PRACTICAL OCCLUSION CULLING FOR PS3

Will Vale
Second Intention Limited

GUERRILLA

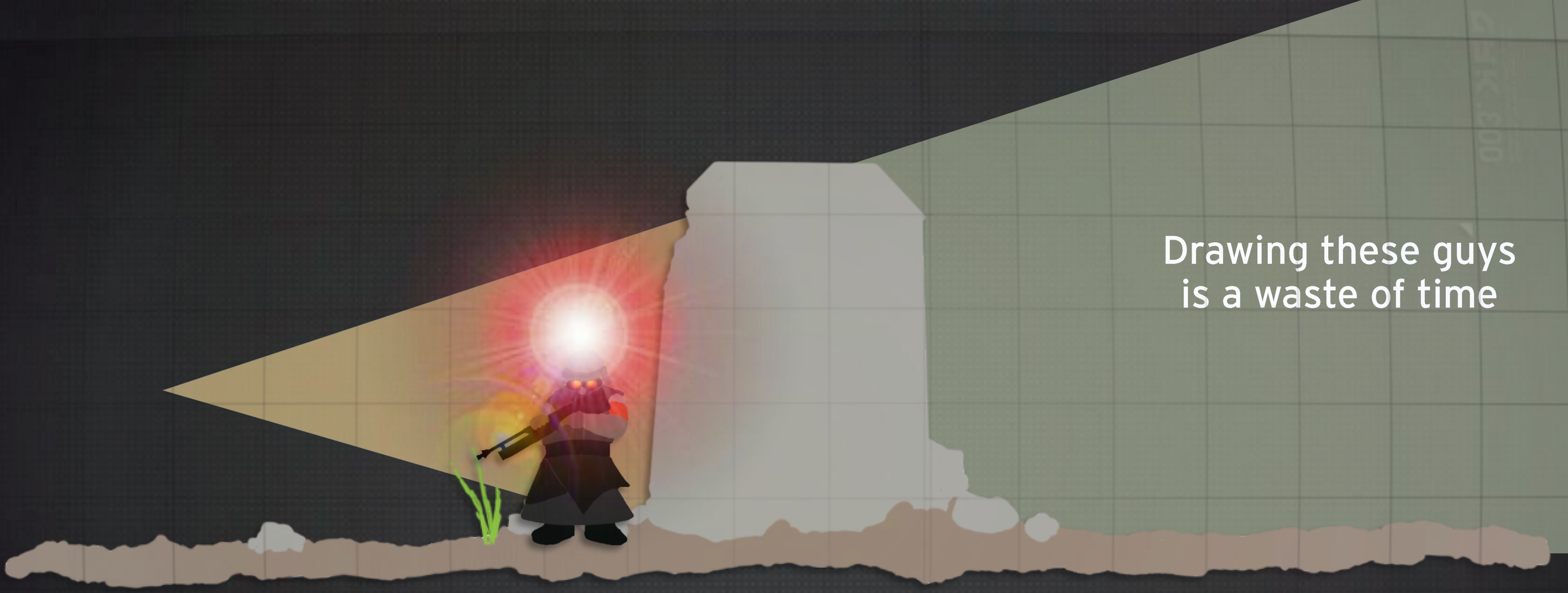
WHO IS THIS GUY?

- Freelancer with graphics tech bias
- Working with Guerrilla since 2005
- Talking about occlusion culling in KILLZONE 3

TAKEAWAY

- Background: Why do occlusion culling?
- The SPU runtime (rendering and testing)
- Creating workable occluders
- Useful debugging tools
- Problems and performance
- Results, and thoughts on where to go next

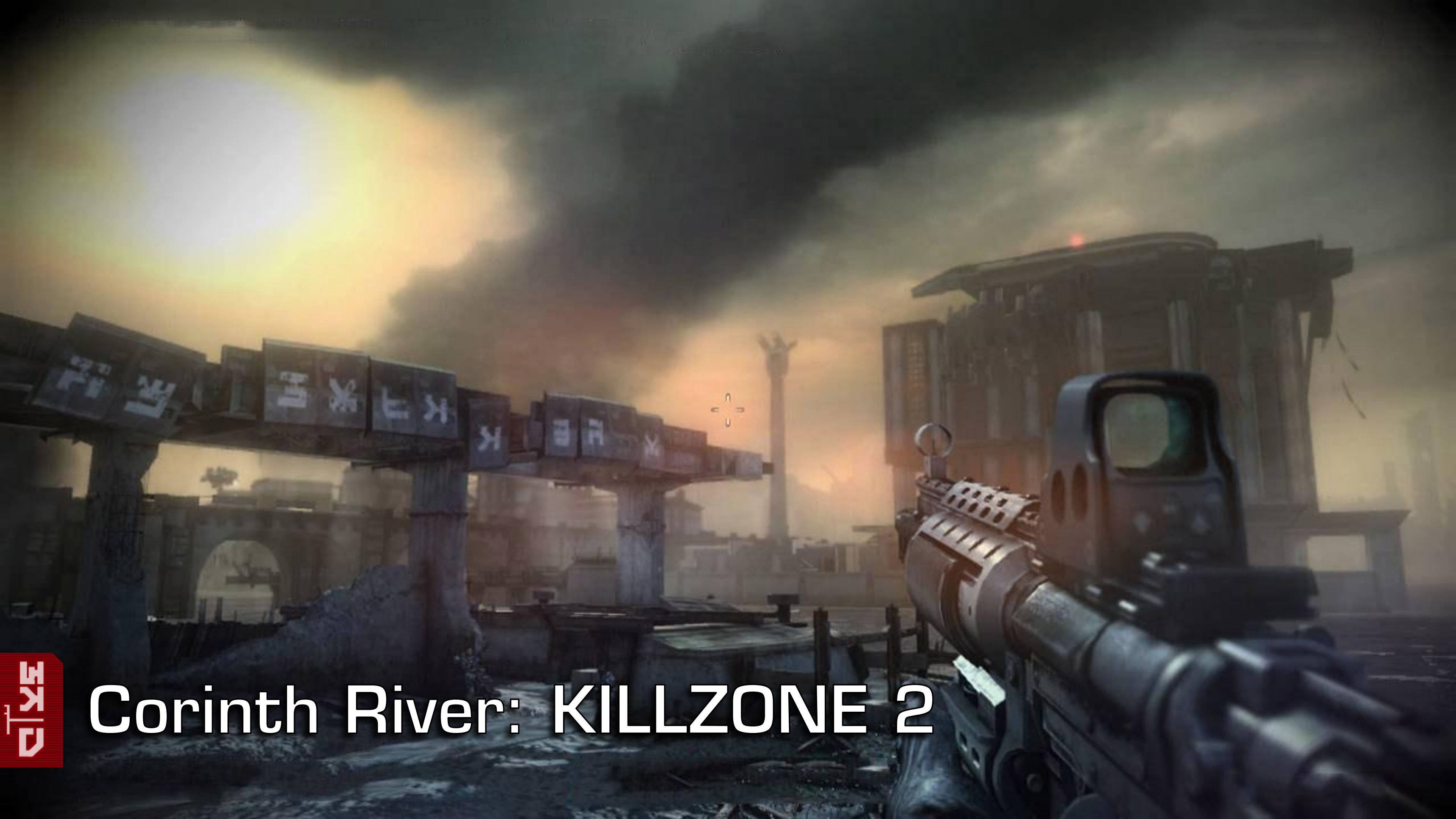
WHY DO OCCLUSION CULLING?



Drawing these guys is a waste of time

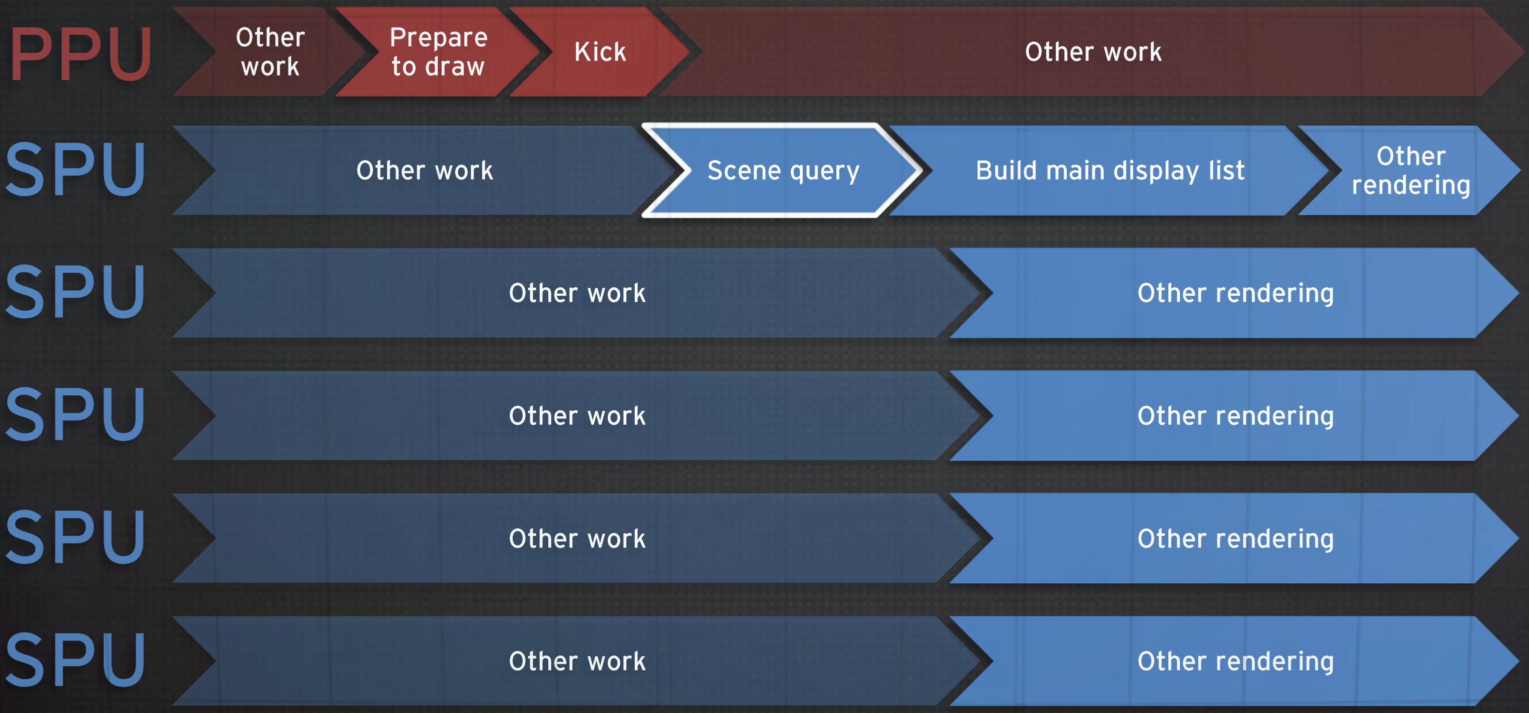
KILLZONE 2: STARTING POINT

- Scene geometry in a Kd-Tree
- Culled using zones, portals and blockers
- Problems:
  - Lots of artist time to place and tweak
  - Entirely static
  - Geometric complexity
    - Lots of tests, fiddly code
  - Too much time – around 10-30% of one SPU (serial)
- Can't feed RSX until it's done



Corinth River: KILLZONE 2

KILLZONE 2: RENDERING PIPELINE



PPU
Other work
Prepare to draw
Kick
Other work

SPU
Other work
Scene query
Build main display list
Other rendering

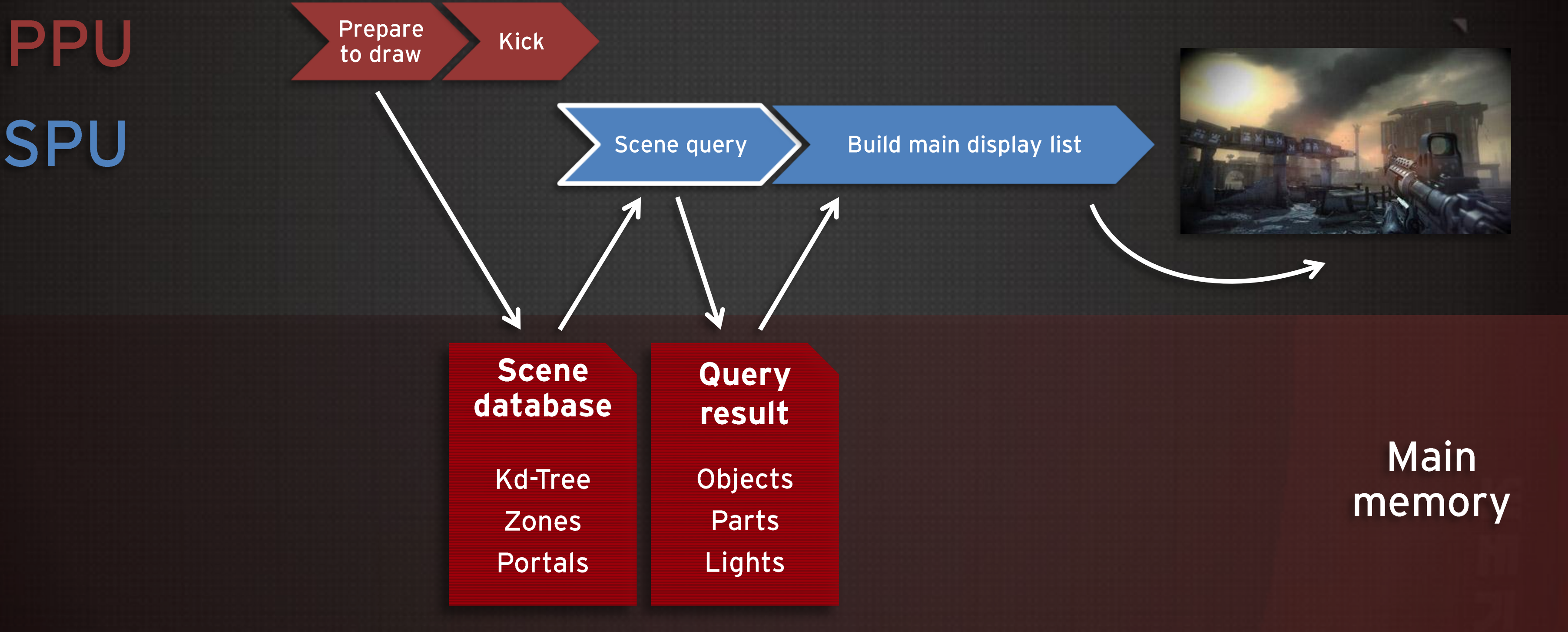
SPU
Other work
Other rendering

SPU
Other work
Other rendering

SPU
Other work
Other rendering

SPU
Other work
Other rendering

KILLZONE 2: RENDERING PIPELINE



PPU

SPU

Prepare to draw

Kick

Scene query

Build main display list

Scene database

Kd-Tree

Zones

Portals

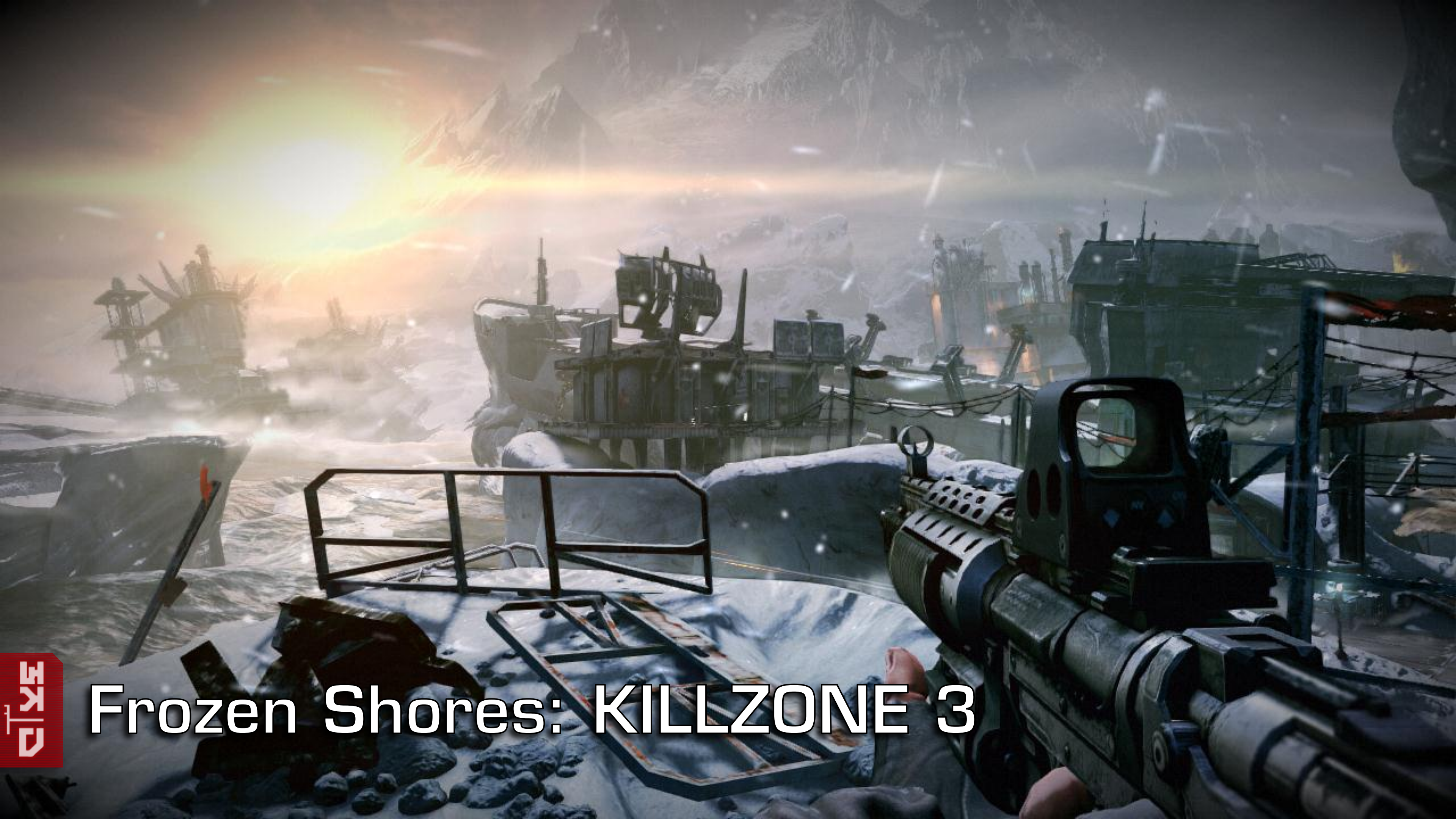
Query result

Objects

Parts

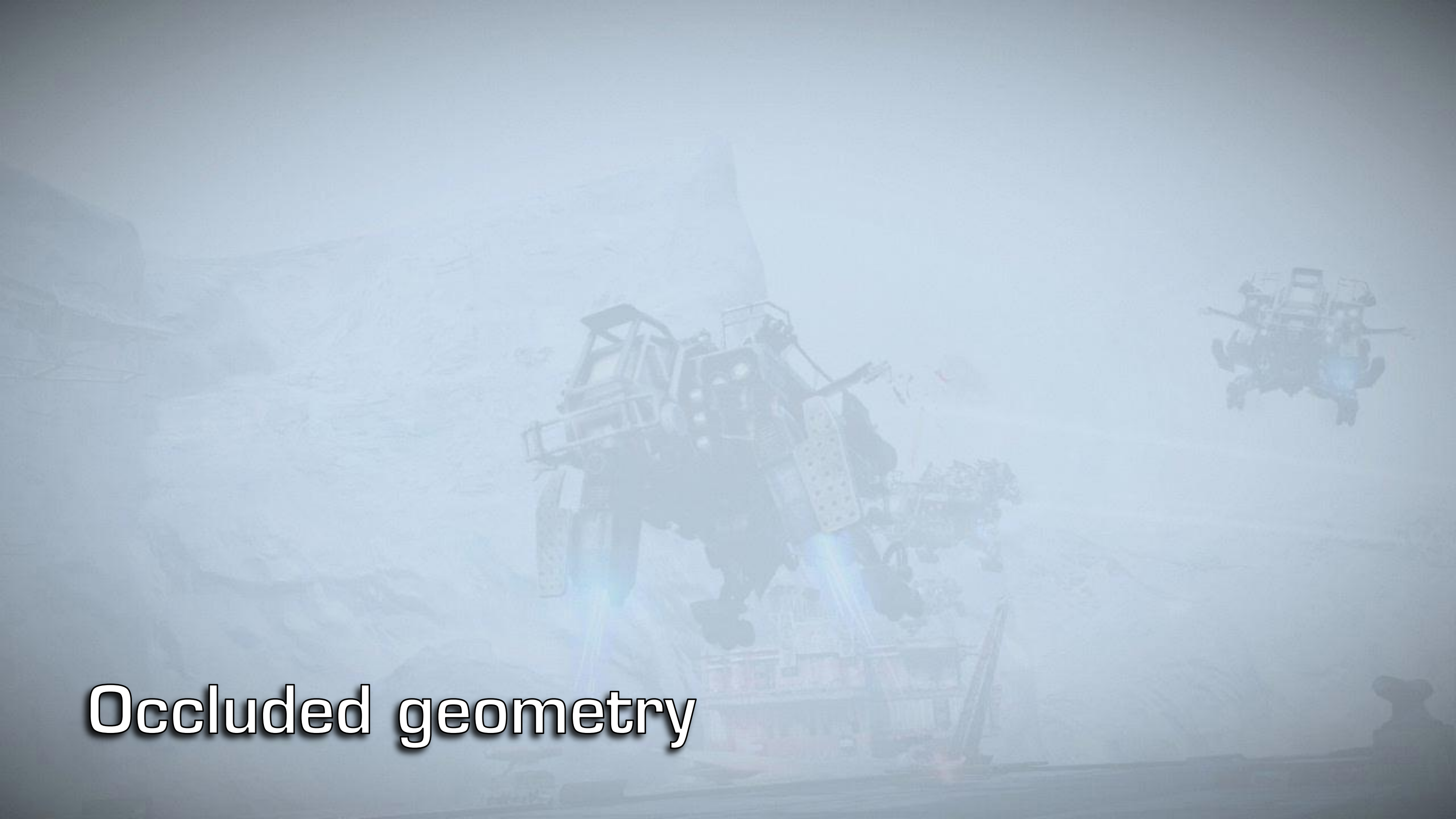
Lights

Main memory



Frozen Shores: KILLZONE 3

Frozen Shores: KILLZONE 3



Occluded geometry

KILLZONE 3: ART GOALS

- Increase scene complexity
  - Larger, more open environments
  - With more stuff in them
- Simplify content pipeline
  - Don't waste artist time on things which aren't pretty
  - Don't require artist tweaks – but allow them
- 80% solution
  - Want it to “just work” 80% of the time

KILLZONE 3: TECH GOALS

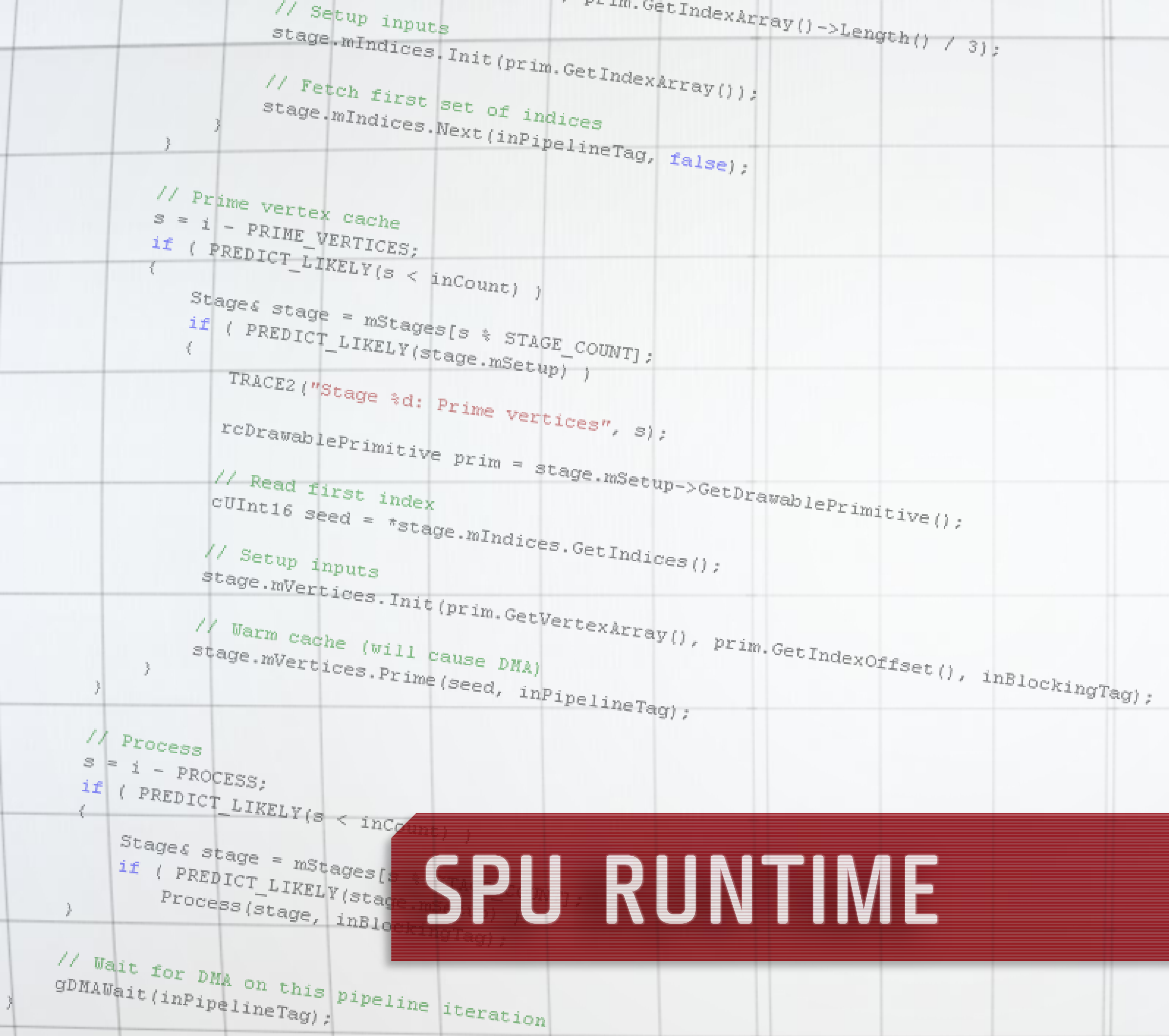
- Don't increase RSX load
    - Never enough GPU time that we can waste it
- Fully conservative solution
    - No popping when you go around corners
- Drop into pipeline without restructuring
    - Reduce risk
    - Allow swapping between implementations at runtime

THE IDEA

- Some spare memory
- Some spare SPU time
- Best guess: **create and test a depth buffer on SPUs**
    - Decouples tests and occluders
    - Rendering linear in number of occluders
    - Testing linear in number of objects
- Plays to SPU strengths
- Culls early

THE PLAN

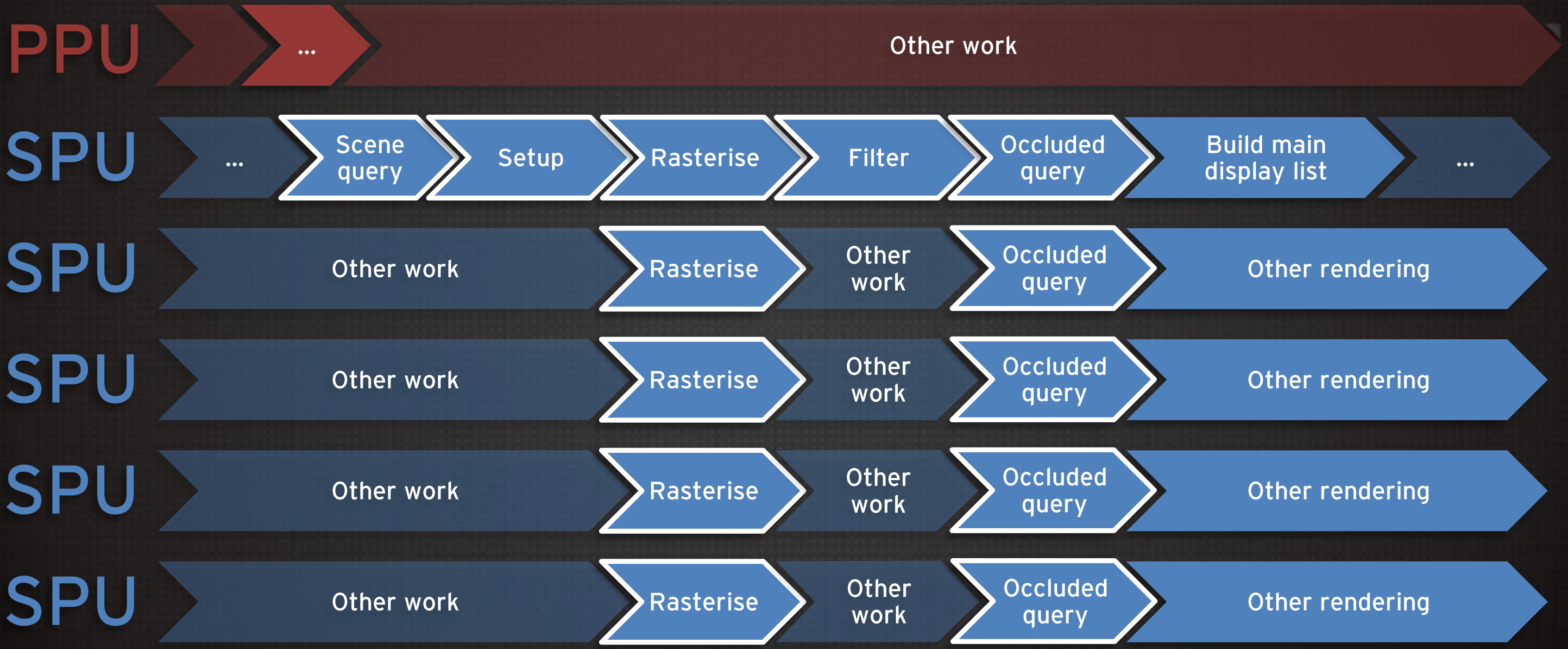
- Create occluder geometry offline
- Each frame, SPUs render occluders to 720p depth buffer
- Split buffer into 16 pixel high slices for rasterisation
- Down-sample buffer to 80x45 (16x16 max filter)
- Test bounding boxes against this during scene traversal
  - Accurate: Rasterisation + depth test
  - Coarse: Some kind of constant-time point test



```
                        , prim.GetIndexArray()->Length() / 3);
                // Setup inputs
                stage.mIndices.Init(prim.GetIndexArray());

                // Fetch first set of indices
                stage.mIndices.Next(inPipelineTag, false);
            }
        }

        // Prime vertex cache
        s = i - PRIME_VERTICES;
        if ( PREDICT_LIKELY(s < inCount) )
        {
            Stage& stage = mStages[s % STAGE_COUNT];
            if ( PREDICT_LIKELY(stage.mSetup) )
            {
                TRACE2("Stage %d: Prime vertices", s);

                rcDrawablePrimitive prim = stage.mSetup->GetDrawablePrimitive();

                // Read first index
                cUInt16 seed = *stage.mIndices.GetIndices();

                // Setup inputs
                stage.mVertices.Init(prim.GetVertexArray(), prim.GetIndexOffset(), inBlockingTag);

                // Warm cache (will cause DMA)
                stage.mVertices.Prime(seed, inPipelineTag);
            }
        }

        // Process
        s = i - PROCESS;
        if ( PREDICT_LIKELY(s < inCount) )
        {
            Stage& stage = mStages[s % STAGE_COUNT];
            if ( PREDICT_LIKELY(stage.mSetup) )
                Process(stage, inBlockingTag);
        }

        // Wait for DMA on this pipeline iteration
        gDMAWait(inPipelineTag);
    }
```

SPU RUNTIME

KILLZONE 3: MODIFIED PIPELINE



PPU: … | Other work

SPU: … | Scene query | Setup | Rasterise | Filter | Occluded query | Build main display list | …

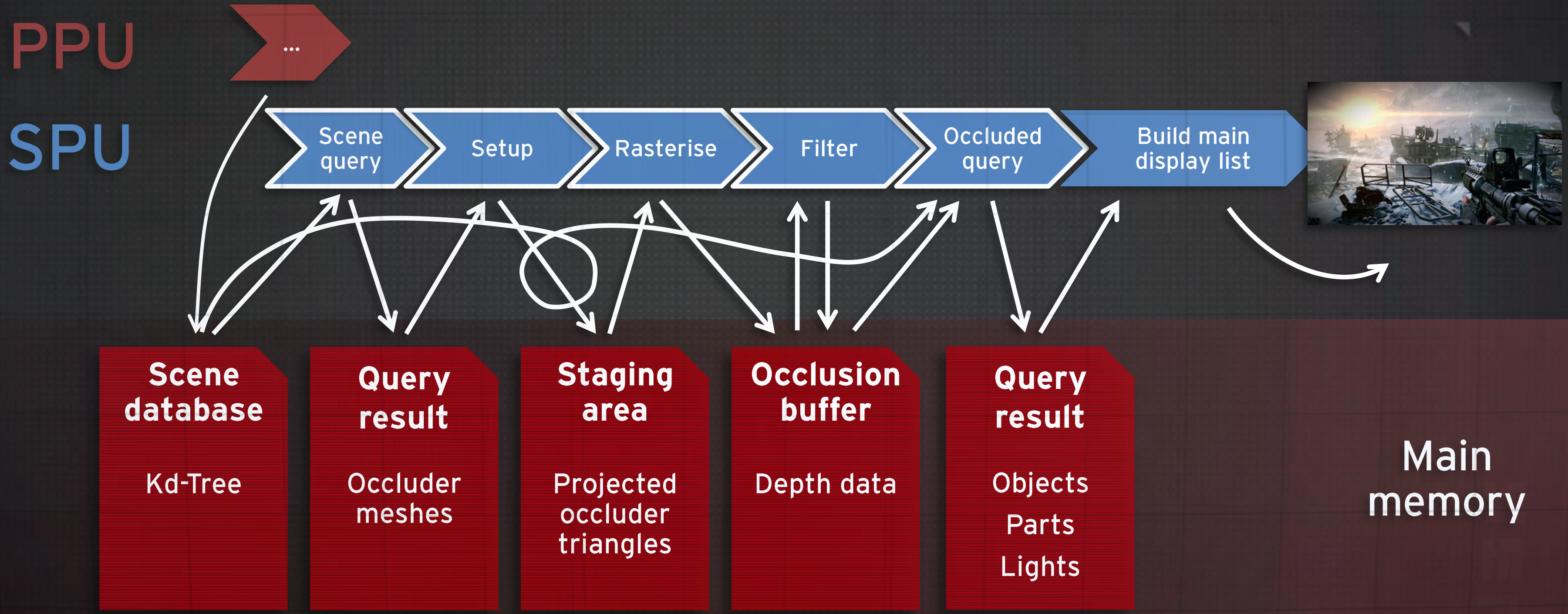
SPU: Other work | Rasterise | Other work | Occluded query | Other rendering

SPU: Other work | Rasterise | Other work | Occluded query | Other rendering

SPU: Other work | Rasterise | Other work | Occluded query | Other rendering

SPU: Other work | Rasterise | Other work | Occluded query | Other rendering

KILLZONE 3: MODIFIED PIPELINE



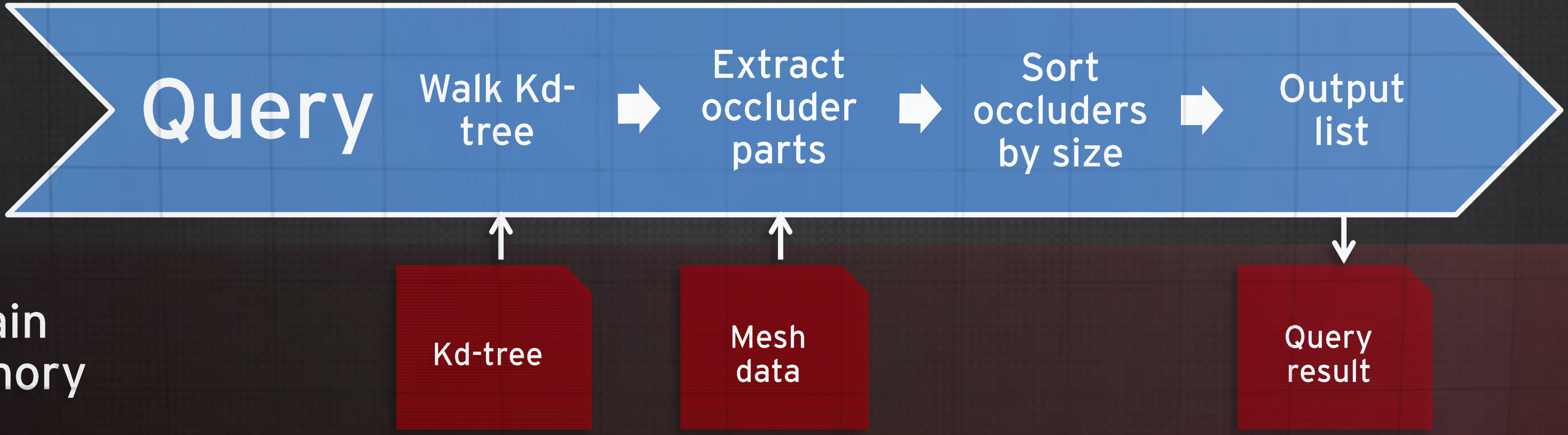
MAIN MEMORY STAGING LAYOUT

Block	Size	Count	Total
Global triangles	48 bytes	4096	192KB
DMA list entries	8 bytes	23K	184KB
Job commands			3KB
		Grand total	**379KB**

- NB:
  - We originally rasterised at 720p
  - Ended up shipping with 640x360 (see later)
  - Memory and performance figures are for this option

OCCLUDER QUERY JOB

- Finds occluders in the (truncated) view frustum
- Occluders are normal rendering primitives
  - Live with the rest of a drawable object, identified by flag bit



Query

Walk Kd-tree → Extract occluder parts → Sort occluders by size → Output list

Main memory

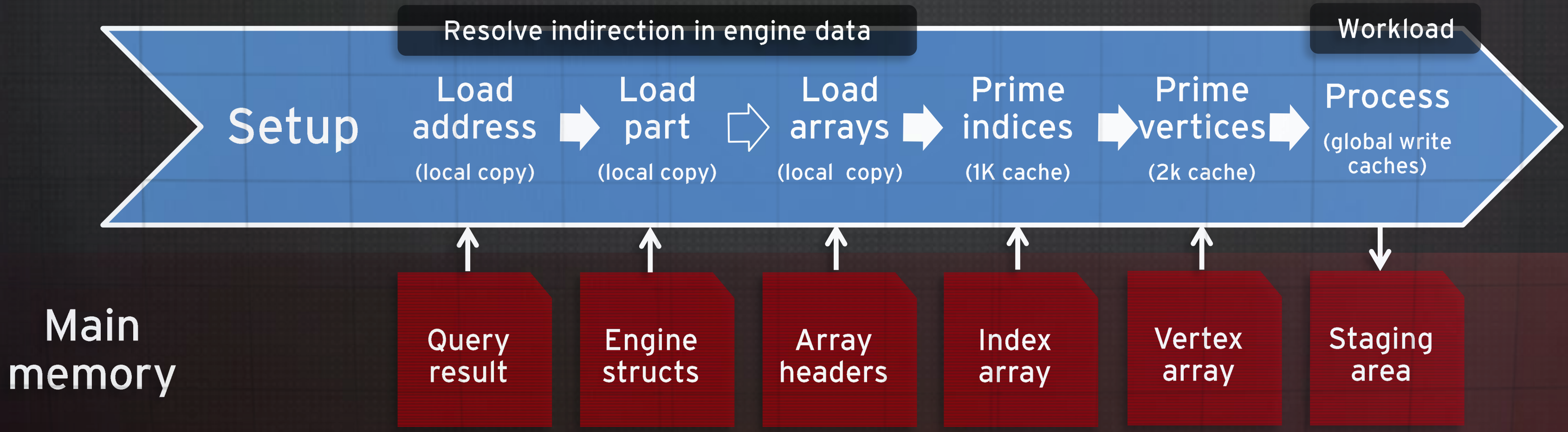
Kd-tree

Mesh data

Query result

OCCLUDER SETUP JOB

- Decodes RSX-style vertex and index arrays
- Outputs clipped + projected triangles to staging area
- Internal pipeline to hide DMA latency



SETUP: LS MEMORY LAYOUT

Block	Size	Count	Total
Triangle write cache	6KB	1	6KB
DMA list write cache	1.5KB	23	34.5KB
Index cache	1KB	6	6KB
Vertex data cache	2KB	6	12KB
Post-transform cache	~600b	6	~3.5KB
		Smaller data, alignment slop etc.	11KB
		Total data	73KB
		Stack	8KB
		Code	40KB
		Grand total	**105KB**

SETUP: LOAD DATA

- First three stages load small (bytes rather than KB) engine structs
- Index data streamed through 1K cache
  - First read pipelined, later reads block
  - Occluders diced so they usually fit in one go
- Vertex data streamed through 2K cache
  - First read pipelined again
  - 90% hit rate
- 32-entry post-transform cache
  - Direct mapped, not a FIFO
  - 60% hit rate

SETUP: DECODE AND TRANSFORM

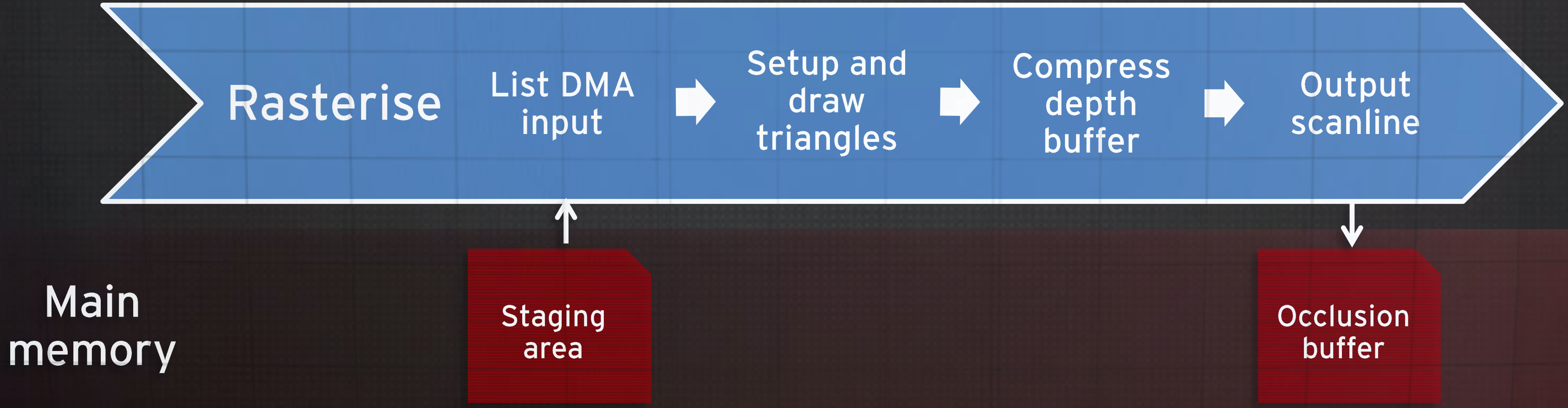
- Last stage does all the heavy lifting
- Decode vertices from 32-bit float or 16-bit integer
  - RSX formats
- No-clipping path
  - Primitive bounds lie inside frustum
  - Store projected vertices in post-transform cache
- Clipping path
  - Only when required
  - Cull/clip triangles against near and far planes
    - ‘Scissor’ test handles image extents later
  - Store clip-space vertices in post-transform cache
  - Branchless clipper

SETUP: CULL AND DISPATCH

- Cull projected triangles against image extents
- Send visible triangles to staging area in main memory
  - Store one copy of each triangle
    - via 6KB double-buffered write cache
  - Store DMA list entry for each strip under the triangle
    - via 1.5KB double-buffered write cache
    - Saves memory (8 byte entries vs. 48-byte triangles)
  - If we run out of staging space, ignore excess triangles
- Then setup and kick rasteriser jobs

RASTERISE JOB

- Launch one rasterise job per strip
- Load triangles from staging area using list DMA
- Draw triangles to a floating point 640x16 depth buffer in LS
- Compress depth buffer to uint16 and store

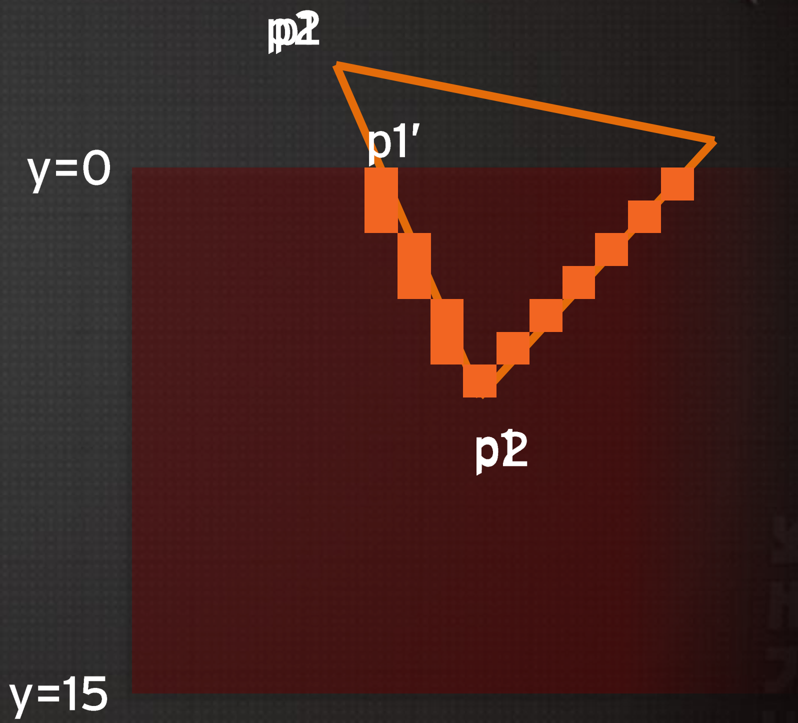


RASTERISE: LS MEMORY LAYOUT

Block	Size	Count	Total
Input triangle buffer	48KB	1	48KB
Depth buffer	20KB	1	20KB
Output scanline	<1KB	1	<1KB
		Smaller data, alignment slop etc.	~1KB
		Total data	70KB
		Stack	8KB
		Code	11KB
		Grand total	**89KB**

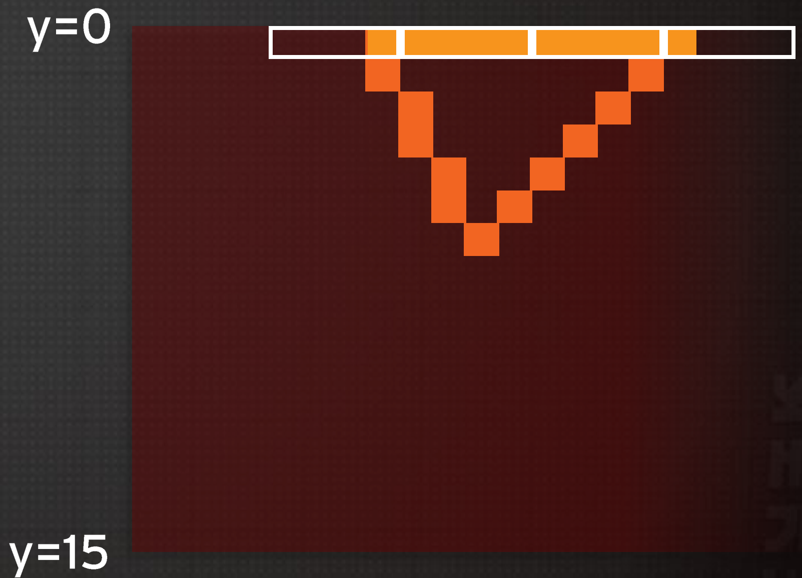
RASTERISE: TRIANGLE SETUP

- Traditional scanline rasterisation
  - Paid attention to triangle setup speed
  - Also considered half spaces
  - Not tried this yet though
- Handle four edges at once (SoA)
- Sort edges start and end on Y
  - Using shuffle table
- Clip edges to strip
- Walk edges and write X, Z pairs into 16-entry span table



RASTERISE: DRAWING SCANLINES

- Outer loop over span table
  - C-with-intrinsics
- Inner loop along scanline
  - SPA assembler
- 4-pixel-wide SIMD
  - Interpolate depth values
  - Depth test and write if nearer
  - Mask writes at start and end



RASTERISE: COMPRESS

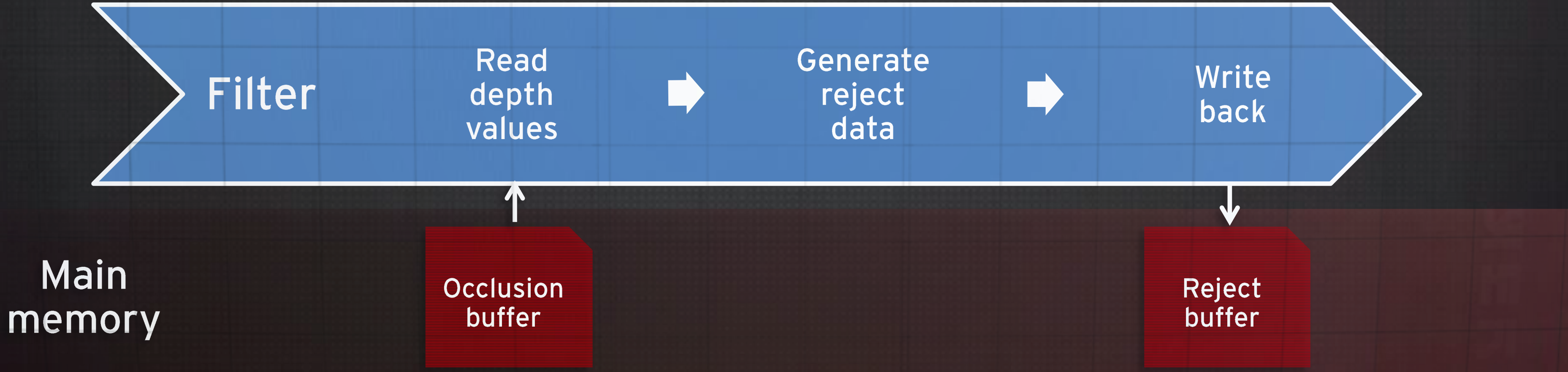
- Down-sample each 16x16 tile to 1 depth value
  - Take maximum so depth is conservative
- Encode depth values as unsigned short
  - Scale float value such that the far plane is at 0xfffe
  - Take the ceiling of the scaled value
  - Reserve 0xffff for infinity
    - Occlusion frustum is much less deep than view frustum
- Each rasterise job produces one row of the occlusion buffer (one tile high)
  - DMA row back to main memory
  - Full-size image never really exists
- Last job out kicks the filter job
  - Synchronise with simple semaphore

RASTERISE: PATCHING HOLES

- Actually we cheat a bit ☺
  - Take 2x2 minimum before 16x16 maximum down-sample
  - Patches holes cheaply – input isn't perfect
  - Otherwise a single pixel hole becomes an entire tile!

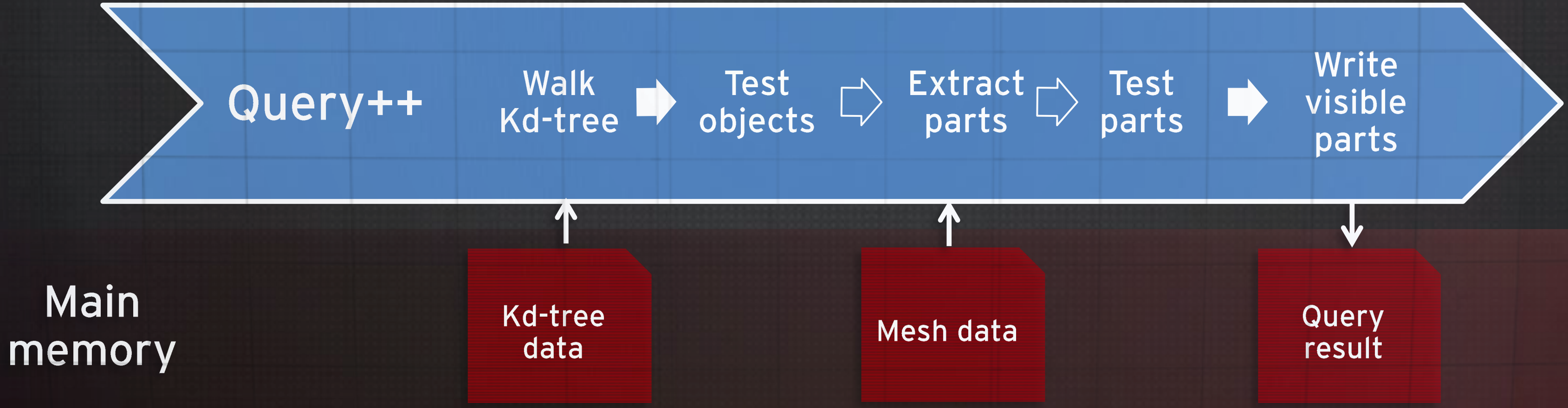
FILTER JOB

- Runs after rasterisation is complete
- Generates coarse reject data
  - Used during query to cull small objects
- Writes back to reject buffer (contiguous with occlusion buffer)



OCCLUDED QUERY JOB

- Tests work in a two-level hierarchy
  - Objects live in the Kd-tree, and contain multiple parts
- Test objects to avoid extracting parts
- Test parts to avoid drawing them
- Final query result written to main memory and used to build display list



QUERY: RASTER TESTS

- Rasterise front-facing quads of bounding box
  - At resolution of occlusion buffer (40x23)
  - Early out and return visible as soon as a scan-line passes depth test
- Same code as occluders, smaller buffer
  - Small quad optimisations important
  - Native quad support avoids setting up two triangles
- Relatively expensive
  - Rasterisation is costly
  - Times 500-1500 objects with 2000-4000 parts

QUERY: SPHERE TESTS

- Use extra levels of reject data
  - Same size as top level
  - Conservatively re-sampled
  - Test bounding spheres against this buffer if they are small enough
  - Fall back to raster test if a given object or part is too large
- Much cheaper
  - Constant time tests
- But limited
  - Only support spheres of certain radii
  - Large radius: more sphere tests, but more false positives
  - Small radius: more raster tests, but fewer false positives

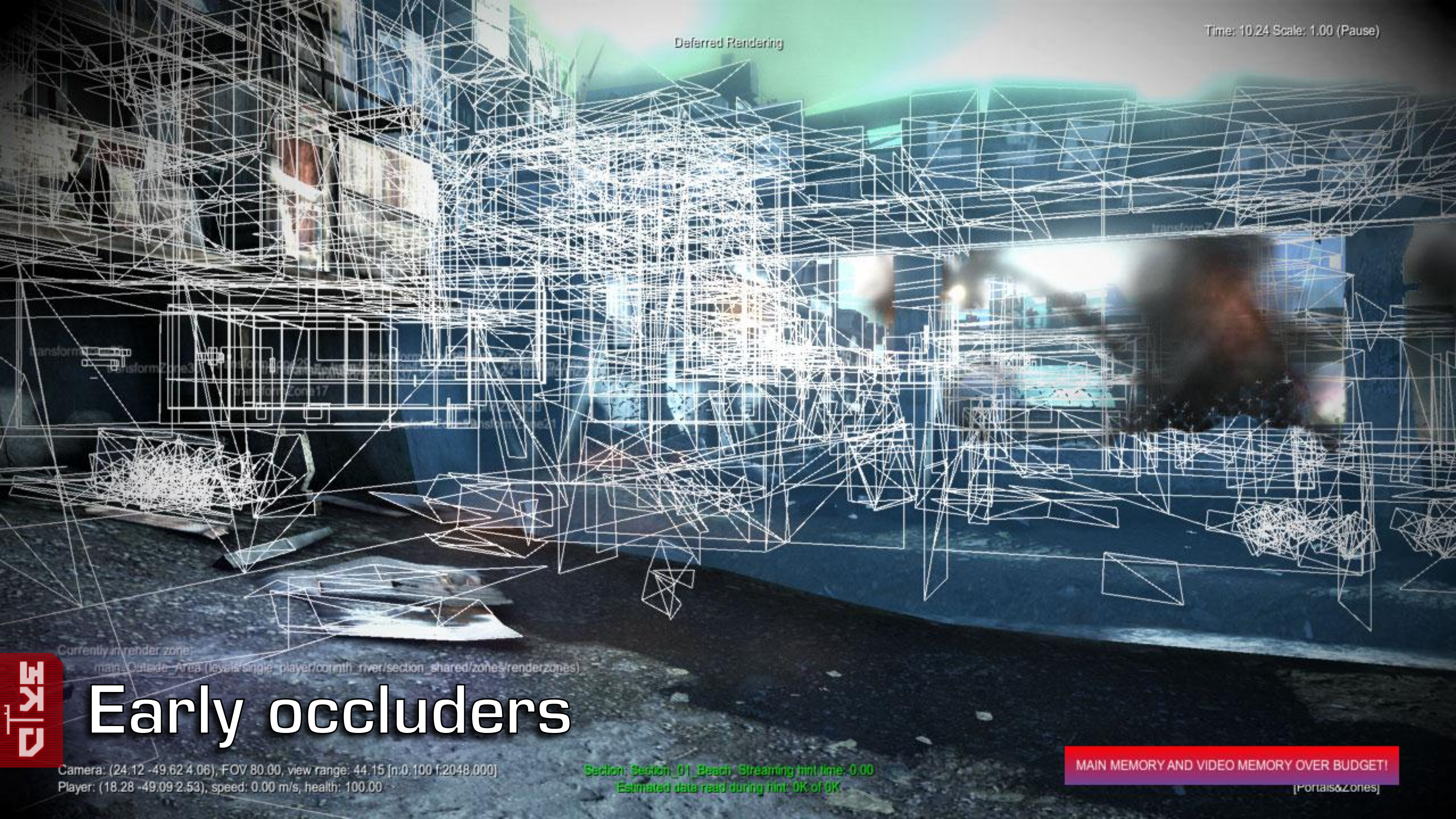
RUNTIME STATS

- Sample occlusion frame
  - Transform 100 occluders
  - With 1500 global triangles (2200 across strips)
  - Fill 500K pixels @ 640x360
  - Test 1000 objects
  - Test 2700 parts
- Timings
  - Setup job: 0.5ms
  - Rasterise job: 2.0ms (on 5 SPUs)
  - Query job: 4.5ms (on 5 SPUs) with 1.0ms doing occlusion queries
  - Overall latency: ~2ms

CREATING OCCLUDERS

- Intended to build occluders from existing visual meshes
  - Avoid creating too much data
  - Use same vertex buffer with new index buffer
  - QEMM-simplified or some other method of reduction
- Easy to get started with test data
  - Just render everything
- We used this for initial runtime development

Early occluders



Deferred Rendering

Time: 10.24 Scale: 1.00 (Pause)

Currently in render zone:
main_Outside_Area (levels/single_player/corinth_river/section_shared/zones/renderzones)

Camera: (24.12 -49.62 4.06), FOV 80.00, view range: 44.15 [n:0.100 f:2048.000]
Player: (18.28 -49.09 2.53), speed: 0.00 m/s, health: 100.00

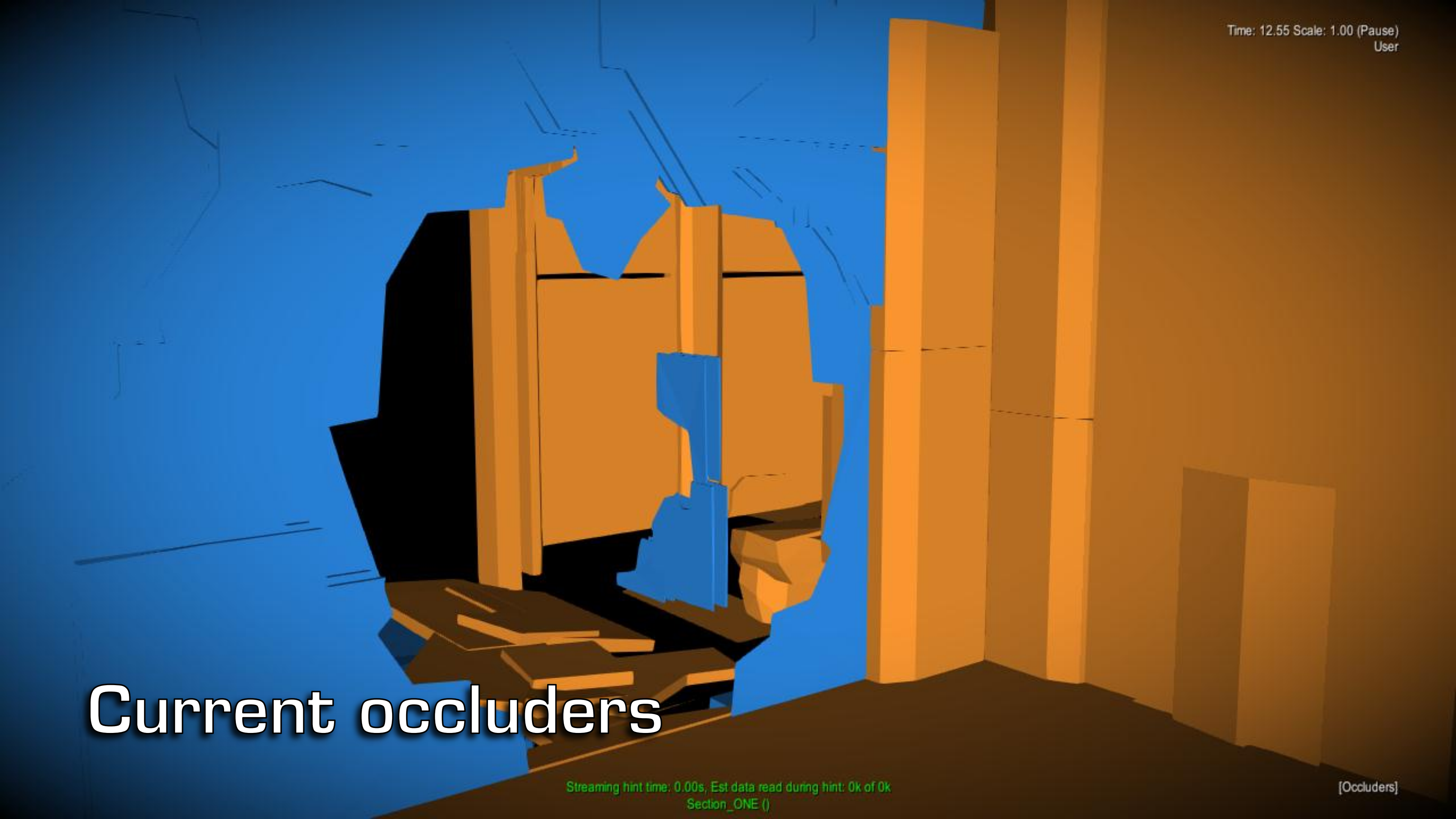
Section: Section_01_Beach, Streaming hint time: 0.00
Estimated data read during hint: 0K of 0K

MAIN MEMORY AND VIDEO MEMORY OVER BUDGET!

[Portals&Zones]

VISUAL OCCLUDERS

- Problems with this approach
  - Far too much data
  - Hard to reduce without losing silhouette
  - Not closed
    - Holes in the occlusion buffer
    - Even worse with back-face culling
- Needed something better
  - Turned out we already had a good candidate



Time: 12.55 Scale: 1.00 (Pause)
User

Current occluders

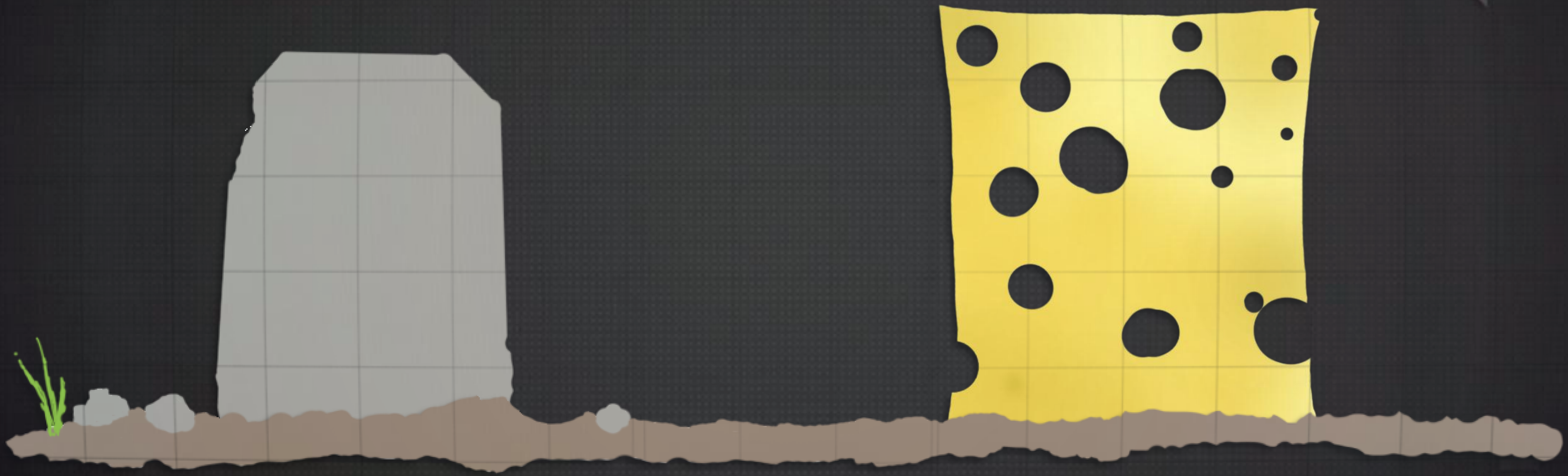
Streaming hint time: 0.00s, Est data read during hint: 0k of 0k
Section_ONE ()

[Occluders]

PHYSICS MESH OCCLUDERS

- Good properties:
  - Closed
  - Simple
- Somewhat accurate
  - We have to be able to shoot it
  - Not really accurate enough
    - Didn't realise this until quite late
- Still way too much of it
  - Small occluders
  - Holey occluders

CHOOSING THE RIGHT OCCLUDERS

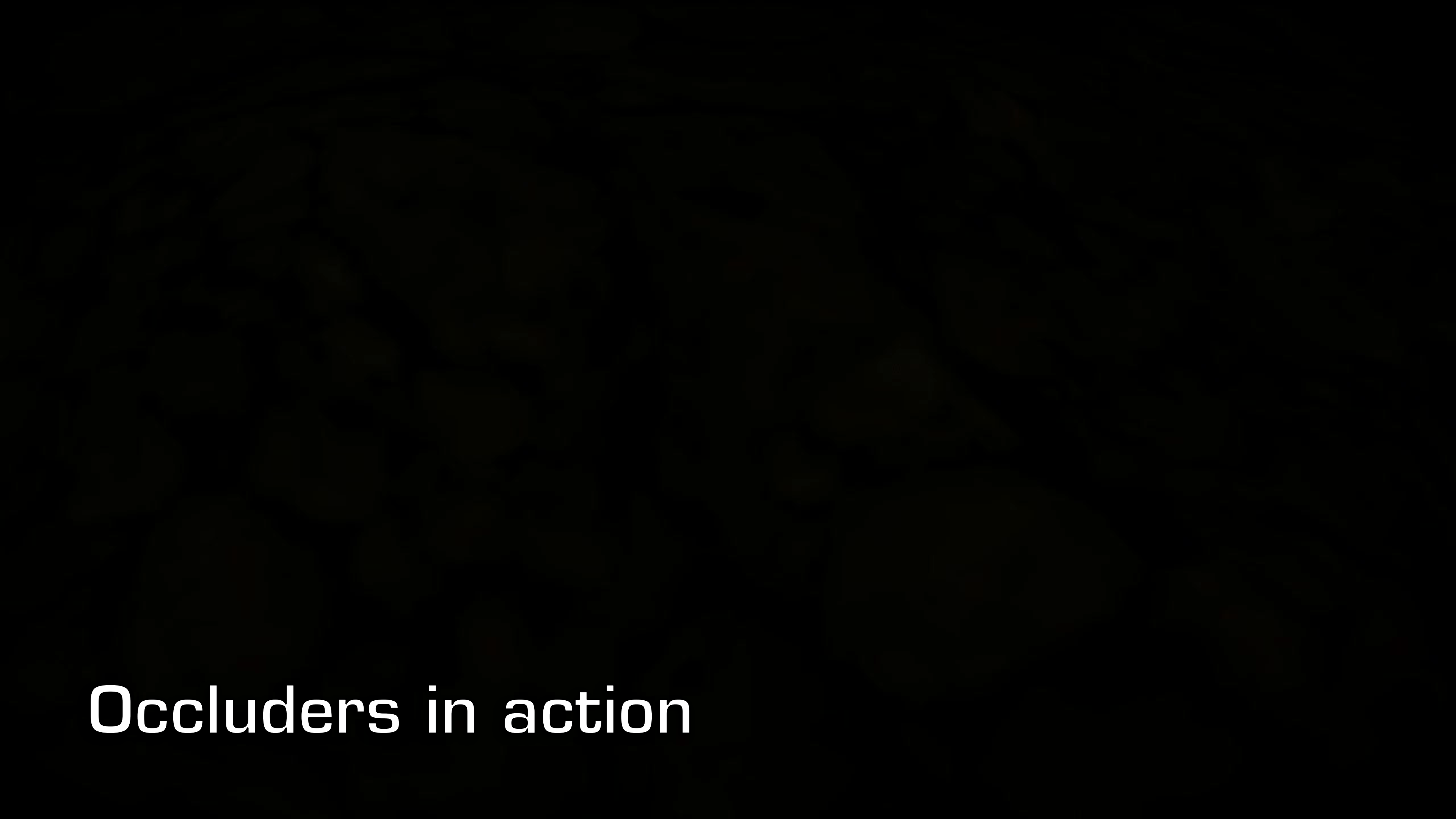


OCCLUDER HEURISTICS

- Filter based on content meta-data
  - Walls, floors, ceilings, terrain = good
  - Set dressing, props, clutter = bad
- Filter based on geometry
  - Don't create really small occluders at all
  - Compare total area of triangles against surface area of bounds
  - Throw away anything below a given threshold
    - Actually use several thresholds, based on size
- Gives us reasonable starting point

ARTIST CONTROLS

- Needed the human touch to get the best out of the system
- Tagging
  - Never, Always, or Auto (in which case let heuristics decide)
- Custom occluders
  - Artists can provide their own meshes
  - Give these priority at runtime during sort
  - Often just 2-sided quads, or boxes
  - Cheap to author, cheap to render



Occluders in action

```
c70 | statis
ves  = 314
memory used (KB) = 285
x buffer memory used (KB) = 131
ex buffer memory used (KB) = 128
OAT32 vertices (KB) = 47
NT16 vertices (KB) = 81
ustom primitives  = 59
ustom primitive memory (KB) = 69
me (us) = 474
Primitives = 101
Primitives area culled = 73
Primitive triangles = 1495
: Primitives clipped = 7
d: Triangles transformed = 1495
d: Triangles near/far culled = 241
nd: Triangles clipped = 58
nd: Triangles projected = 1180
end: Triangles area/backface culled = 101
end: Triangles extent culled = 263
-end: Triangles discarded (global overflow) = 0
t-end: Triangles discarded (bin overflow) = 0
nt-end: Triangles written to main memory = 917
ont-end: Triangles written to bin DMA lists = 2253
ont-end: Vertices = 4485
ront-end: Vertices transformed (vertex cache misses) = 2061
Front-end: Vertex buffer loads (data cache misses) = 33
Front-end: Vertex buffer KB = 23
Front-end: Index buffer loads = 28
Front-end: Index buffer KB = 8
Front-end: Time (us) = 1975
Back-end: Bins = 23
Back-end: Triangles = 2253
Back-end: Spanning triangles = 0
Back-end: Spanning edges = 0
Back-end: Short edges = 0
nd:                   17623
```

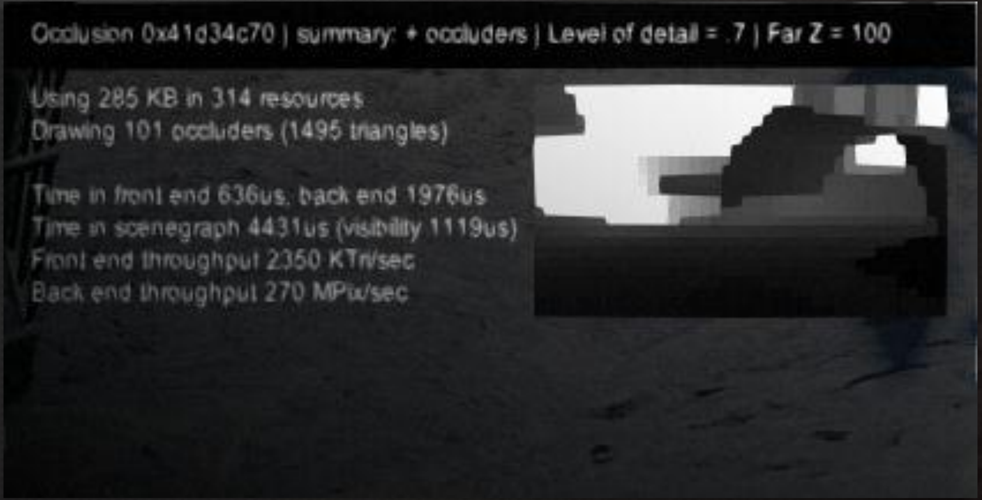
DEBUG TOOLS

HOW DO WE SEE THIS?

- Testing a system like this is hard
- If you did a good job…
  - …then there are no visible results ☹
- Makes it hard to prove you've done any work…

COMMON TOOLS

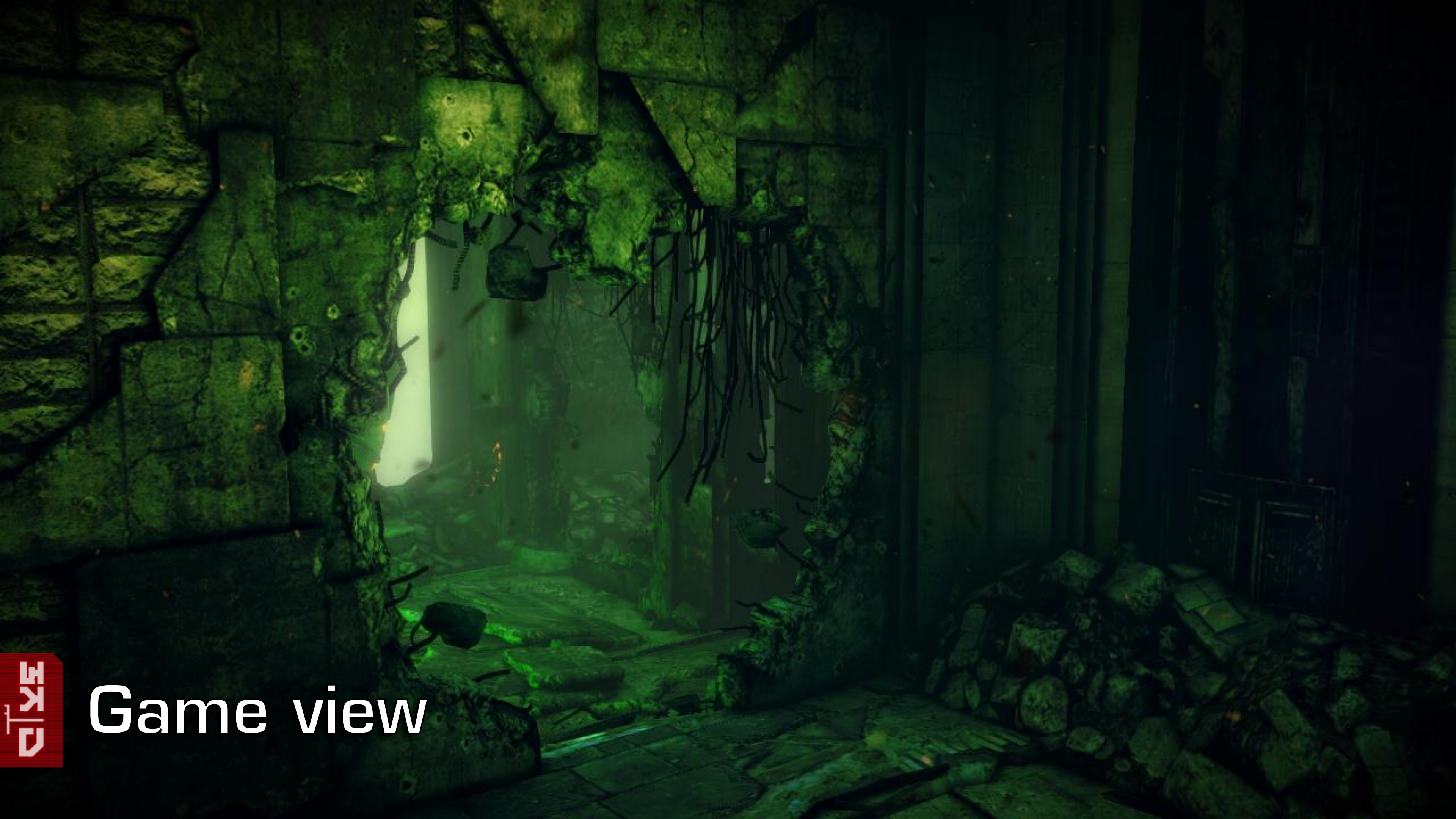
- Three main modes
- Common display:
  - Occlusion buffer
  - Timings, stats, warnings
- Can take screen shot of occlusion buffer
  - For debugging the rasteriser



```
Occlusion 0x41d34c70 | summary: + occluders | Level of detail = .7 | Far Z = 100

Using 285 KB in 314 resources
Drawing 101 occluders (1495 triangles)

Time in front end 636us, back end 1976us
Time in scenegraph 4431us (visibility 1119us)
Front end throughput 2350 KTri/sec
Back end throughput 270 MPix/sec
```

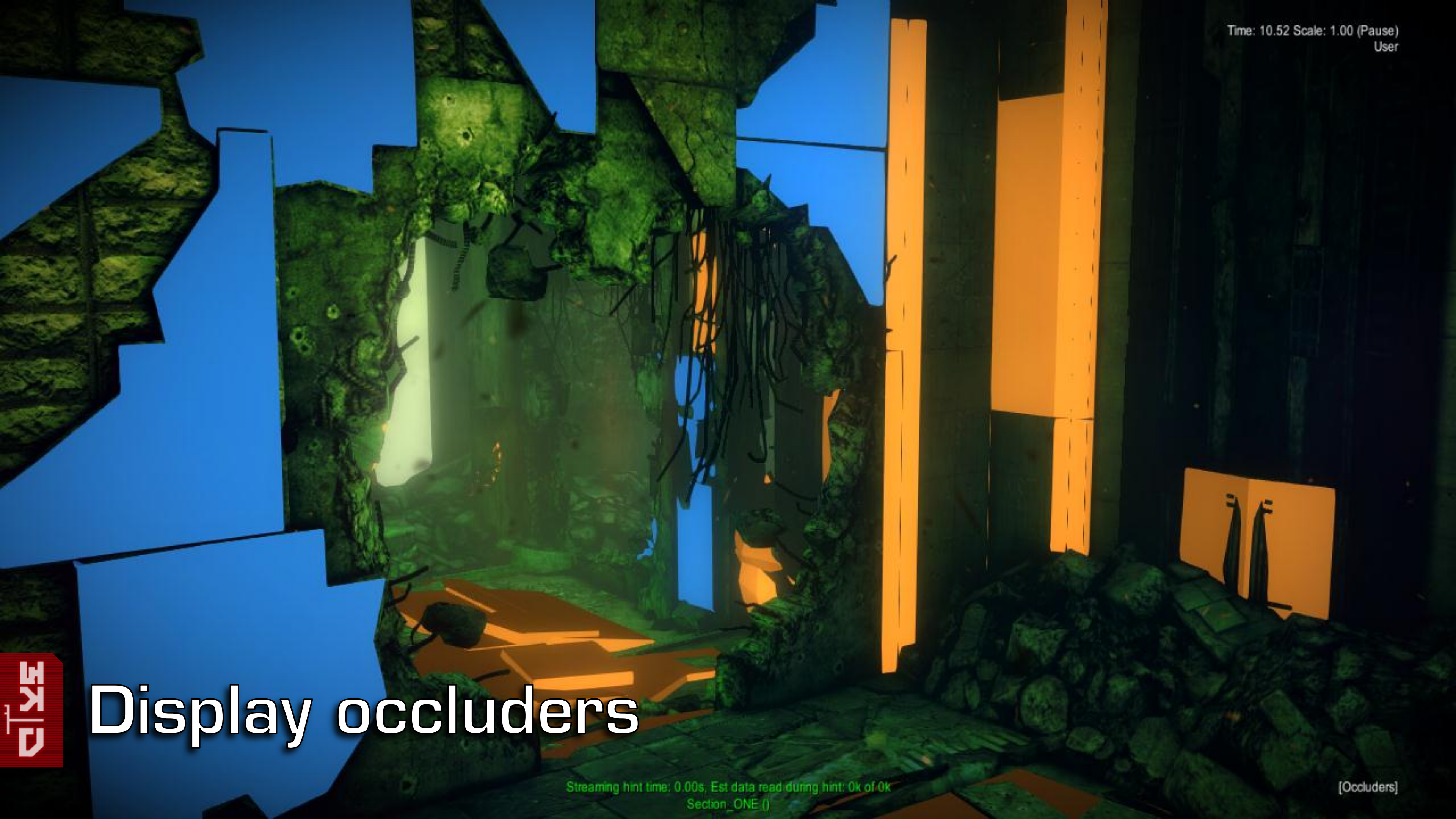
DISPLAY OCCLUDERS

- Lit depth-tested translucent geometry
  - Cheap, since we're rendering from hardware-style vertex and index buffers.
- Colour coding
  - Important/custom
  - Active/inactive
  - Overflows
- Important mode for artists
  - Did my custom occluder work?



Game view

Display occluders

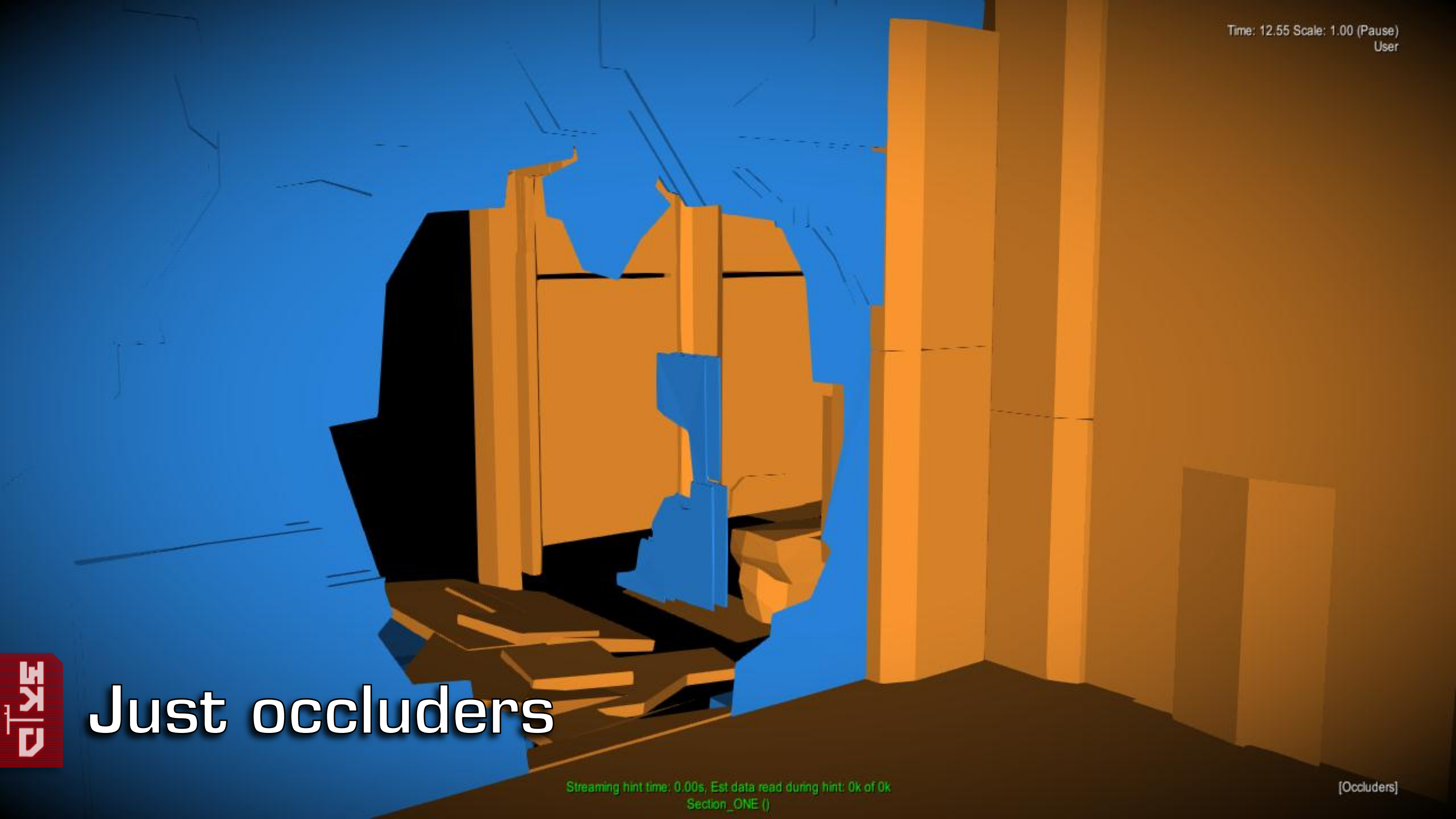


```
Time: 10.52 Scale: 1.00 (Pause)
User
```

```
Streaming hint time: 0.00s, Est data read during hint: 0k of 0k
Section_ONE ()
```

```
[Occluders]
```

Just occluders



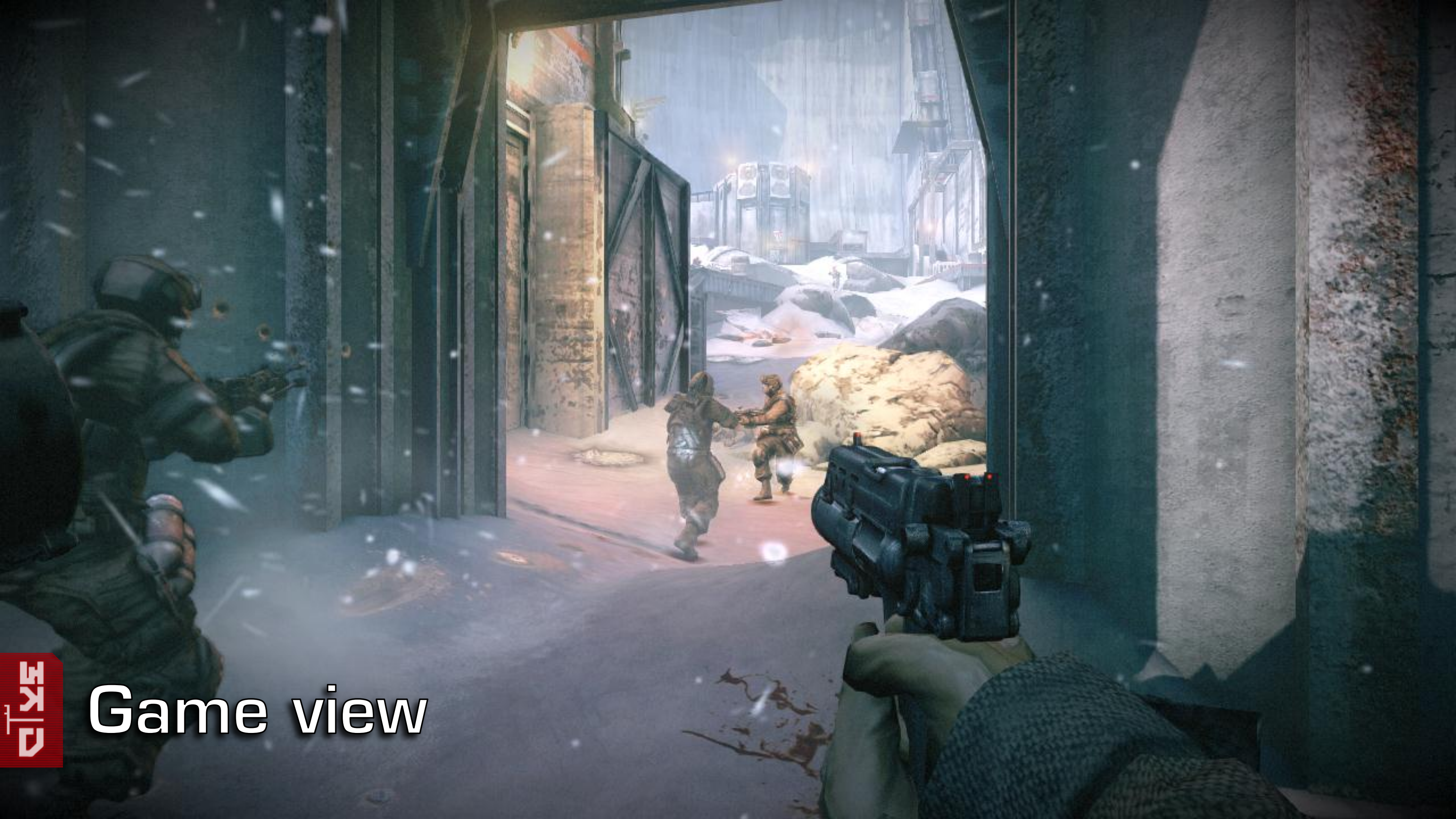
Time: 12.55 Scale: 1.00 (Pause)
User

Streaming hint time: 0.00s, Est data read during hint: 0k of 0k
Section_ONE ()

[Occluders]

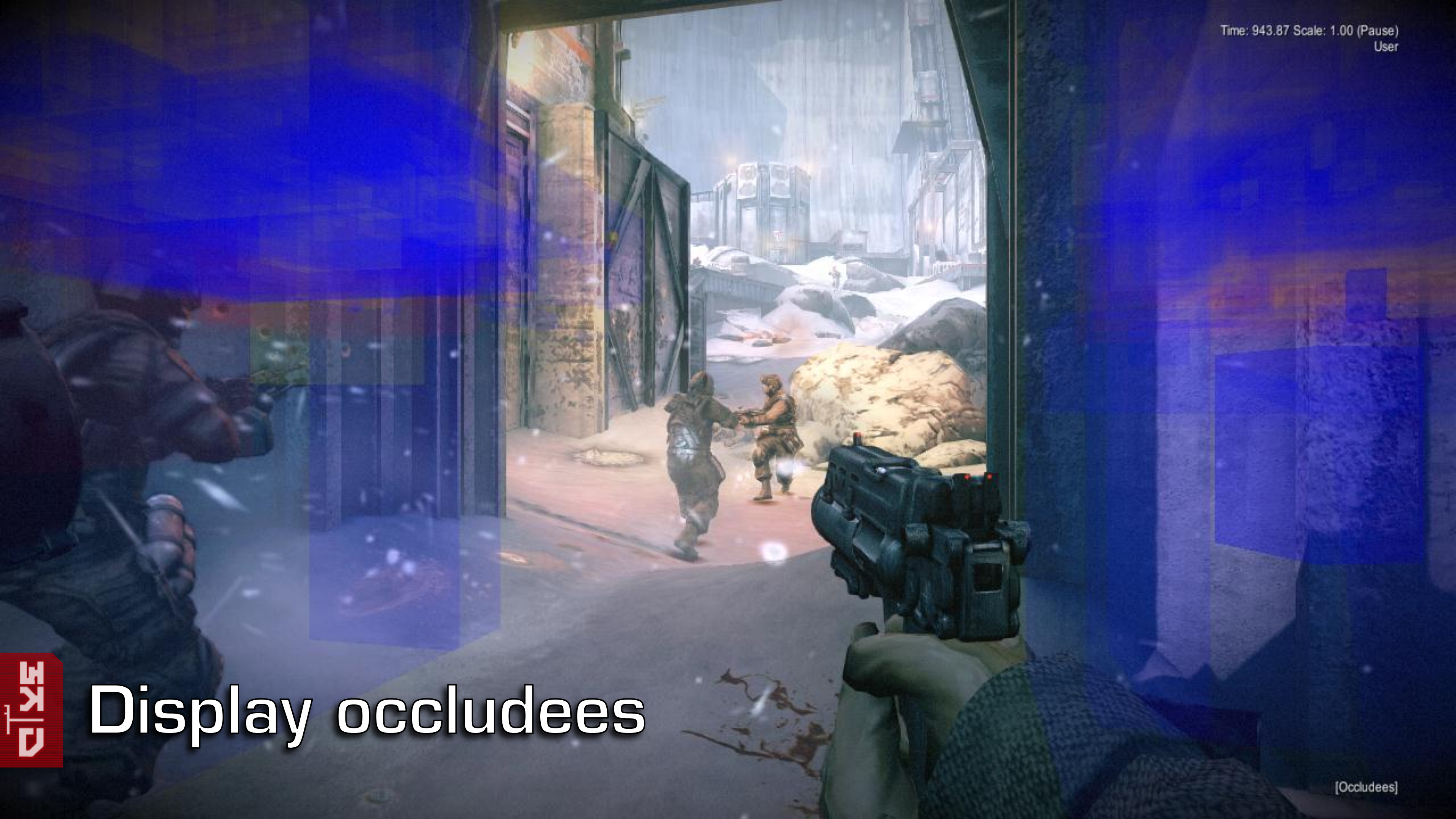
DISPLAY OCCLUDEES

- Draw bounding boxes when objects are culled out
  - Drawing the not-drawn stuff
- Colour coded by type
  - Objects, parts, lights, etc.
- Good for demos
  - “Wow, look at all that stuff I can't see!”
  - And rough performance tuning



Game view

Display occludees



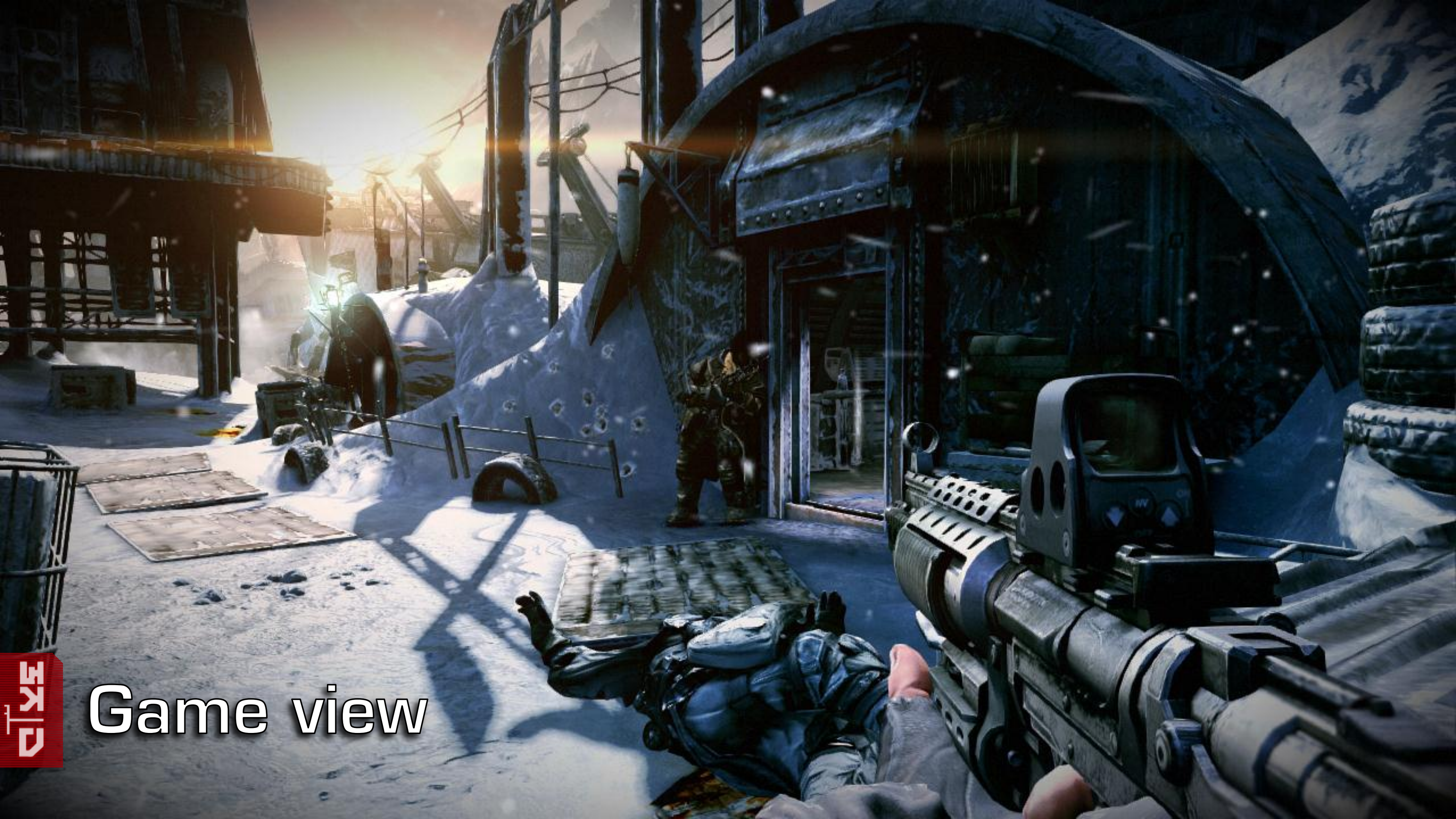
Time: 943.87 Scale: 1.00 (Pause)
User

[Occludees]

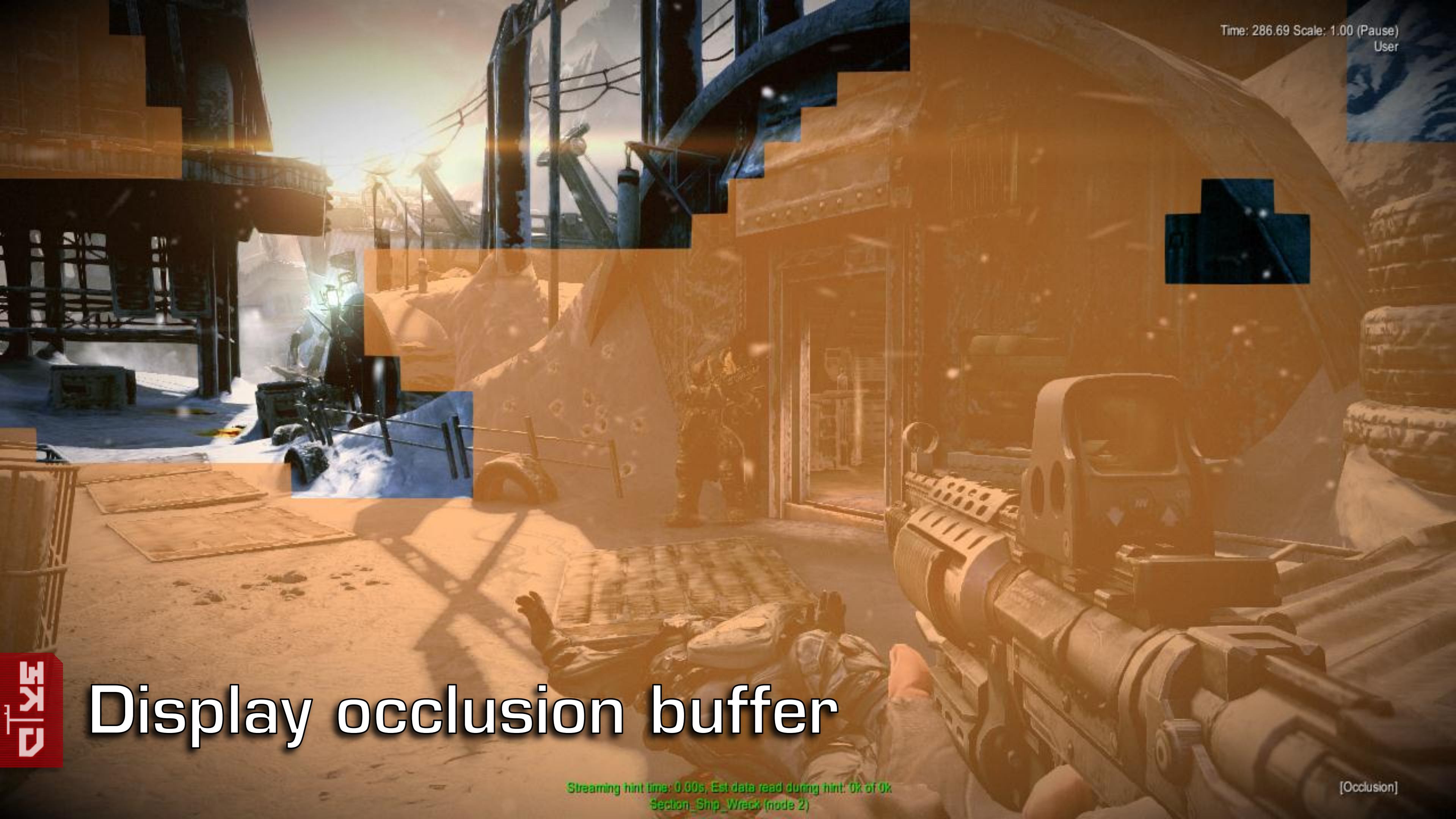
DISPLAY BUFFER

- Draw the occlusion buffer tiles on the screen
- Transparency-coded for depth
- Good for checking conservatism is working right
- Also used by artists when checking for occlusion leaks

Game view



Display occlusion buffer

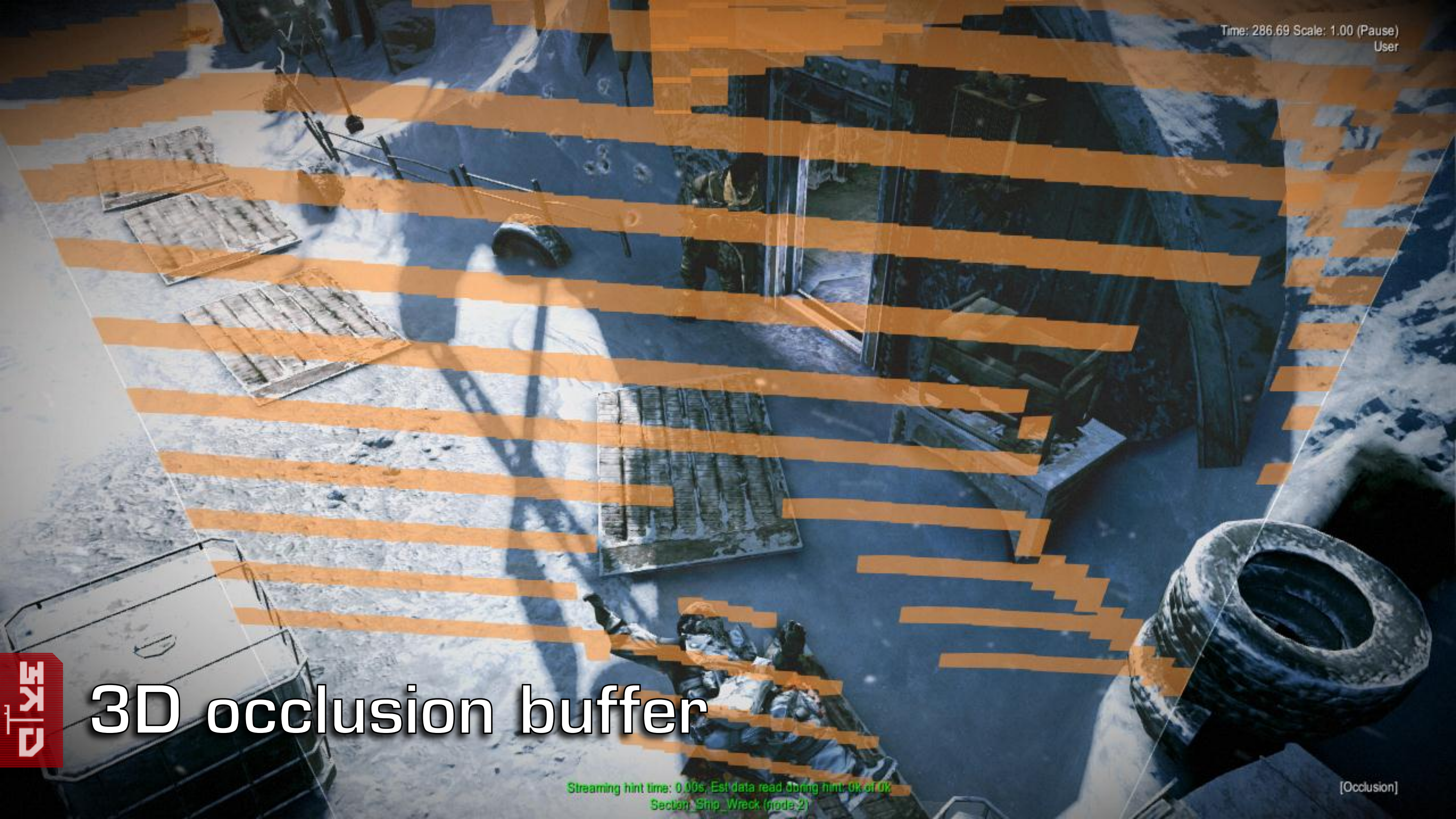


Time: 286.69 Scale: 1.00 (Pause)
User

Streaming hint time: 0.00s, Est data read during hint: 0k of 0k
Section_Ship_Wreck (node 2)

[Occlusion]

3D occlusion buffer



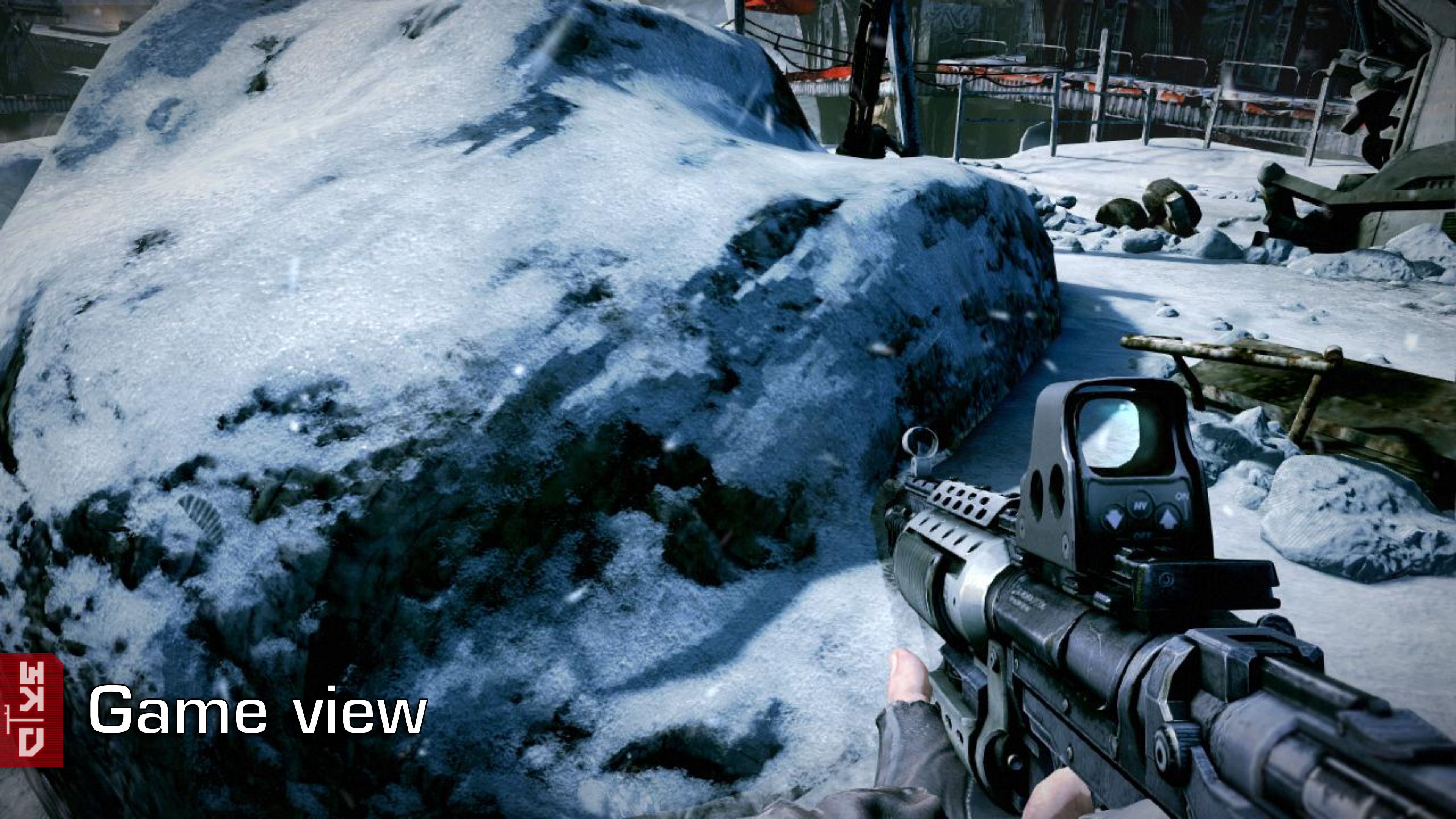
Time: 286.69 Scale: 1.00 (Pause)
User

Streaming hint time: 0.00s, Est data read during hint: 0k of 0k
Section: Ship_Wreck (node 2)

[Occlusion]

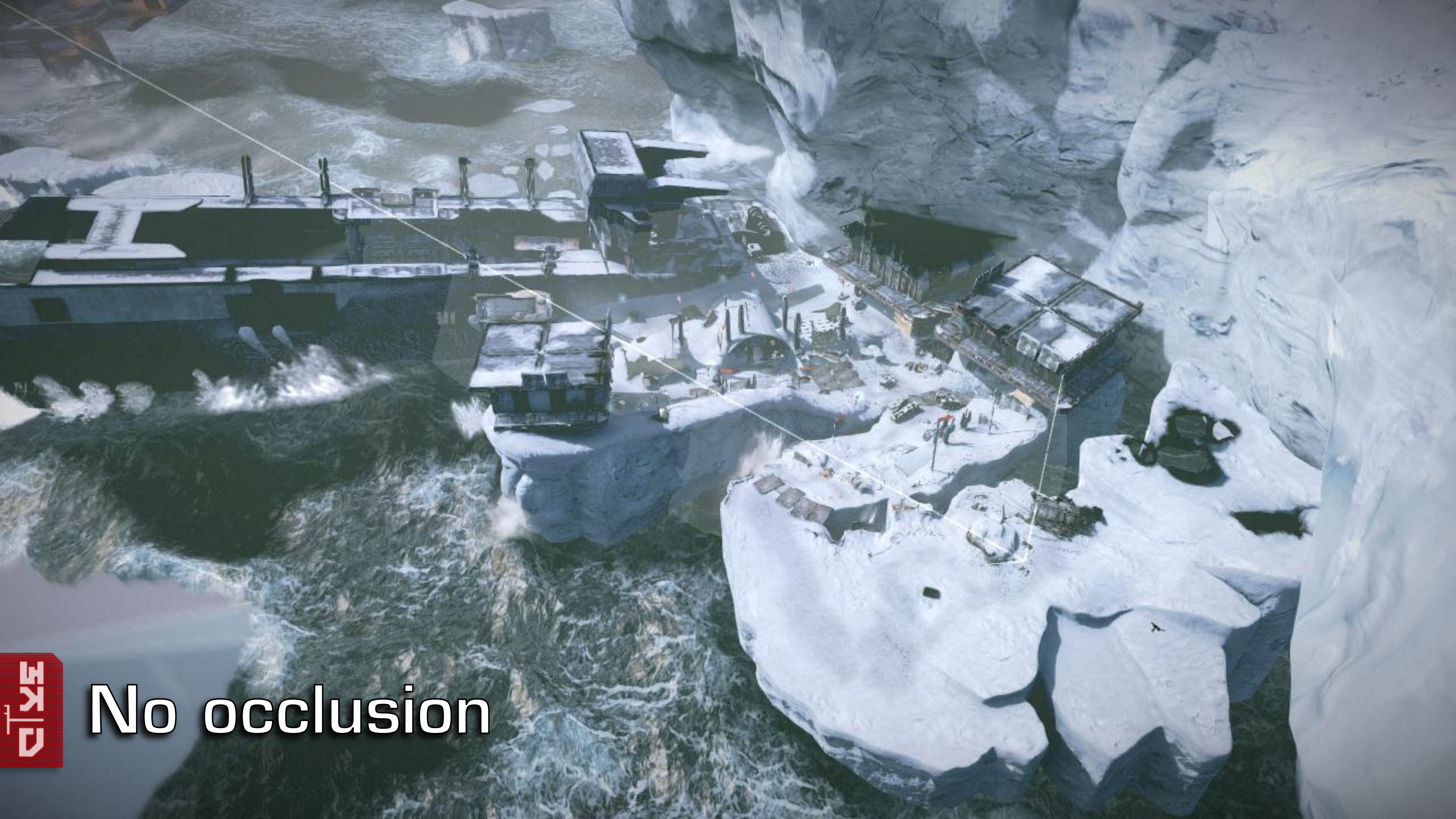
DEBUG MODE 9

- Existing scene debug tool
  - Uses player camera for the frustum
  - But debug camera for the renderer
- Very useful for testing culling
  - Frustum and occlusion
- All games should have this!

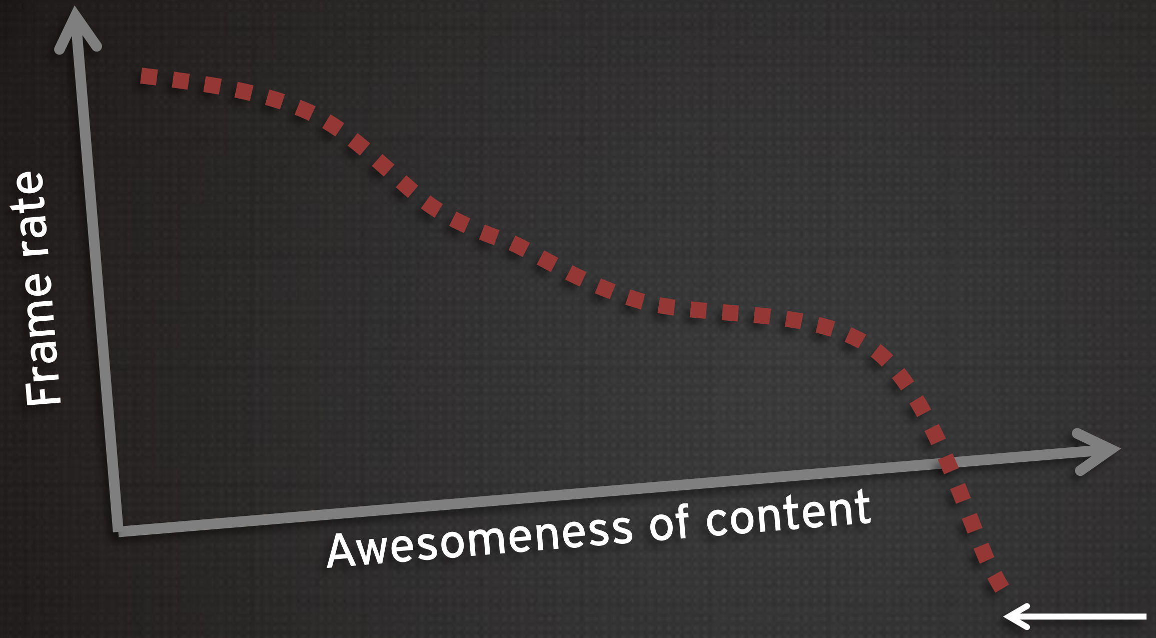


Game view

Debug 9



No occlusion



Might get fired here...

PERFORMANCE ANXIETY

IMPROVING THE CULL RATE

- Initially tried a bit too hard to get good SPU performance
  - Used sphere tests only for some situations
  - So we didn't get enough RSX culling
- Had to make sure we do full tests on all objects
  - Makes optimisation even more important

EASY OPTIMISATION

- Rasterisation and testing too expensive
  - Both largely bound by fill rate
  - Already worked on triangle setup
- 720p occlusion buffer was a headache…
- But it did provide a lot of headroom:
  - Downsize to 360p – 40x23 tiles rather than 80x45
  - Instant fourfold speedup!
- Possible artifacts
  - Already have 2 pixel artifacts from gap removal – this makes them 4 pixel
  - Still quite small ☺
- Checked it in quietly…

MORE DIFFICULT

- Other significant optimisations
  - Temporal coherence
    - Assume an object is visible if it was last frame
    - Use feedback from testing parts to update status
  - Split query job into serial and parallel parts
    - Serial: Walk Kd-Tree and generate object list
    - Parallel: Test objects and extract parts
  - Sub-mesh culling
    - Use spatial Kd-Tree inside a mesh to cull it in arbitrarily small pieces
    - Keep slicing and testing until we get to the break-even point

LONG THIN OBJECTS

- We had some specific problems with skinny objects
  - Radius too large for sphere tests
  - Not enough pixels generated during raster tests for reliable result
- Didn't want to use conservative rasterisation
  - Too risky too late
  - Would have made all objects bigger, but we had only a few problem objects
- Test a diagonal of each quad
  - If the diagonal is visible, the quad is visible
  - If not, test the quad
- Fixed the skinny objects, plus it saved some time overall

CONCLUSION

COSTS

- Code production costs
  - Two man-months to get initial implementation running
  - One month to switch to physics meshes and make everything robust for production
  - Two months bug fixing and optimisation
- Runtime costs
  - ~400KB scratch memory per view
  - 20-50% of one SPU (for everything)

RESULTS

- Pros
  - Relatively fast, with room to optimise further
  - Stable and predictable performance
  - Completely dynamic
  - Excellent occluder fusion
- Cons
  - Still needed artist work to get levels running fast enough
  - But easier to do than with old system
  - Campaign levels: 5-10% of occluders custom made
  - Multi player levels: 20-50% of occluders custom made ☹
  - Should have anticipated this earlier and changed workflow to suit

QUESTIONS?

- **will@secondintention.com**
- Slides will be in the GDC Vault
- Also on **http://www.secondintention.com/**

THE END

- Thanks for listening!
  - Thanks to everyone at Guerrilla for being awesome!
  - Special thanks to Incognito for their sample code

BONUS SLIDE: FUTURE WORK

- Tame software rasteriser = opportunity
- Rendering AI depth cubemaps
  - Combined with caching and offline generation
  - Use idle time
  - Reasoning in dynamic worlds!
  - Experiments are promising...
- Rendering sun occlusion
  - For really cool glares
  - Use idle time
  - Like the speeder bike sequence in Jedi

BONUS SLIDE: CHC DETAILS

- Reduce the number of tests
  - CHC - Wimmer + Bittner - GPU gems 2 Ch. 6
  - Much easier without the GPU involved
  - Visible last frame?
    - Assume visible this frame, and skip the test
  - Update assumption based on low level tests
    - Usually use GPU feedback for this
    - We use feedback from part tests instead

BONUS SLIDE: PARALLEL QUERY

- Parallelise scenegraph job
  - Kd-Tree part is fast but complicated - serial
  - Extracting parts from objects is slow but dumb - parallel
  - Works really well
  - Same IO as usual
  - Output into array without locks using atomic reservations
  - Lots more space in LS for each job
    - Bigger caches

BONUS SLIDE: SPU CODE MAKEUP

- Job code starts as C/C++
- Optimise when we have real data
- Structural code stays C/C++
  - Lots of C style
  - Mix in intrinsics where necessary
    - To get SIMD
    - To get to instruction set
  - Can be 10x faster
- SPA for inner loops
  - 2x faster again