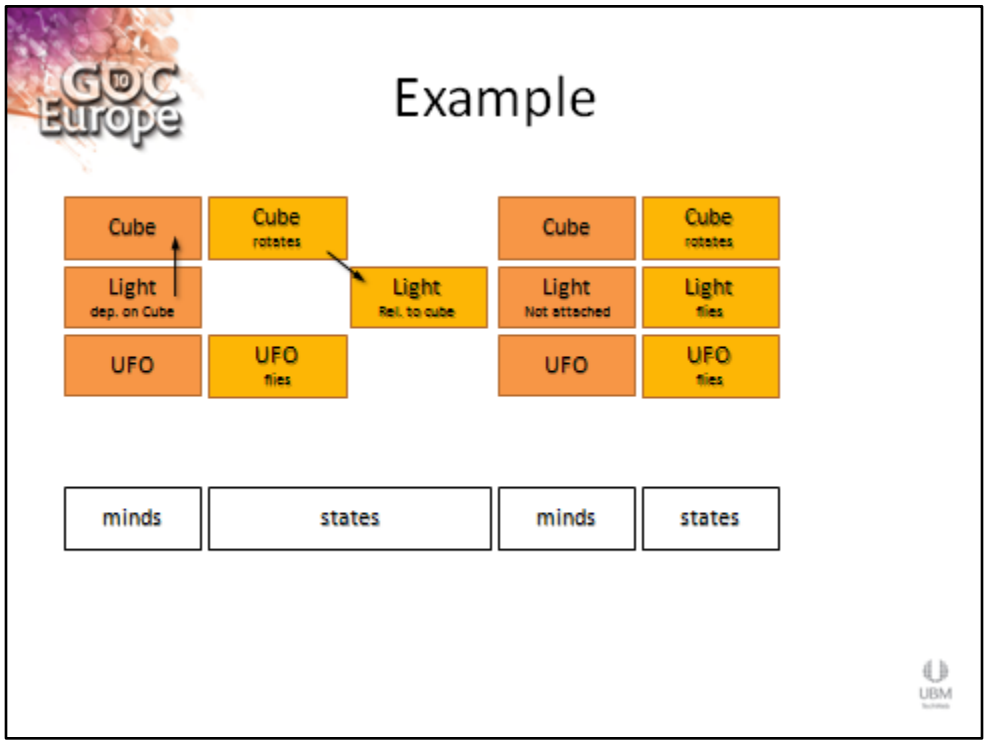


Cube's end of state update schedules Light's state update

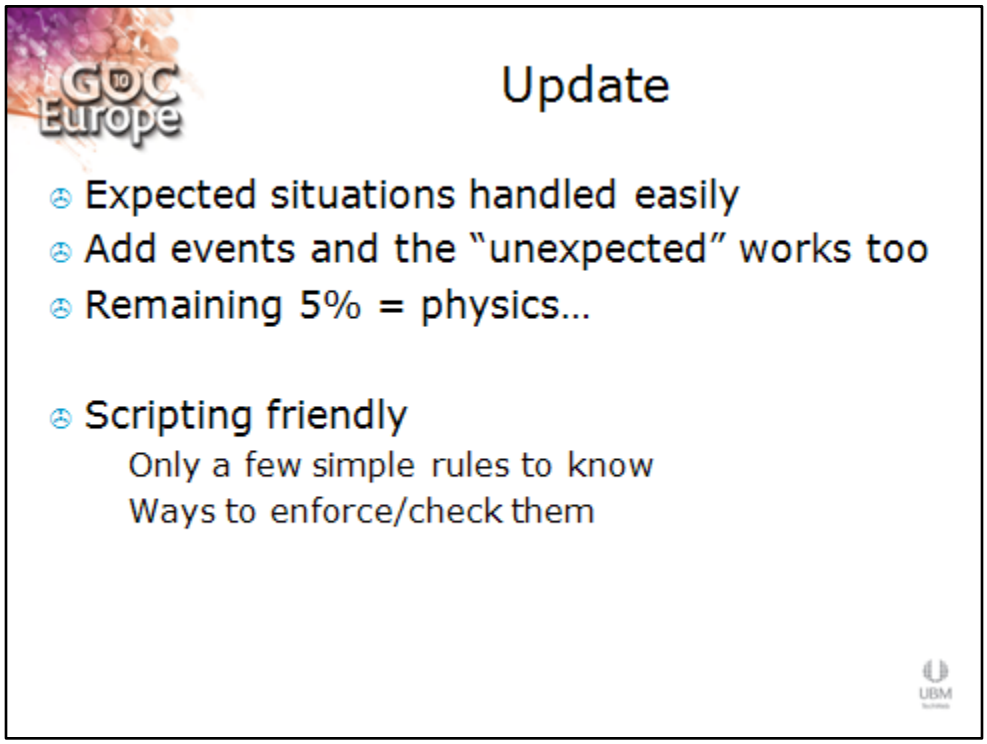


Here we go again.

Ok, this time light decides to change parent…



All entities independent, easy update.



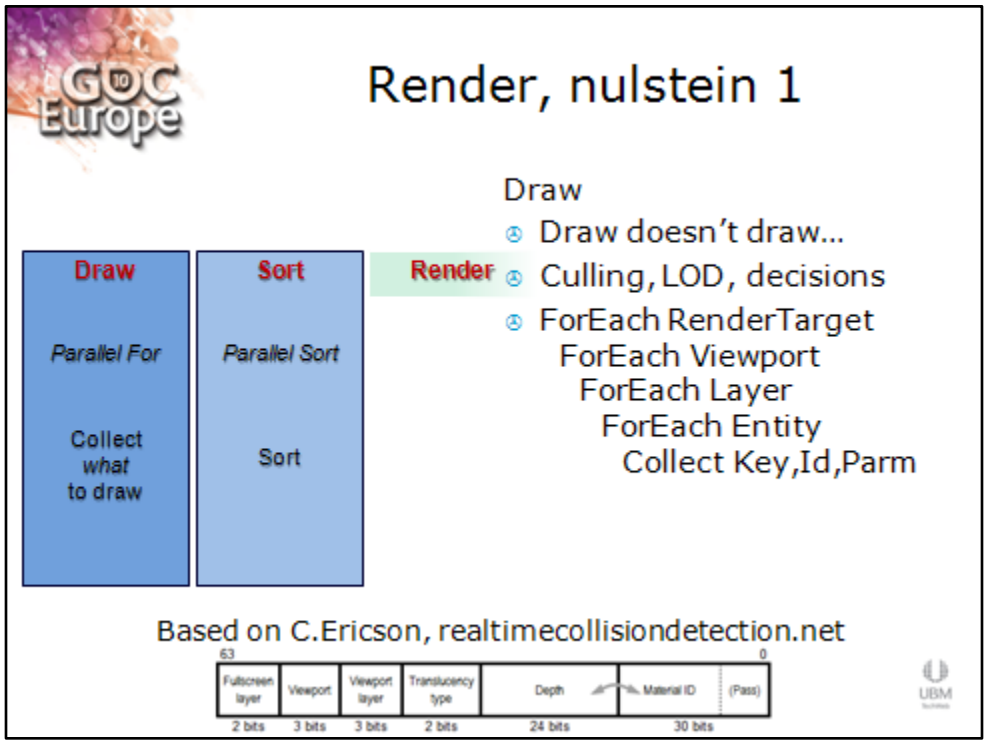
Time check : half-way through

Expected situations are situations an entity can efficiently check for while updating its mind
Unexpected situations are ones like being hit by a bullet. Bullet entity needs to send an event at impact frame, and next frame, the victim entity's mind will realize it's dead.

Physics: cue ball can hit a corner and bounce back on a nearby ball, and as there was no way for it to predict that during Mind Update, it can only be told on next frame, which will break the simulation. I don't think Physics can be accurately processed this way, need more work…

Scripting: high level coders and scripters don't want to have to deal with locks and, really, given you want to minimize locks, nobody wants that anyway. With the system presented here, they don't need locks at all, they just need to understand the difference between Mind and State, and it's straightforward.

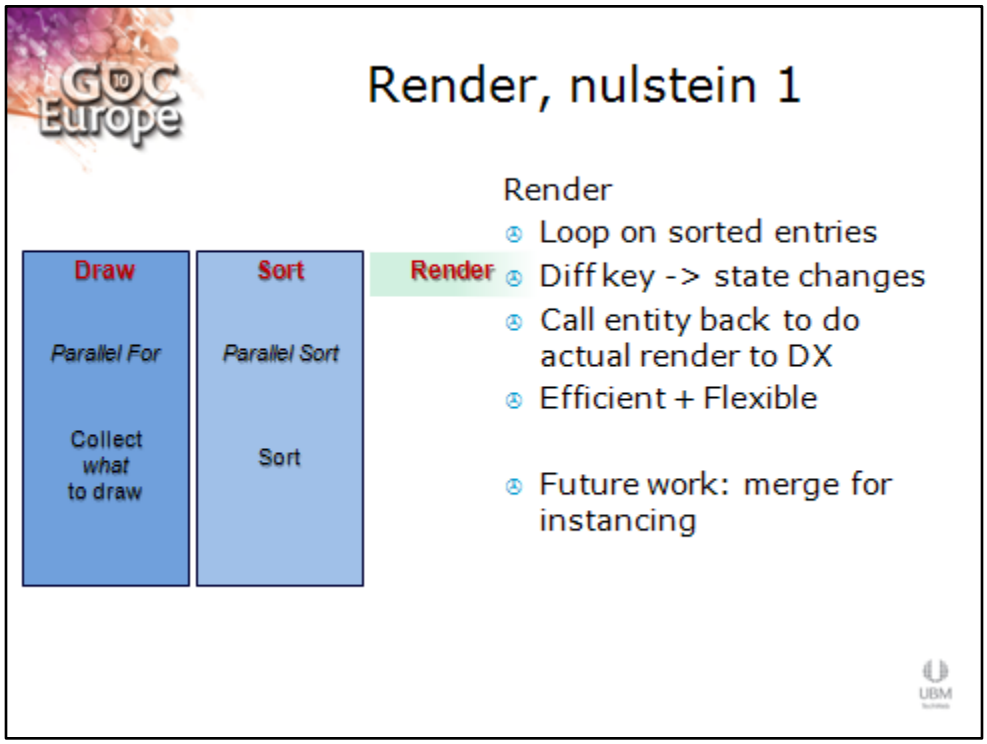
Also, it is possible to add logic in the engine that CRC's the areas that should not be touched and pop an error if it has.



Draw is fully parallel as each thread collects data in its own queue, sort will put everything together later.

In nulstein, each entry weights 128bits (64 bits key, 32bits Entity ID, 32bits Parameter) For 4096 entries, that's 64K of data to sort == barely enough to justify a parallel sort…

A lot of the expensive parts of rendering happen here, in parallel, most notably culling.

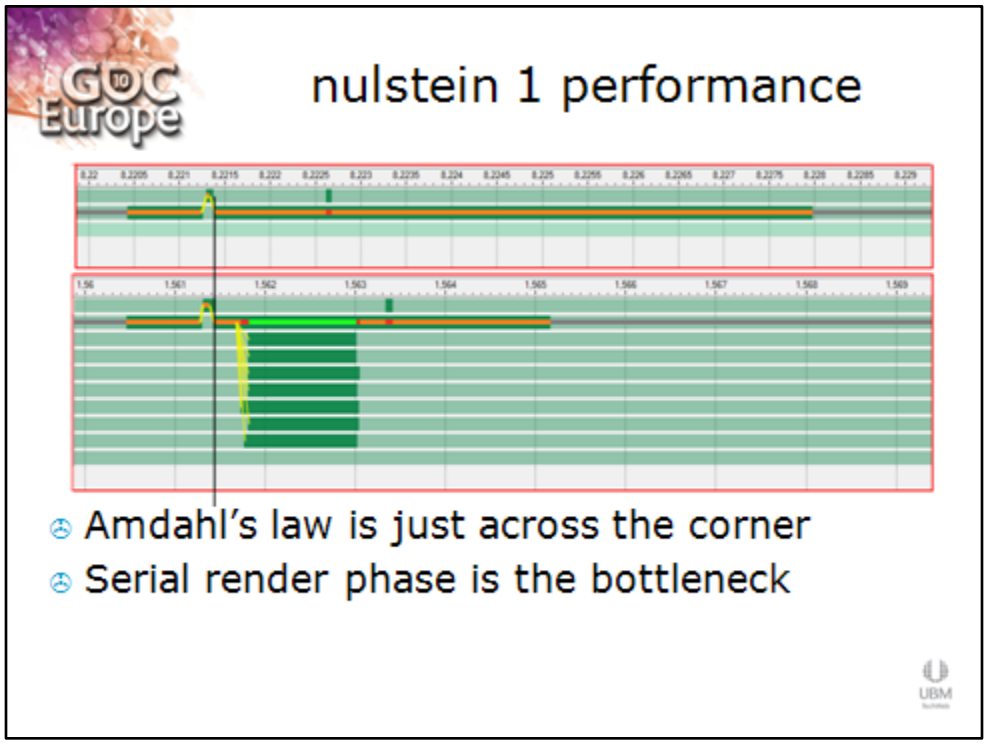


Render happens on the **one** thread, serially, by nature.

Looking at the difference between the current key and the previous one, the engine can very efficiently update the pipeline's state.

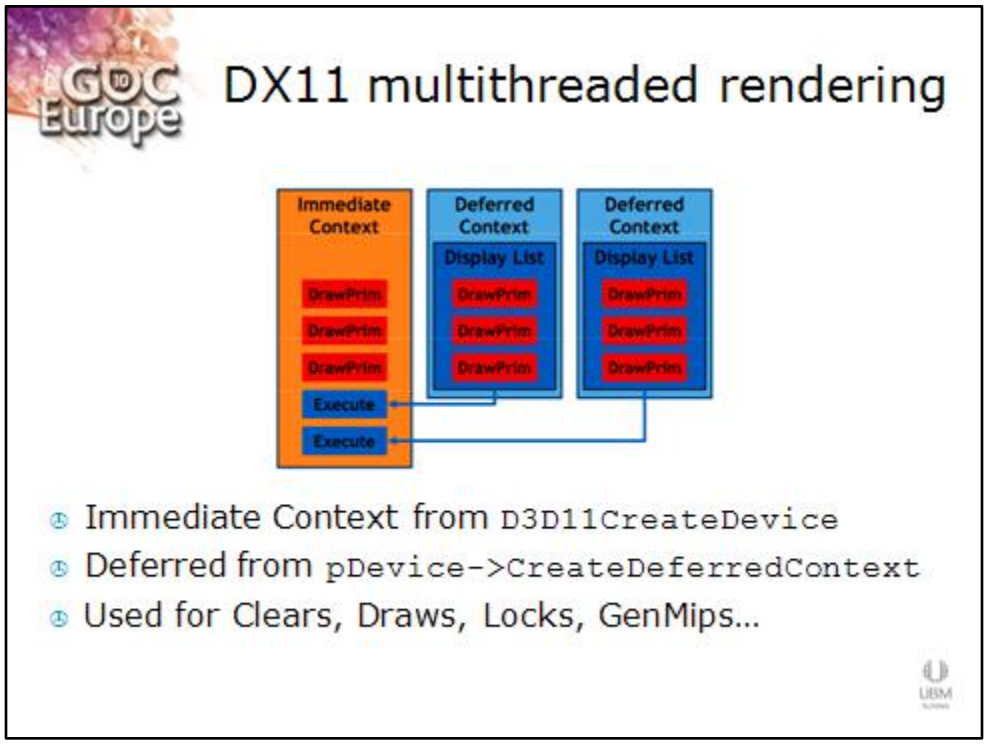
We start with an out of bounds "Last Key", which makes the engine do select the correct render target, viewport and various states.
Entity is called back to draw whatever it needs to draw.
Current key becomes last key
Repeat until done

Instancing would really only be a matter of accumulating keys with same instancing id and calling back the entity with the resulting list.



This is nulstein 1 running on one thread, on top, and on eight below.
The remaining serial part is the bit where we loop on keys, actually submitting draw calls to DX.
Everything else is parallel…

If we were to add more threads, we'd just see diminishing returns. Rendering wants to be done in parallel too !



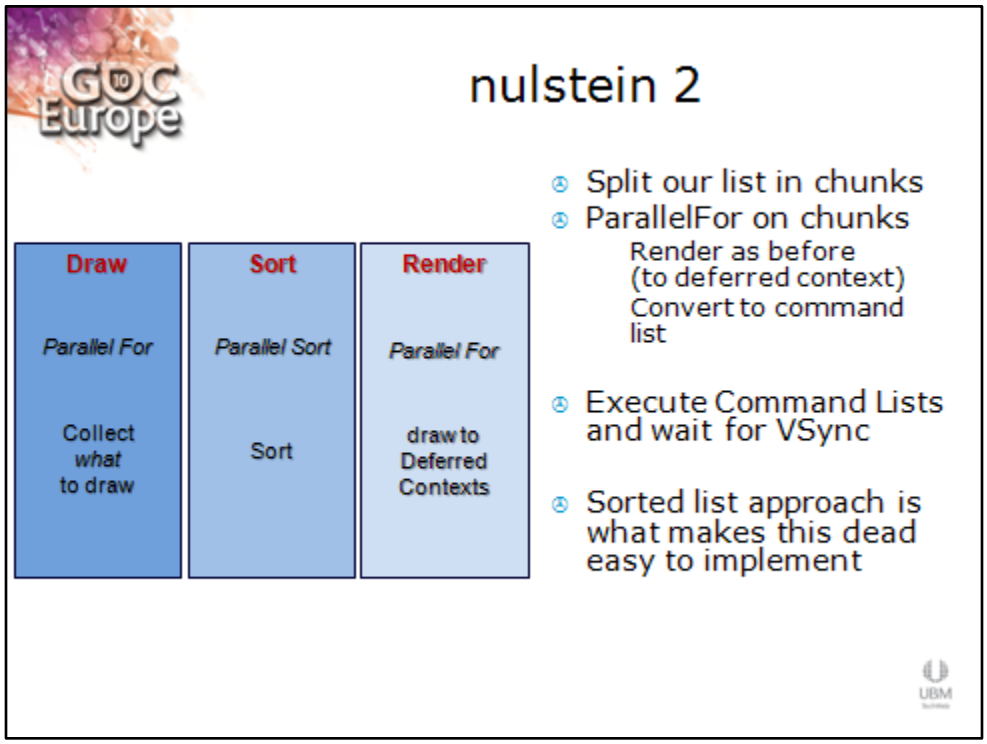
Very straightforward
- Multiple threads render to Deferred Contexts
- Deferred Contexts generate Command Lists
- Main thread submits them to Immediate Ctxt

DX11 multithreaded rendering

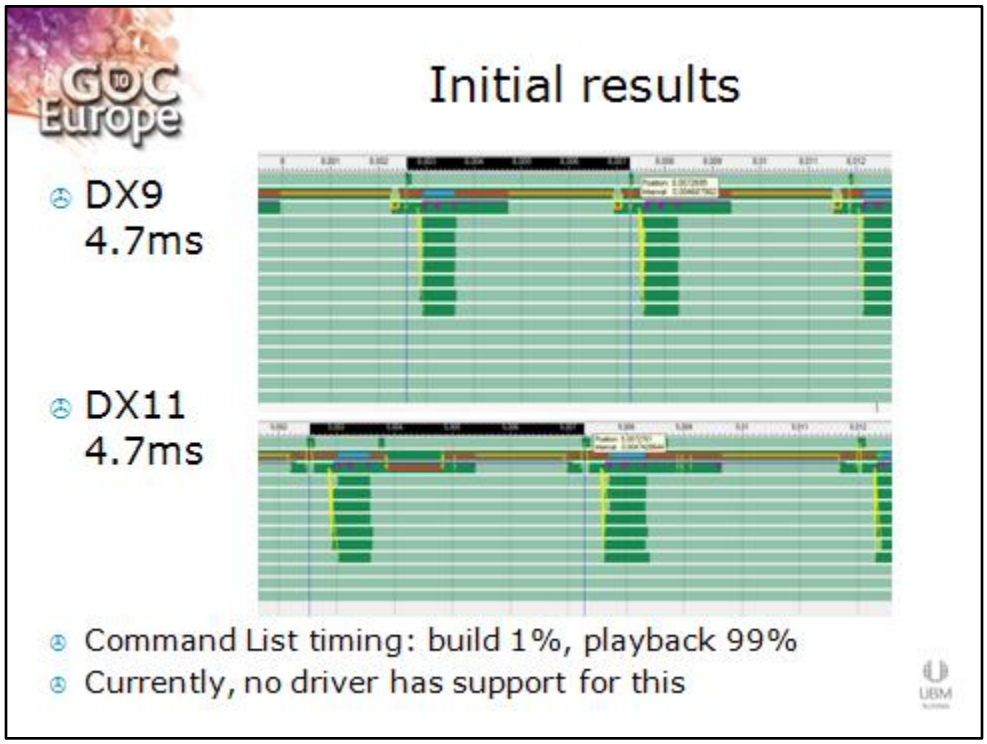
- Deferred context converts to command list
- Execute command lists in ImmediateContext

- Uses only UpdateSubResource and Lock(DISCARD)
- Engine doesn't have to manage resources

- Pipeline state not modified by CommandList
- Each CommandList must be self sufficient



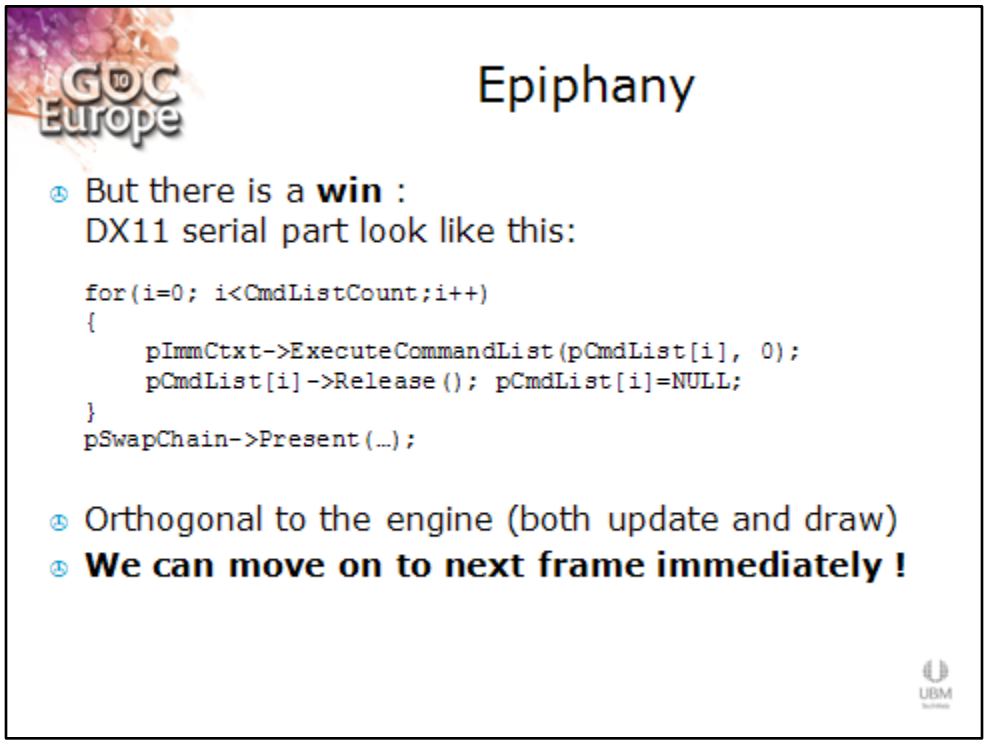
The neatest thing, this approach also deals with the requirement that command list don't rely on the current pipeline's state:
We always start with an out of range "last key": each command lists starts with the full pipeline state.



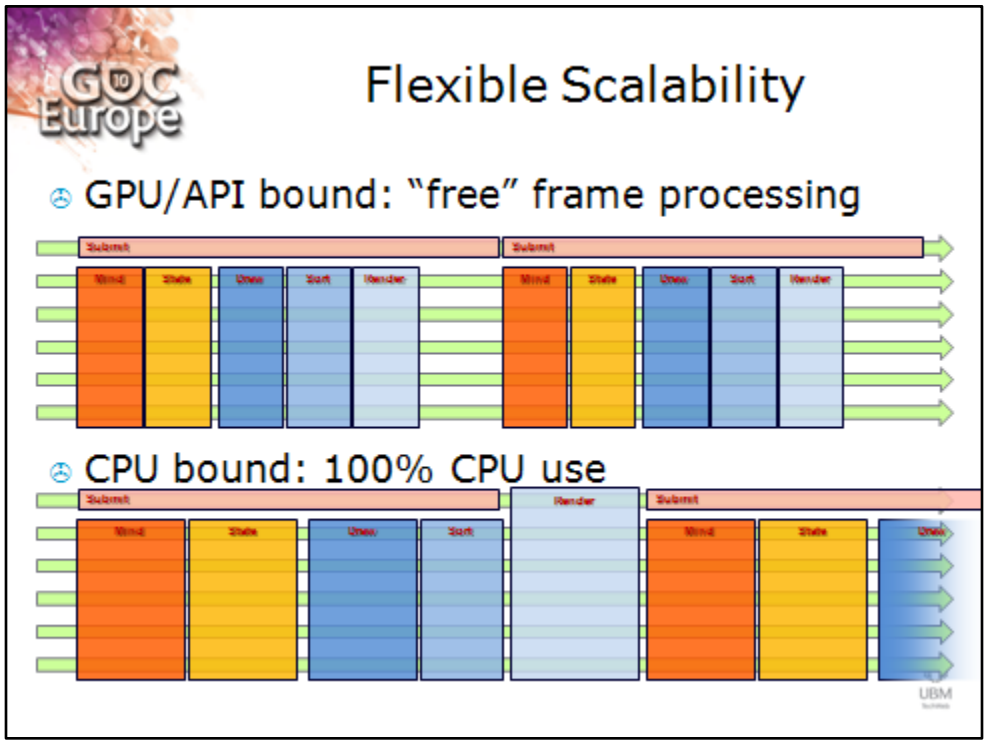
Initial results have been disappointing because the part we execute in parallel happens really really fast, and the part we run serially takes the same time as if we were submitting draw calls like before…

DX11 deferred contexts can take advantage of "Driver Command Lists", but no driver currently supports this. If and when they do, these graphs may change.

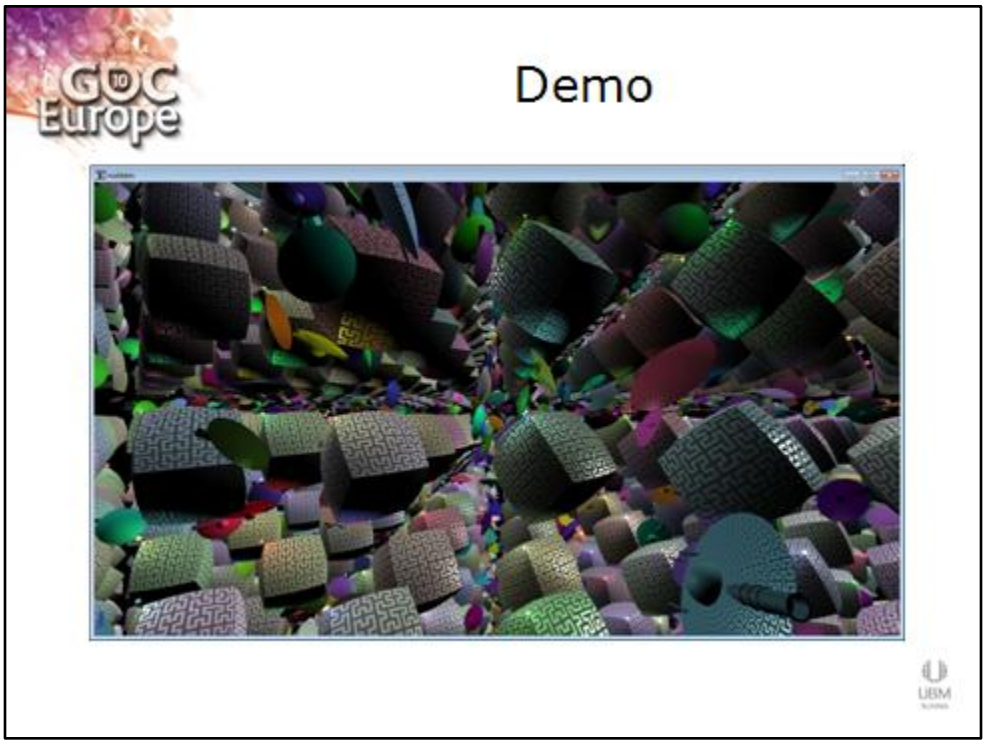
But…



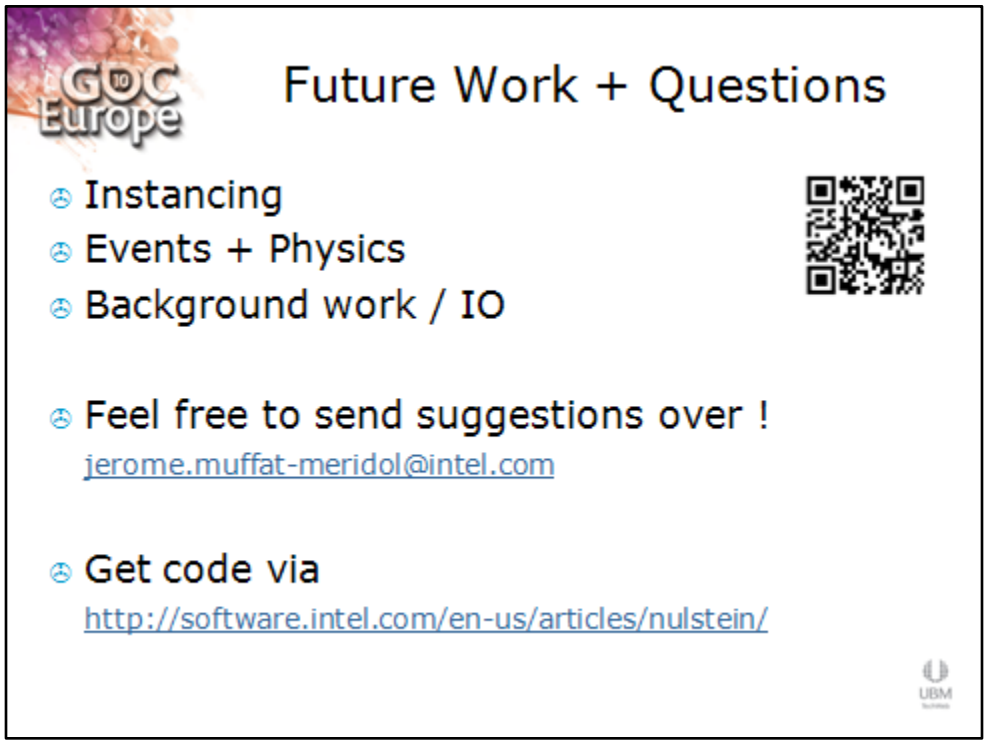
We can modify anything we want while Command Lists are being submitted : they don't reference any of it !



When we're bound by graphics (GPU bound or stuck in DX/driver), next frame is prepared while we're bottlenecked and the frame time only depends on rendering time.
When we're CPU bound, on the other hand, we're always making use of all available resources.



Run demo again, varying amounts of GPU work and CPU work, explaining what happens with respect to the previous graph.


GDC Europe
Future Work + Questions
Instancing
Events + Physics
Background work / IO
Feel free to send suggestions over !
jerome.muffat-meridol@intel.com
Get code via
http://software.intel.com/en-us/articles/nulstein/
UBM