GDC
09 learn network inspire

www.GDConf.com

Game Developers Conference®

March 23-27, 2009 | Moscone Center, San Francisco

The PlayStation®3's SPUs in the Real World

A KILLZONE 2 Case Study

Michiel van der Leeuw

Technical Director - Guerrilla Games

PLEASE MAKE A SELECTION 001 KZ

TAKEAWAY

- Things we did on SPUs
- What worked or didn't work for us
- Practical advice on using SPUs
- Some food for thought

- Firstly a post-mortem
- Talk about what we did and if we liked it
- I'll try to add lots of numbers, because I like it in other presentations

- Will try to put this into perspective
- So you can apply this to your own game

KILLZONE 2

- Announced E3 2005
- Gold Master end 2008

- ~ 120 Average team size
- ~ 27 Programmers

Team:
- Peak 140 in Amsterdam, 50 at other Sony studios
- Average around 120

Coders (peak):
- 5 Network coders
- 9 Game coders
- 6 AI coders
- 7 Tech coders
- 27 Total

KILLZONE 2

Franchise and engine goals:

- Cinematic
- Dense
- Realistic
- Intense

Can SPU's help?

Cinematic:
- Movie-like quality in everything (animation, special effects)
- Make people forget it's a video game

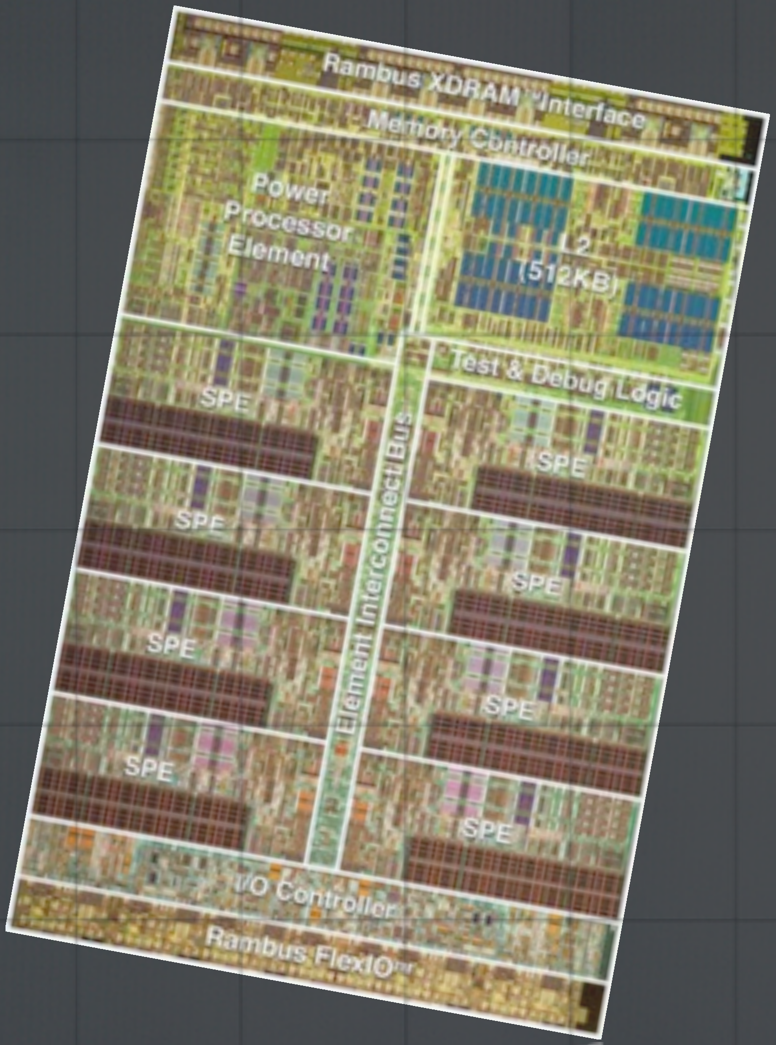
Dense:
- Dense, lots of everything
- Over-the-top

Realistic:
- Not just from a graphics point of view
- Responds to player actions
- AI behaves logically

Intense:
- Frantic, lots of everything (see a pattern)

SPUs OVERVIEW

- 6 x 3.2GHz of processing power
- General purpose instruction set
- 256KB local memory per SPU
  - No caches! Only local code
- Very fast DMA in/out
- Libraries for scheduling



Problems:
- Hard to program, especially at the start
- Lack of instruction of data and instruction cache difficult
- Basically impossible to use as a ‘normal’ core out of the box

Virtues:
- Truly parallel
- Incredibly incredibly fast
- DMA allows hiding of *all* memory latency

Difficult to unlock the power

KILLZONE 2 SPU USAGE

002. KZ

PLEASE MAKE A SELECTION 001 KZ

KILLZONE 2 SPU USAGE

- Animation
- AI Threat prediction
- AI Line of fire
- AI Obstacle avoidance
- Collision detection
- Physics
- Particle simulation
- Particle rendering
- Scene graph
- Display list building
- IBL Light probes
- Graphics post-processing
- Dynamic music system
- Skinning
- MP3 Decompression
- Edge Zlib
- etc.

44 Job types

KILLZONE 2 SPU USAGE

- Per frame
  - ~ 100 physics objects simulated
  - ~ 600 image-based light probes
  - ~ 250 rays cast per frame
  - ~ 200 animations sampled
  - ~ 250 particle systems active
  - ~ 6 scene graph traversals
  - ~ 2000 graphics objects drawn
  - ~ 75 skinning batches

Notes:
- Started out with simple stuff (skinning, particle vertices)
- Ditched fluid sim running on SPUs (artistic reasons, couldn't get it to look right)

Next up:
- Movie from game

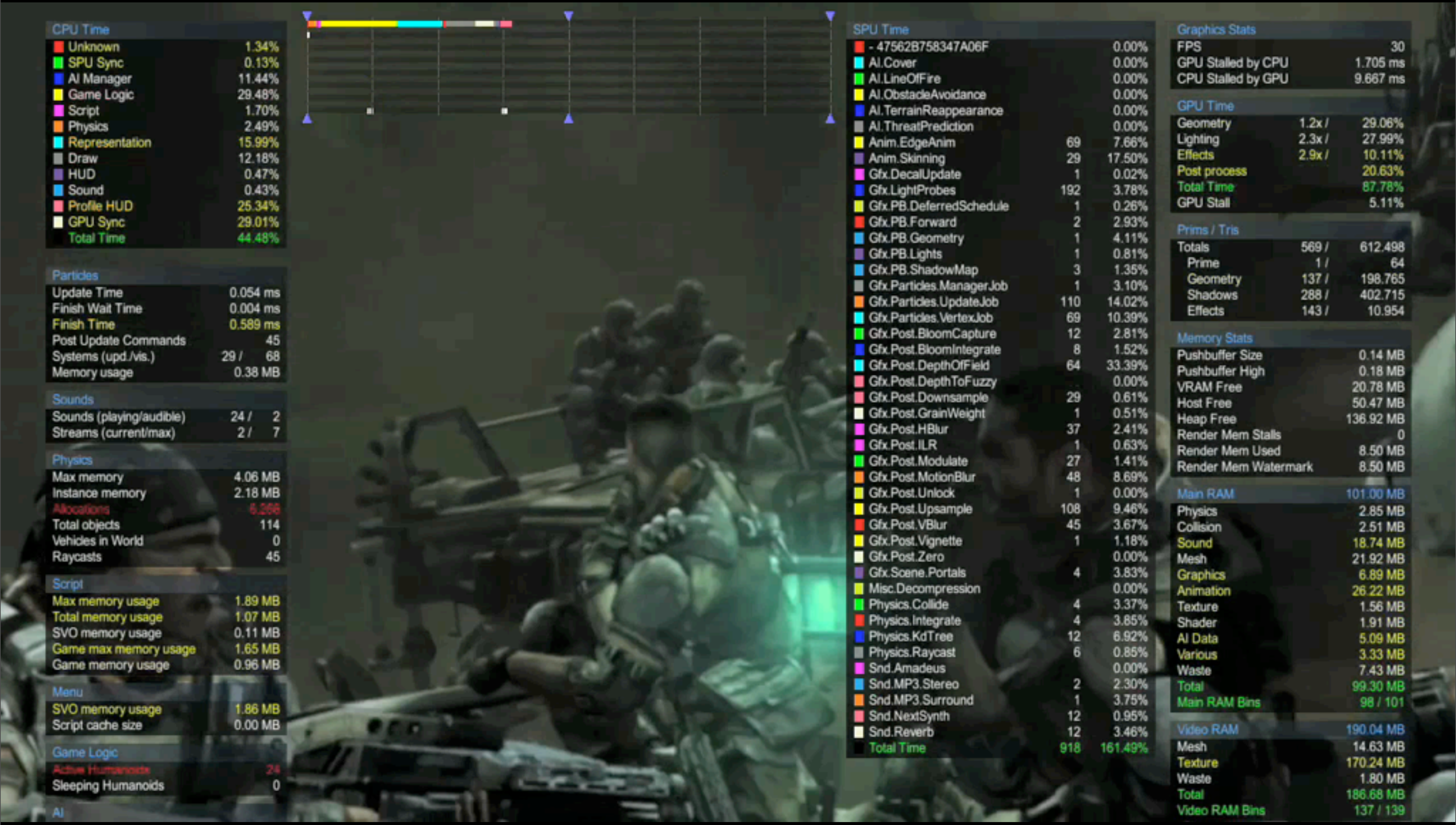
KILLZONE 2 SPU USAGE

Killzone 2 - SPU usage in cut scene

On-screen:
- Left side = PPU and SPU activity bars
    - Top is PPU
    - Below is six SPUs (Flickering due to bug in my code)
- Right side = SPU activity by job
    - Column 1: Name
    - Column 2: Job count
    - Column 3: Time taken in percentage of one SPU on a 30 Hz frame

Usage:
- Lots of different things
- Lots of jobs, about 1000
- This is still light usage, 1.5 SPUs used
    - Cut-scene
    - No AI or gameplay
    - Normally about 4 to 5 SPUs used

KILLZONE 2 SPU USAGE

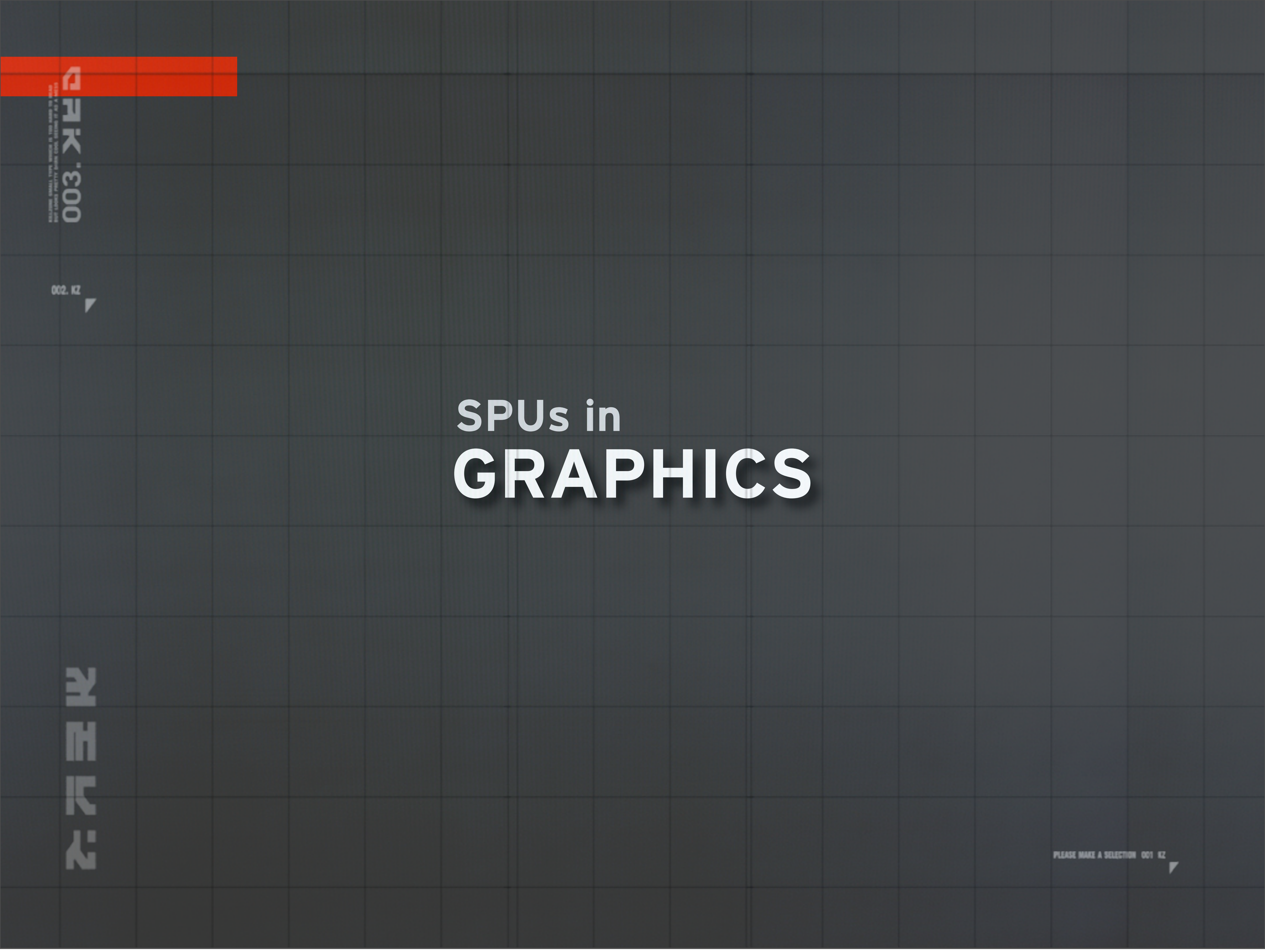


```
CPU Time
🟥 Unknown                      1.34%
🟩 SPU Sync                     0.13%
🟦 AI Manager                  11.44%
🟨 Game Logic                  29.48%
🟪 Script                       1.70%
🟧 Physics                      2.49%
🟦 Representation              15.99%
⬜ Draw                        12.18%
🟪 HUD                          0.47%
🟦 Sound                        0.43%
🟥 Profile HUD                 25.34%
⬜ GPU Sync                    29.01%
   Total Time                  44.48%

Particles
Update Time                  0.054 ms
Finish Wait Time             0.004 ms
Finish Time                  0.589 ms
Post Update Commands               45
Systems (upd./vis.)        29 /    68
Memory usage                  0.38 MB

Sounds
Sounds (playing/audible)   24 /     2
Streams (current/max)       2 /     7

Physics
Max memory                    4.06 MB
Instance memory               2.18 MB
Allocations                     6,266
Total objects                     114
Vehicles in World                   0
Raycasts                           45

Script
Max memory usage              1.89 MB
Total memory usage            1.07 MB
SVO memory usage              0.11 MB
Game max memory usage         1.65 MB
Game memory usage             0.96 MB

Menu
SVO memory usage              1.86 MB
Script cache size             0.00 MB

Game Logic
Active Humanoids                   24
Sleeping Humanoids                  0

AI
```

```
SPU Time
🟥 - 47562B758347A06F                    0.00%
🟦 AI.Cover                              0.00%
🟩 AI.LineOfFire                         0.00%
🟨 AI.ObstacleAvoidance                  0.00%
🟦 AI.TerrainReappearance                0.00%
⬜ AI.ThreatPrediction                   0.00%
🟨 Anim.EdgeAnim                  69     7.66%
🟪 Anim.Skinning                  29    17.50%
🟪 Gfx.DecalUpdate                 1     0.02%
🟦 Gfx.LightProbes               192     3.78%
🟨 Gfx.PB.DeferredSchedule         1     0.26%
🟥 Gfx.PB.Forward                  2     2.93%
🟦 Gfx.PB.Geometry                 1     4.11%
🟪 Gfx.PB.Lights                   1     0.81%
🟦 Gfx.PB.ShadowMap                3     1.35%
⬜ Gfx.Particles.ManagerJob        1     3.10%
🟧 Gfx.Particles.UpdateJob       110    14.02%
🟦 Gfx.Particles.VertexJob        69    10.39%
🟩 Gfx.Post.BloomCapture          12     2.81%
🟦 Gfx.Post.BloomIntegrate         8     1.52%
🟦 Gfx.Post.DepthOfField          64    33.39%
🟥 Gfx.Post.DepthToFuzzy                 0.00%
🟥 Gfx.Post.Downsample            29     0.61%
⬜ Gfx.Post.GrainWeight            1     0.51%
🟪 Gfx.Post.HBlur                 37     2.41%
🟪 Gfx.Post.ILR                    1     0.63%
🟩 Gfx.Post.Modulate              27     1.41%
🟧 Gfx.Post.MotionBlur            48     8.69%
🟨 Gfx.Post.Unlock                 1     0.00%
🟨 Gfx.Post.Upsample             108     9.46%
🟥 Gfx.Post.VBlur                 45     3.67%
🟨 Gfx.Post.Vignette               1     1.18%
⬜ Gfx.Post.Zero                         0.00%
🟪 Gfx.Scene.Portals               4     3.83%
🟨 Misc.Decompression                    0.00%
🟩 Physics.Collide                 4     3.37%
🟥 Physics.Integrate               4     3.85%
🟦 Physics.KdTree                 12     6.92%
⬜ Physics.Raycast                 6     0.85%
🟪 Snd.Amadeus                           0.00%
🟦 Snd.MP3.Stereo                  2     2.30%
🟧 Snd.MP3.Surround                1     3.75%
🟥 Snd.NextSynth                  12     0.95%
⬜ Snd.Reverb                     12     3.46%
   Total Time                    918   161.49%
```

```
Graphics Stats
FPS                                 30
GPU Stalled by CPU            1.705 ms
CPU Stalled by GPU            9.667 ms

GPU Time
Geometry              1.2x /    29.06%
Lighting              2.3x /    27.99%
Effects               2.9x /    10.11%
Post process                    20.63%
Total Time                      87.78%
GPU Stall                        5.11%

Prims / Tris
Totals                 569 /   612.498
  Prime                  1 /        64
  Geometry             137 /   198.765
  Shadows              288 /   402.715
  Effects              143 /    10.954

Memory Stats
Pushbuffer Size                0.14 MB
Pushbuffer High                0.18 MB
VRAM Free                     20.78 MB
Host Free                     50.47 MB
Heap Free                    136.92 MB
Render Mem Stalls                    0
Render Mem Used                8.50 MB
Render Mem Watermark           8.50 MB

Main RAM                     101.00 MB
Physics                        2.85 MB
Collision                      2.51 MB
Sound                         18.74 MB
Mesh                          21.92 MB
Graphics                       6.89 MB
Animation                     26.22 MB
Texture                        1.56 MB
Shader                         1.91 MB
AI Data                        5.09 MB
Various                        3.33 MB
Waste                          7.43 MB
Total                         99.30 MB
Main RAM Bins                 98 / 101

Video RAM                    190.04 MB
Mesh                          14.63 MB
Texture                      170.24 MB
Waste                          1.80 MB
Total                        186.68 MB
Video RAM Bins               137 / 139
```

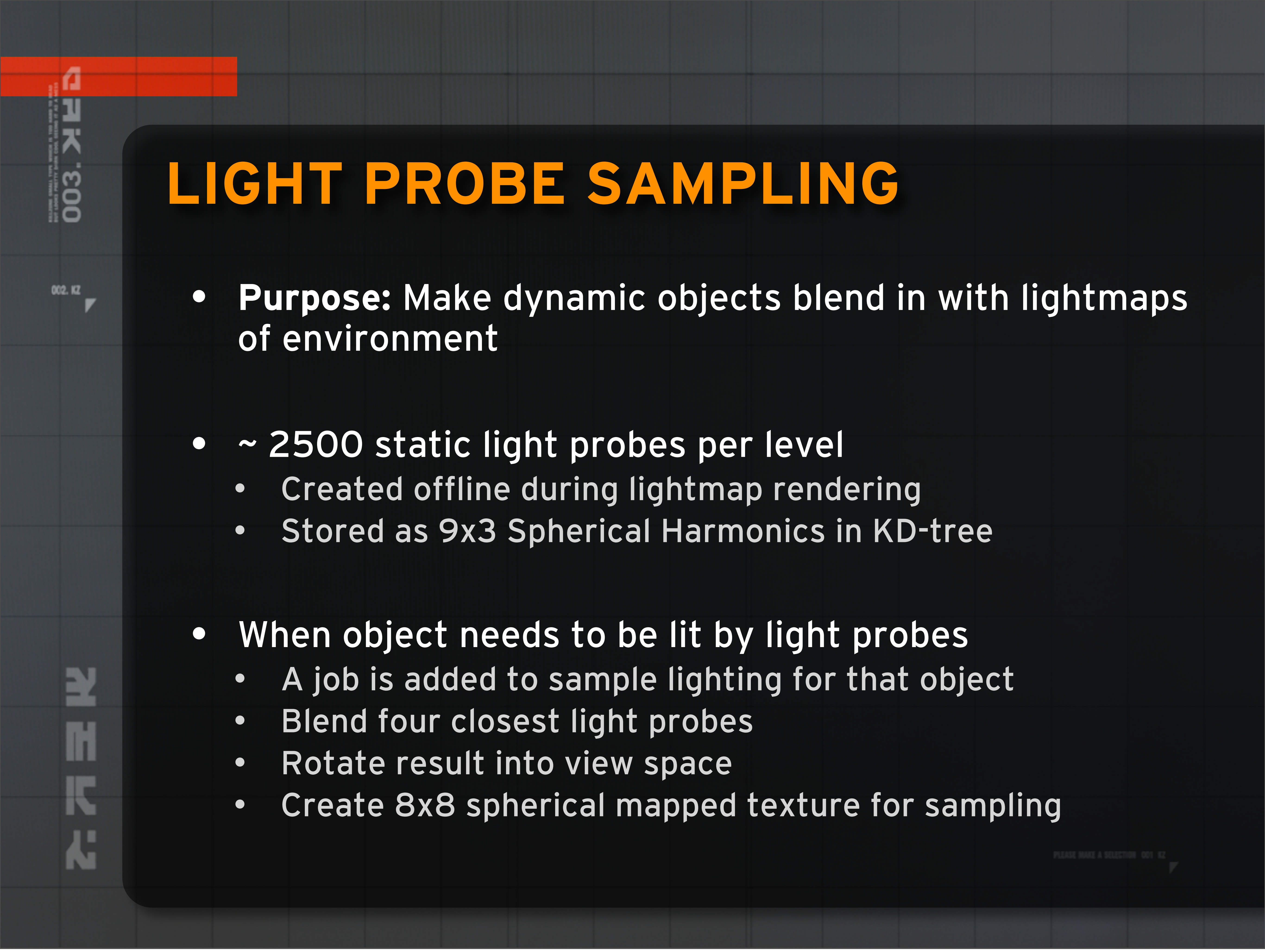
Killzone 2 - SPU usage in cut scene



SPUs in GRAPHICS

002. KZ
PLEASE MAKE A SELECTION 001 KZ

Presentation overview
- Take a look at different areas
    - Graphics
    - Game code
    - AI

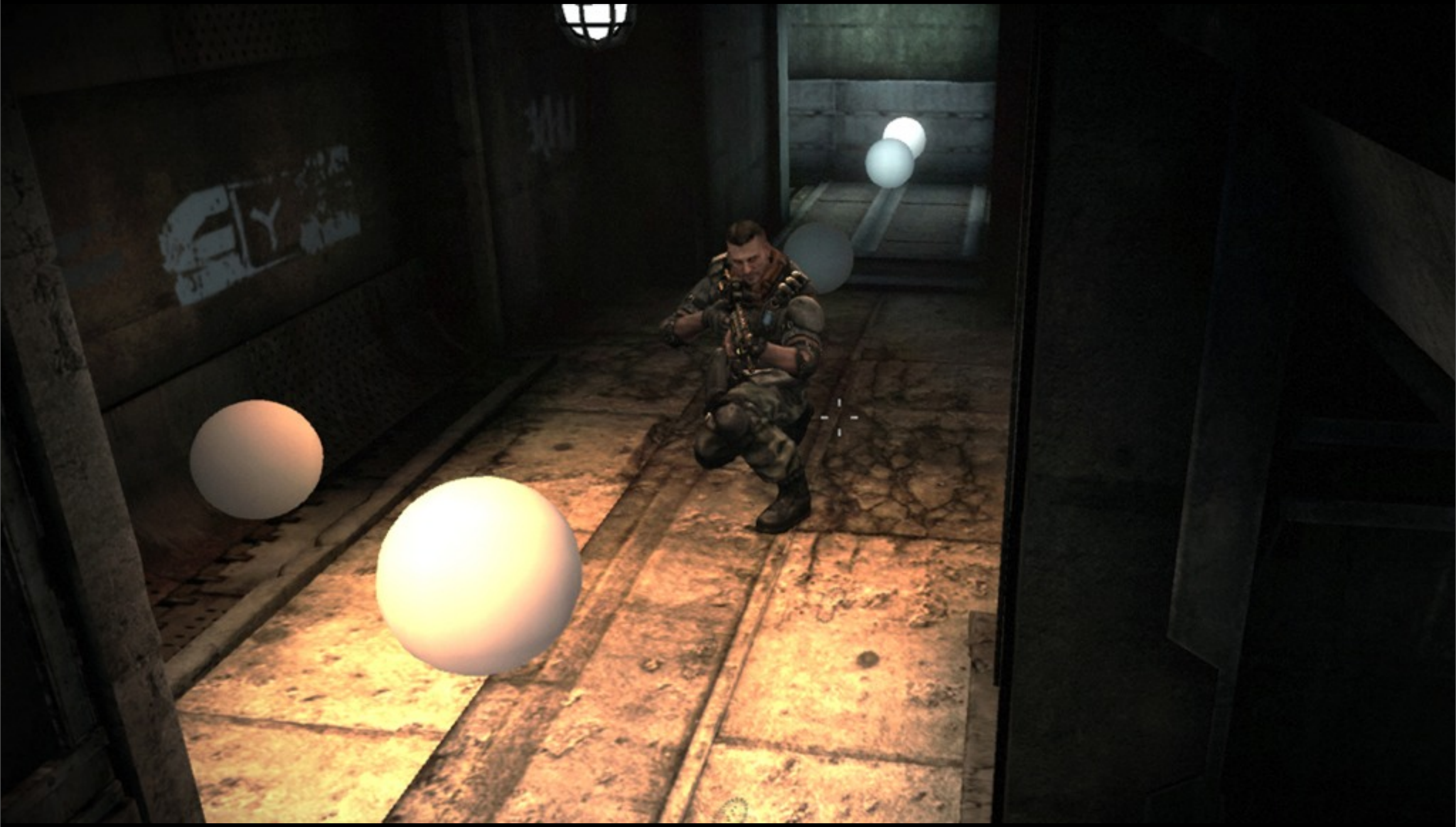


LIGHT PROBE SAMPLING

- **Purpose:** Make dynamic objects blend in with lightmaps of environment
- ~ 2500 static light probes per level
  - Created offline during lightmap rendering
  - Stored as 9x3 Spherical Harmonics in KD-tree
- When object needs to be lit by light probes
  - A job is added to sample lighting for that object
  - Blend four closest light probes
  - Rotate result into view space
  - Create 8x8 spherical mapped texture for sampling

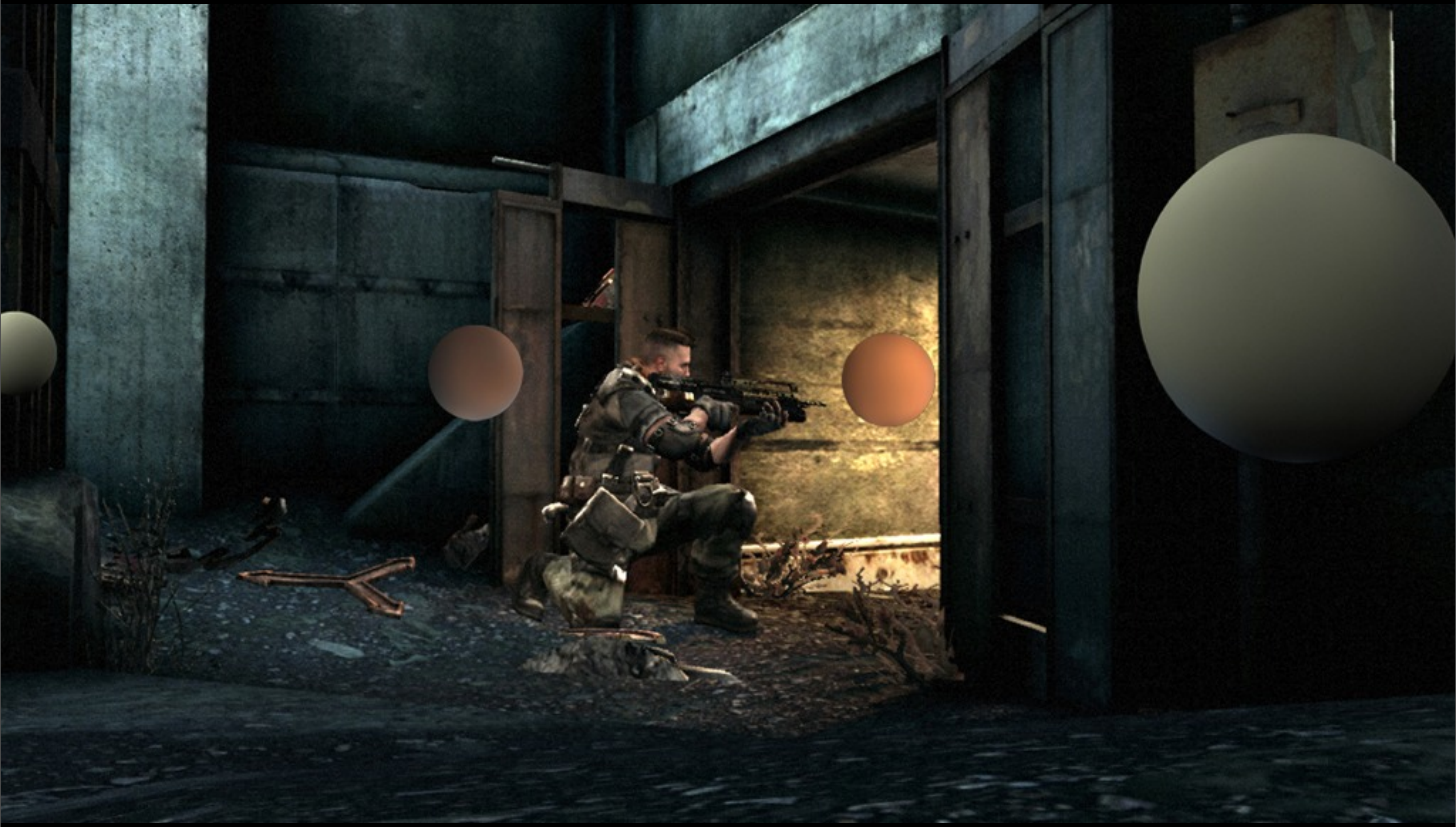
Notes:
- Will explain with pictures to make it clear
- Sounds expensive, but it’s not too bad
- Do roughly 600 light probes per frame (13% of one SPU)
- Sometimes more
- Multiple probes per object (legs, torso, arms), never bothered to optimize

LIGHT PROBE SAMPLING



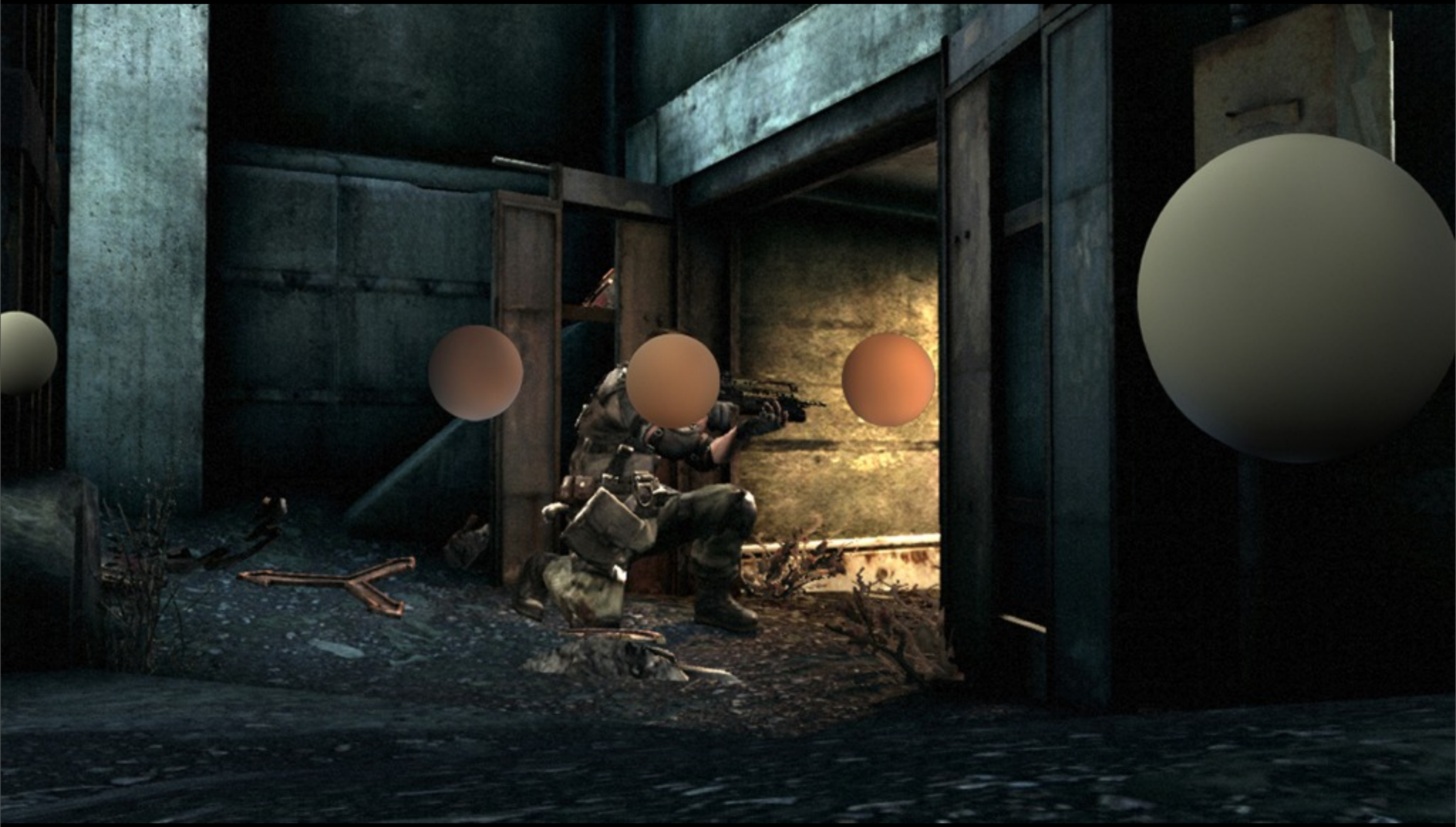
Light probe placement in world

LIGHT PROBE SAMPLING



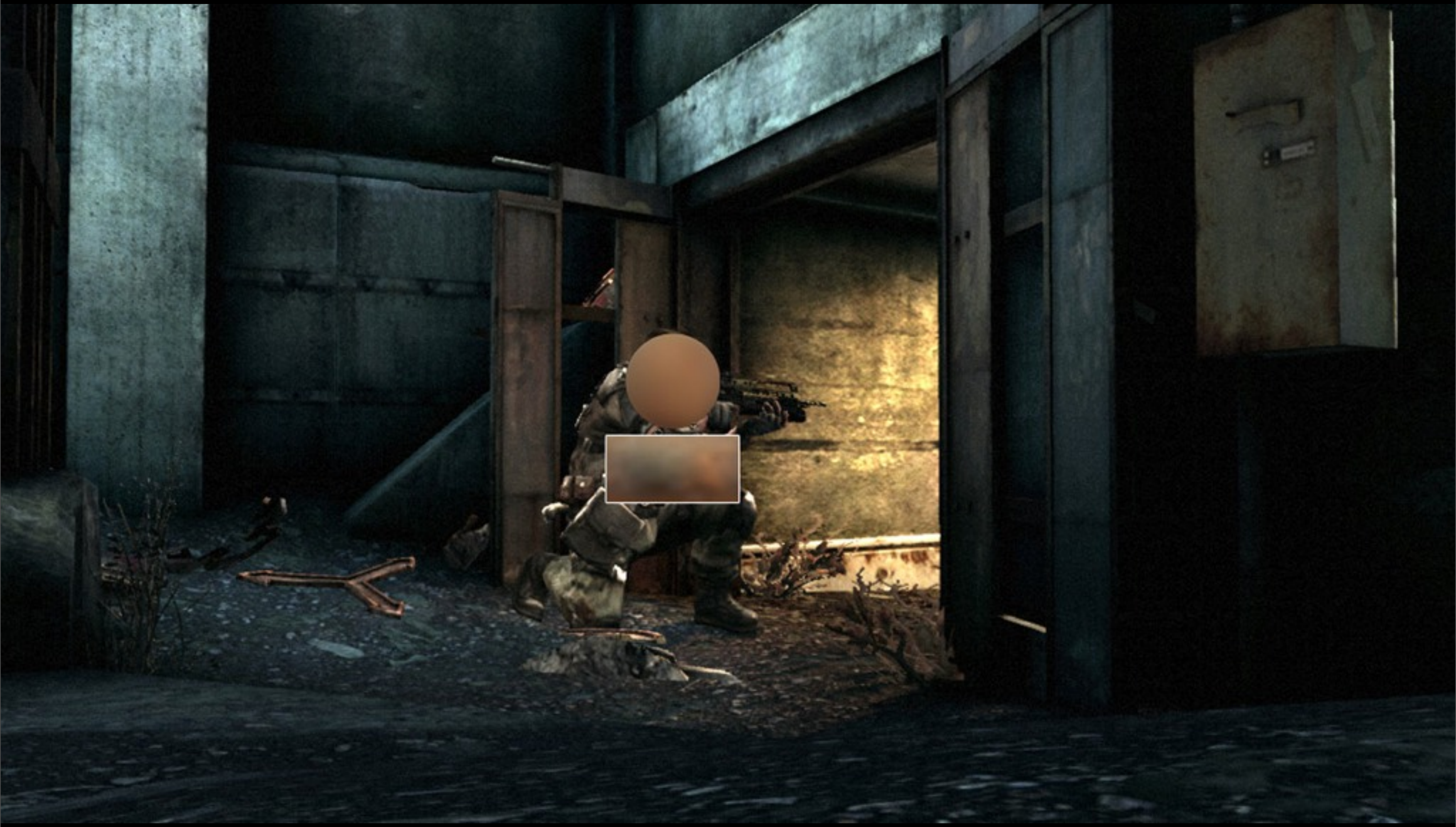
Light probes around a dynamic object

LIGHT PROBE SAMPLING



Blended light probe at object's position

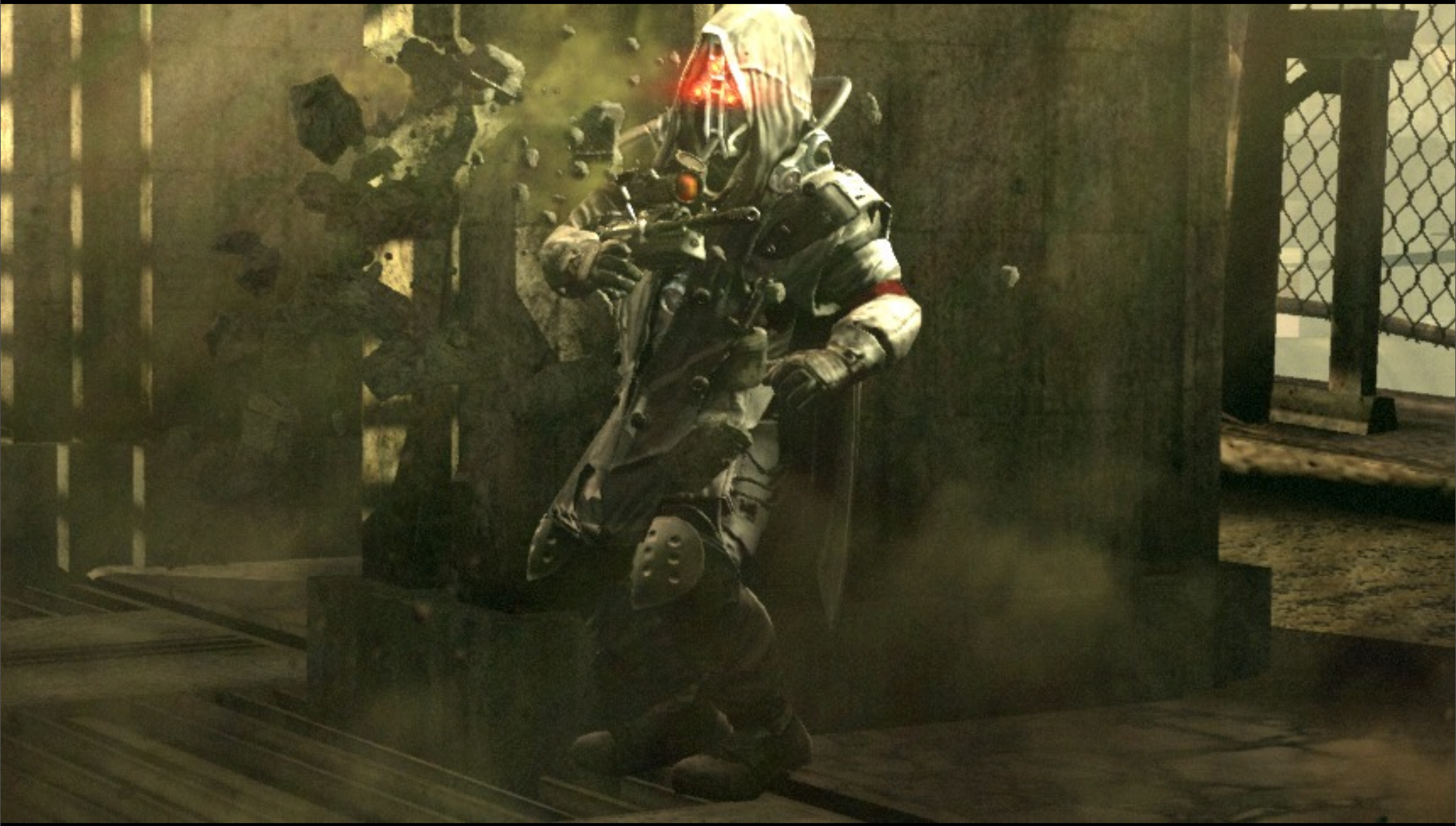
LIGHT PROBE SAMPLING



Blended probe sampled in texture space

Notes:
- This is actually fake
    - This is a latitude-longitude map
    - In-game is spherical map
- But this seemed to explain better by the artist

LIGHT PROBE SAMPLING



Final shot with lightmaps and light probes

Notes:
- This is the final scene as you see it in-game
  - Or on TV, in this case
  - Or on PlayStation®Network, soon

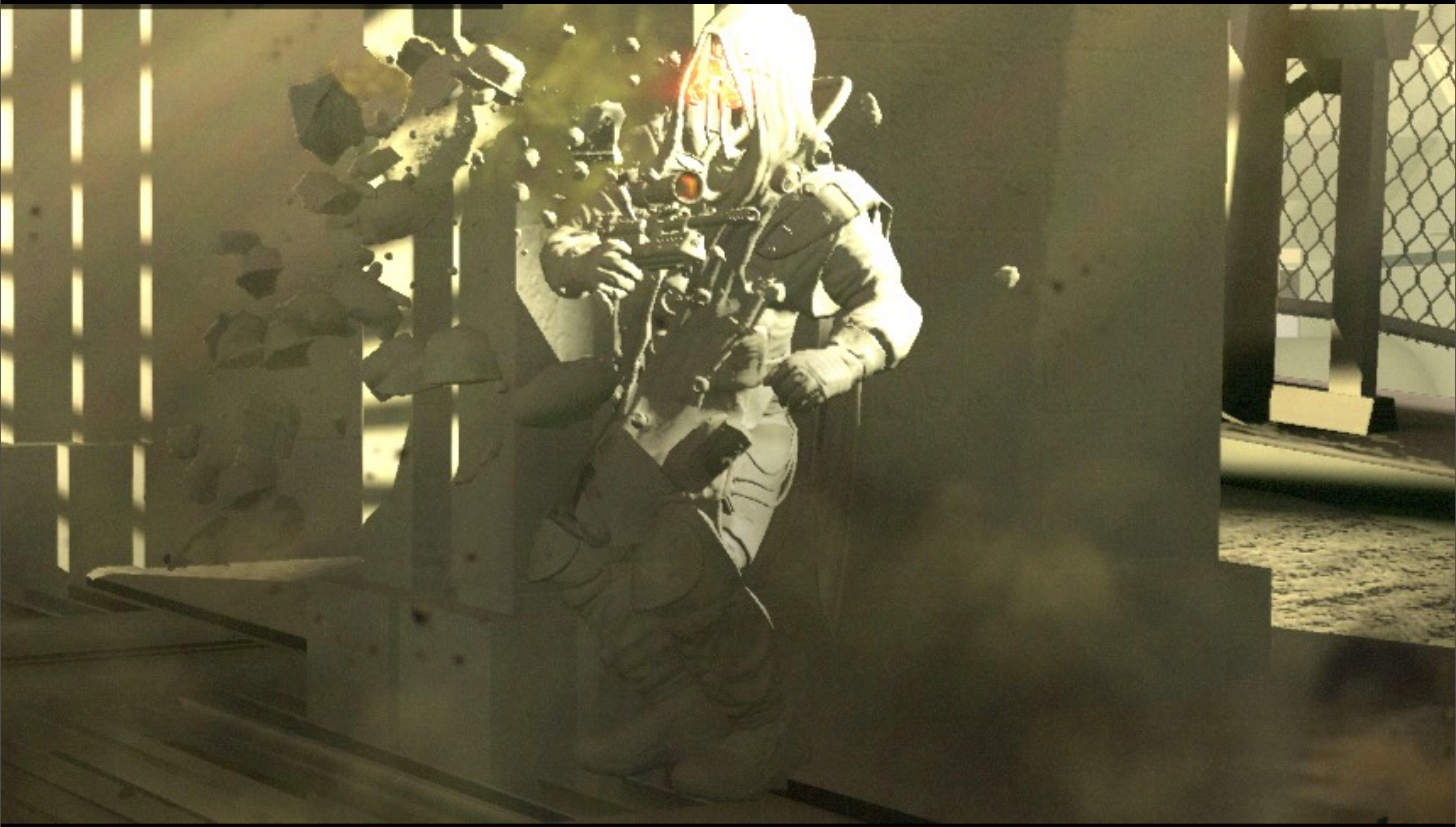
- Can you see what is
  - Lightmapped
  - Lit by light probes?

LIGHT PROBE SAMPLING

Base scene, no light probes, no textures

Notes:
- Light probes replaced with single white probe
- All dynamic objects and destructibles are lit with probes
- Only floor is static in this case

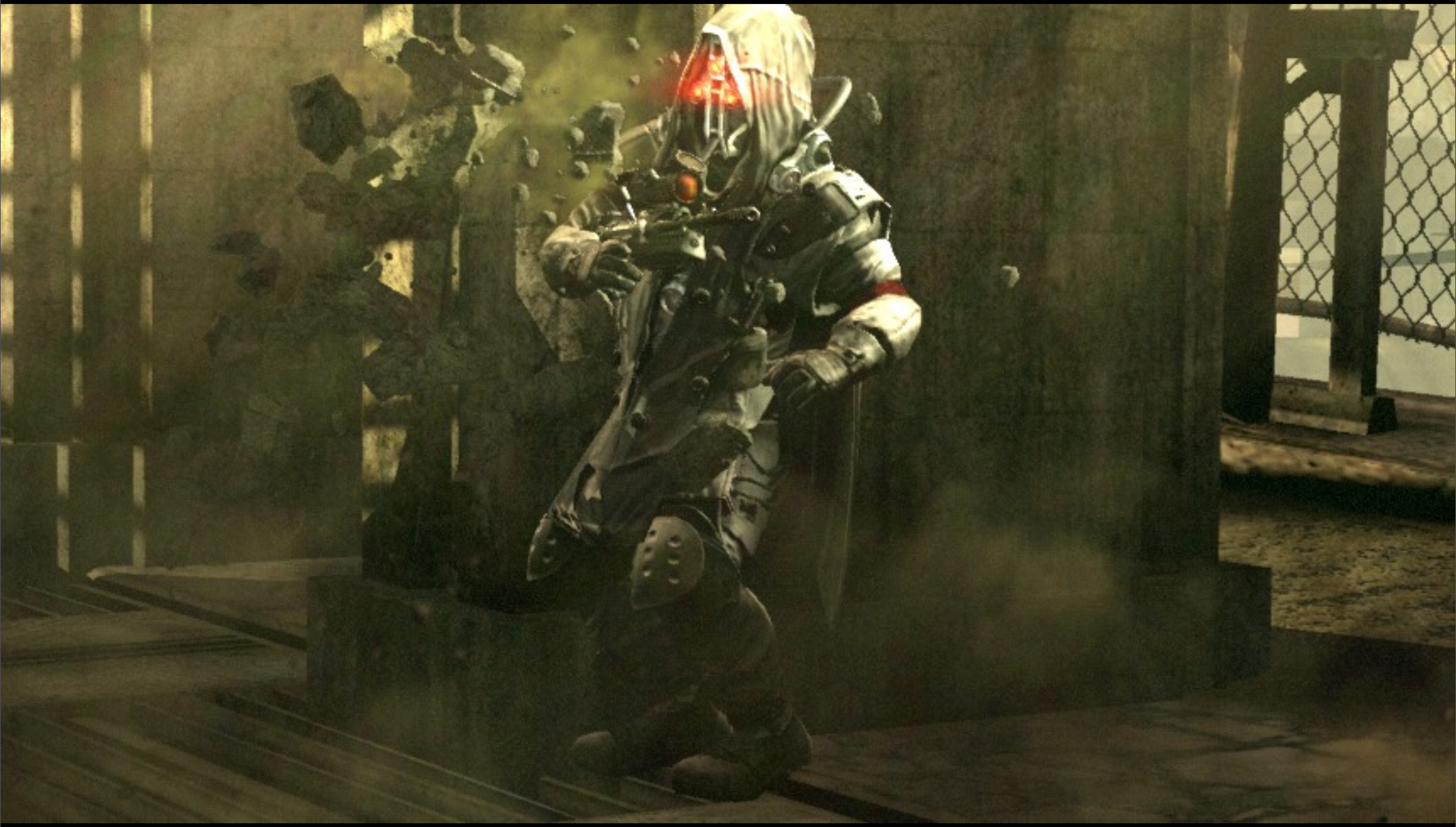
LIGHT PROBE SAMPLING



...add light probes sampling and sunlight

Notes:
- All lighting and directional information in shadow is light probes
- Blend of sun light and light probes ‘pull’ the scene together

LIGHT PROBE SAMPLING



...add textures

Notes:
- Textures add less ‘feeling’ than lighting

CINEMATIC - Because you forget it’s a game, objects don’t “pop out” of the scene

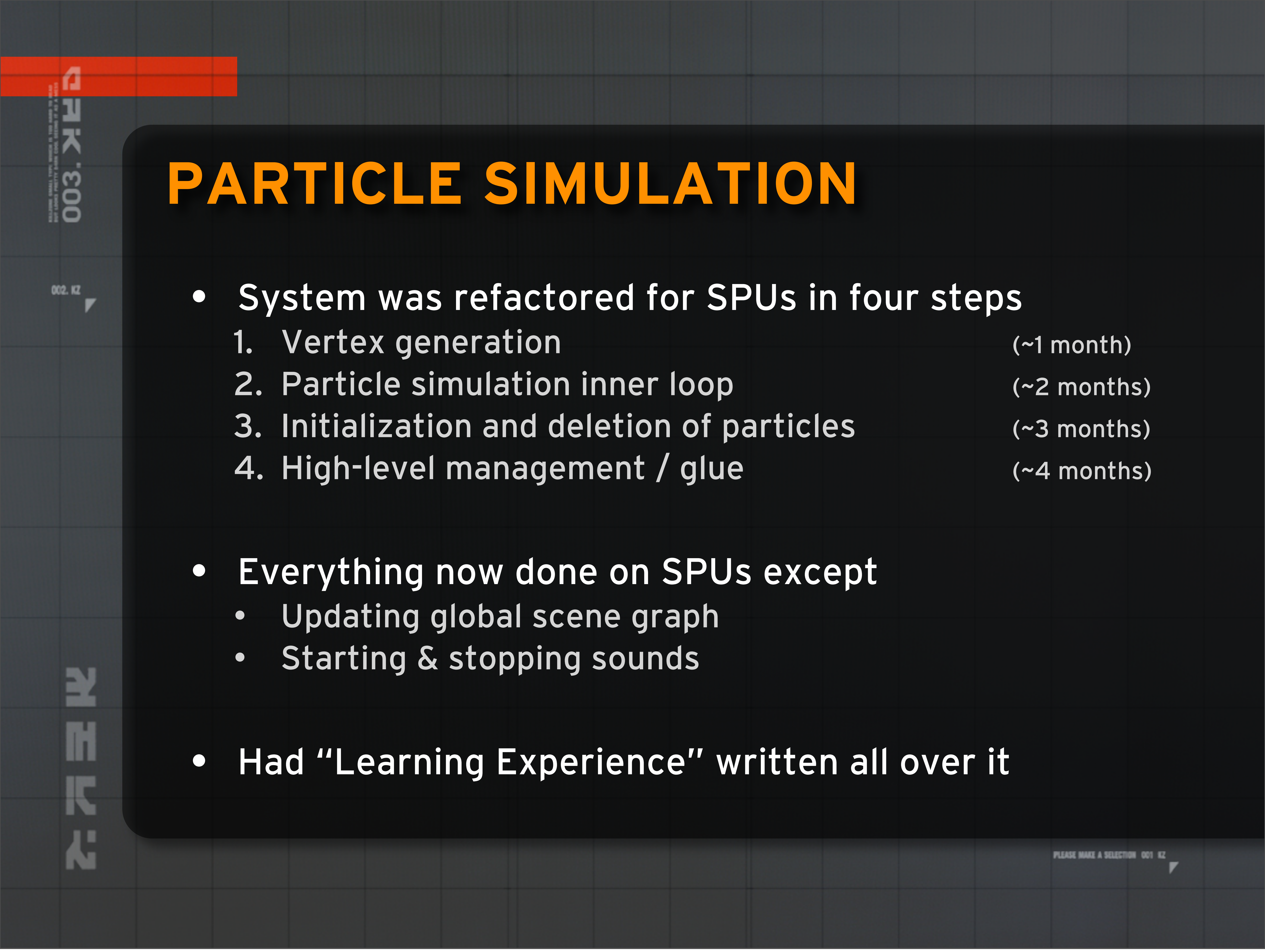
PARTICLE SIMULATION

- We're quite particle heavy, per 30 Hz frame:
  - ~ 250 particle systems simulated
  - ~ 150 systems drawn
  - ~ 3000 particles updated
  - ~ 200 collision ray casts
- Difficult to optimize for multi-core
  - System had grown feature-heavy over time
  - Had to refactor in-place, incremental steps
  - Code was quite optimized, but generic C++
  - Memory accesses all over the place

Notes:
- A lot of our look comes from effects
- Particles very key to our look
- *Not* designed for PlayStation®3! Like, really...

Features:
- Geometry/mesh particles
- Collision particles (ray casts)
- Recursive system (impact causes new particle system)
- Attach sounds
- Lots of curves and tweakability

You:
- Everyone has one of these hideous beasts in their code somewhere



PARTICLE SIMULATION

- System was refactored for SPUs in four steps
  1. Vertex generation (~1 month)
  2. Particle simulation inner loop (~2 months)
  3. Initialization and deletion of particles (~3 months)
  4. High-level management / glue (~4 months)
- Everything now done on SPUs except
  - Updating global scene graph
  - Starting & stopping sounds
- Had “Learning Experience” written all over it

Steps:

1. Simple, vectorised when moved as well
2. Also quite straightforward, particles already configured, just need simulation
3. Tough, had to read all artist curves, tweaks, variable-sized arrays, linked lists, etc. (data all over the place)
4. Elite, had to remove memory allocation altogether, difficult edge cases
    - But definitely the biggest win, SPU pain caused us to do stuff ‘the right way’ which in the end helped

Notes:

- Now runs almost completely on SPUs
- Almost zero PPU overhead left
- And many many times faster than old version
    - SPU memory locality = cache locality
    - Custom memory allocator
- Every time we made it faster, the artists started using more of it
    - Requiring us to further optimize
    - Because we liked the look it gave us
- “Incremental” was the keyword
- Learning experience because we learned how memory-bound we really were, but never knew

CODE STYLE DIFFERENCE

PPU Version

- Single-threaded
- Malloc & free
- Pointers to objects
- Input control curves
- Raycasts during update

- ~ 20ms update on PPU

SPU Version

- Heavily parallel
- Linear memory block
- Embedded objects
- Sampled lookup tables
- Queues raycast jobs

- < 1ms update on PPU
- ~ 15ms update on SPUs

20x Less PPU overhead!
Incredible amount of work

Notes:
- First looked more like game code than engine code
- Turned into lean and mean, memory-local system
- Forced to change it by SPUs architecture, but don't regret it in the end

Performance:
- Optimized a lot along the way
- Final system now runs faster on SPUs than same code on PPU
    - DMA hides memory latency
- Runs faster on Intel Xeons as well (no longer fall off D$)

PARTICLE SIMULATION

Example movie

Notes:
- Explosions in presentation are cool
- Higher budgets than the actual game (3x higher)

At peak:
- 1700 particle emitters
- 1100 visible systems
- 120% of an SPU for update
- 110% of an SPU for vertex generation

Future:
- Code is fairly unoptimized, with optimization it'll probably run in-game

DENSE & INTENSE - This is a large part of what makes everything so over-the-top, in your face

IMAGE POST-PROCESSING

- Effects done on SPU
  - Motion blur
  - Depth of field
  - Bloom

- SPUs assist the RSX with post-processing
  1. **RSX** prepares **low-res image buffers** in XDR
  2. **RSX** triggers **interrupt** to start SPUs
  3. **SPUs** perform **image operations**
  4. **RSX** already **starts on next frame**
  5. Result of SPUs processed by RSX **early in next frame**

- Improved version available in PlayStation®Edge 1.1!
- Similar (but different) approach in PhyreEngine now!

Notes:
- Moved some work normally done on RSX to SPUs
  - 20% on RSX was too much
  - Freed up to draw more geometry/lighting
- Ported only the low-res effects
  - SPUs not fast enough for full-res effects (for us)
- 4pm talk on PhyreEngine that uses SPU-assisted RSX-post processing
  - Room 3002, West Hall, called “Deferred Lighting and Post Processing on PlayStation®3”
  - Different tradeoff, may suit you better
  - Two techniques now available to all 3rd parties!

IMAGE POST-PROCESSING

- Comparison
  - @ 30 Hz

	RSX Time	SPU Time	Quality
RSX	20%	0%	Medium
RSX + 5 SPUs	14%	12%	High

- SPUs are compute-bound
  - Bandwidth not a problem
  - Code can be optimized further
- Our trade-of: RSX vs. SPU time
  - SPUs take longer
  - But SPUs look better
  - And RSX was our bottleneck

Notes:
- Top line is the RSX, bottom SPUs
- SPUs are slower
  - At time of Killzone 2 shipping, Edge is faster!
  - Dropped in at last moment

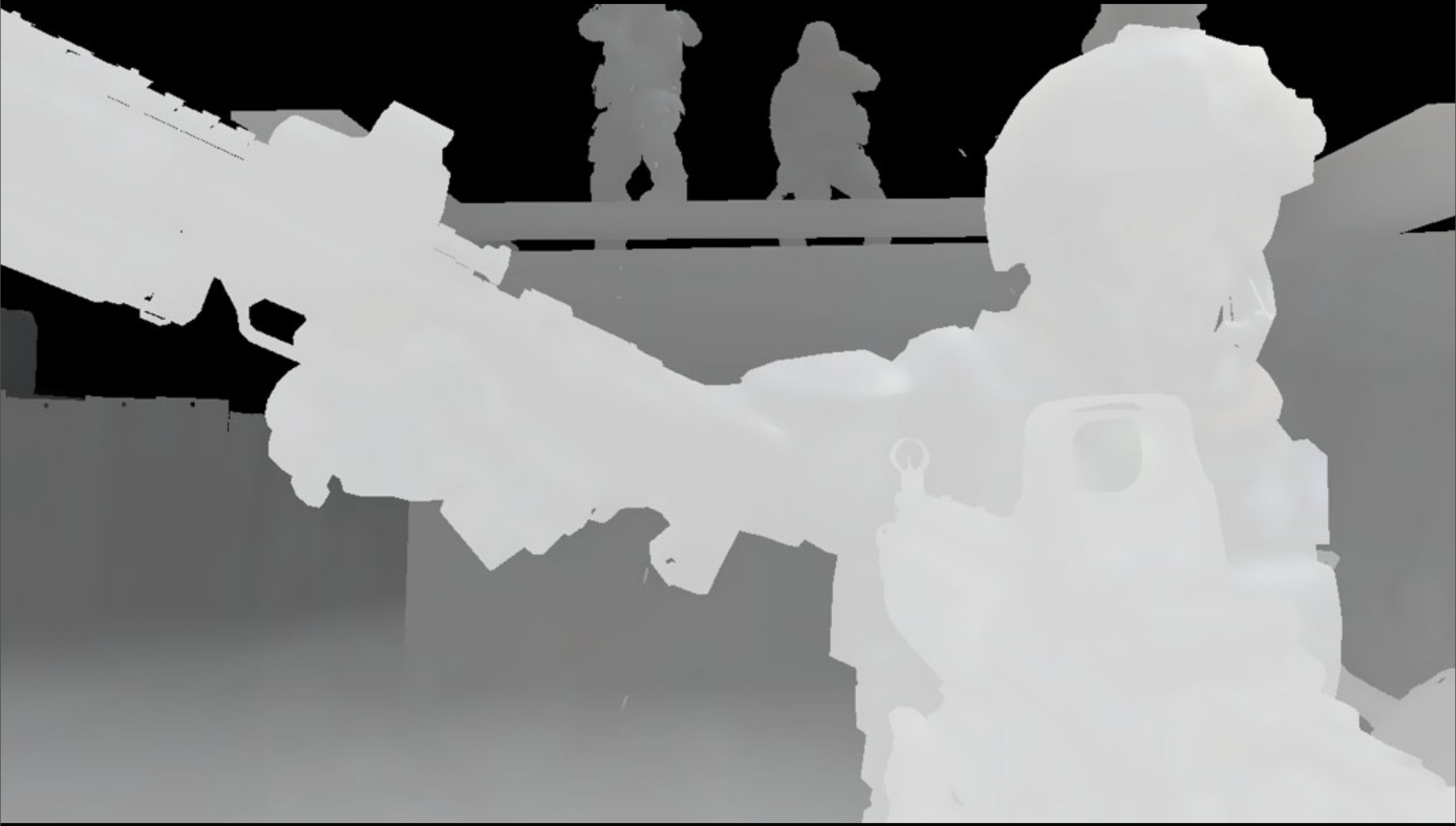
Quality:
- SPUs have higher internal precision
- Dropped in late in the day (everybody was optimizing)
- IGN picked up on it (preview vs. review code)
  - Some of the quality itself
  - Most of it the tweaks done with the extra budget (increased light map resolution, detail textures, etc.)

IMAGE POST-PROCESSING

Input quarter-res image in XDR

Notes:
- Output from deferred renderer after MSAA resolve
- 640x360
- No visible motion, quite static

IMAGE POST-PROCESSING



Input quarter-res depth buffer

Notes:
- Depth buffer at 640x360

IMAGE POST-PROCESSING

Input quarter-res 2D motion vectors

Notes:
- Projected 2D motion vector
- Written out to a rendertarget in deferred pass

BLOOM + LENS REFLECTION

- Takes roughly 13.1% of one SPU
- SPUs do
  - Depth-dependent intensity response curve
  - Hierarchical 7x7 gaussian blur (16 bit fixed point)
  - Upscaling results from different levels
  - Internal Lens Reflection (inspired by Masaki Kawase)
  - Accumulating into results buffer

Notes:
- First effect we ported, most linear
- 2.6% of five SPUs
- Repeated downsample until 1/64th buffer size
- Accumulate into premultiplied alpha compositing buffer

MOTION BLUR

- Takes roughly 9.5% of one SPU
- Input
  - Quarter-res image from deferred renderer
  - Sixteenth-res 2D motion vector
- Steps:
  - Blur motion vectors (dilation)
  - Blur image along motion vectors (8-tap point samples)
  - Combine blurred image with source
    - Use motion vector amplitude as alpha
- Low-motion areas are unaffected (alpha = 0)

Notes:
- Second effect we ported
- 1.9% of five SPUs
- Worried about texture sampling
- Ended up not doing bilinear filtering
  - Filtering on SPUs is very expensive
  - Looks more grainy
    - Exactly what we wanted!
    - Nice artifacts may stay
- Leaves static parts untouched

DEPTH OF FIELD

- Takes roughly 23% of one SPU
- Input
  - Quarter-res image from deferred renderer
  - Quarter-res depth buffer
- Samples image in floating point
  - 36 jittered disc point samples
    - Much more than the 8 bilinear samples on RSX!
    - Got rid of all banding / stair-stepping
  - Weighted by ‘fuzziness’ value based on focus point

Notes:
- Last effect we ported
- 4.6% of five SPUs
- Never thought it would work
	- But ended up being biggest success!
- The point samples are cheap
	- So we could do a lot more!

Quality:
- Quality of effect determined largely by radius of circle of confusion
- Less by sampling quality
- The 36 jittered samples were a test
- But looked so good we kept them in

Edge bleeding:
- Intricate sampling methods to avoid edge bleeding
	- From background to foreground
	- But heavily blend front-to-background
- See Edge code

IMAGE POST-PROCESSING

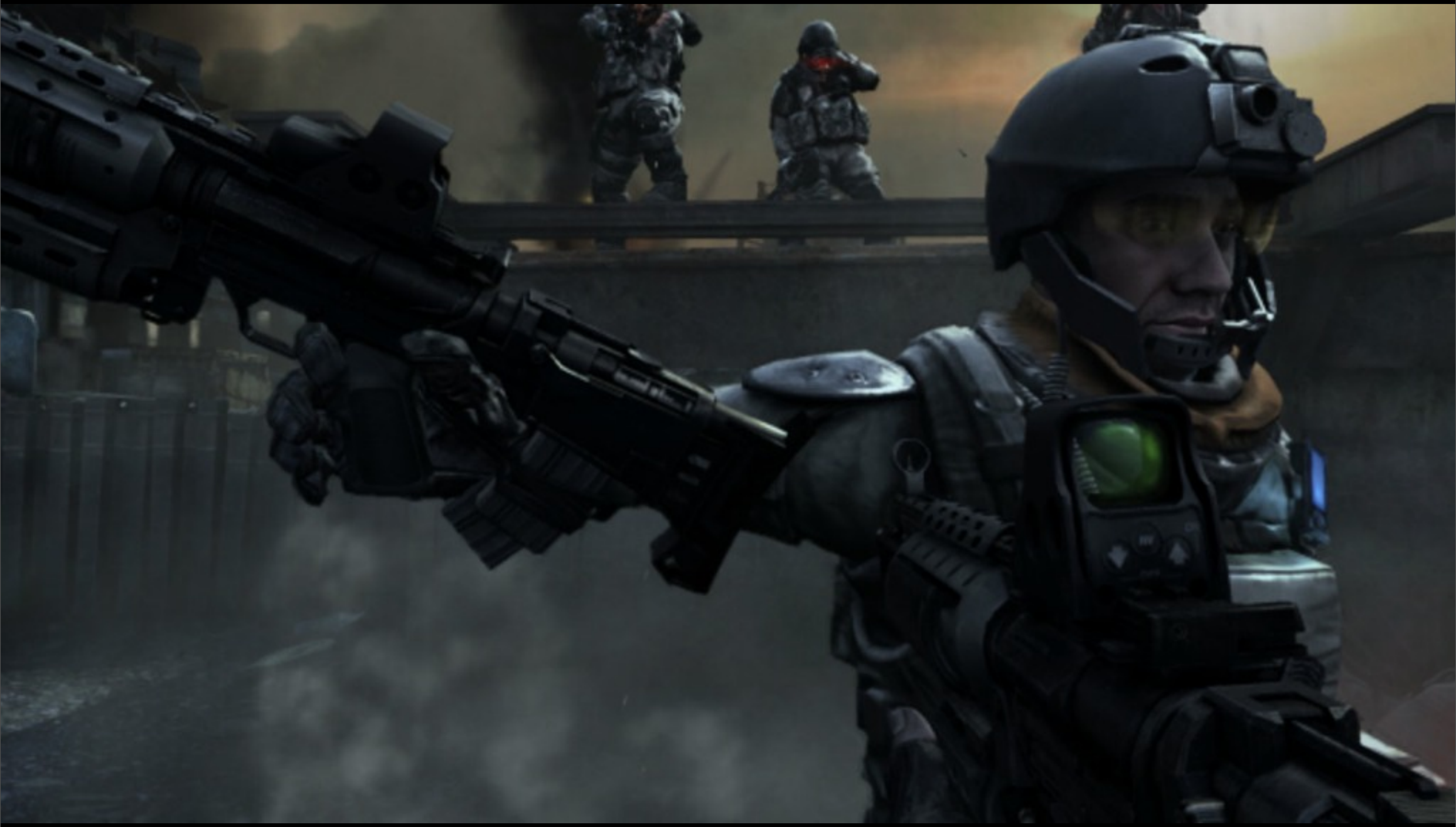


Image before post

Notes:
- Putting it all together
    - Image without post

IMAGE POST-PROCESSING

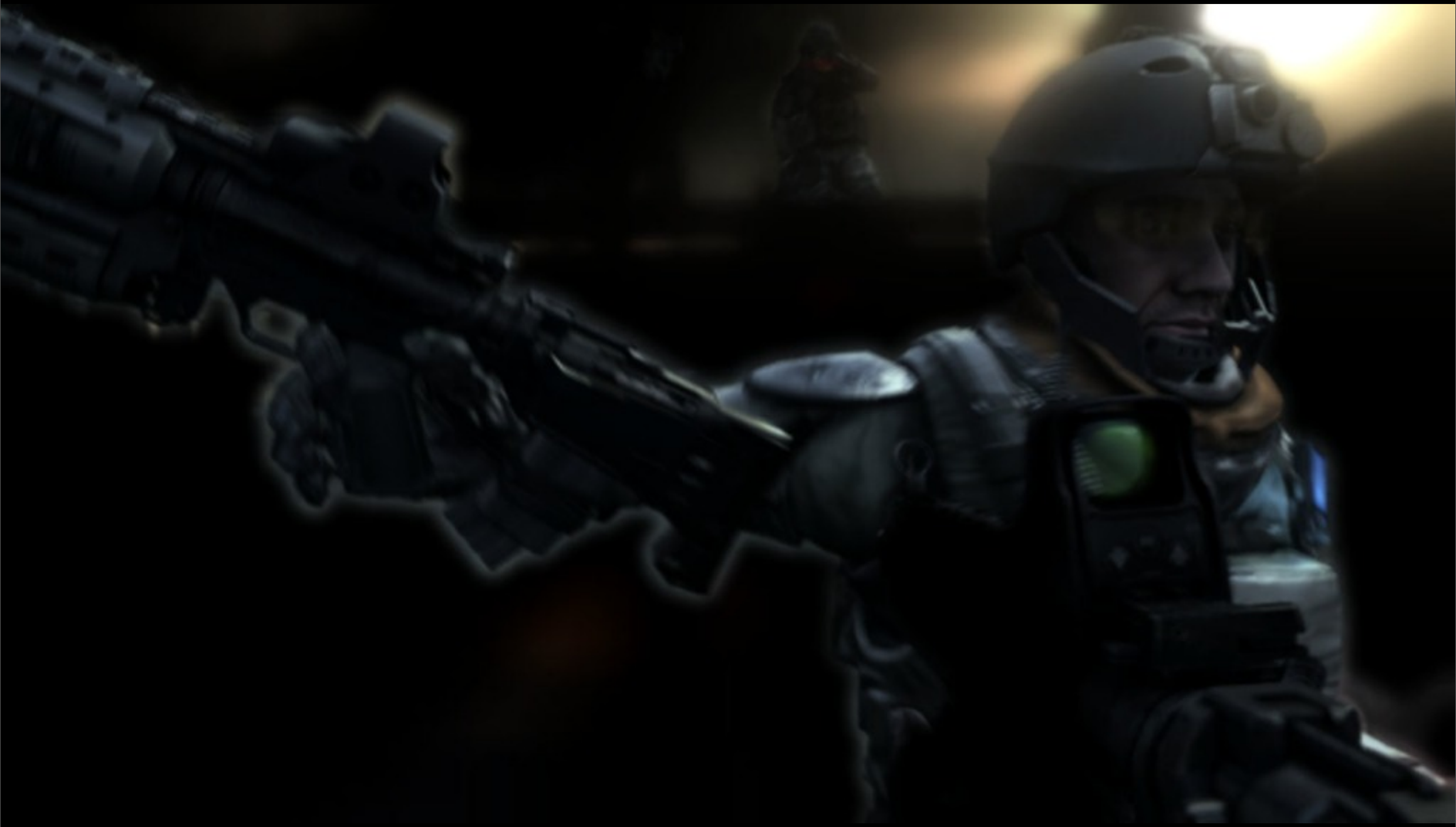
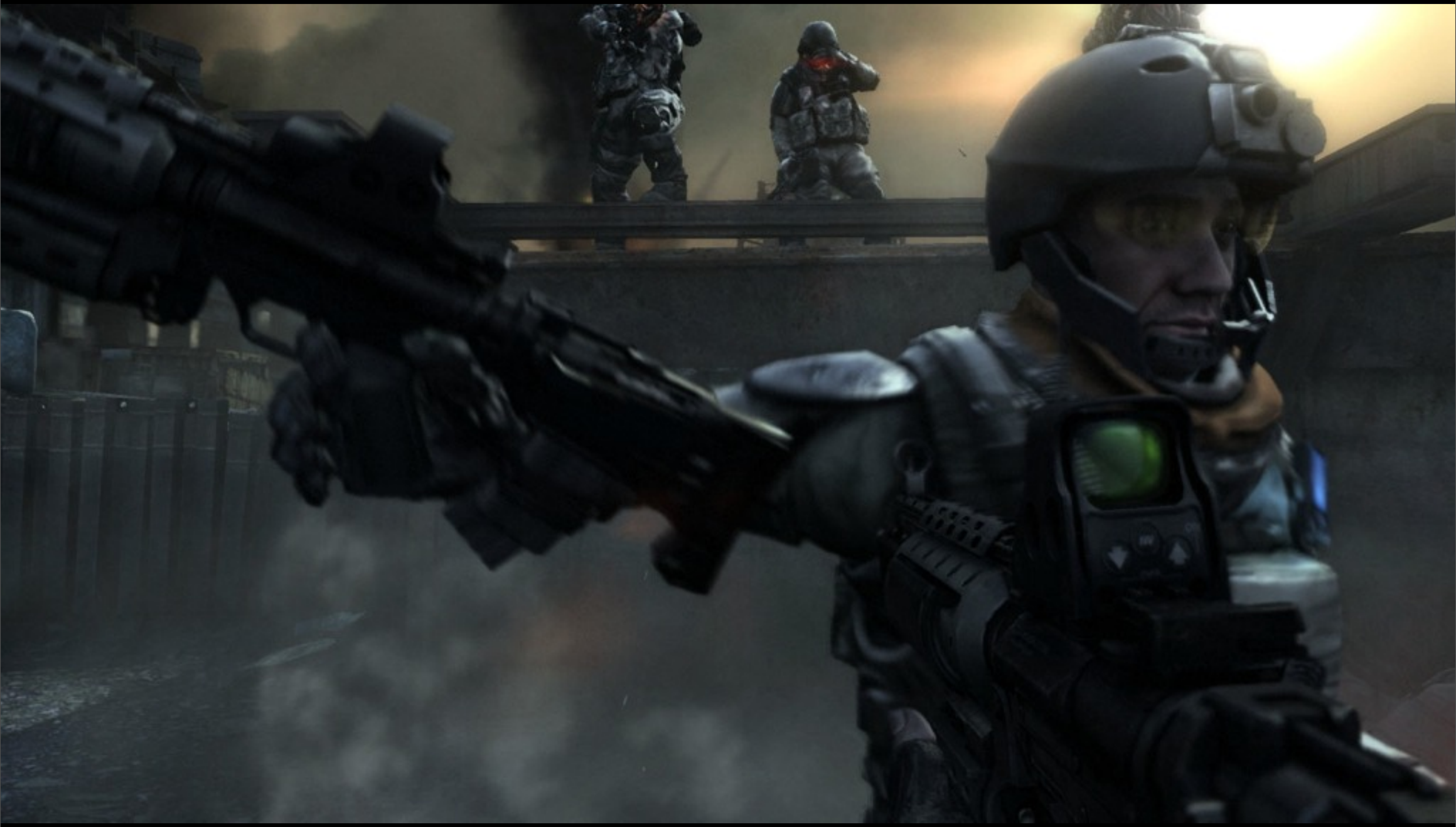


Image generated on the SPUs (bloom, DoF, motion blur)

Notes:
- Result of all of the SPU jobs combined
- Quarter-res premultiplied alpha format
- Only areas affected by post-processing are present

Visual:
- **Motion blur:** Actor at front
- **Bloom:** Sunlight
- **Depth of field:** Gun is out of focus (it actually always is, too bad for the detail)

IMAGE POST-PROCESSING



Composited image

Notes:
- Final image as composited by the RSX
- Blending the out-of-focus and blurry areas in

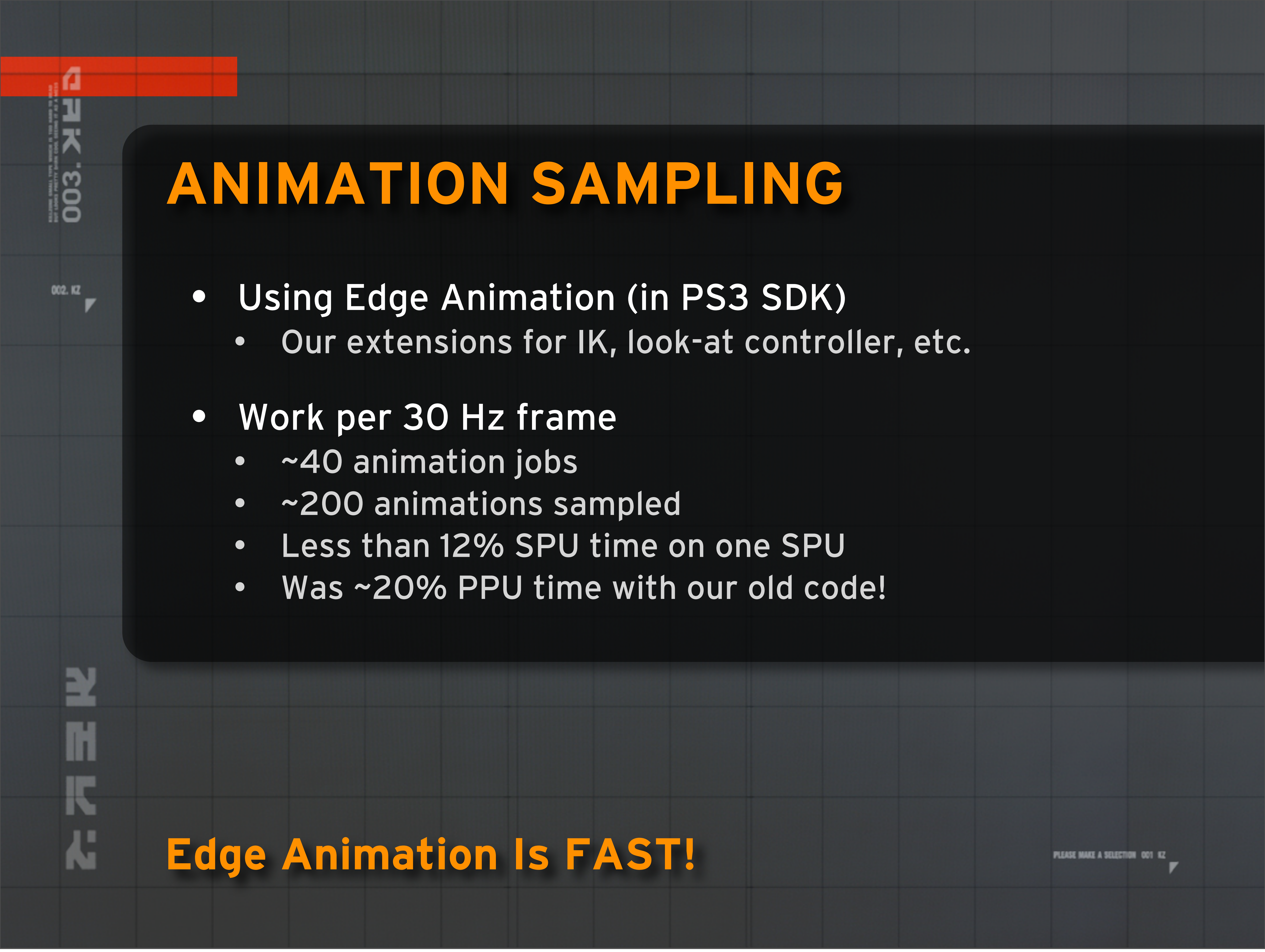
IMAGE POST-PROCESSING

Input quarter-res image in XDR

CINEMATIC - All of these techniques come directly from film. If we hadn’t done them on SPUs we may not have been able to do them at this quality

SPUs in GAME CODE

Notes:
- Enough graphics
- On to game code, the most difficult area to use SPUs



ANIMATION SAMPLING

- Using Edge Animation (in PS3 SDK)
  - Our extensions for IK, look-at controller, etc.
- Work per 30 Hz frame
  - ~40 animation jobs
  - ~200 animations sampled
  - Less than 12% SPU time on one SPU
  - Was ~20% PPU time with our old code!

Edge Animation Is FAST!

Notes:
- Used to be bottleneck on PS2 and on PPU
- Very heavy on animation
  - Blending dozens of animations (2 to 50 per humanoid)
- Large part of Edge Animation design inspired by our requirements
- High-level animation code still a bottleneck

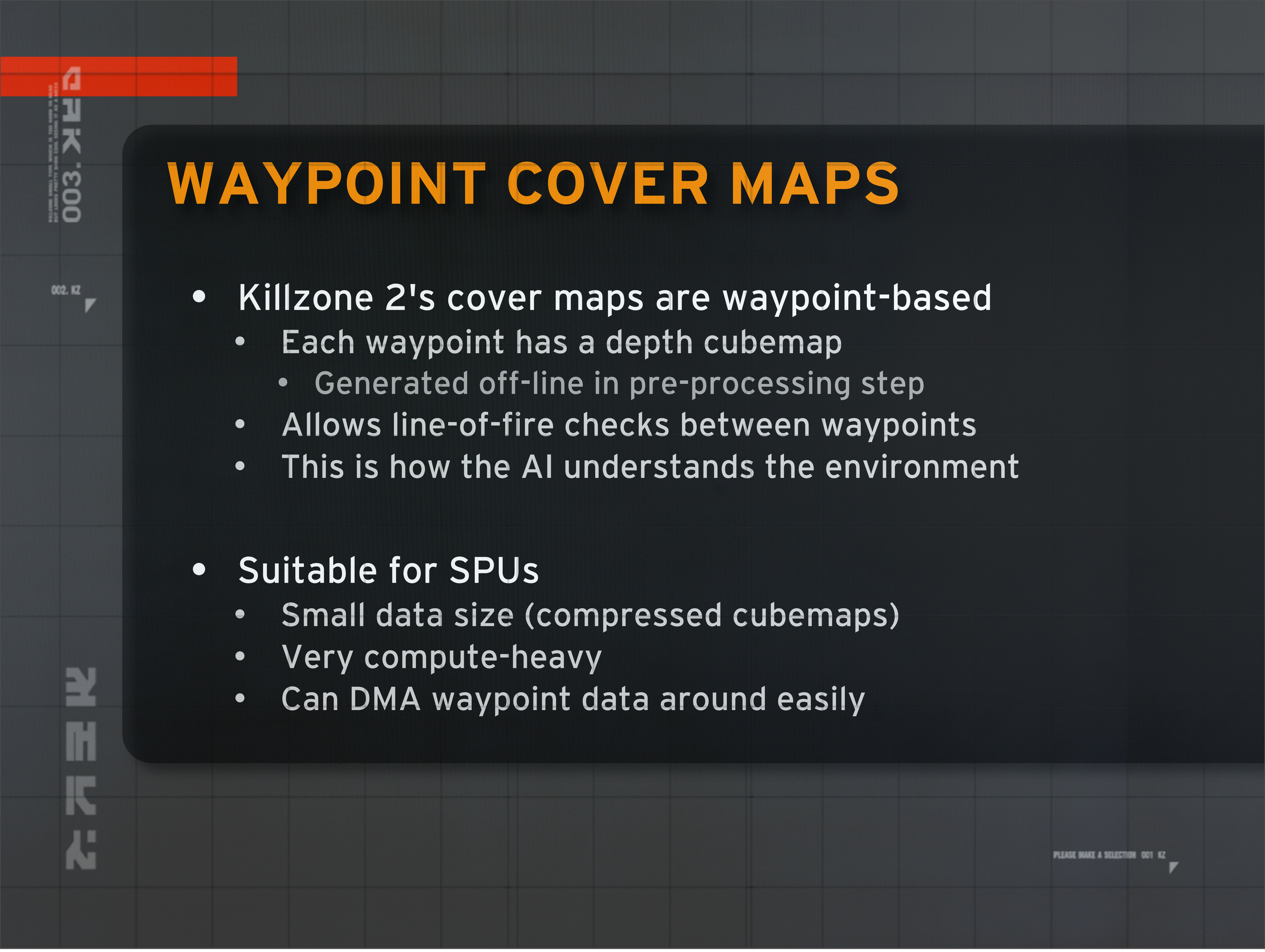
Advice:
- If you have animation sampling showing up on PPU
  - Use this, it's *incredibly* fast

REALISTIC - We require hundreds of animations to make the characters stick to the ground, move realistically. Animation is key to this, and with it performance

SPUs in
ARTIFICIAL INTELLIGENCE

Notes:

- That’s largely it for game code
- We also do raycasts from game code
- Says something about what we can use SPUs for
    - Very difficult to use SPUs in pure logic code
    - Also felt it wouldn’t give us much to focus there
    - Didn’t suit our franchise needs



WAYPOINT COVER MAPS

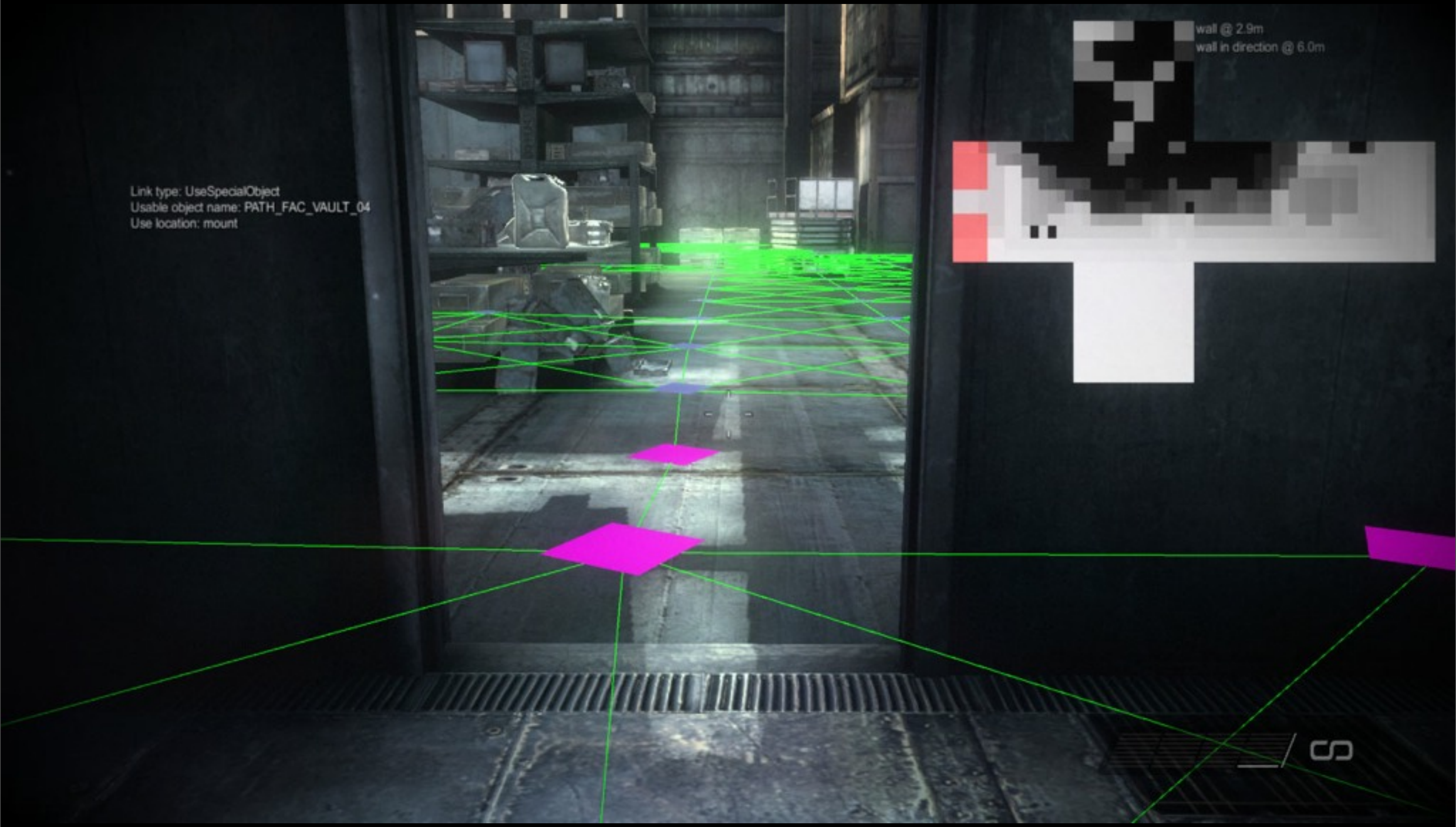
- Killzone 2's cover maps are waypoint-based
  - Each waypoint has a depth cubemap
    - Generated off-line in pre-processing step
  - Allows line-of-fire checks between waypoints
  - This is how the AI understands the environment

- Suitable for SPUs
  - Small data size (compressed cubemaps)
  - Very compute-heavy
  - Can DMA waypoint data around easily

Notes:
- The AI in Killzone 2 ‘sees’ based on waypoints
- Levels are populated with thousands of waypoints, divided into zones

Reasoning:
- Allows reasoning about tactical waypoint positions
    - Can a person standing on waypoint A see me if I’m on waypoint B?
    - How can I plan a path from A to B without somebody on waypoint C seeing me?
- Large problem space to search
- But allows intelligent behavior
    - More checks than we could ever do before

WAYPOINT COVER MAPS



Link type: UseSpecialObject
Usable object name: PATH_FAC_VAULT_04
Use location: mount

wall @ 2.9m
wall in direction @ 6.0m

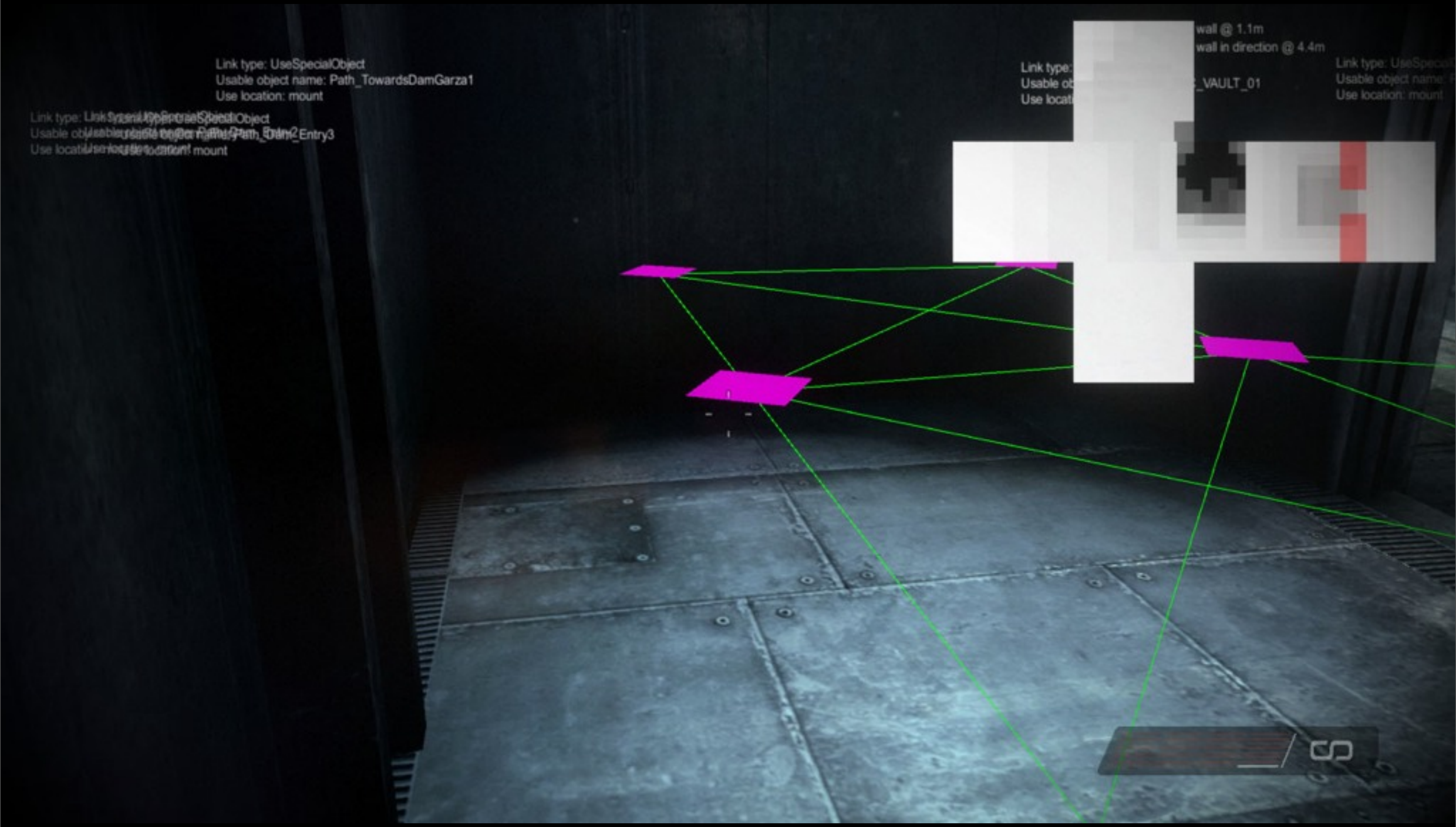
Example waypoint cover map

Notes:
- White areas are closeby
- Red area is player position
- Depth compressed to 6 bits
- Less detail in floor/ceiling than walls

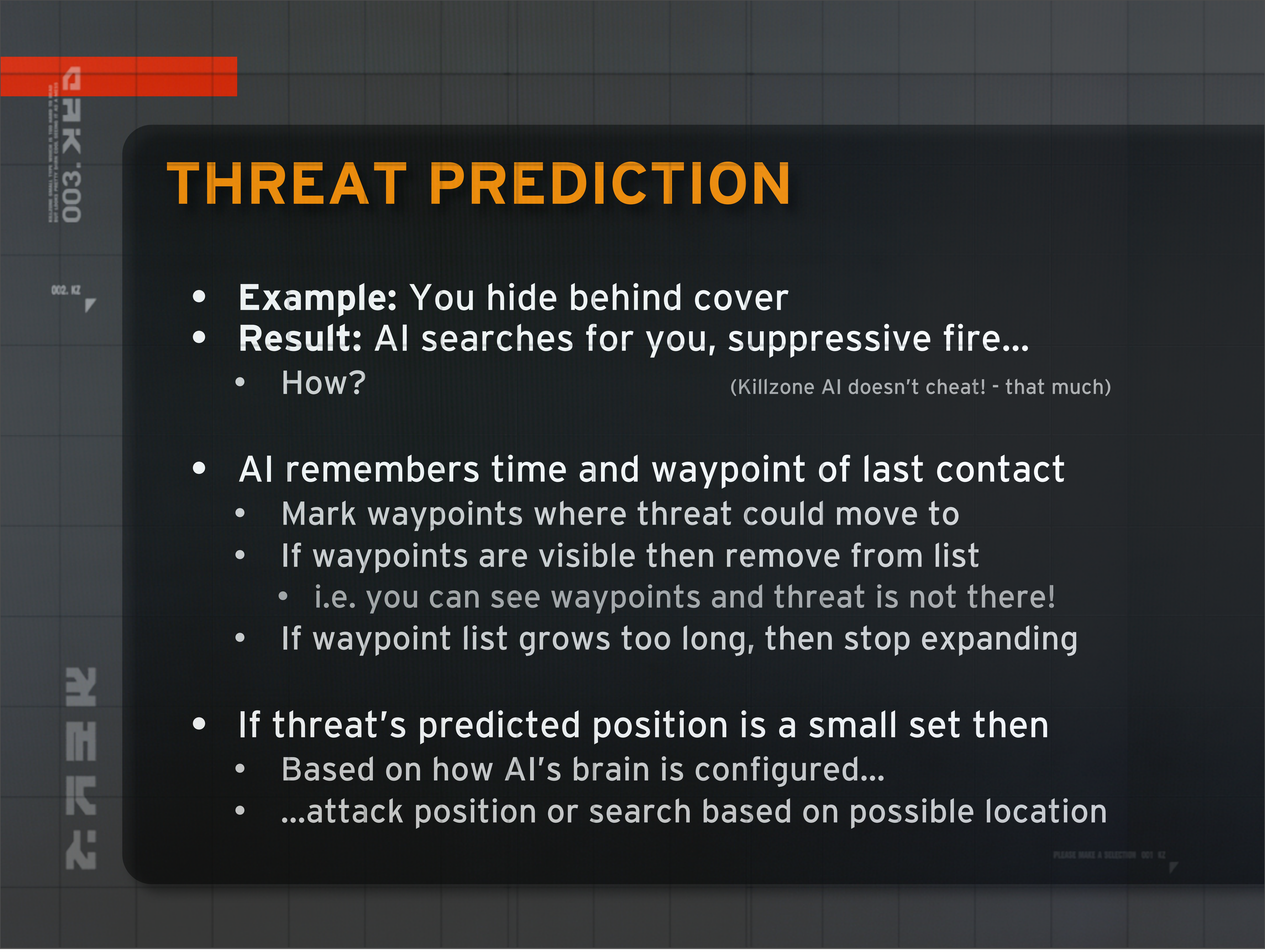
FAIL:
- Tried all sorts of novel compression techniques on cubemaps
    - FAIL
    - Simple solution worked best

Visual:
- Small hallway, so lots of closeby walls
- Dark area looks out into room in camera-facing direction

WAYPOINT COVER MAPS


Link type: UseSpecialObject
Usable object name: Path_TowardsDamGarza1
Use location: mount
wall @ 1.1m
wall in direction @ 4.4m
_VAULT_01

Example waypoint cover map



THREAT PREDICTION

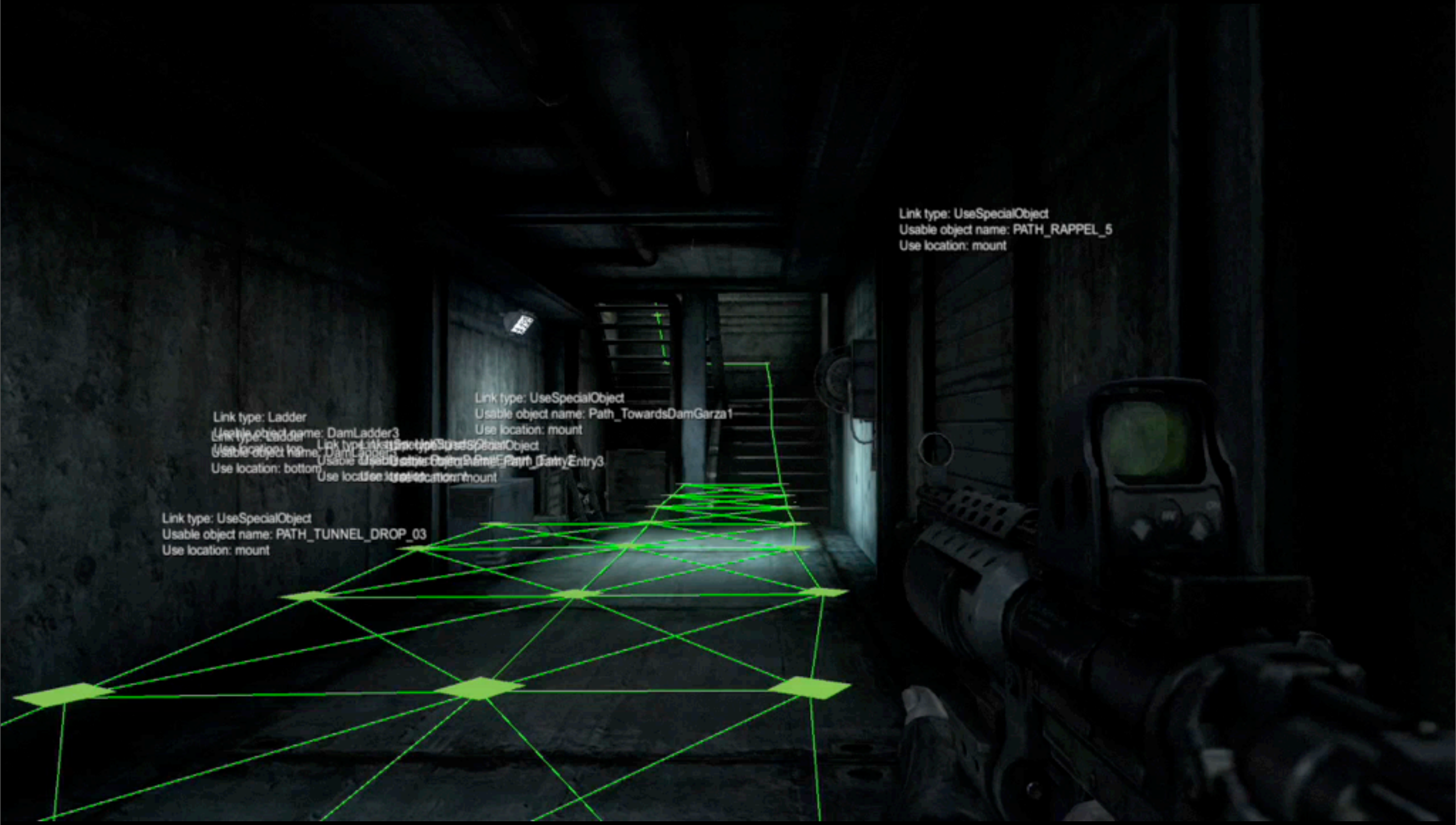
- **Example:** You hide behind cover
- **Result:** AI searches for you, suppressive fire...
  - How? (Killzone AI doesn't cheat! - that much)
- AI remembers time and waypoint of last contact
  - Mark waypoints where threat could move to
  - If waypoints are visible then remove from list
    - i.e. you can see waypoints and threat is not there!
  - If waypoint list grows too long, then stop expanding
- If threat's predicted position is a small set then
  - Based on how AI's brain is configured...
  - ...attack position or search based on possible location

Notes:
- (run through slide first)
- AI doesn't cheat
  - We don't tell it the player position, *ever*
  - (though game scripters may cheat...)

How this works:
- When an AI loses sight or sound of a threat
  - At start of AI update, fire max 4 jobs (one for each 'lost threat')
  - Process current set of possible positions
  - Scan all waypoints in this list to see if the player is visible there
    - If found -> threat found!
    - If not, expand adjacent waypoints to see which ones are visible
  - Result is:
    - Possible location list
    - Fed back into the AI planner and 'brain'

Deals with:
- People running into a building
- You hiding behind cover and standing up to shoot
- People inside a building shooting out of windows, etc.

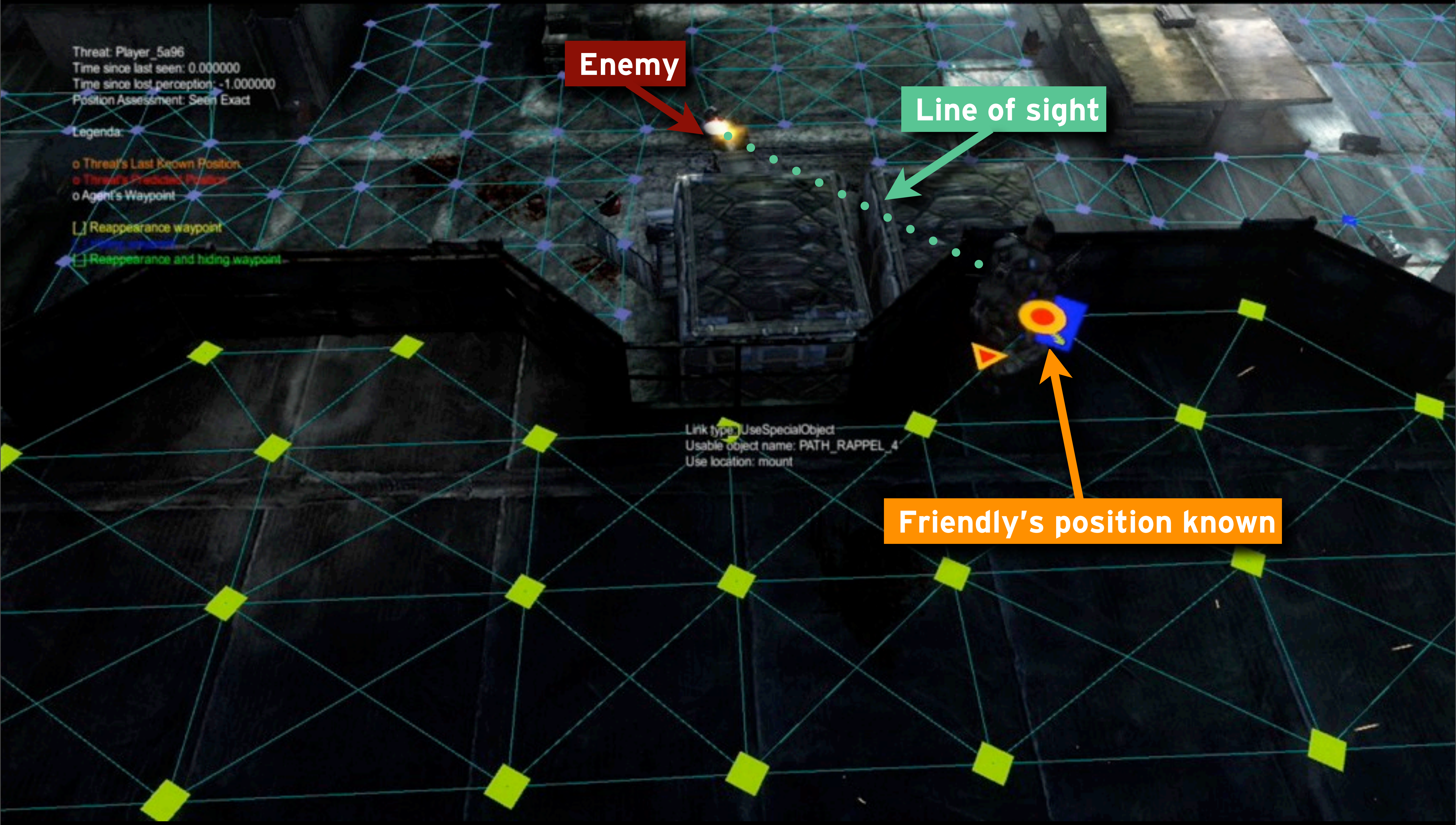
THREAT PREDICTION


Link type: UseSpecialObject
Usable object name: PATH_RAPPEL_5
Use location: mount
Link type: UseSpecialObject
Usable object name: Path_TowardsDamGarza1
Use location: mount
Link type: Ladder
Usable object name: DamLadder3
Use location: bottom
Path_EntryEntry3
Link type: UseSpecialObject
Usable object name: PATH_TUNNEL_DROP_03
Use location: mount

Threat Prediction using SPUs and cover maps

Notes:
- Part of Killzone 2 demo level
- Warehouse section
- Green = waypoint grid

THREAT PREDICTION



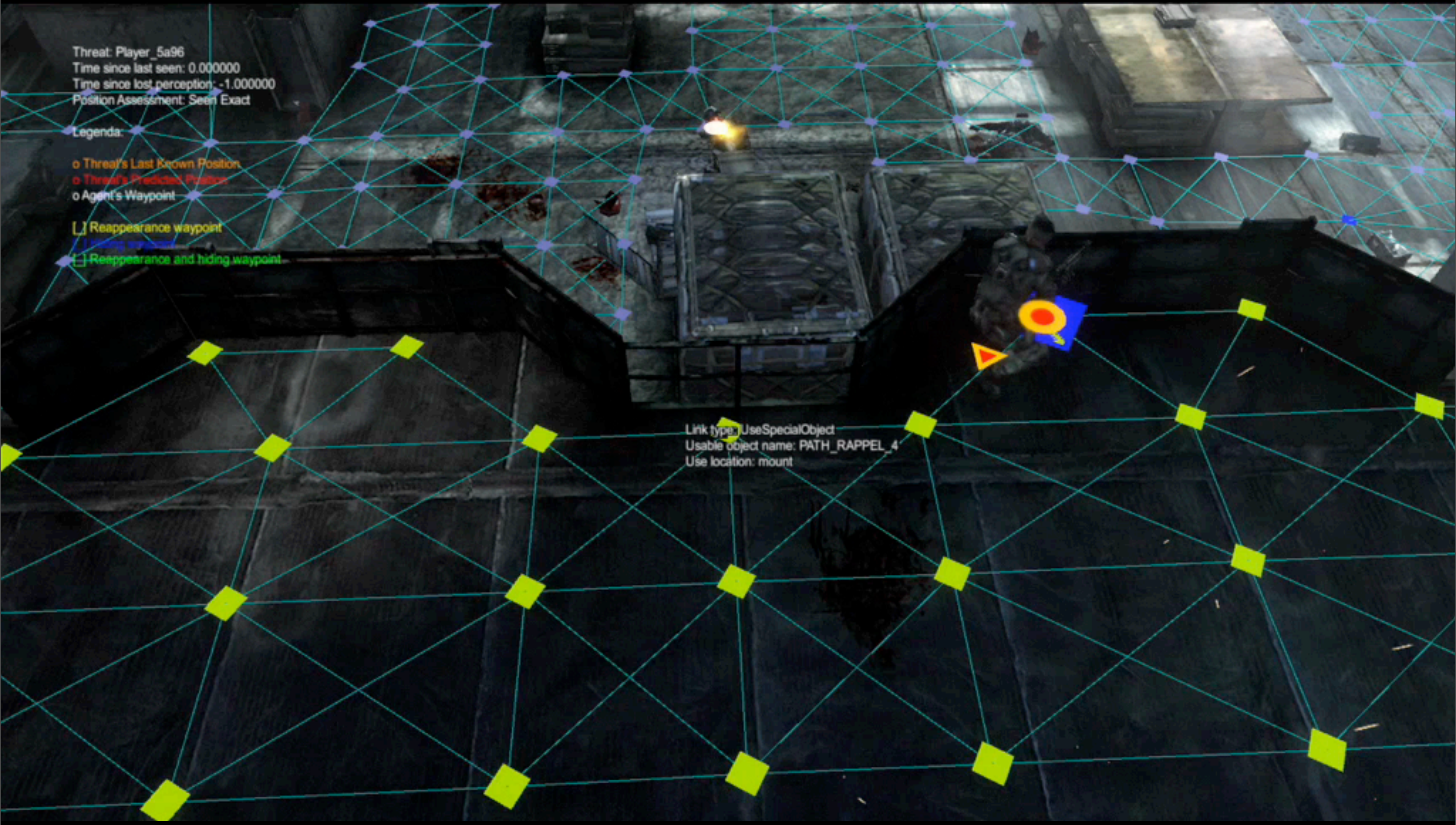
```
Threat: Player_5a96
Time since last seen: 0.000000
Time since lost perception: -1.000000
Position Assessment: Seen Exact

Legenda:

o Threat's Last Known Position
o Threat's Predicted Position
o Agent's Waypoint

[_] Reappearance waypoint
[illegible]
[_] Reappearance and hiding waypoint
```

```
Link type: UseSpecialObject
Usable object name: PATH_RAPPEL_4
Use location: mount
```

Encounter between enemy and friendly

Notes:
- Immediately engaged by enemy
- Knows player position

THREAT PREDICTION



Threat: Player_5a96
Time since last seen: 0.000000
Time since lost perception: -1.000000
Position Assessment: Seen Exact

Legenda:

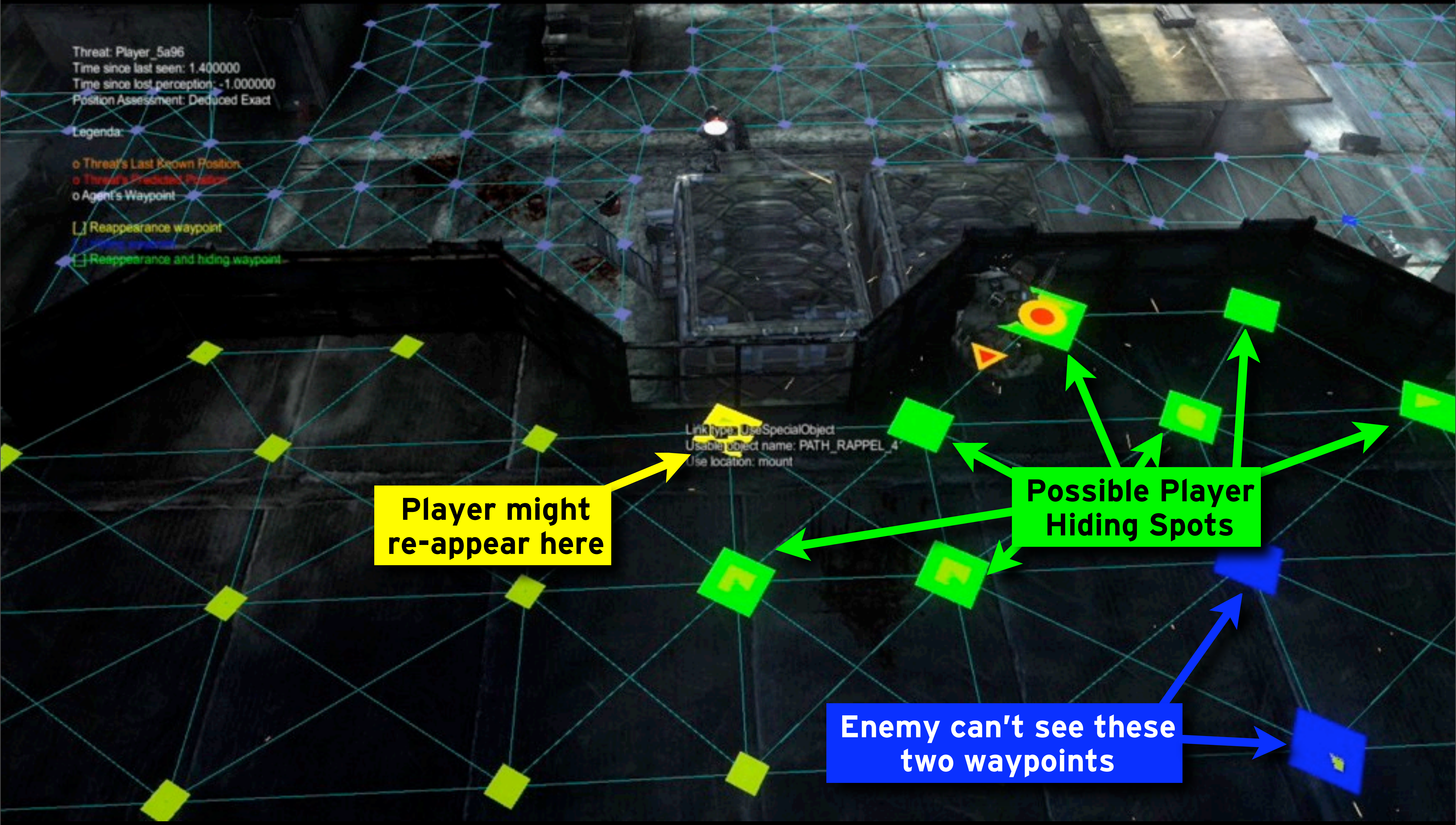
o Threat's Last Known Position
o Threat's Predicted Position
o Agent's Waypoint

[ ] Reappearance waypoint
[ ] Hiding waypoint
[ ] Reappearance and hiding waypoint

Link type: UseSpecialObject
Usable object name: PATH_RAPPEL_4
Use location: mount

Player enters cover position

THREAT PREDICTION



Threat: Player_5a96
Time since last seen: 1.400000
Time since lost perception: -1.000000
Position Assessment: Deduced Exact

Legenda:

o Threat's Last Known Position
o Threat's Predicted Position
o Agent's Waypoint

[ ] Reappearance waypoint
[ ] Hiding waypoint
[ ] Reappearance and hiding waypoint

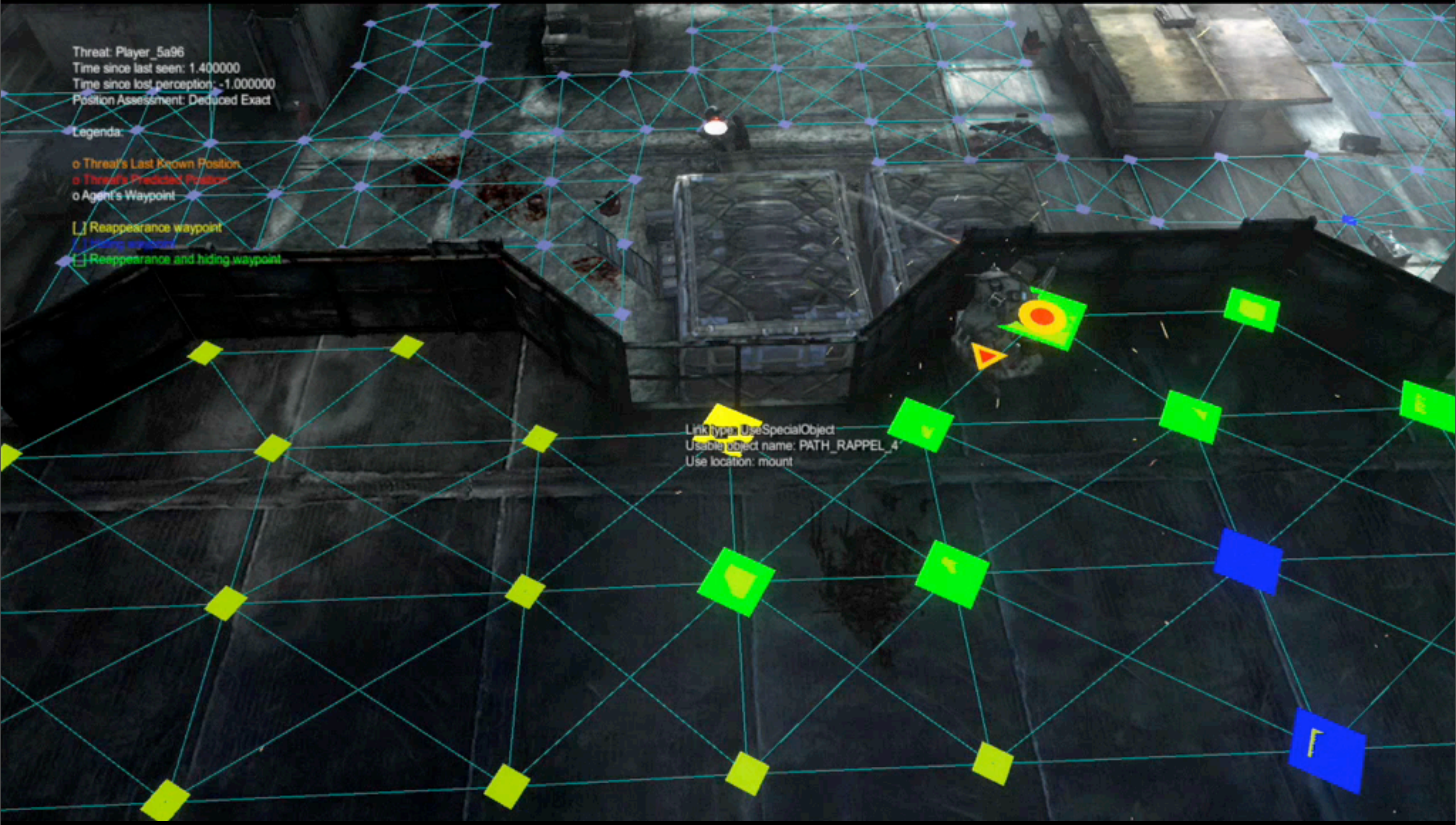
Link type: UseSpecialObject
Usable object name: PATH_RAPPEL_4
Use location: mount

Player might re-appear here
Possible Player Hiding Spots
Enemy can't see these two waypoints

Player enters cover position

Notes:
- Re-appearance waypoint is very important, good place to shoot!
- Helghast shoot at windows, etc. to pin you down
    - NOT pre-programmed!
    Adjusts to your actions!

THREAT PREDICTION



```
Threat: Player_5a96
Time since last seen: 1.400000
Time since lost perception: -1.000000
Position Assessment: Deduced Exact

Legenda:

o Threat's Last Known Position
o Threat's Predicted Position
o Agent's Waypoint

[_] Reappearance waypoint
[_] Hiding waypoint
[_] Reappearance and hiding waypoint
```

```
Link type: UseSpecialObject
Usable object name: PATH_RAPPEL_4
Use location: mount
```

Enemy looking for player

LINE OF FIRE

- **Problem:** AI agents running into each other's line-of-fire
- **Solution:** Line-of-fire annotations

- Each AI agent publishes 'hints'
  - Calculate which waypoints may be in my line-of-fire
  - Published as 'advice' for other entities
  - Most AI tasks use this advice in path planning

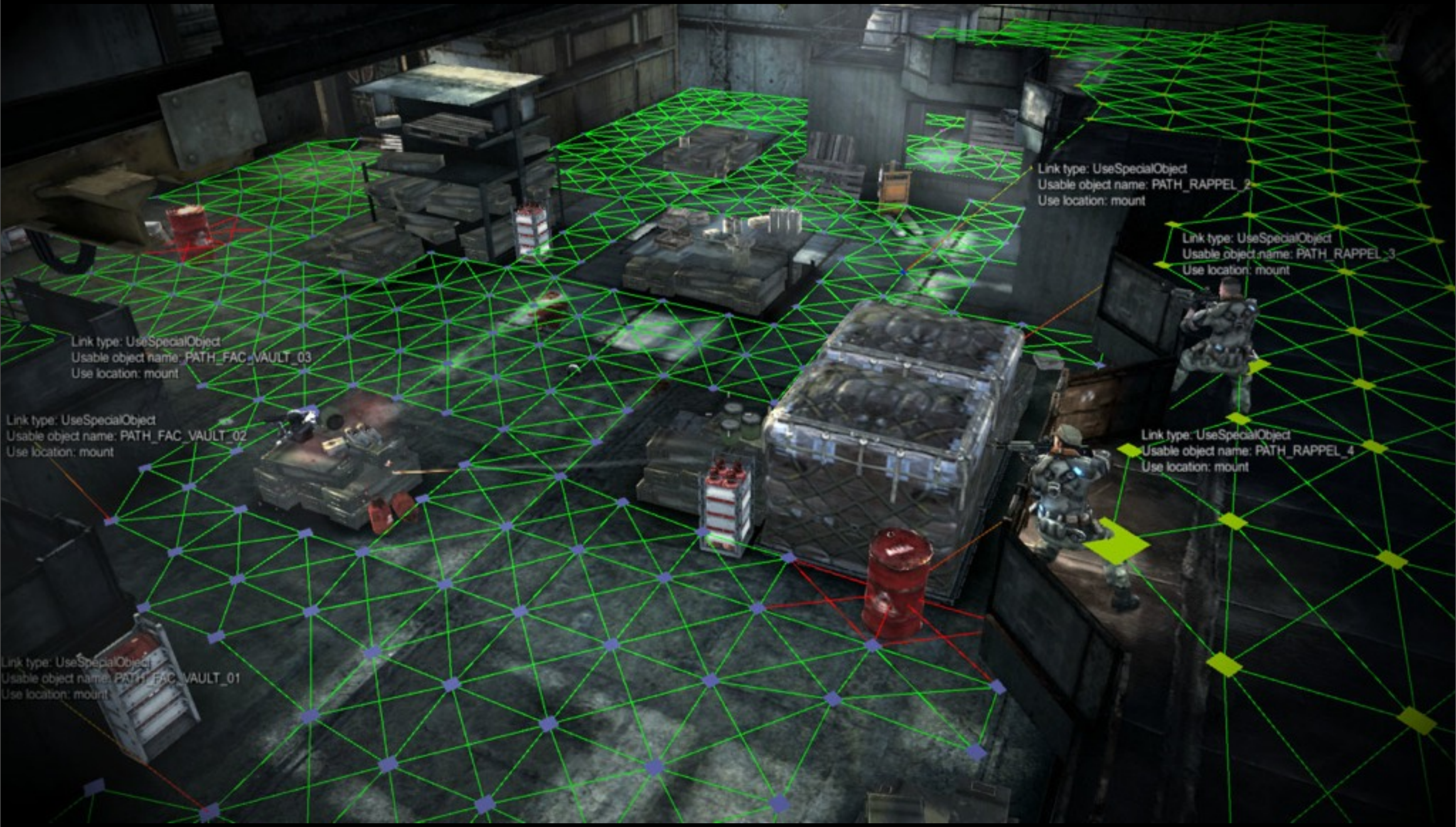
- SPU does many tests
  - Each line-of-fire against each waypoint link
  - Tapered-capsule vs. tapered-capsule

Notes:
- Challenge to make AI look believable in dynamic situations
- Standing in your or their friend's line of fire is STUPID
  - Breaks all immersion

Approach:
- For each entity engaged in combat
- Check current threat and see which waypoint links lie in the line of fire
- Hints are in the form "don't stand here or be shot in the head, KTHXBYE"

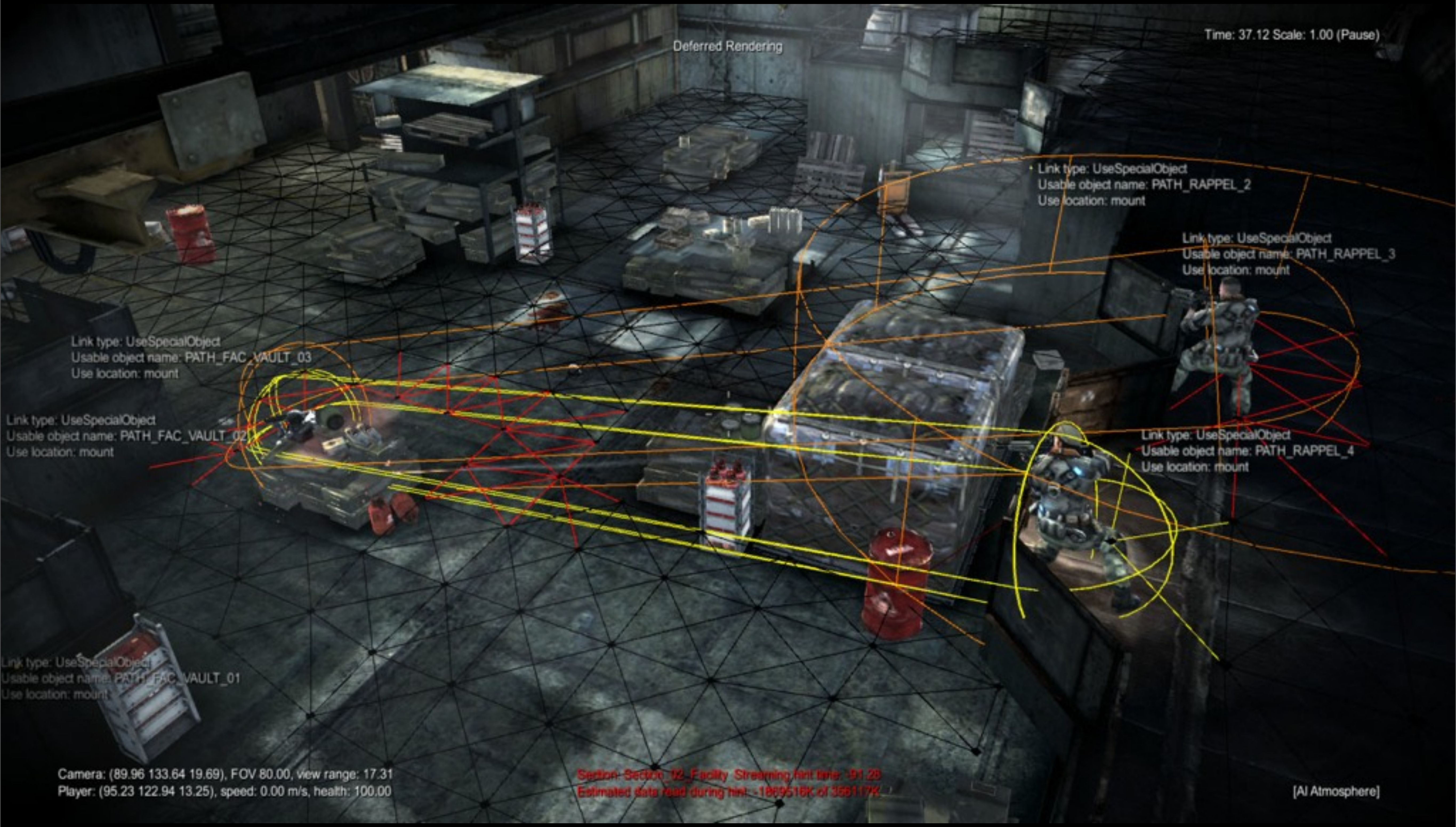
Result:
- Flanking, walking around enemies engaged in battle
- Behavior is 'interpreted' by humans as being more intelligent

LINE OF FIRE


Link type: UseSpecialObject
Usable object name: PATH_RAPPEL_2
Use location: mount
Link type: UseSpecialObject
Usable object name: PATH_RAPPEL_3
Use location: mount
Link type: UseSpecialObject
Usable object name: PATH_FAC_VAULT_03
Use location: mount
Link type: UseSpecialObject
Usable object name: PATH_FAC_VAULT_02
Use location: mount
Link type: UseSpecialObject
Usable object name: PATH_RAPPEL_4
Use location: mount
Link type: UseSpecialObject
Usable object name: PATH_FAC_VAULT_01
Use location: mount

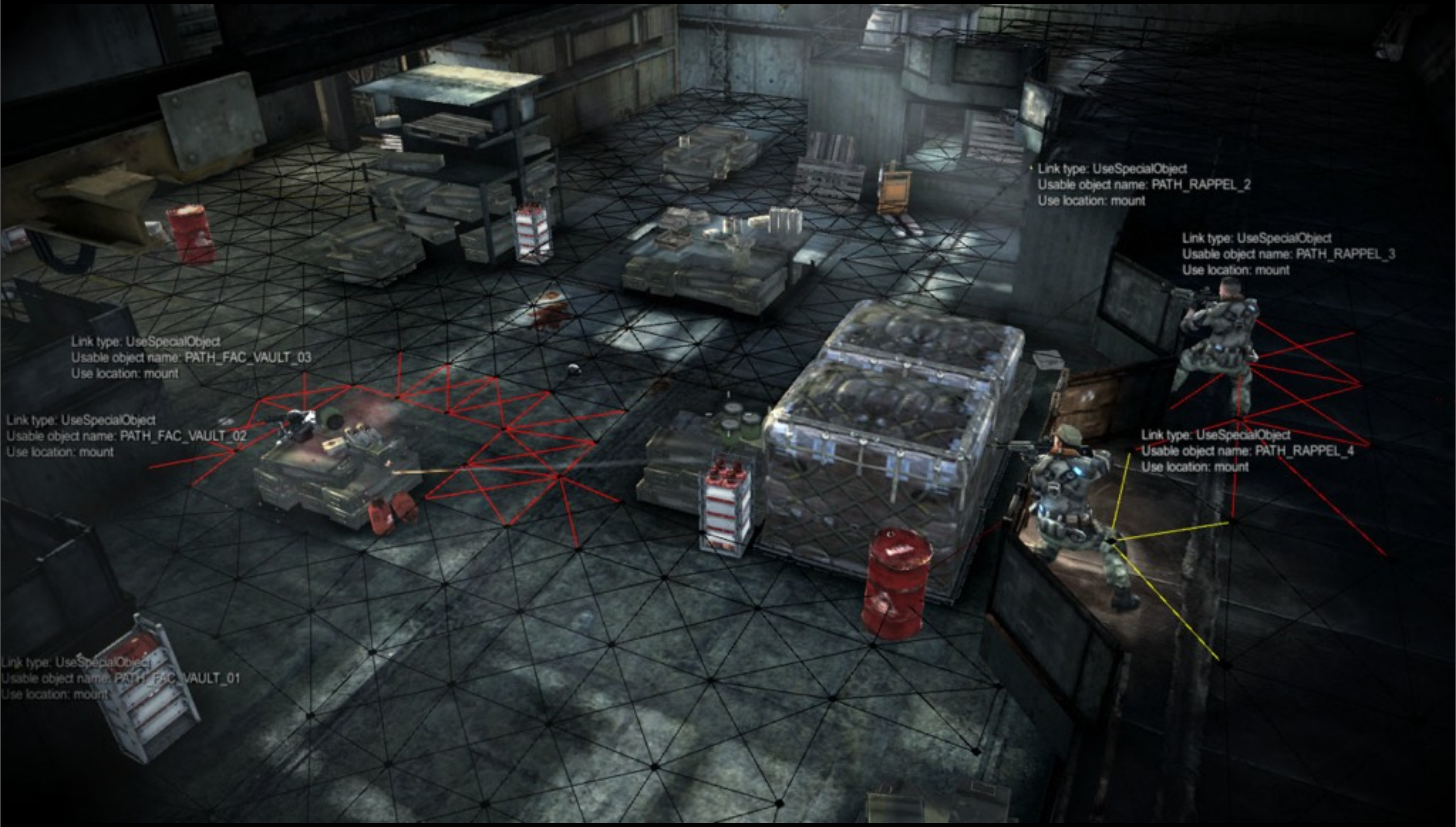
Two friendlies engaging a foe

LINE OF FIRE


Deferred Rendering
Time: 37.12 Scale: 1.00 (Pause)
Link type: UseSpecialObject
Usable object name: PATH_RAPPEL_2
Use location: mount
Link type: UseSpecialObject
Usable object name: PATH_RAPPEL_3
Use location: mount
Link type: UseSpecialObject
Usable object name: PATH_FAC_VAULT_03
Use location: mount
Link type: UseSpecialObject
Usable object name: PATH_FAC_VAULT_02
Use location: mount
Link type: UseSpecialObject
Usable object name: PATH_RAPPEL_4
Use location: mount
Link type: UseSpecialObject
Usable object name: PATH_FAC_VAULT_01
Use location: mount
Camera: (89.96 133.64 19.69), FOV 80.00, view range: 17.31
Player: (95.23 122.94 13.25), speed: 0.00 m/s, health: 100.00
[AI Atmosphere]

Capsule indicate Line of Fire

LINE OF FIRE


Link type: UseSpecialObject
Usable object name: PATH_RAPPEL_2
Use location: mount
Link type: UseSpecialObject
Usable object name: PATH_RAPPEL_3
Use location: mount
Link type: UseSpecialObject
Usable object name: PATH_FAC_VAULT_03
Use location: mount
Link type: UseSpecialObject
Usable object name: PATH_FAC_VAULT_02
Use location: mount
Link type: UseSpecialObject
Usable object name: PATH_RAPPEL_4
Use location: mount
Link type: UseSpecialObject
Usable object name: PATH_FAC_VAULT_01
Use location: mount

Red links = In line of fire of upper NPC

REALISTIC & INTENSE - Our AI code on SPUs allow the AI agents to act realistically

The most intense battles in our game are the ones where the AI is chasing you, hunting you down, responding to you.

It doesn't become just another pattern-matching game. These guys can be tough as nails, because they're smart.

002. KZ

PUTTING IT ALL TOGETHER

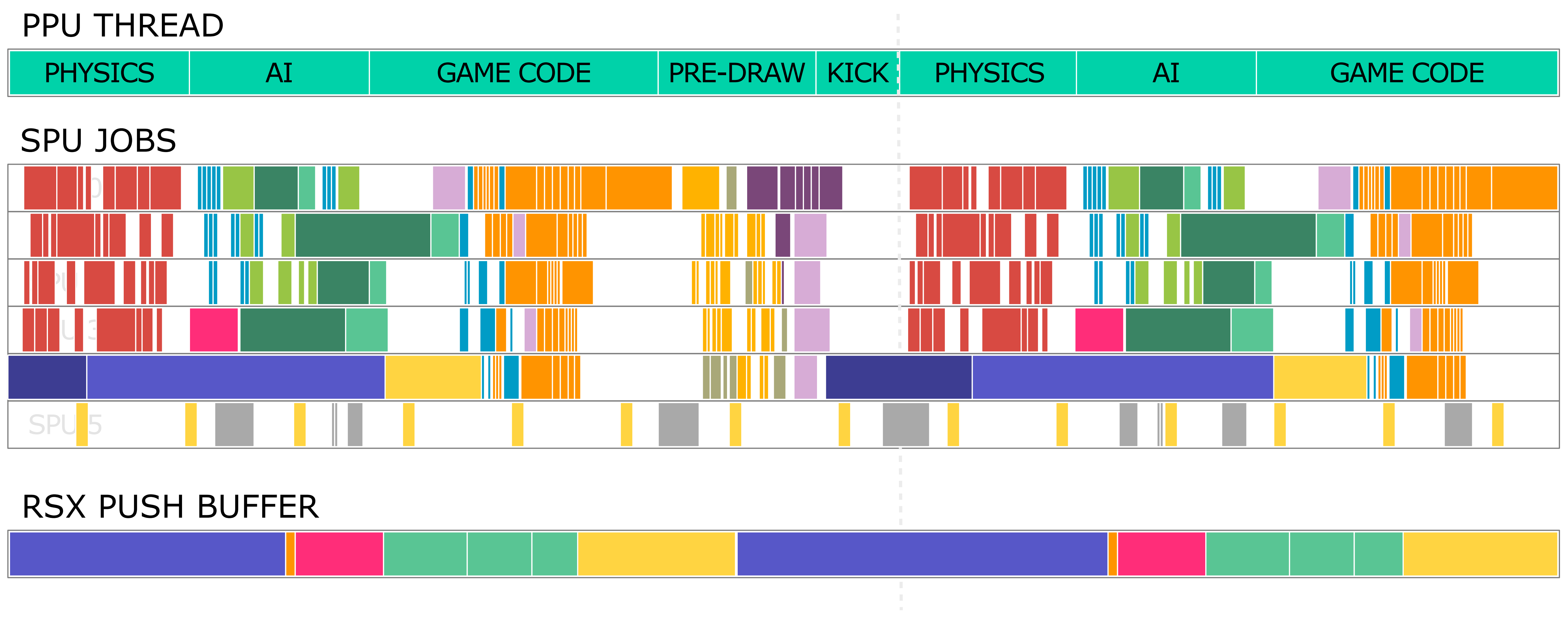
PLEASE MAKE A SELECTION 001 KZ

SCHEDULING

- Almost all SPU code we have are jobs
  - Many different job managers (middleware)
  - Managed by own high-level job manager
    - Manages the other job managers / workloads
    - Easy to write your own, or grab somebody else's
- High-level scheduling hardcoded in main loop
  - Use barriers, mutexes and semaphores sensibly
  - Could use generalization, but good enough for now
- None of this is rocket science, just work

Notes:
- High-level scheduling decisions done by hand
- Point of this all is that we had ideas on how to over-engineer it
- But under-engineering works fine and it probably won't change
- Pick whatever suits you best

- PPU and SPU can both add jobs to the queue



LEGEND

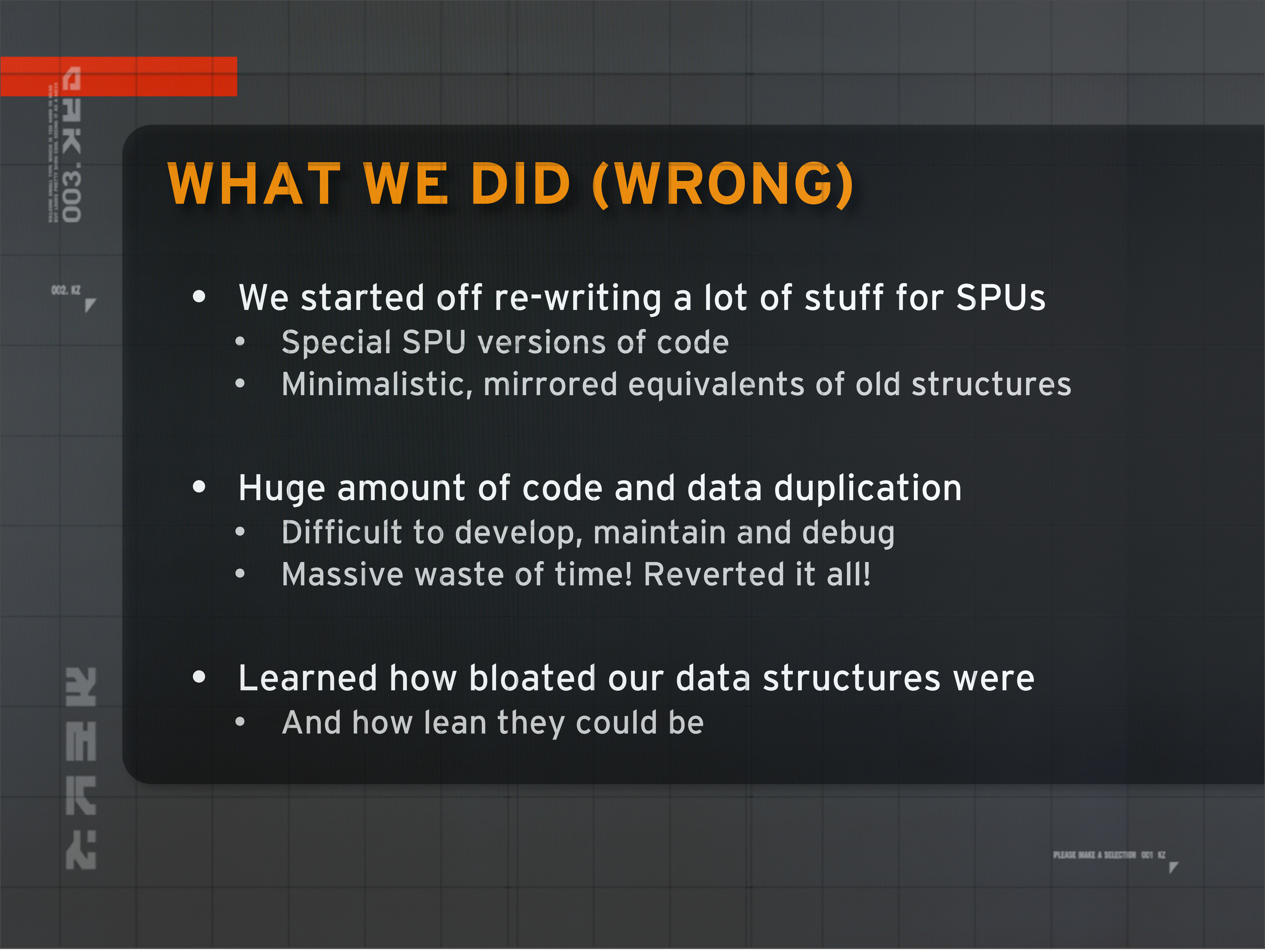
🟨 Audio	🟧 Particle update	🟨 Forward display list
⬜ System, unzip, etc.	🟫 Particle vertices	🟥 Lighting display list
🟥 Havok Physics	🟪 Skinning	🟩 Shadowmap scene graph
🟩 AI Jobs	🟦 Main scene graph	🟩 Shadowmap display list
🟦 Animation	🟦 Main display list	🟧 Post-processing
🟪 Ray casts	🟦 IBL Sampling	

Buildup is roughly:
- draw SPU6 audio background jobs, audio encoding, nextsynth
- draw other background jobs, decompression
- show AI kicking off logic jobs, picking up the results in the same frame
- show monolithic Havok block with one SPU unclaimed
- show animation jobs starting during the game update
- show particle update jobs starting
- show PPU draw activity
- show skinning jobs starting
- show particle results being picked up on PPU
- while PPU continues, SPU starts firing off particle draw jobs
- main scene graph query begins
- query continues while particle and skinning jobs run
- when query is done, the transparent, lighting and normal geometry jobs start
- the lighting job fires off a few extra scene traversals for shadowcasting
- the geometry job fires off hundreds of lightprobe jobs
- a few shadowmap scene graph queries end and start a shadowmap displaylist builder job
- the SPU post-process jobs are added later in the frame
- this concludes all of the data that needs to be done in the frame

LOOKING BACK

RESULTS

- SPUs cost us a lot of development time
- But our code is now future-proof
  - Scales well to many cores
  - Built a toolset so we can continue this path
- Supported our franchise values
  - Cinematic
  - Dense
  - Realistic
  - Intense



WHAT WE DID (WRONG)

- We started off re-writing a lot of stuff for SPUs
  - Special SPU versions of code
  - Minimalistic, mirrored equivalents of old structures
- Huge amount of code and data duplication
  - Difficult to develop, maintain and debug
  - Massive waste of time! Reverted it all!
- Learned how bloated our data structures were
  - And how lean they could be

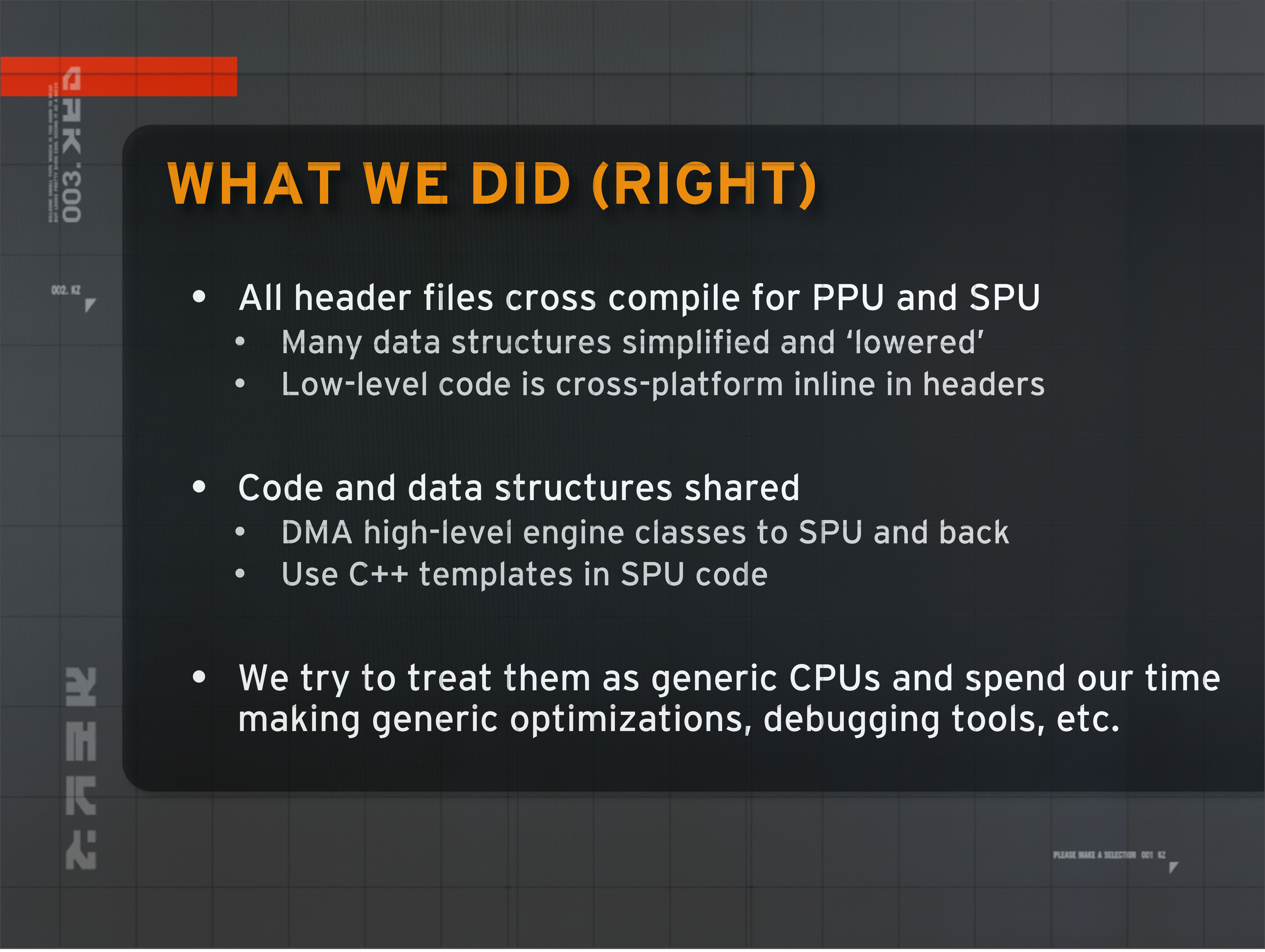
We first found it hard to optimize our code for SPUs because our data structures were full of pointer indirections and virtual functions. They were not friendly to the SPU’s hardcore local memory scheme. We also had this feeling that you had to have all your data structures for SPUs really, really optimized.

So we ended up doing what many others did and that was creating special code for SPUs that worked on simplified data structures, derived from the bloated data structures we had. This ended up as a lot of double bookkeeping and special-case code that was hard to maintain, ugly, often incompatible, and just a bad idea.

We reached some sort of mentality shift at some point when we started to notice we could actually compile our entire engine for SPU’s - even if it wouldn’t run, we could at least parse the data structures and use them directly on SPUs. So we took another good look at our data structures, and started optimizing them.

- AI coders:
  - difficult to find these tasks that have small domain and large loops
  - was relatively easy, not too complicated
  - hard to find code whose design fits the SPUs and even worse to change design

- Failed
  - A* planning in parallel (now ported to SPU for new game, but only on one SPU)

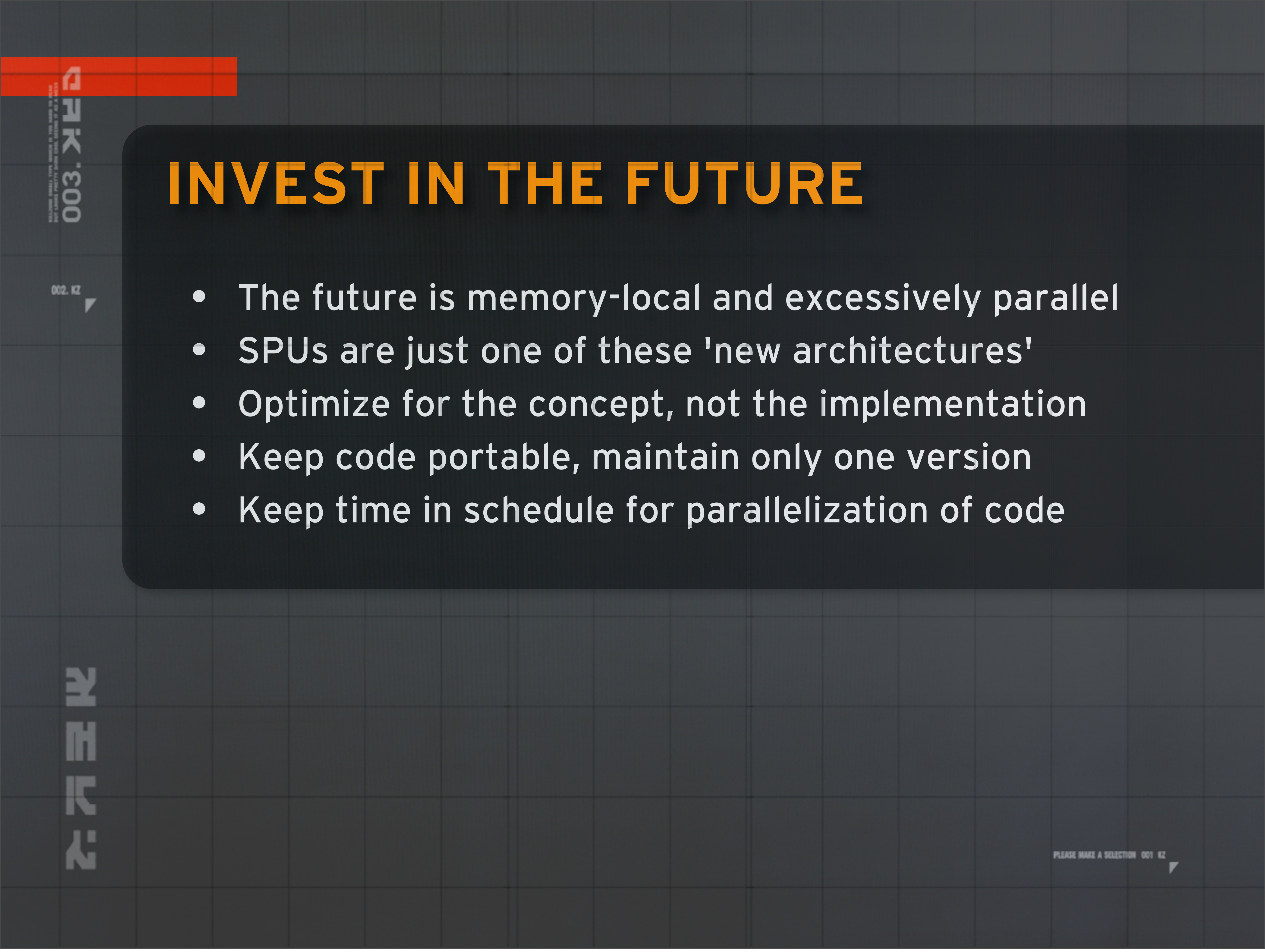


WHAT WE DID (RIGHT)

- All header files cross compile for PPU and SPU
  - Many data structures simplified and 'lowered'
  - Low-level code is cross-platform inline in headers
- Code and data structures shared
  - DMA high-level engine classes to SPU and back
  - Use C++ templates in SPU code
- We try to treat them as generic CPUs and spend our time making generic optimizations, debugging tools, etc.

Notes:
- This was a big mind-switch
- In the end SPUs are *not* 'just another core'
  - But it helps if you try to squint your eyes and treat them as one as much as possible

RECOMMENDATIONS



INVEST IN THE FUTURE

- The future is memory-local and excessively parallel
- SPUs are just one of these 'new architectures'
- Optimize for the concept, not the implementation
- Keep code portable, maintain only one version
- Keep time in schedule for parallelization of code

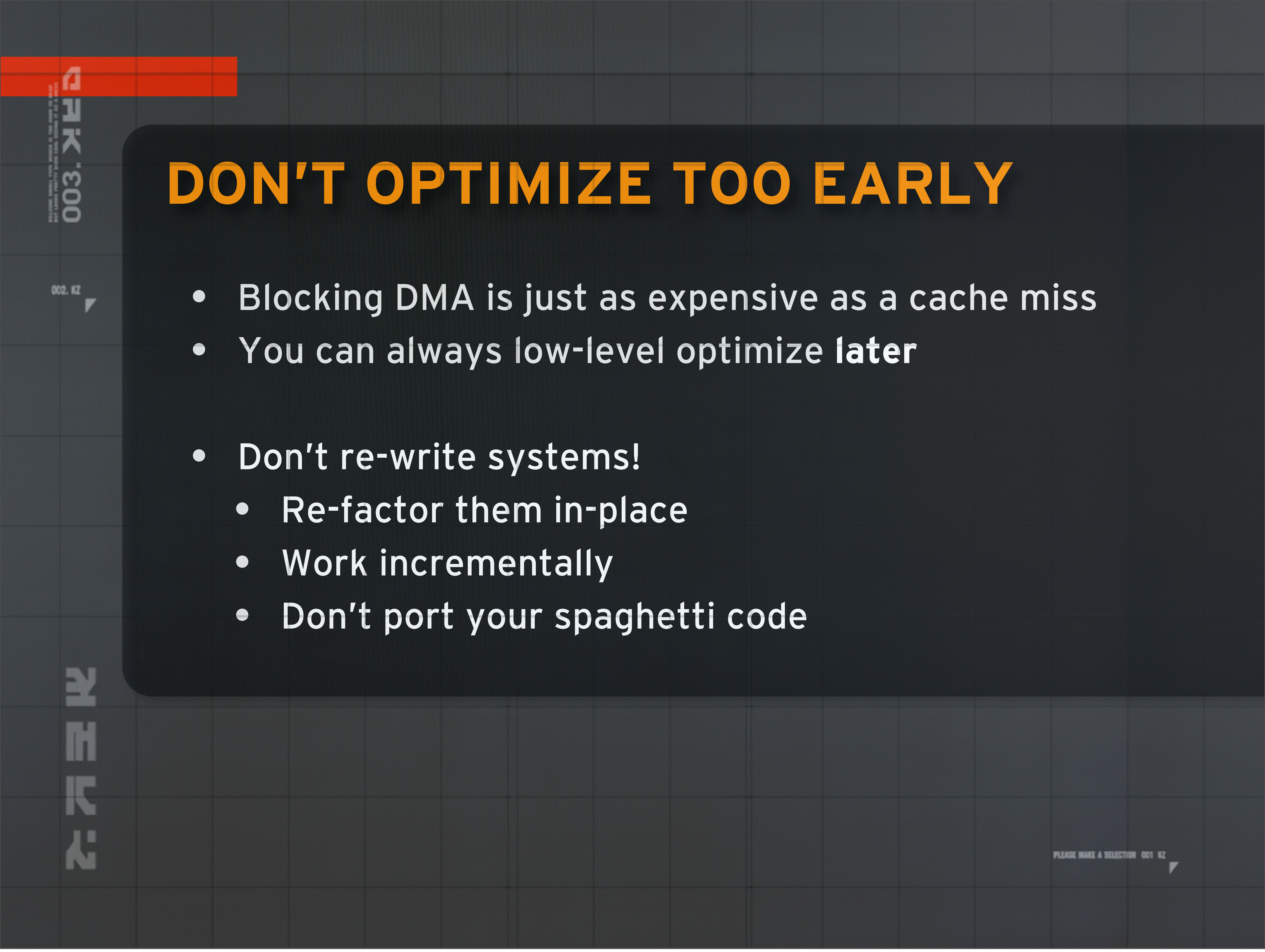
Notes:
- GPGPU, Larrabee, etc. all work well with similar algorithms

Side-notes:
- SPUs wouldn't have been so difficult if the PPU would have been faster
- But they're here and if you invest in them you'll get payoff in the future
- If you don't the problem won't go away, yes it's hard
    - But if you embrace them you'll at least get the benefit (on *all* platforms)

TREAT CPU POWER AS A CLUSTER

- Many separate equal resources
- Think in workloads / jobs
- Build latency into algorithms
- Favor computation-intensive code
- Avoid random memory accesses, globals

Notes:
- Makes it easier to think about how to schedule things
- Excessively parallel is something we already know, use that experience



DON'T OPTIMIZE TOO EARLY

- Blocking DMA is just as expensive as a cache miss
- You can always low-level optimize **later**

- Don't re-write systems!
  - Re-factor them in-place
  - Work incrementally
  - Don't port your spaghetti code

Notes:
- Optimized code is difficult to maintain

Our approach:
- If if it runs fine on the PPU -> DON'T port it
- If you get it on the SPU -> COOL!
- If it's slow but you don't stall on it -> DON'T optimize it

In general:
- BE LAZY!!

FOCUS ON YOUR GAME

- SPUs can add a lot to your game
- Do things you didn't think were possible
- Takes a lot of effort to get right

- So choose the things that will matter!

Notes:
- Almost too simple of a message
- But can't stress this enough
- Don't just port anything
- Think:
    - Will gamers notice this?
    - Improve my MetaCritic score?

- SPUs helped define the look and feel of Killzone 2
- What will they do to *your* game?

RECOMMENDATIONS

- Invest in the Future
- Treat Your CPU Power as a Cluster
- Don't Optimize Too Early
- Focus on Your Game

Thank You for Listening
Any Questions?

Notes:
- Run around and find me
- Up for sharing experiencing or code