
33 Milliseconds
In the life of a Space Marine…
WARHAMMER
40,000
SPACE MARINE
GPU Optimization

Introduction

- Space Marine is a third person action game
  - PS3, 360, PC-DX9!
- We started from the DarkSiders engine (Vigil)
  - We had DX9(-ish) across the board
    - 360 and PC, at the time the engine had some PS3 support
    - On Ps3 shipped using a DX9 layer, we translate DX9->GCM on the fly on SPUs.
  - Nice workflows. Great game.
  - Not too fast, very OO engine
    - Didn't matter…
    - …DS looks nice!

How much did we want to change?

Many more objects… Many more lights… Many more partices, decals…

What did we do?

- Changed all the rendering! Integrate in-house game libs.
  - Kept (most) of the base abstractions
  - Moved to deferred Lighting
    - Very well known. Details are tricky
    - Pushing LOTS of geometry
    - ZERO precomputation
      - Nice art workflow
  - New materials, post-fx, particles, etc...
  - Added streaming
    - Region based, not «continous». Easier/Better!
  - New job system, math, culling, containers, memory...
    - Neat stuff. SIMD everywhere, SPU friendly, lockless structures...
- We passed the "digital foundry" test!
  - You might be surprised: Six months from shipping, our framerate was not great
    - You **SHOULD**, I'm serious!

Digital Foundry: *Space Marine is remarkable in that in our tests we saw a locked 30FPS with v-sync throughout the entirety of the in-game action […] The locked frame-rate remains no matter which console you're playing on: both 360 and PS3 are remarkably solid throughout. Space Marine's sheer consistency is a great asset: controller response feels good and there's no lag regardless of how much is happening on-screen.*

Good games are not made by caring about bugs, memory, performance and so on at the last minute. Keep your framerate _always_ during the _whole_ production. When you finish the «space», optimize, as you go... We were lucky, we had an amazing optimization team, and we did a LOT of overtime. If you consider overtime «normal», please change industry ☺

Performance. How?

- We started caring too late… ☹
- …Everything matters!
  - **LOTS of <0.5ms optimizations**
  - Tried multiple variations/tricks everywhere
  - First thing: Do less! Cull!
  - Occlusion: hardware, software. Frustum, size, LOD…
- The game was not 100% job-based ☹
  - Classic sim/render thread split.
  - GPU draws were single-threaded ☹ No time to change this.
- Bonus difficulty: **No buffering**, no latency by design
  - Only concession: a one frame buffer between sim/render threads
  - Hand crafted render thread execution "blocks" to avoid stalls between CPU and GPU
    - Cull a bit, kick draws, cull more, kick draws etc… Balancing is an art… Next time, let the scheduler (job dependencies) do the magic.

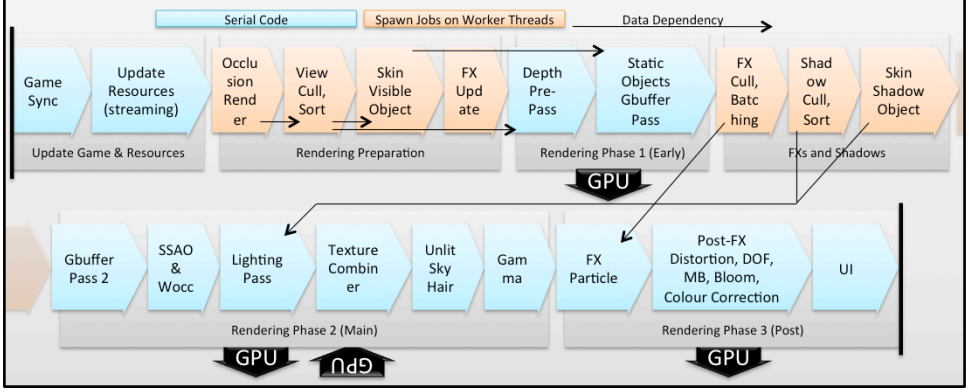
We didn't allow the GPU to run one frame behind of the CPU. This was tricky not only because the CPU rendering has to be fast enough and generate enough work to keep the GPU from stalling, but because we also had dependencies from the GPU to the CPU, reading back the occlusion counters and waiting on fences to update dynamic streams and textures.

GPU Optimization…

- Everybody knows: **bottlenecks, balancing**
  - ALU vs Memory bandwidth
  - Memory latency vs «Threads» (registers)
  - SM also has: CPU vs GPU (no waits)
  - And everything else...
- But also remember: **do less! cull & sort**
  - Early-z Early-Stencil, “ifall”, texkill…
  - Overdraw, sort order, z-prepass...
  - Small triangles/non-full quads
  - Culling! Occlusion!
    - CPU, GPU queries, GPU HW predication...

Render Frame Overview

- Rendering:
  - Split serial rendering code “when needed”
  - Kick processing on worker threads…
  - …while creating GPU commands with the data that is ready



Rendering Phase 2 emits a command buffer but also waits on the GPU for HW occlusion queries to be used in the Texture Combiner stage. There is no stall as typically Lighting and SSAO take enough time to cover the GPU latency.
Timing is fundamental, not only between serial parts and spawned CPU tasks, but also between CPU and GPU.

Brace yourself...

...delving into a frame

1) Game sync

- Start of the frame is serial, needs to be quick
  - GPU is usually still busy with draws submitted in the last phase of previous frame here
    - But we want to minimize the amount of serial CPU code
  - Game syncronization
    - Copy animation to rendering
  - Resource update
    - Streaming, create and destroy GPU data

2) Rendering Preparation

- Start processing **main view only** (for the first batch of GPU submission). Queue a chain:
  - Kick **software occlusion rendering** jobs
    - Merge sw occlusion Zbuffers (min of all)
  - Kick frustum/LOD culling jobs
  - Kick visible object skinning (on 360 and PS3, PC does Gpu skinning)
- Kick particle emitters update
  - In SM particle effects can also submit meshes and decals

Preparation: Culling

- Frustum
  - Sphere and OOBB
- Software Occlusion Rasterization
  - SIMD triangle scanline rendering, no tiles, no multi-res
    - Occluder primitives hand-authored ☹, previous system used anti-portal planes (without merging…), we used the already made planes as occluders…
    - We didn't do Z-Buffer reprojection ala Crysis. No time, and no need, the renderer was fast enough…
  - Divide primitives evenly between multiple small ZBuffers
  - Merge Zbuffers together (nearest)
- SW Occlusion Query: rasterize bounding boxes, don't write z, just test. Again in parallel on SPUs.

GPU Z-Buffer reprojection does not work well with moving or skinned objects, either we have to ignore these in the reprojection (marking them in the stencil) and leaving holes or we have to reproject everything potentially leading to false occlusions. Crysis 2 does the latter, using a median filter to fill small holes after the scattering.

Coupling Z-Buffer reprojection with simple occlusion meshes would be the best solution.

Automatic generation of LODs is simpler than the automatic generation of occluders, as the latter have to be inscribed inside meshes, and these are often open or present complex degenerate cases in games.

Standard vertex LODs are not so important for us also because our current engine is very heavy on material costs (GPU context switching, more CPU objects, resources in flight). LODs were needed to collapse materials and reduce the number of CPU objects in flight in a frame, more than reducing vertex counts.

Two interesting approaches for occluders are:
- Simplification Envelopes (guaranteed maximum deviation from original mesh) by Cohen-Varshney-Manocha-Turk-Weber-Agarwal-Brooks-Wright
- Voxel Mesh Processing. Voxel rasterization can deal with holes (by scanning using

Preparation: Culling

- **Many fine tuned heuristics** to remove elements based on distance/screen size/view angle etc…
  - Lights, Objects, Decals, Shadow casters (objects, lights)...
- LODs
  - In our engine each object has a single material/GPU state set. Fat structures, indirections, setting materials (textures!) was expensive…
  - …Reduce the number of objects in flight! **LODS merge materials**.
    - Unfortunately, hand-authored ☹.
    - Next time: bake texture -> vertex colors far away and use for these objects a diffuse only, vertex colored material. Big win!
  - LODs also reduce "partial quads" due to too small triangles, increase shading efficiency
    - Less vertex processing usually not as important

Culling Results

- Low overdraw in the gBuffer pass
  - Even lower in the more expensive material pass due to further HW Occlusion Query culling… by then we only draw what's surely visible, and we have zero overdraw
- Good quad efficiency (75%+)

3) Rendering Stage 1

- Update the Colour-Grade (if needed)
  - 3D Volume Texture
  - Blending on the GPU when transitioning regions
- Depth Pre-Pass
  - **Very few static objects**
  - Heuristics based on distance, screenspace size
  - Sorted front-to-back
- G-Buffer Pass
  - **Sorted by material and roughly front-to-back**
    - Split into two chunks (near and far objects)
    - Again, setting materials was expensive… indirections...
  - Single RGBA 8-bit buffer
    - RGB: Normal. Alpha: Specular and a "rough" surface bit flag
  - Stencil flags for moving objects and "no decals"
  - **Emit GPU Visibility Queries** around all draws
    - Not on PS3… only PC and 360

Do you want hot-swappable assets? Streamable assets? Don't use reference counters with indirections (handles). Instead, keep a list of things that you need to patch if the asset swaps and use direct pointers. Or if you want handles, do handles to arrays of resources, not single ones…

4) Rendering Stage 2 (Main)

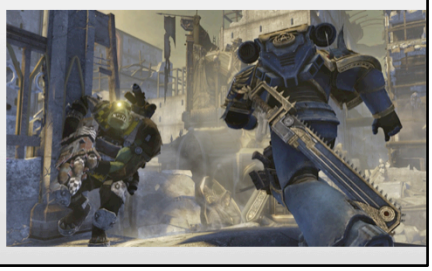
- Gbuffer "late pass"
  - We still have to render some "late" objects in this CPU phase, these were waiting on earlier jobs…
  - Wait on the FX system update tasks
    - FX can **submit meshes and decals**!
    - Render those here if needed
  - Add depth-only objects
    - Hair, and "Unlit"
    - Won't be shaded by the deferred lighting system
- Splat volumetric decals normals (more on later)
- Resolve buffers
  - We generate a R32F linear view depth buffer
  - A half-res min-max linear depth one
  - A half-res normal buffer
- SSAO and World Occlusion

We use the zbuffer often to convert to view and world space. We convert the depth to a linear R32F, it seems to be worth the cost. On PC we use MRT to render to the R32F while doing the Gbuffer pass (as we can't easily read the hardware Z). We use hardware Z readback on PC only for PCF shadows (as that is way more supported across our hw range than the raw depth reads).

4) Rendering Stage 2 (Main)

- Lighting
  - Deferred shadows buffer
  - Deferred lighting

- Texture Combiner
  - First all the objects
  - Then we splat the volumetric decals textures
- Sky and Hair



- Gamma conversion

Screen Space Volumetric Decals

- Deferred lighting: not only for dynamic lights.
- **Decals are cheap too** (no need to compute lighting twice).
  - Volumetric decals, apply textures on the intersection of a box and the scene…
  - We also supported standard geometry based decals. No need to blend too many textures everywhere in the material shaders.

1. Intersect boxes with the scene
   - Prime stencil (early-stencil!) by drawing box backfaces
   - Handle camera inside the decal: disable Depth Test and draw directly front faces
   - **Early-Stencil is the key – optimized per platform!**
     - Used to mask objects that don't receive decals
     - Characters and object moving relative to the camera
   - Optimization for:
     - Thick decals – box does not intersect scene "much"
     - Camera inside decals – would execute shader on the whole screen otherwise
2. Draw boxes front faces
   - Reconstruct scene world space position from zbuffer
   - Transform into local space of the box
   - Project textures
3. Clear stencil

Volume Drawing, we use it a lot

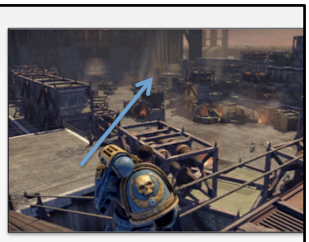
- Lights, shadows, decals… In theory it's easy...
  - Three pass method, backfaces, frontfaces, draw. Two pass variants well known.
- …In practice, lots of different small tricks
  - On 360 alone, experimented with at least 5/6 different variants!
  - …use cw/ccw stencil, stencil zfail… what happens with early-z… when to clear stencil?
  - More variants due to different use cases, lights are not 100% the same as the decals, big lights versus small lights, camera inside the volume etc...
- Optimized volume geometry
  - Avoid having too many edges -> partial quads
  - Fullscreen draws: per platform! Quad-list? Single big triangle? Point-sprite grid?
  - Many other options, i.e. "rasterize" the primitive in software into a point-sprite grid?
- **Know the details of your architecture!**
  - Especially, **early-reject** buffers... (PC landscape is messy if you consider old cards and laptops). Per pixel? Per quad? Heuristic? < or <=? Trivial reject or also trivial accept?
  - Many tricks are really GPU dependent... I.E. Branches in the shader or priming early rejects? Can you alias buffers with MSAA ones? Separate controls for early-reject versus final test?

Ambient Occlusions

- SSAO (~1.4 ms)
  - 2d circular kernel, clamp min and max proj. size
  - No noise pattern, no post filtering
    - Noise does not work well at half-res
    - We prefer having more samples than using time to blur
  - Simplified line sampling (ala EA/MMA)
  - Normal-aware
    - Fundamental to avoid the "spotlight" look
- **"World Occlusion"** (~0.6 ms – no update)
  - AO from above
    - Render orthographic map top-down. Blur
  - Works under arching structures
    - Uses 4 layers of depth peeling (packed into a RGBA target)
    - Difference between distances = occlusion
    - Expensive to compute**, updated in small tiles** (only static objects) **when needed** (around player movement)
- Both half-res
  - Using half-res min-max depth
    - Tricky when sampling normals (don't have matching normals on max or min depth). Edge artifacts…
    - We use a combination of min and max during SSAO, and sample from full-res normals

Fun fact: We didn't include sky in the gbuffer pass, so the sky region would contain random depths on PC (depth written with a MRT r32f buffer, which is not cleared every frame the actual Z is). That generated a high variance in the kernel sizes for AO in the area which trashed the cache ☺

4a) Shadows



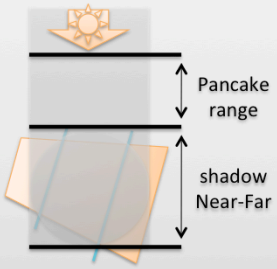
- **Big shadowing distances**
  - In/Outdoor mixed scenes, couldn't cheat much
  - Spot, Ortho and Poing light shadows
  - Stable Sun Cascade Shadow Maps
  - Again, Details/Optimization are not easy…
- **Deferred shadow buffer**
  - Render all the shadows in screenspace first…
    - Can't have more than two intersecting lights plus sun
    - RGBA buffer. R = sun, G and B = other lights
  - …then render lighting
  - Only one shadow buffer in memory at any time
    - Simpler and uses less memory. Nice cascade blending... We didn't want to use tiles on 360
- Culling: Frustum, Occlusion and Size!
  - Similar to main view, plus **HW predication**…
    - Plus artist-set ☹ flags (not caster, caster only)
    - Already-in shadow (fixed sun direction) by another caster...
  - Culling is VERY important for the performance

The only platform on which the single shadow buffer does not work too well is Intel Sandy Bridge, which for now hosts only a single early-z rejection buffer, thus switching depth buffers continously invalidates it all the time. It's fundamental in a deferred renderer to preserve the early-z rejection of the main scene depth across the pipeline.

The culling system is similar to the one we have for the main view, but the occlusion geometry was authored too loosely and would discard important casters in some situations. As we couldn't fix that art problem at the time, we did employ only single occluder planes instead of relying on the software occlusion rasterizer.

Sun shadowmap generation

- One cascade at a time
  - We don't pack the cascades. We resolve the whole shadowmap.
- We fit the cascades tightly on the depth axis (**pancaking**)
  - Objects intersecting near will be deformed if vertices behind near are flattened on the plane.
  - We pull back the near plane a bit, so that does not happen often for objects
  - Rarely still have bad self-shadows -> ask the artists to subdivide large polygons
  - Think: what happens to early-z to the squished objects? Per platform? Use stencil?
- Experiment: 5 CSM, 4 computed in an interleaved fashion, every other frame
  - Worked, but artifacts with huge moving objects
  - Could have scripted it disabling in given set-pieces



The half-frame rate shadows are implemented in Crysis2. Crytek confirmed that they disable them in situations where the trick won't work.

We experimented with splatting back the moving objects every frame. Resolve and memory costs didn't make this worth for us.

Notice that stable cascades can be seen as a "window" over a big orthographic projects of the whole scene. We just shift this window around frame to frame. If everything is static, we could see the previous frame data as a cache, we could just render the new small border that results from the intersection of the previous frame window with the current one.
The problem there is still with dynamic objects and with the fact that at each frame we find a new near-far z-ranges. The z-range problem can be solved by reprojecting (but that would lose some resolutions) or by writing an index in a buffer (stencil?) which corresponds to an array that remembers which near-far was used for that pixel, thus allowing correct reprojection when computing the shadows.

Note that all this also means that using the previous frame depth for occlusion culling of the next one is going to work well, because other than moving objects and the border where we don't have data, the rest will be exact, no perspective distortion and no scattering is needed.

Sun shadowmap splatting

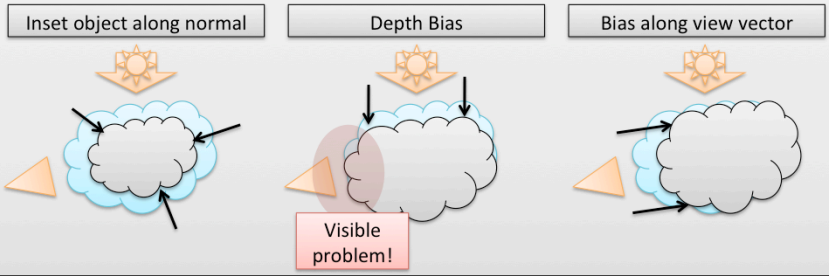
- Four 800x800 stable CSM cascades per frame
  - Depending on the platform (and cascade) 4x4, 3x3, 2x2 PCF
- We rely on early-stencil when splatting shadows in screenspace:
  - Render cascades front to back, using planes in the world and z-test…
  - Equivalent to selecting the cascade with the CSM frustum splitting plane.
  - Stencil mask any pixel that intersects the cascade
  - In the subsequent cascades, stencil test to avoid shading pixels already handled by previous shadows
  - Small regions shaded twice to interpolate between cascades

The shadowmap size was choosen for performance and to fit into 360 EDRAM…

Note: Best-cascade selection is possible too: draw on screen a volume that corresponds to the frustum of a given cascade minus the geometry of the subsequent one

Shadowmap Biasing

- Avoid self-shadowing artifacts due to precision
  - **Good**: Inset along the geometric normal
  - Bad: depth bias in shadow space
  - **Cheap hack**: a combination of a shift along the shadow z and (more important) along the view vector (optionally, scaled by V dot N)



Other shadows

- More occlusion culling…
  - Via HW Predicated Rendering
    - **Pre-pass for shadowing lights**: draw volume only, predicate on the second pass
    - We use the results of this predication around shadow map generation draw calls (and lighting, as we have it)
  - Done before sun shadows, so we have a lot of latency for the result to "come back"
- Shadowing (all of this is predicated)
  - Generate shadowmap – 800x800
  - Splat into deferred shadow buffer (R-G channels) by drawing light volume. 2x2 PCF (no LOD)
    - Again, we use stencil and early-stencil to avoid computing the shadows on pixels outside the light volume

4b) Lighting

- ~30 active lights per frame, not uncommon (worse case, hundreds)
- Blinn-Phong specular. Our own **Oren-Nayar diffuse** approximation (for Sun, Lambert for other lights)
  - Not all lighting is deferred: Character shaders add their own "beauty" lights
  - **Gobo/Masks support**, saves on dynamic shadow caster rendering
  - Shadow buffer is an input, but not AO, that is mixed by the material shaders for extra control
- Like shadow splatting: Shade only pixels inside light volumes
  - Stencil used to mark volumes
    - Only one bit of early-stencil on 360 and PS3
    - Can't use early-stencil for anything else!
      - Can't do multiple BRDF models
  - We don't **compute sun in full shadow**
    - Before rendering sunlight, we prime early-stencil using shadow buffer!

Next time: Tiled deferred lighting. Tile classification can operate both on light volumes and on materials

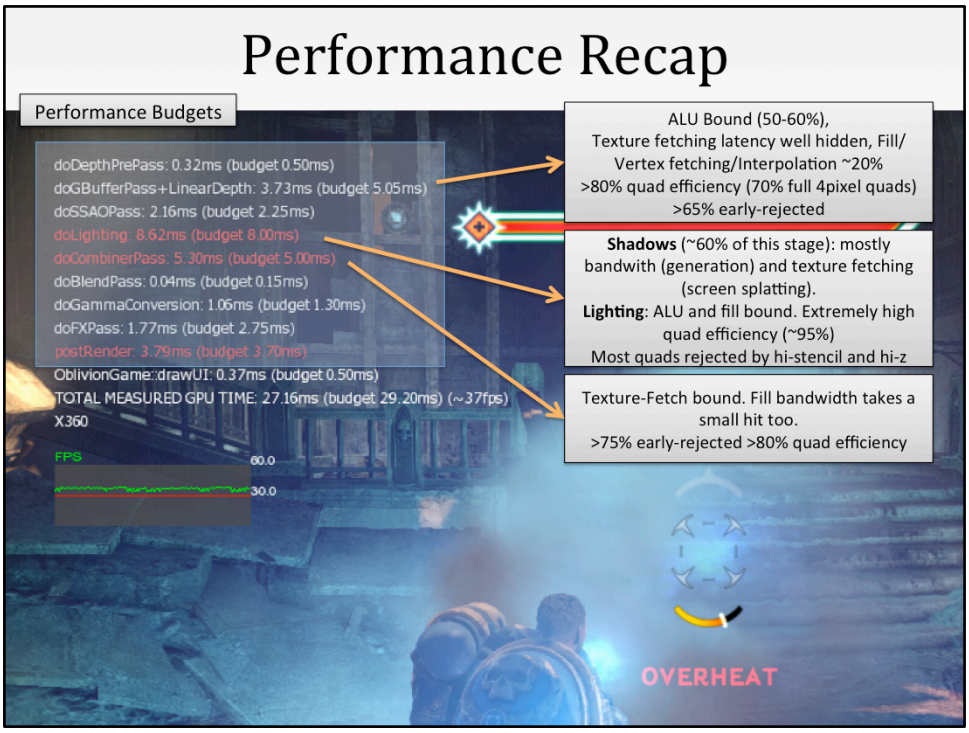
See – Rendering Tech of Space Marine (KGC 2011) for the details of the Oren Nayar approximation!

4c) Texture Combiner (Material pass)

- **Zero overdraw** is fundamental
  - **More culling,** read the HW occlusion query results. Better CPU performance, less draws
  - Perfect Early-depth rejection (know your hardware early-z!)
  - Draws are only sorted by material now
    - Minimize state changes, expensive on CPU in our engine
- Our shaders are rather complex
  - One or two diffuse and normalmaps
  - Anisotropic filtering, only on diffuse
  - Handle ambient lighting and occlusion, fog
  - "Character lighting": two cheap camera-relative rim lights (in darkness only)
  - Shaders are generated with a graph-based tool
    - Graph-based for deferred materials is not bad
    - Most (all) of what the graph does is texture layering
    - Unique shaders and post-effects are hand coded
- After geometry, we do a second decal pass
  - First decal stage is only for normals, this is for colour
  - Combiner stage marks stencil again (same as gbuffer, for pixels which don't receive decals)
  - This pass uses occlusion query results as well

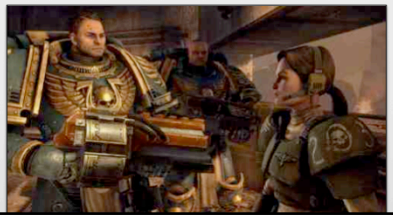
We have normal-only decals and colour-only decals too. Most are both though and get rendered in both passes.

Performance Recap



Sky and Hair

- Skybox
  - Two dxt textures, **yCoCg encoding**
  - Fogged (ground, not distance)
  - Writes in the alpha an amount that will override the luminosity used for bloom computation
- Hair
  - Alpha-blended strands
    - No forward rendering (we did it, was expensive)
    - Solution: **"Suck" lighting** from the skull lighting behind
  - Depth written in the gbuffer pass (not normals) so it will be resolved in the depth buffers, used to occlude particles, decals.
  - Normals used as shift amounts, how much we have to offset in the lighting buffer to find the skull lighting



Gamma Conversion

- Everything is gamma corrected
  - …but the particles (historical reasons)
  - Even alpha-blending and decals!
  - Hardware gamma texture fetching
    - Matching the output on all the platforms is not easy...
    - Performance implications, but we were not texture bound...
    - Next time: software gamma 2 is not a bad option
- 16bit linear buffers for lighting and combiner
  - Beware of banding!
    - Again, per platform tricks, understand your platform details...
  - We don't care much about having a large HDR range
    - A bit is useful for bloom, DOF, Motion Blur
    - But no game renders in physical measures, everything is authored with some tone mapping pre-applied
- HDR Resolve / Gamma conversion
  - Packs into alpha the luminosity for bloom
  - Sky overrides luminosity with its own alpha output
    - Artists wanted a precise, authored bloom control on sky

5) Rendering Stage 3 (Post)

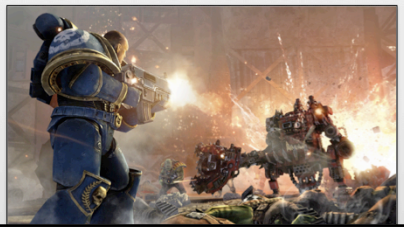
- Particles
  - Full or half-res, depending on load
- Post-render
  - Resolve a half-res copy of the buffer
  - Distortion Texture
    - More particles, write distortion vectors to a half-res target
  - Depth of Field and Motion Blur
  - Bloom
- Final mix all + colour correction
  - Distort, AA filter, Bloom, DOF/MB, Vignette, colour volume

- Per region settings
  - Artists can set parameters per region (world boxes)
  - Parameters (and colour correction) crossfade
  - Not only for post-effects… lighting and more…

We use the zbuffer often to convert to view and world space. We convert the depth to a linear R32F, it seems to be worth the cost. On PC we use MRT to render to the R32F while doing the Gbuffer pass (as we can't easily read the hardware Z). We use hardware Z readback on PC only for PCF shadows (as that is way more supported across our hw range than the raw depth reads).

FX Particles

- Our game is very particle heavy
  - When particles take too much time on screen, we switch to half-frame resolution
- Particle emitter
  - Tree-based, single effect
  - Has a strict priority ordering of its particles
  - Each emitter is culled (Frustum, occlusion), not each particle
  - Emitters are depth-sorted
- Particles are batched all together in big streams
  - We use a single 4K texture across all the effects of a level
  - Only two shaders: standard (10 instructions) and animated (flipbook – 12 instructions)
  - Pre-multiplied alpha lets us mix alphablending and additive in the same batch
  - Particles are "soft", faded with the z-buffer
- Particle types: Screen aligned, Normal, Streams



Half-res particles...

- Again: details are different per platform!

1. Render to a half-res offscreen buffer
   - ColorBlend: src->one, dest->invsrcalpha, op->add. AlphaBlend: src->zero, dest->invsrcalpha, op->add
2. Identify edges and prime the early reject buffers
   - We use the min-max zbuffer to identify edges. We take 5 samples in a + arrangement
   - Create an edge mark in early-stencil
   - Hi-stencil test can't be inverted via state...
   - "invert" hi-stencil into early-z!
3. Upsample half-res onto full res
   - Mask out edges using early-z
   - Same blending as first step but dest->srcalpha
4. Fill edges with a second particle pass
   - Render particles again, full-res
   - Only in the edges using early-stencil
   - Need to bias mip levels to match half-res
   - Same blending as first step

Half-res particles artifacts

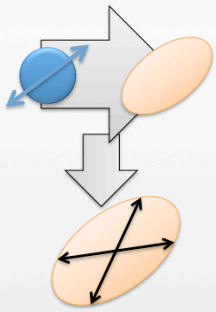
- Half-res soft particles are not easy!
  - Depth-fade means that they are not either in front or behind an object
  - Many strategies, **none** is perfect
    - Simple: render half-res with z-buffer subsampled taking the farthest depth. During upsample, if full-res z-buffer is in front of half-res, mask the particle
    - Bandwidth intensive: render two particle colours correspoding to nearest and farthest depths, during upsample, interpolate
  - Our choice is **almost perfect**, we re-render edges…
  - Where is the defect?
    - Half-res buffer starts black, Full-res edges not. Clamp to white at different points, defects are evident if many additive particles are drawn and then a multiplicative one comes on top.
    - Possible solution would be to render edges to offscreen and then blit them back, but it's expensive (resolve time...)
    - We tuned the game, it's hard to see it. It's even very hard to see the half-res switch!

Bloom and Distortion

- Pretty standard techniques
  - But we were careful in the implementation
- Distortion particles
  - Used also for portals and environment elements
  - Careful with depth-testing, avoid dragging in foreground elements
  - Had to disable linear interpolation
- Bloom
  - Colour buffer alpha = input for bloom amount
  - Pyramid, filtered with separable gaussian
    - ½->¼, ¼->1/16, blur 1/16, 1/16->¼, blur ¼
  - Careful to avoid **linear upsampling "blocks", strobing and aliasing effects**... Also beware of interpolator precision on different platforms!
  - Anyway, after quarter-res, everything is dead cheap
  - Using linear interpolation to reduce the taps

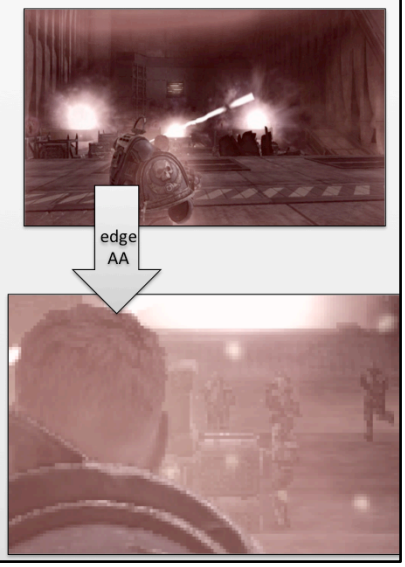
Depth of Field and Motion Blur

- The two effects are very similar
  - We compute them at the same time (total: 1.3ms, always on!)
  - Motion blur is along a line, DOF has its kernel
  - We skew DOF kernel along MB line
  - We fake a bit of HDR by reading the "bloom" channel
- DOF Kernel
  - "Separable" gaussian
  - We choose the two axes per pixel
    - We start with the motion blur direction
    - And an orthogonal axis scaled according to DOF
    - Then we compute two diagonal axes from this coordinate frame
      - Otherwise, with zero DOF the second axis (blur pass) would have no influence (waste samples)
    - Problem when MB is too small
      - We have to fade towards a fixed reference frame
  - Sample weighting
    - Scatter-inspired
    - If a sample max blur lenght would reach the center of the kernel, then gather
- Post processing of the axes (was cut from final game)
  - Scatter (to allow DOF to spread from foreground objects)
  - Edge-filter AA (as we compute DOF on a half-res texture)
  - We also had MB on moving objects, now we mask them



Final pass: Merge, Colour Correct, AA

- Fetch screen buffer using UV from distortion
- Edge-Antialiasing
  - Works surprisingly well: similar to Nvidia's FXAA, FXAA was not out yet ☺
    - Find edges and their gradients in image space
    - Filter orthogonally to the gradient
- Merge DOF/MB
  - Always on, not only for cinematics
  - Motion Blur is also used for "explosion blur", by tweaking the blur axes...
- Add Bloom
- Add Vignette
  - A cheap superellipsoid
- Colourgrade
  - 32x32x32 lookup texture
- Single pass for all the above: ~1.5ms



Optimization recap…

- **Hand-tuned all shaders**. Instructions, Register Usage, Halfs etc...
  - In the end our pipeline was really balanced, VS vs PS usage, almost always ALU bound...
- **Many little things** matter (accumulate), i.e. avoid non-needed clears and resolves
  - I.E. If sun, we draw that first, as it's fullscreen it avoids a clear of the lighting buffer. We collapsed all post together etc...
- **Early-rejection is fundamental**, we rely on in for many different effects! (stencil and depth)
- **Per-platform details can make the difference**. Not that many msecs, but are the key to find enough space to achieve a stable framerate
- **Optimized quad usage** (LODs, and selecting the "right" primitives for screenspace operations)
- **Good culling is fundamental**. Not too hard, but more heuristics to tune
- We rely on the CPU/SPUs quite a bit during rendering (skinning, culling...), **in a game with little latency, CPU-GPU dependencies are tricky**
- **Chose the right battles.** Much performance can be wasted in effects that don't make the same visual difference than others... Not all math is useful.
- **Authoring matters.** If artists have to deal with technical issues (LODs, splitting/merging of objects and instances, Occlusion volumes etc...) not only you'll waste their time, but these choices can't change!
- On the CPU, by far the first issue was to try to reduce **the cache misses** (unsurprisingly)
  - Unfortunately, many of these are generated by a given architecture and can't be optimized late...
  - Post-Mortem we estimated that Space Marine could have pushed even 50% more objects if we could have done some fundamental changes in the architecture