Stockholm Game Developer Forum 2010

Parallel Futures of a Game Engine (v2.0)

Johan Andersson
Rendering Architect, DICE

Background

- DICE
  - Stockholm, Sweden
  - ~250 employees
  - Part of *Electronic Arts*
  - *Battlefield* & *Mirror's Edge* game series
- Frostbite
  - Proprietary game engine used at DICE & EA
  - Developed by DICE over the last 5 years

DICE

FROSTBITE
A DICE TECHNOLOGY

http://badcompany2.ea.com/

http://badcompany2.ea.com/

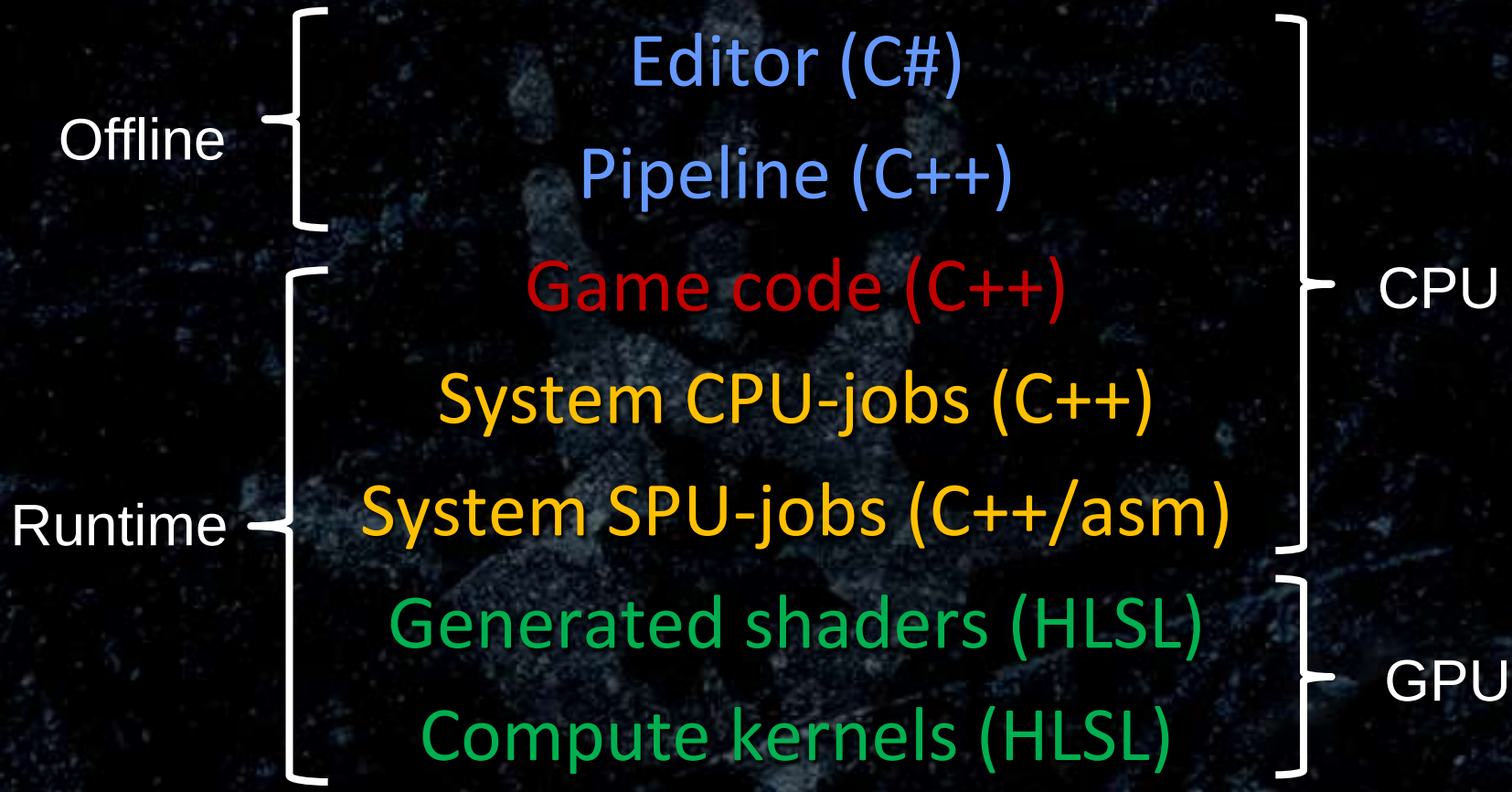
Outline

- Current parallelism
- Futures
- Q&A

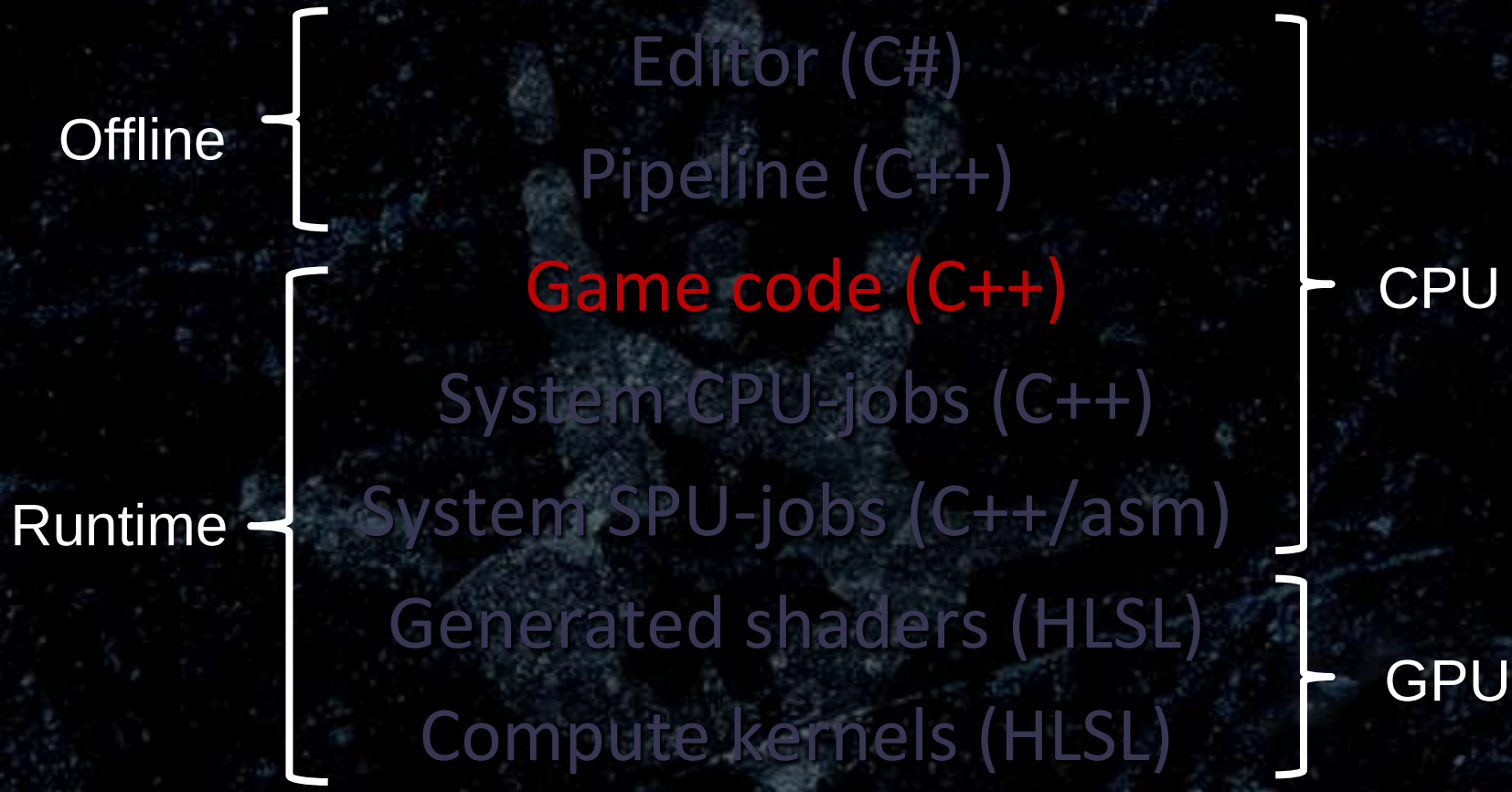
- Slides will be available online later today
  - http://publications.dice.se and http://repi.se

Current parallelism

Levels of code in Frostbite



Levels of code in Frostbite



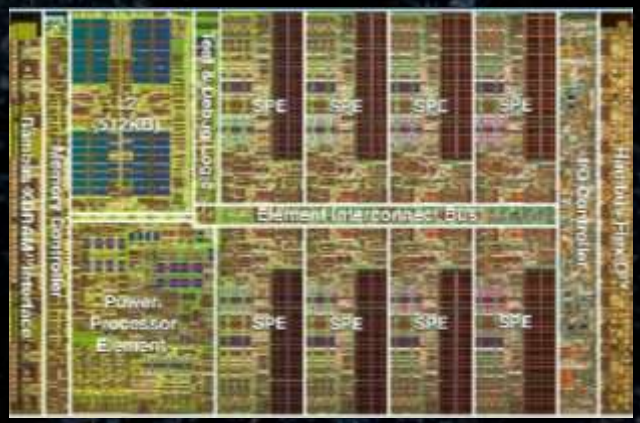
General ”game code” (1/2)

- This is the majority of our 1.5 million lines of C++
  - Runs on Win32, Win64, Xbox 360 and PS3
  - We do not use any scripting language
- Similar to general application code
  - Huge amount of code & logic to maintain + continue to develop
  - Low compute density
    - ”Glue code”
    - Scattered in memory (pointer chasing)
    - Difficult to efficiently parallelize
  - Out-of-order execution is a big help, but consoles are in-order ☹
- Key to be able to quickly iterate & change
  - This is the actual game logic & glue that builds the game
  - C++ not ideal, but has the invested infrastructure

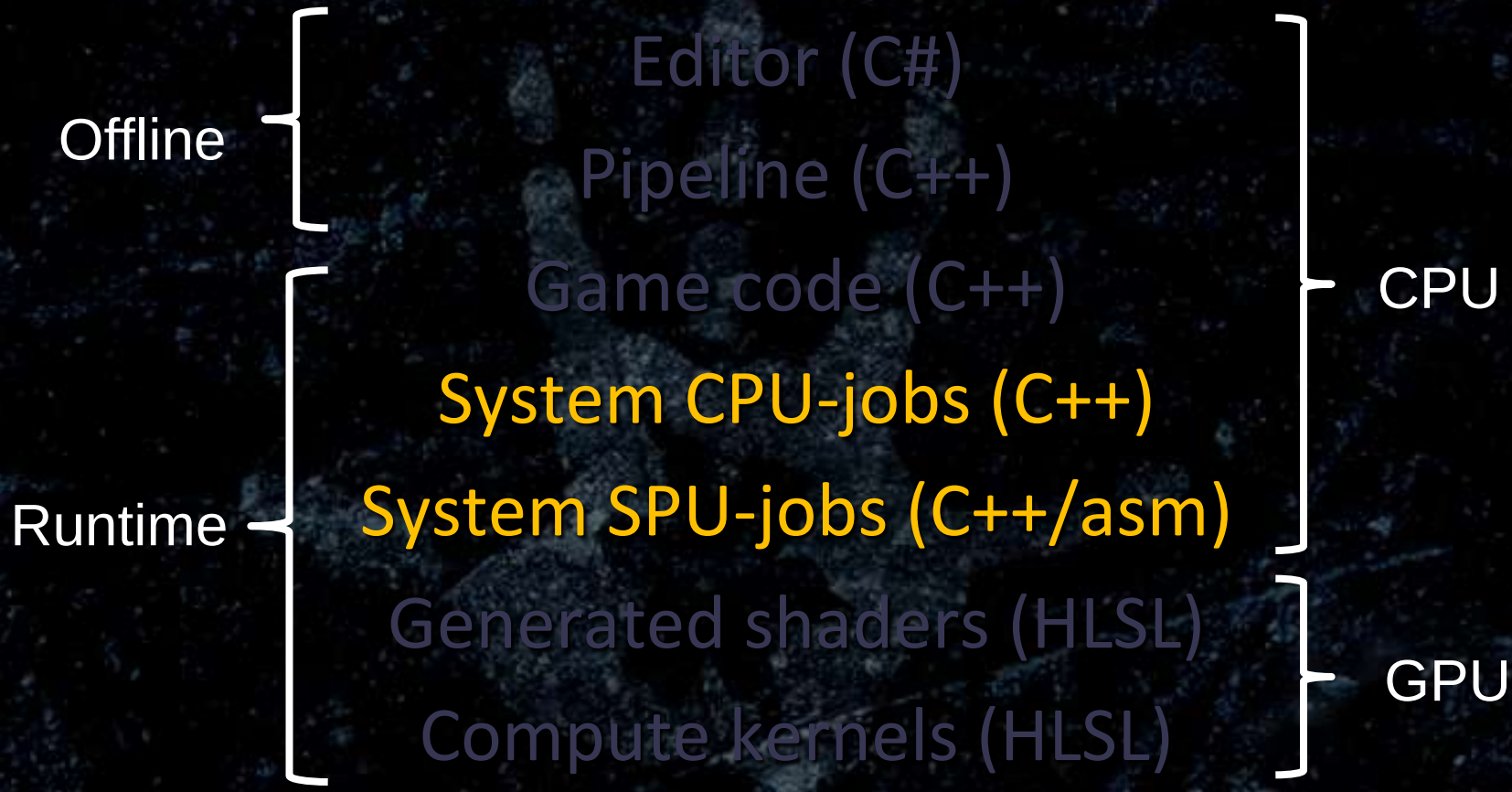
General ”game code” (2/2)

- PS3 is one of the main challenges
  - Standard CPU parallelization doesn’t help (much)
  - CELL only has 2 HW threads on the PPU
- Split the code in 2: game code & system code
  - Game logic, policy and glue code only on CPU
  - *”If it runs well on the PS3 PPU, it runs well everywhere”*
  - Lower-level systems on *PS3 SPUs*
- Main goals going forward:
  - Simplify & structure code base
  - Reduce coupling with lower-level systems
  - Increase in task parallelism for PC

CELL processor



Levels of code in Frostbite



Job-based parallelism

- *Essential* to utilize the cores on our target platforms
  - Xbox 360: 6 HW threads
  - PlayStation 3: 2 HW threads + 6 powerful SPUs
  - PC: 2-16 HW threads
- Divide up system work into *Jobs* (a.k.a. *Tasks*)
  - *15-200k* C++ code each. *25k* is common
  - Can depend on each other (if needed)
  - Dependencies create job graph
- All HW threads consume jobs
  - ~200-300 / frame

What is a Job for us?

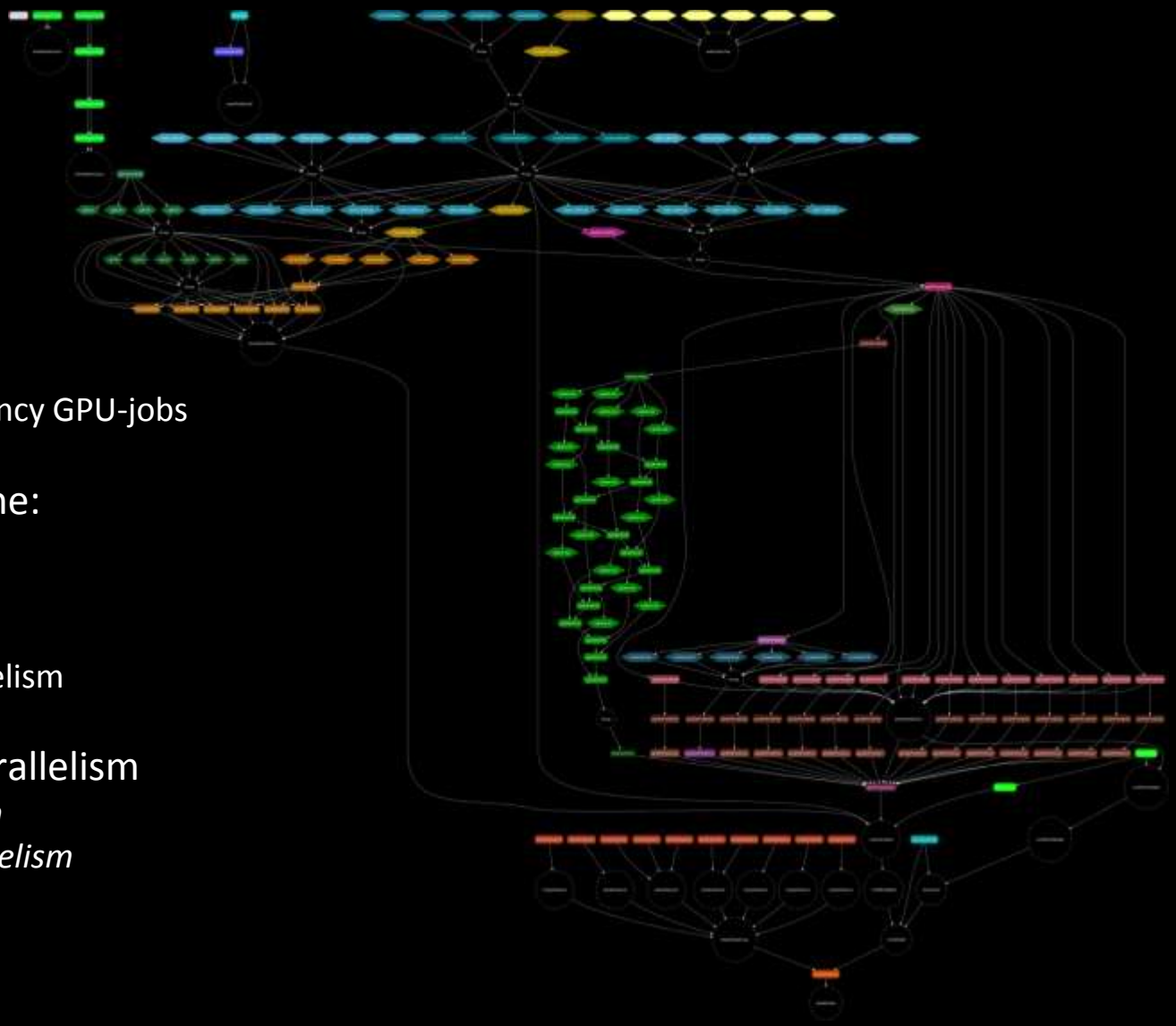
An asynchronous function call

- Function ptr + 4 *uintptr_t* parameters
- Cross-platform scheduler: *EA JobManager*
- Uses *work stealing*

2 types of Jobs in Frostbite:

- *CPU job* (good)
  - General code moved into job instead of threads
- *SPU job* (great!)
  - Stateless pure functions, no side effects
  - Data-oriented, explicit memory DMA to local store
  - Designed to run on the PS3 SPUs = also very fast on in-order CPU
  - Can hot-swap → quick iterations ☺

Frostbite CPU job graph



Build big job graphs:

- Batch, batch, batch
- Mix CPU- & SPU-jobs
- Future: Mix in low-latency GPU-jobs

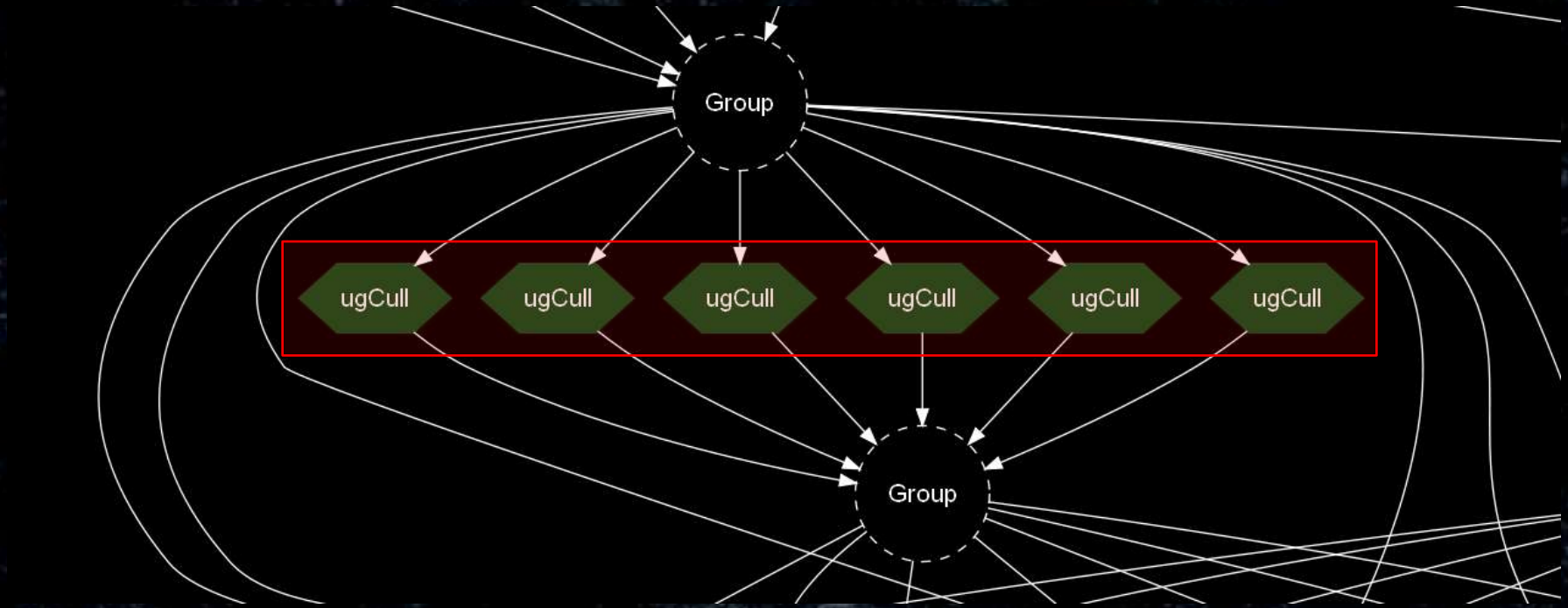
Job dependencies determine:

- Execution order
- Sync points
- Load balancing
- i.e. the *effective* parallelism

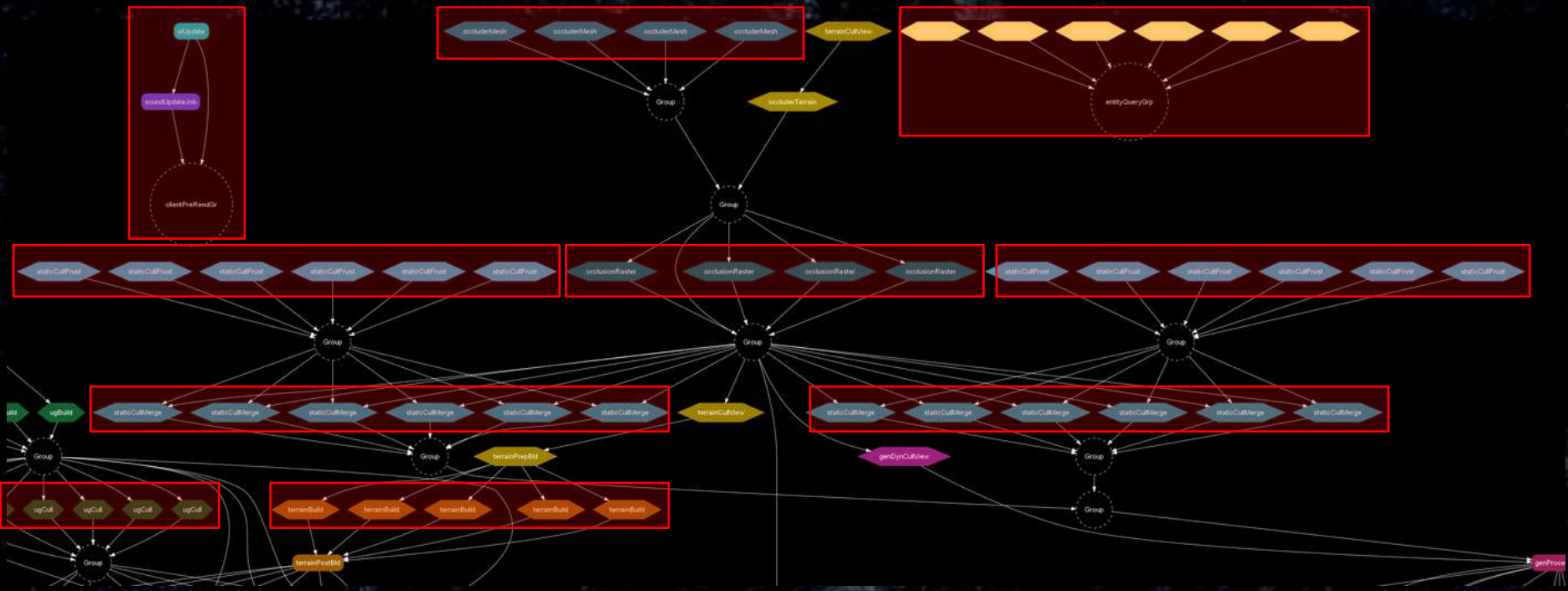
Intermixed task- & data-parallelism

- aka *Braided Parallelism*
- aka *Nested Data-Parallelism*
- aka *Tasks and Kernels*

Data-parallel jobs



Task-parallel algorithms & coordination



Timing View: Battlefield Bad Company 2 (PS3)

Rendering jobs

Rendering systems are heavily divided up into CPU- & SPU-jobs

Most will move to GPU

- Eventually.. A few have already!
- PC CPU<->GPU *latency wall* ☹
- Mostly one-way data flow

Jobs:

- Terrain geometry [3]
- Undergrowth generation [2]
- Decal projection [4]
- Particle simulation
- Frustum culling
- Occlusion culling
- Occlusion rasterization
- Command buffer generation [6]
- PS3: Triangle culling [6]

Occlusion culling job example

Problem: Buildings & env occlude large amounts of objects

Obscured objects still have to:

- Update logic & animations
- Generate command buffer
- Processed on CPU & GPU

= expensive & wasteful ☹

Difficult to implement full culling:

- Destructible buildings
- Dynamic occludees
- Difficult to precompute

From Battlefield: Bad Company PS3

DICE

FROSTBITE
A DICE TECHNOLOGY

Solution: Software occlusion culling

Rasterize coarse zbuffer on SPU/CPU

- 256x114 float
- Low-poly occluder meshes
  - 100 m view distance
  - Max 10000 vertices/frame
- Parallel vertex & raster SPU-jobs
- Cost: a few milliseconds

Cull all objects against zbuffer

- Screen-space bounding-box test
- Before passed to all other systems
- Big performance savings!

GPU occlusion culling

Ideally want to use the GPU, but current APIs are limited:

- Occlusion queries introduces overhead & latency
- Conditional rendering only helps GPU
- Compute Shader impl. possible, but same *latency wall*

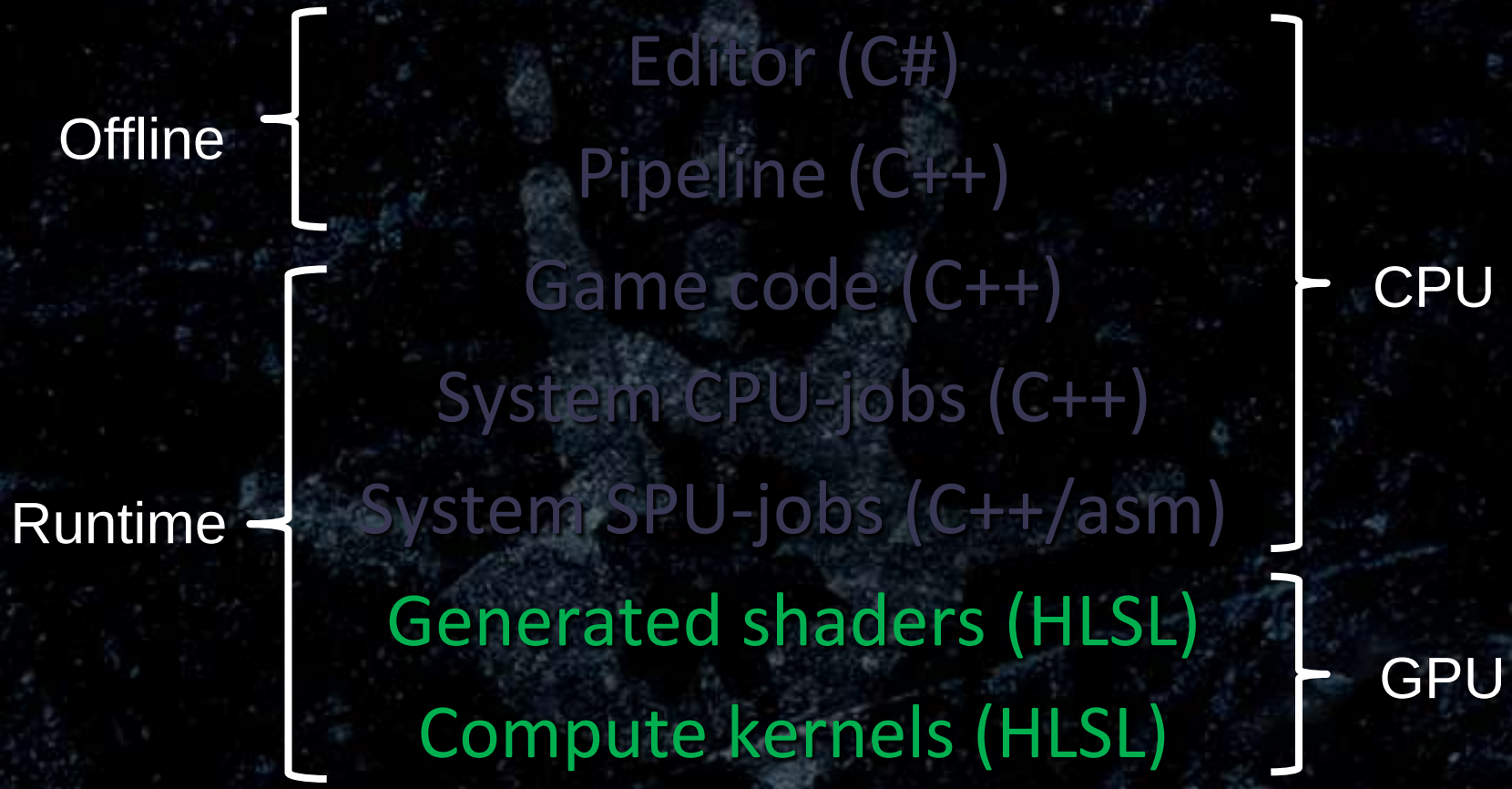
Future 1: Low-latency GPU execution context

- Rasterization and testing done on GPU where it belongs
- Lockstep with CPU, need to read back within a few ms
- Should be possible on *Larrabee* & *Fusion*, want on all PC

Future 2: Move entire cull & rendering to ”GPU”

- World, cull, systems, dispatch. End goal
- *Very* thin GPU graphics driver – GPU feeds itself

Levels of code in Frostbite



Shader types

- Generated shaders [1]
  - Graph-based *surface shaders*
  - Treated as content, not code
  - Artist created
  - Generates HLSL code
  - Used by all meshes and 3d surfaces
- Graphics / Compute kernels
  - Hand-coded & optimized HLSL
  - Statically linked in with C++
  - Pixel- & compute-shaders
  - Lighting, post-processing & special effects

Graph-based surface shader in FrostEd 2

Futures

Challenges

3 major challenges/goals going forward:

1. How do we make it easier to develop, maintain & parallelize general game code?
2. What do we need to continue to innovate & scale up real-time computational graphics?

   Most likely the same solution(s)!
3. How can we move & scale up advanced simulation and non-graphics tasks to data-parallel manycore processors?

Challenge 1

"How do we make it easier to develop, maintain & parallelize general game code?"

- *Shared State Concurrency* is a killer
  - Not a big believer in *Software Transactional Memory* either
  - Because of performance and too "optimistic" flow
- A more strict & adapted C++ model
  - Support for true immutable & r/w-only memory access
    - Per-thread/task memory access opt-in
    - Reduce the possibility for side effects in parallel code
    - As much compile-time validation as possible
  - Micro-threads / coroutines as first class citizens
  - More?
- Other languages?

Challenge 1 - Task parallelism

- Multiple task libraries
  - EA JobManager
    - Dynamic job graphs, on-demand dependencis & continuations
    - Targeted to cross-platform SPU-jobs, *key* requirement for this generation
    - Not geared towards super-simple to use CPU parallelism
  - MS ConcRT, Apple GCD, Intel TBB
    - All has some good parts!
    - Neither works on all of our current platforms
  - OpenMP
    - Just say no. Parallelism can't be tacked on, need to be an explicit core part

- Need C++ enhancements to simplify usage
  - *C++ 0x* lambdas / GCD blocks ☺
  - Glacial C++ development & deployment ☹
  - Want on all platforms, so lost on this console generation

Challenge 2 - Definition

”Real-time interactive graphics & simulation at Pixar level of quality”

- Needed visual features:
  - Complete anti-aliasing + natural motion blur & DOF
  - Global indirect lighting & reflections
  - Sub-pixel geometry
  - OIT
  - Huge improvements in character animation

These require massively more compute, BW and improved model!

(animation can’t be solved with just more/better compute, so pretend it doesn’t exist for now)

Challenge 2 - Problems

Problems & limitations with current GPU model:

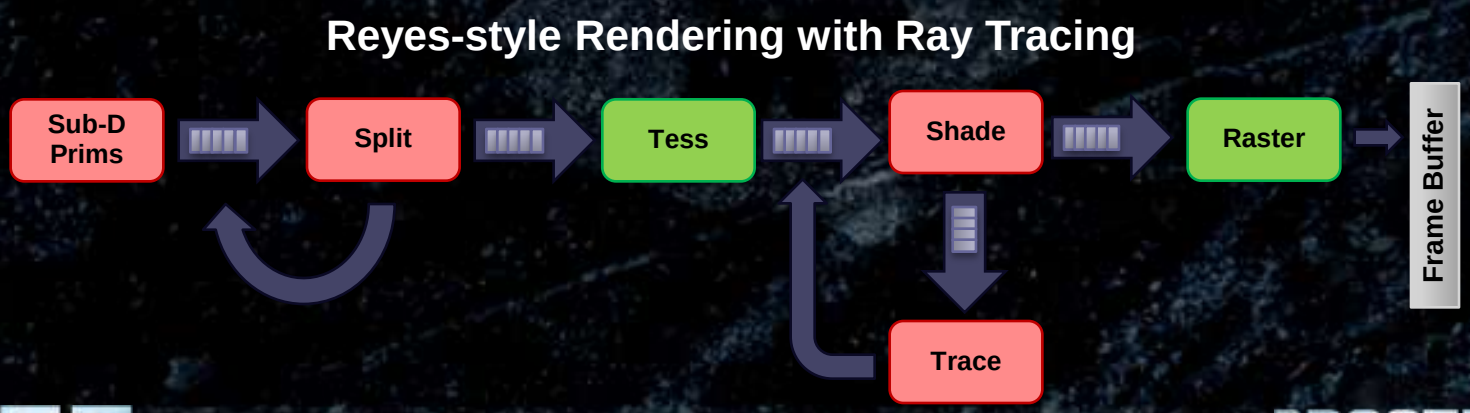
- Fixed rasterization pipeline
  - Compute pipeline not fast enough to replace it
- GPU is handicapped by being spoon-fed by CPU
- *Irregular workloads* are difficult & inefficient
- Can't express *nested data-parallelism* well
- Current HLSL is a very limited language

Challenge 2 - Solutions

- Absolutely a job for a *high-throughput oriented data-parallel processor*
  - With a highly flexible programming model
  - The CPU, as we know it, and APIs/drivers are only in the way
- Pure software solution not practical as next step after DX11 PC
  - Multi-vendor & multi-architecture marketplace
  - But for future consoles, the more flexible the better (perf. permitting)
- Long term goal:
  - Multi-vendor & multi-platform standard/virtual DP ISA
- Want a rich programmable compute model as next step
  - Nested data-parallelism & GPU self-feeding
  - Low-latency CPU<->GPU interaction
  - Efficiently target varying HW architectures

”Pipelined Compute Shaders”

- *Queues* as streaming I/O between compute kernels
  - Simple & expressive model supporting *irregular workloads*
  - Keeps data on chip, supports variable sized caches & cores
    - Can target multiple types of HW & architectures
- Hybrid graphics/compute user-defined pipelines
  - Language/API defining fixed stages inputs & outputs
  - Pipelines can feed other pipelines (similar to *DrawIndirect*)



”Pipelined Compute Shaders”

- Wanted for next major *DirectX* and *OpenCL/OpenGL*
  - As a standard, as soon as possible
  - Run on all: discrete GPU, integrated GPU and CPU

- Model is also a good fit for many of our CPU/SPU jobs
  - Parts of job graph can be seen as queues between stages
  - Easier to write kernels/jobs with streaming I/O
    - Instead of explicit fixed-buffers and ”memory passes”
    - Or dynamic memory allocation

Language?

- Future DP language is a big question
  - But the concepts & infrastructure are what is important!
- Could be an extended HLSL or ”data-parallel C++”
  - Data-oriented imperative language (i.e. not standard C++)
  - Think HLSL would probably be easier & the most explicit
  - Amount of code is small and written from scratch
- Shader-like implicit SoA vectorization (SIMT)
  - Instead of SSE/LRBni-like explicit vectorization
  - Need to be first class language citizen, not tacked on C++
  - Easier to target multiple evolving architectures implicitly

Future hardware (1/2)

- Single main memory & address space
  - Critical to share resources between graphics, simulation and game in immersive dynamic worlds
- Configurable kernel local stores / cache
  - Similar to *Nvidia Fermi* & *Intel Larrabee*
  - Local stores = reliability & good for regular loads
    - Together with user-driven async DMA engines
  - Caches = essential for irregular data structures
- Cache coherency?
  - Not always important for kernels
  - But essential for general code, can partition?

Future hardware (2/2)

- 2015 = 40 TFLOPS, we would spend it on:
  - 80% graphics
  - 15% simulation
  - 4% misc
  - 1% game (wouldn't use all 400 GFLOPS for game logic & glue!)
- OOE CPUs more efficient for the majority of our game code
  - But for the *vast* majority of our FLOPS these are fully irrelevant
  - Can evolve to a small dot on a sea of DP cores
  - Or run scalar on general DP cores wasting some compute
- In other words: no need for separate CPU and GPU!

-> single heterogeneous processor

Conclusions

- Future is an interleaved mix of task- & data-parallelism
  - On both the HW and SW level
  - But programmable DP is where the massive compute is done
- Data-parallelism requires *data-oriented design*
- Developer productivity can't be limited by model(s)
  - It should enhance productivity & perf on all levels
  - Tools & language constructs play a critical role
- We should welcome our parallel future!

Thanks to

- DICE, EA and the Frostbite team
- The graphics/gamedev community on Twitter

- Steve McCalla, Mike Burrows
- Chas Boyd
- Nicolas Thibieroz, Mark Leather
- Dan Wexler, Yury Uralsky
- Kayvon Fatahalian

References

Previous Frostbite-related talks:

- [1] Johan Andersson. ”Frostbite Rendering Architecture and Real-time Procedural Shading & Texturing Techniques ”. GDC 2007. http://repi.blogspot.com/2009/01/conference-slides.html
- [2] Natasha Tartarchuk & Johan Andersson. ”Rendering Architecture and Real-time Procedural Shading & Texturing Techniques”. GDC 2007. http://developer.amd.com/Assets/Andersson-Tatarchuk-FrostbiteRenderingArchitecture(GDC07_AMD_Session).pdf
- [3] Johan Andersson. ”Terrain Rendering in Frostbite using Procedural Shader Splatting”. Siggraph 2007. http://developer.amd.com/media/gpu_assets/Andersson-TerrainRendering(Siggraph07).pdf
- [4] Daniel Johansson & Johan Andersson. “Shadows & Decals – D3D10 techniques from Frostbite”. GDC 2009. http://repi.blogspot.com/2009/03/gdc09-shadows-decals-d3d10-techniques.html
- [5] Bill Bilodeau & Johan Andersson. “Your Game Needs Direct3D 11, So Get Started Now!”. GDC 2009. http://repi.blogspot.com/2009/04/gdc09-your-game-needs-direct3d-11-so.html
- [6] Johan Andersson. ”Parallel Graphics in Frostbite”. Siggraph 2009, Beyond Programmable Shading course. http://repi.blogspot.com/2009/08/siggraph09-parallel-graphics-in.html

Questions?

Email: johan.andersson@dice.se **Blog:** http://repi.se **Twitter:** @repi

For more DICE talks: http://publications.dice.se

Bonus slides

Game development

- 2 year development cycle
  - New IP often takes *much* longer, 3-5 years
  - Engine is continuously in development & used
- AAA teams of 70-90 people
  - 40% artists
  - 30% designers
  - 20% programmers
  - 10% audio
- Budgets $20-40 million

- Cross-platform development is market reality
  - Xbox 360 and PlayStation 3
  - PC (DX9, DX10 & DX11 for BC2)
  - Current consoles will stay with us for many more years

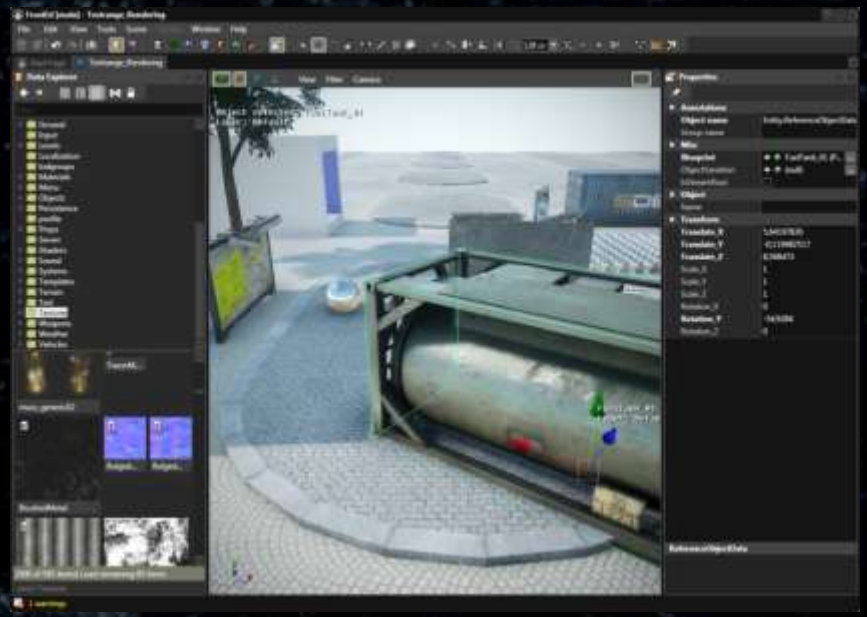
Game engine requirements (1/2)

- Stable real-time performance
  - Frame-driven updates, 30 fps
  - Few threads, instead per-frame jobs/tasks for everything
- Predictable memory usage
  - Fixed budgets for systems & content, fail if over
  - Avoid runtime allocations
  - Love *unified memory*!
- Cross-platform
  - The consoles determines our base tech level & focus
  - PS3 is design target, most difficult and good potential
  - Scale up for PC, dual core is min spec (slow!)

Game engine requirements (2/2)

- Full system profiling/debugging
  - Engine is a vertical solution, touches everywhere
  - PIX, xbtracedump, SN Tuner, ETW, GPUView
- Quick iterations
  - Essential in order to be creative
  - Fast building & fast loading, hot-swapping resources
  - Affects both the tools and the game
- Middleware
  - Use when it make senses, cross-platform & optimized
  - Parallelism have to go through *our* systems

Editor & Pipeline



Editor (”FrostEd 2”)

- WYSIWYG editor for content
- *C#*, Windows only
- Basic threading / tasks

Pipeline

- Offline/background data-processing & conversion
- *C++*, some *MC++*, Windows only
- Typically IO-bound
- A few compute-heavy steps use CPU-jobs
- Texture compression uses CUDA, prefer OpenCL or CS

CPU parallelism models are generally not a problem here

EntityRenderCull job example

```
struct FB_ALIGN(16) EntityRenderCullJobData
{
	enum
	{
		MaxSphereTreeCount = 2,
		MaxStaticCullTreeCount = 2
	};

	uint sphereTreeCount;
	const SphereNode* sphereTrees[MaxSphereTreeCount];

	u8 viewCount;
	u8 frustumCount;

	u8 viewIntersectFlags[32];
	Frustum frustums[32];

	.... (cut out 2/3 of struct for display size)

	u32 maxOutEntityCount;

	// Output data, pre-allocated by callee
	u32 outEntityCount;
	EntityRenderCullInfo* outEntities;
};

void entityRenderCullJob(EntityRenderCullJobData* data);

void validate(const EntityRenderCullJobData& data);
```

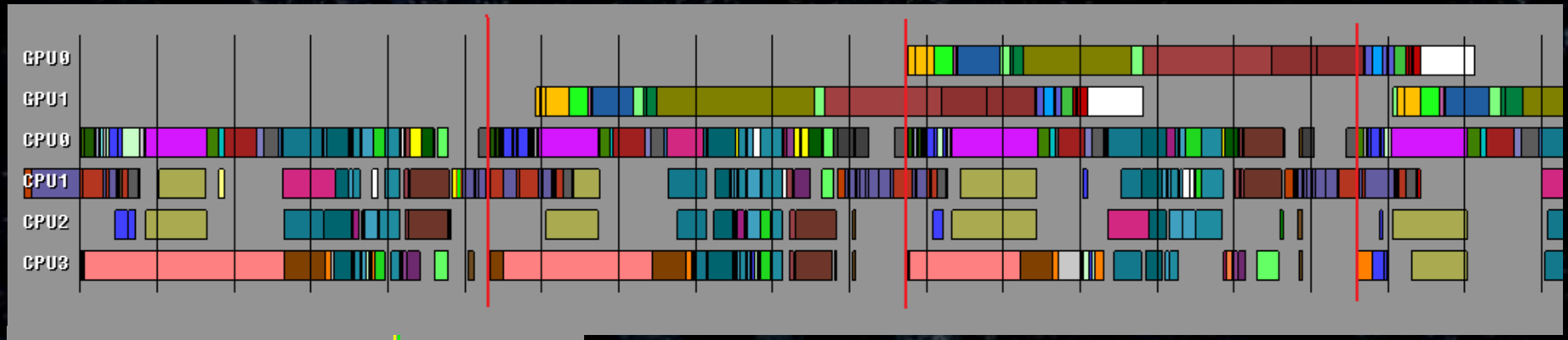
- Frustum culling of dynamic entities in sphere tree
- struct contain all input data needed
- Max output data pre-allocated by callee
- Single job function
  - Compile both as CPU & SPU job
- Optional struct validation func

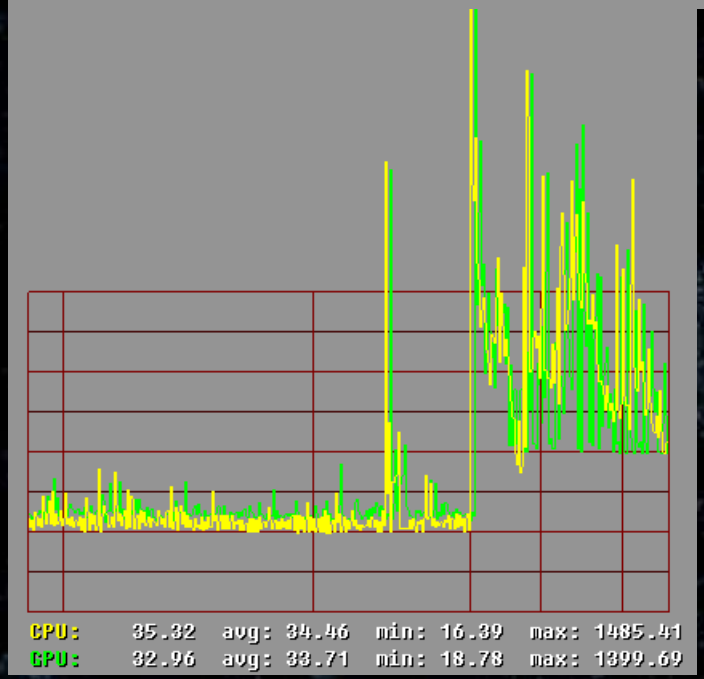
EntityRenderCull SPU setup

```
// local store variables
EntityRenderCullJobData g_jobData;
float g_zBuffer[256*114];
u16 g_terrainHeightData[64*64];

int main(uintptr_t dataEa, uintptr_t, uintptr_t, uintptr_t)
{
    dmaBlockGet("jobData", &g_jobData, dataEa, sizeof(g_jobData));
    validate(g_jobData);

    if (g_jobData.zBufferTestEnable)
    {
        dmaAsyncGet("zBuffer", g_zBuffer, g_jobData.zBuffer, g_jobData.zBufferResX*g_jobData.zBufferResY*4);
        g_jobData.zBuffer = g_zBuffer;

        if (g_jobData.zBufferShadowTestEnable && g_jobData.terrainHeightData)
        {
            dmaAsyncGet("terrainHeight", g_terrainHeightData, g_jobData.terrainHeightData, g_jobData.terrainHeightDataSize);
            g_jobData.terrainHeightData = g_terrainHeightData;
        }
        dmaWaitAll(); // block on both DMAs
    }

    // run the actual job, will internally do streaming DMAs to the output entity list
    entityRenderCullJob(&g_jobData);

    // put back the data because we changed outEntityCount
    dmaBlockPut(dataEa, &g_jobData, sizeof(g_jobData));
    return 0;
}
```

Timing view

Example: PC, 4 CPU cores, 2 GPUs in AFR (AMD Radeon 4870x2)





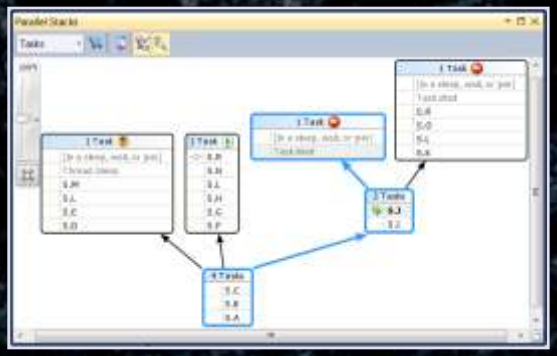
- Real-time in-game overlay
  - See timing events & effective parallelism
  - On CPU, SPU & GPU – for all platforms
  - Use to reduce sync-points & optimize load balancing
- GPU timing through DX event queries
- Our main performance tool!

Language (cont.)

- Requirements:
  - Full rich debugging, ideally in Visual Studio
  - Asserts
  - Internal kernel profiling
  - Hot-swapping / edit-and-continue of kernels
- Opportunity for IHVs and platform providers to innovate here!
  - Goal: Cross-vendor standard
  - Think of the co-development of *Nvidia Cg* and HLSL

Unified development environment

- Want to debug/profile task- & data-parallel code seamlessly
  - On all processors! CPU, GPU & manycore
  - From any vendor = requires standard APIs or ISAs
- *Visual Studio 2010* looks promising for task-parallel PC code
  - Usable by our offline tools & hopefully PC runtime
  - Want to integrate our own JobManager
- *Nvidia Nexus* looks great for data-parallel GPU code
  - Eventual must have for all HW, how?
  - Huge step forward!



VS2010 Parallel Tasks