Parallel Graphics in Frostbite – Current & Future

Johan Andersson
DICE
Menu

- Game engine CPU & GPU parallelism
- Rendering techniques & systems – old & new
- Mixed in with some future predictions & wishes
Quick background

Frostbite 1.x  [1][2][3]
- Xbox 360, PS3, DX10
- Battlefield: Bad Company (shipped)
- Battlefield 1943 (shipped)
- Battlefield: Bad Company 2

Frostbite 2  [4][5]
- In development
- Xbox 360, PS3
- DX11 (10.0, 10.1, 11)

Disclaimer: Unless specified, pictures are from engine tests, not actual games
Job-based parallelism

Must utilize all cores in the engine

- Xbox 360: 6 HW threads
- PS3: 2 HW threads + 6 great SPUs
- PC: 2-8 HW threads
  - And many more coming

Divide up systems into Jobs

- Async function calls with explicit inputs & outputs
- Typically fully independent stateless functions
  - Makes it easier on PS3 SPU & in general
- Job dependencies create job graph

All cores consume jobs
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

Braided Parallelism* [6]
- Intermixed task- & data-parallelism

* Still only 10 hits on google (yet!), but I like Aaron’s term
Rendering jobs

Rendering systems are heavily divided up into jobs

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

Most will move to GPU
- Eventually.. A few have already!
- Mostly one-way data flow

I will talk about a couple of these..
Parallel command buffer recording

Dispatch draw calls and state to multiple command buffers in parallel
- Scales linearly with # cores
- 1500-4000 draw calls per frame
- Reduces latency & improves performance

Important for all platforms, used on:
- Xbox 360
- PS3 (SPU-based)
- PC DX11

Previously not possible on PC, but now in DX11...
DX11 parallel dispatch

First class citizen in DX11
- Killer feature for reducing CPU overhead & latency
- ~90% of our rendering dispatch job time is in D3D/driver

1. DX11 deferred device context per core
   - Together with dynamic resources (cbuffer/vbuffer) for usage on that deferred context

2. Renderer has list of all draw calls we want to do for each rendering “layer” of the frame

3. Split draw calls for each layer into chunks of ~256 and dispatch in parallel to the deferred contexts
   - Each chunk generates a command list

4. Render to immediate context & execute command lists

5. Profit!

Goal: close to linear scaling up to octa-core when we get full DX11 driver support (up to the IHVs now)

Future note: This is “just” a stopgap measure until we evolve the GPU to be able to fully feed itself (hi LRB)
Occlusion culling

Problem: Buildings & env occlude large amounts of objects

Invisible objects still have to:
- Update logic & animations
- Generate command buffer
- Processed on CPU & GPU

Difficult to implement full culling
- Destructible buildings
- Dynamic occludees
- Difficult to precompute
- GPU occlusion queries can be heavy to render

From Battlefield: Bad Company PS3
Our solution:

Software occlusion rasterization
Software occlusion culling

Rasterize coarse zbuffer on SPU/CPU

- 256x114 float
  - Good fit in SPU LS, but could be 16-bit
- Low-poly occluder meshes
  - Manually conservative
  - 100 m view distance
  - Max 10000 vertices/frame
- Parallel SPU vertex & raster jobs
- Cost: a few milliseconds

Then cull all objects against zbuffer

- Before passed to all other systems = big savings
- Screen-space bounding-box test

Pictures & numbers from Battlefield: Bad Company PS3
GPU occlusion culling

Ideally want GPU rasterization & testing, but:
- Occlusion queries introduces overhead & latency
  - Can be manageable, but far from ideal
- Conditional rendering only helps GPU
  - Not CPU, frame memory or draw calls

Future 1: Low-latency extra GPU exec. context
- Rasterization and testing done on GPU where it belongs
- Lockstep with CPU, need to read back data within a few ms
- Should be possible on LRB (latency?), want on all HW

Future 2: Move entire cull & rendering to “GPU”
- World rep., cull, systems, dispatch. End goal.
Problem: Slow GPU triangle & vertex setup on PS3

- Combined with unique situation with powerful & initially not fully utilized “free” SPUs!

Solution: SPU triangle culling

- Trade SPU time for GPU time
- Cull all back faces, micro-triangles, out of frustum
  - Based on Sony’s PS3 EDGE library [7]
  - Also see Jon Olick’s talk from the course last year

- 5 SPU jobs processes frame geometry in parallel
- Output is new index buffer for each draw call
Custom geometry processing

Software control opens up great flexibility and programmability!

Simple custom culling/processing that we’ve added:

- Partition bounding box culling
- Mesh part culling
- Clip plane triangle trivial accept & reject
- Triangle cull volumes (inverse clip planes)

Others are doing: Full skinning, morph targets, CLOD, cloth

Future wish: No forced/fixed vertex & geometry shaders

- DIY compute shaders with fixed-func stages (tessellation and rasterization)
- Software-controlled queuing of data between stages
  - To avoid always spilling out to memory
Decal projection

- Traditionally a CPU process
  - Relying on identical visual & physics representation 😞
  - Or duplicated mesh data in CPU memory (on PC) 😞

- Consoles read visual mesh data directly
  - UMA! 😊
  - Project in SPU-jobs
  - Output VB/IB to GPU
Decals through GS & StreamOut

Keep the computation & data on the GPU (DX10)

- See GDC’09 “Shadows & Decals – D3D10 techniques in Frostbite”, slides with complete source code online [4]

Process all mesh triangles with Geometry Shader

1. Test decal projection against the triangles
2. Setup per-triangle clip planes for intersecting tris
3. Output intersecting triangles using StreamOut

Issues:
- StreamOut management
- Drivers (not your standard GS usage)

Benefits:
- CPU & GPU worlds separate
- No CPU memory or upload
- Huge decals + huge meshes
Deferred lighting/shading

Traditional deferred shading:

1. Graphics pipeline rasterizes gbuffer for opaque surfaces
   - Normal, albedos, roughness

2. Light sources are rendered & accumulate lighting to a texture
   - Light volume or screen-space tile rendering

3. Combine shading & lighting for final output

Also see Wolfgang’s talk “Light Pre-Pass Renderer Mark III” from Monday for a wider description [8]
1. Divide screen up into tiles and determine how many & which light sources intersect each tile

2. Only apply the visible light sources on pixels in each tile
   - Reduced BW & setup cost with multiple lights in single shader

Used in Naughty Dog’s Uncharted [9] and SCEE PhyreEngine [10]

Hmm, isn’t light classification per screen-space tile sort of similar of how a compute shader can work with 2D thread groups?

Answer: YES, except a CS can do everything in a single pass!

From “The Technology of Uncharted”. GDC’08 [9]
CS-based deferred shading

Deferred shading using DX11 CS

- Experimental implementation in Frostbite 2
- Not production tested or optimized
- Compute Shader 5.0
- Assumption: No shadows (for now)

New hybrid Graphics/Compute shading pipeline:

1. Graphics pipeline rasterizes gbuffers for opaque surfaces
2. Compute pipeline uses gbuffers, culls light sources, computes lighting & combines with shading

(multiple other variants also possible)
CS requirements & setup

- Input data is gbuffers, depth buffer & light constants
- Output is fully composited & lit HDR texture
- 1 thread per pixel, 16x16 thread groups (aka tile)

```cpp
Texture2D<float4> gbufferTexture1 : register(t0);
Texture2D<float4> gbufferTexture2 : register(t1);
Texture2D<float4> gbufferTexture3 : register(t2);
Texture2D<float4> depthTexture : register(t3);
RWTexture2D<float4> outputTexture : register(u0);
#define BLOCK_SIZE 16

[numthreads(BLOCK_SIZE,BLOCK_SIZE,1)]
void csMain(
    uint3 groupId : SV_GroupID,
    uint3 groupThreadId : SV_GroupThreadID,
    uint groupIndex: SV_GroupIndex,
    uint3 dispatchThreadId : SV_DispatchThreadID
){
    ...
}
```
1. Load gbuffers & depth

2. Calculate min & max z in threadgroup / tile
   - Using InterlockedMin/Max on groupshared variable
   - Atomics only work on ints 😞
   - But casting works (z is always +)

```cpp
// --- globals above, function below -------
float depth = depthTexture.Load(uint3(texCoord, 0)).r;
uint depthInt = asuint(depth);
minDepthInt = 0xFFFFFFFF;
maxDepthInt = 0;
GroupMemoryBarrierWithGroupSync();
InterlockedMin(minDepthInt, depthInt);
InterlockedMax(maxDepthInt, depthInt);
GroupMemoryBarrierWithGroupSync();
float minGroupDepth = asfloat(minDepthInt);
float maxGroupDepth = asfloat(maxDepthInt);
```

Optimization note:
Separate pass using parallel reduction with Gather to a small texture could be faster

Note to the future:
GPU already has similar values in HiZ/ZCull!
Can skip step 2 if we could resolve out min & max z to a texture directly

Min z looks just like the occlusion software rendering output
CS step 3 – Cull idea

3. Determine visible light sources for each tile
   - Cull all light sources against tile “frustum”
     - Light sources can either naively be all light sources in the scene, or CPU frustum culled potentially visible light sources
   - Output for each tile is:
     - # of visible light sources
     - Index list of visible light sources

<table>
<thead>
<tr>
<th>Lights</th>
<th>Indices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global list</td>
<td>1000+</td>
</tr>
<tr>
<td></td>
<td>0 1 2 3 4 5 6 7 8 ..</td>
</tr>
<tr>
<td>Tile visible list</td>
<td>~0-40+</td>
</tr>
<tr>
<td></td>
<td>0 2 5 6 8 ..</td>
</tr>
</tbody>
</table>

Example numbers from test scene

This is the key part of the algorithm and compute shader, so must try to be rather clever with the implementation!
CS step 3 – Cull implementation

- Each thread switches to process light sources instead of a pixel*
  - Wow, parallelism switcheroo!
  - 256 light sources in parallel per tile
  - Multiple iterations for >256 lights

- Intersect light source & tile
  - Many variants dep. on accuracy requirements & performance
  - Tile min & max z is used as a shader “depth bounds” test

- For visible lights, append light index to index list
  - Atomic add to threadgroup shared memory. “inlined stream compaction”
  - Prefix sum + stream compaction should be faster than atomics, but more limiting

- Synchronize group & switch back to processing pixels
  - We now know which light sources affect the tile

```cpp
struct Light{
    float3 pos;
    float sqrRadius;
    float3 color;
    float invSqrRadius;
};
int lightCount;
StructuredBuffer<Light> lights;

groupshared uint visibleLightCount = 0;
groupshared uint visibleLightIndices[1024];

// ----- globals above, cont. function below ---------

uint threadCount = BLOCK_SIZE*BLOCK_SIZE;
uint passCount = (lightCount+threadCount-1) / threadCount;

for (uint passIt = 0; passIt < passCount; ++passIt) {
    uint lightIndex = passIt*threadCount + groupIndex;
    // prevent overrun by clamping to a last “null” light
    lightIndex = min(lightIndex, lightCount);
    if (intersects(lights[lightIndex], tile)) {
        uint offset;
        InterlockedAdd(visibleLightCount, 1, offset);
        visibleLightIndices[offset] = lightIndex;
    }
}
GroupMemoryBarrierWithGroupSync();
```

*Your grandfather’s pixel shader can’t do that!
4. For each pixel, accumulate lighting from visible lights
   - Read from tile visible light index list in threadgroup shared memory

```c
float3 diffuseLight = 0;
float3 specularLight = 0;

for (uint lightIt = 0; lightIt < visibleLightCount; ++lightIt)
{
    uint lightIndex = visibleLightIndices[lightIt];
    Light light = lights[lightIndex];

    evaluateAndAccumulateLight(
        light,
        gbufferParameters,
        diffuseLight,
        specularLight);
}
```

5. Combine lighting & shading albedos / parameters
   - Output is non-MSAA HDR texture
   - Render transparent surfaces on top

Computed lighting

Combined final output
(not the best example)
Example results
Example: 25+ analytical specular highlights per pixel
Frostbite 2 - Deferred Shading through Compute Shader test
1000 point lights
Compute Shader-based Deferred Shading demo
CS-based deferred shading

The Good:

- Constant & absolute minimal bandwidth
  - Read gbuffers & depth once!

- Doesn’t need intermediate light buffers
  - Can take a lot of memory with HDR, MSAA & color specular

- Scales up to huge amount of big overlapping light sources!
  - Fine-grained culling (16x16)
  - Only ALU cost, good future scaling
  - Could be useful for accumulating VPLs

The Bad:

- Requires DX11 HW (duh)
  - CS 4.0/4.1 difficult due to atomics & scattered groupshared writes

- Culling overhead for small light sources
  - Can accumulate them using standard light volume rendering
  - Or separate CS for tile-classific.

- Potentially performance
  - MSAA texture loads / UAV writing might be slower then standard PS

The Ugly:

- Can’t output to MSAA texture
  - DX11 CS UAV limitation.
**Future programming model**

*Queues* as compute shader streaming in/outs
- In addition to buffers/textures/UAVs
- Simple & expressive model supporting irregular workloads
- Keeps data on chip, supports variable sized caches & cores

Build *your* pipeline of stages with queues between
- Shader & fixed function stages (sampler, rasterizer, tessellator, Zcull)
- Developers can make the GPU feed itself!

GRAMPS model example [8]
What else do we want to do?

**WARNING:** Overly enthusiastic and non all-knowing game developer ranting

- **Mixed resolution MSAA particle rendering**
  - Depth test per sample, shade per quarter pixel, and depth-aware upsample directly in shader

- **Demand-paged procedural texturing / compositing**
  - Zero latency “texture shaders”

- **Pre-tessellation coarse rasterization for z-culling of patches**
  - Potential optimization in scenes of massive geometric overdraw
  - Can be coupled with recursive schemes

- **Deferred shading w/ many & arbitrary BRDFs/materials**
  - Queue up pixels of multiple materials for coherent processing in own shader
  - Instead of incoherent screen-space dynamic flow control

- **Latency-free lens flares**
  - Finally! No false/late occlusion
  - Occlusion query results written to CB and used in shader to cull & scale

- **And much much more...**
Conclusions

A good parallelization model is *key* for good game engine performance (duh)

- Job graphs of mixed task- & data-parallel CPU & SPU jobs works well for us
- SPU-jobs do the heavy lifting

Hybrid compute/graphics pipelines looks promising

- Efficient interopability is super important (DX11 is great)
- Deferred lighting & shading in CS is just the start

Want a user-defined streaming pipeline model

- Expressive & extensible hybrid pipelines with queues
- Focus on the data flow & patterns instead of doing sequential memory passes
Acknowledgements

- DICE & Frostbite team
- Nicolas Thibieroz, Mark Leather
- Miguel Sainz, Yury Uralsky
- Kayvon Fatahalian
- Matt Swoboda, Pål-Kristian Engstad
- Timothy Farrar, Jake Cannell
References


We are hiring senior developers
Questions?

- Please fill in the course evaluation at: http://www.siggraph.org/courses_evaluation

- You could win a Siggraph’09 mug (yey!)

- One winner per course, notified by email in the evening

Email: johan.andersson@dice.se
Blog: http://repi.se
Twitter: http://twitter.com/repi
Bonus slides
Timing view

- Real-time in-game overlay
  - See CPU, SPU & GPU timing events & effective parallelism
  - What we use to reduce sync-points & optimize load balancing between all processors
- GPU timing through event queries
  - AFR-handling rather shaky, but works!*

Example: PC, 4 CPU cores, 2 GPUs in AFR

*At least on AMD 4870x2 after some alt-tab action