GDC

Higher Res Without Sacrificing Quality

+ Other Lessons From

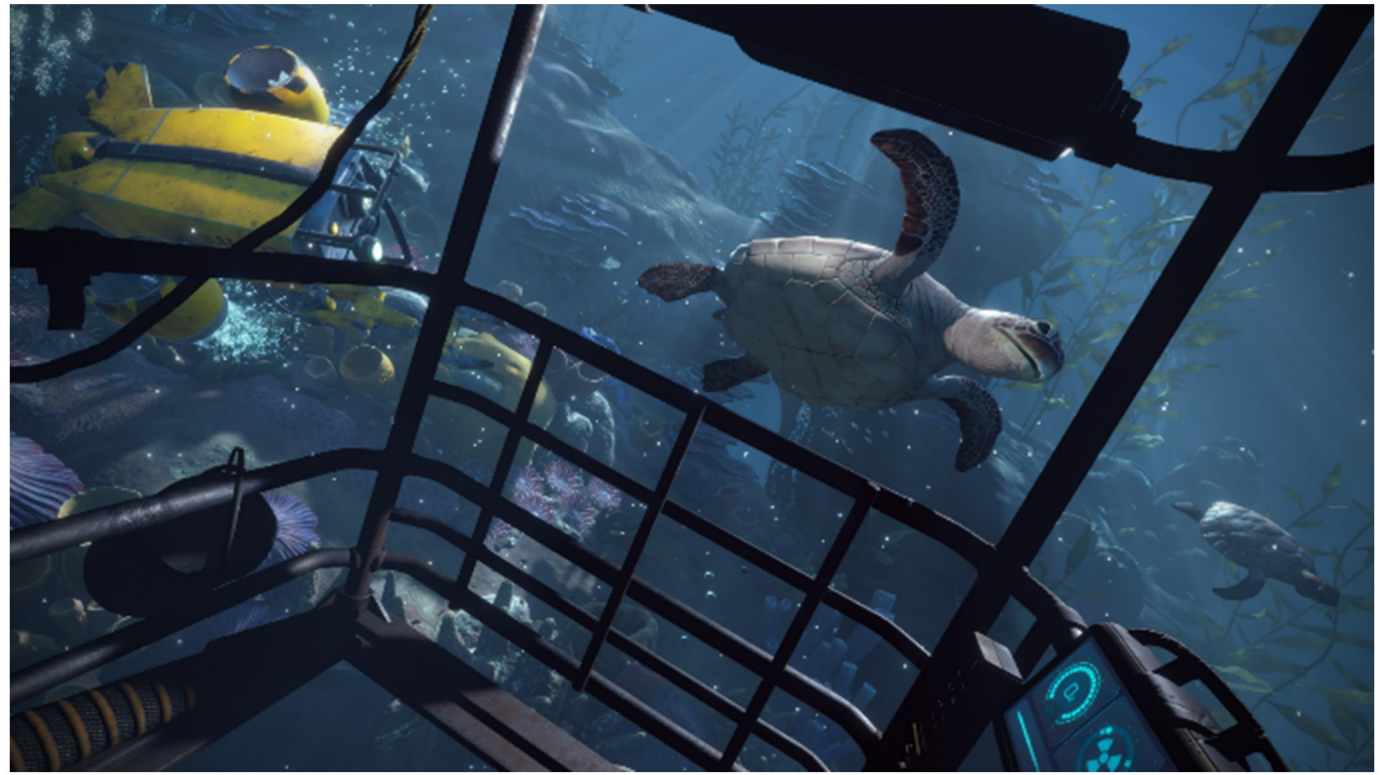
"PlayStation® VR Worlds"

Simon Hall
Joe Milner-Moore

PlayStation® VR Worlds

- Launch title for PlayStation®VR
  - showcase for VR
- Developed by London Studio
  - over three years
  - full-sized title
- Made up of five completely different experiences

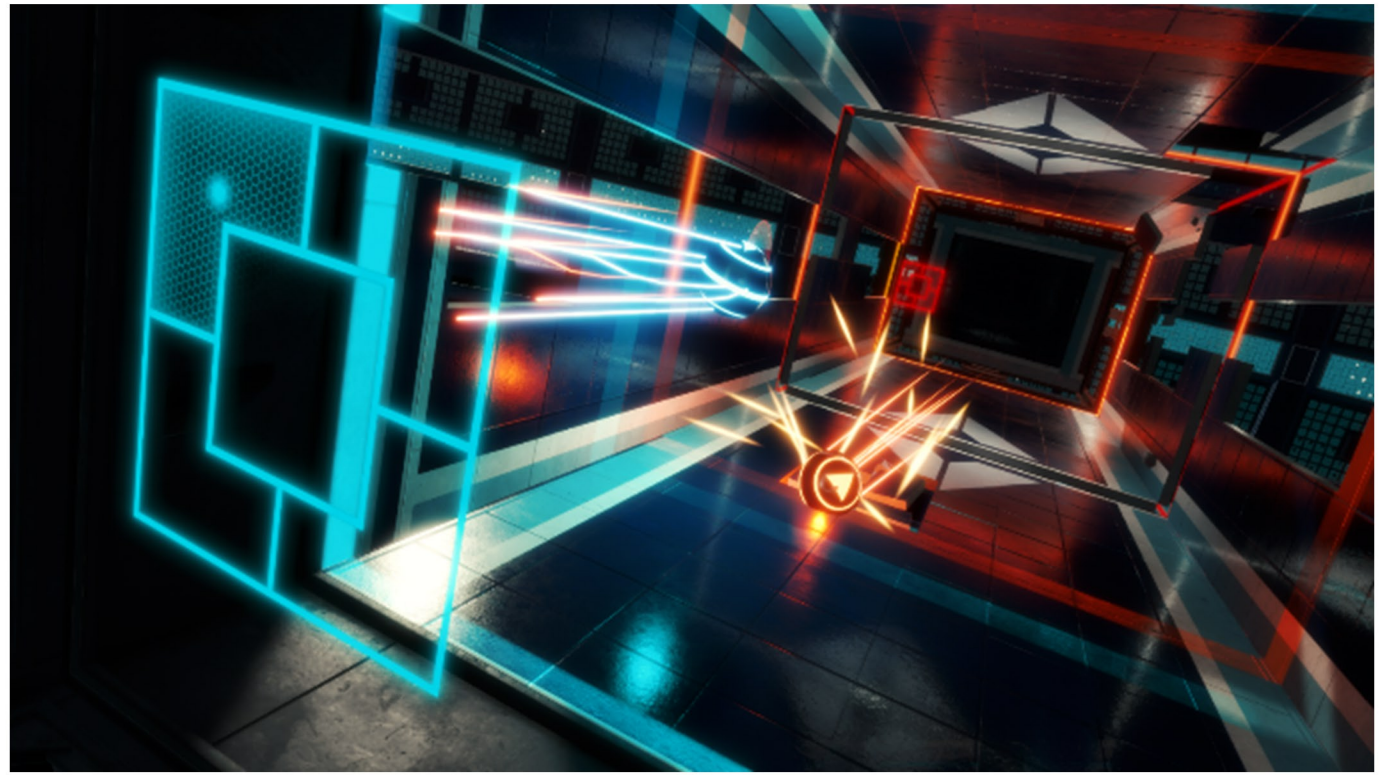
Ocean Descent



VR Luge

London Heist

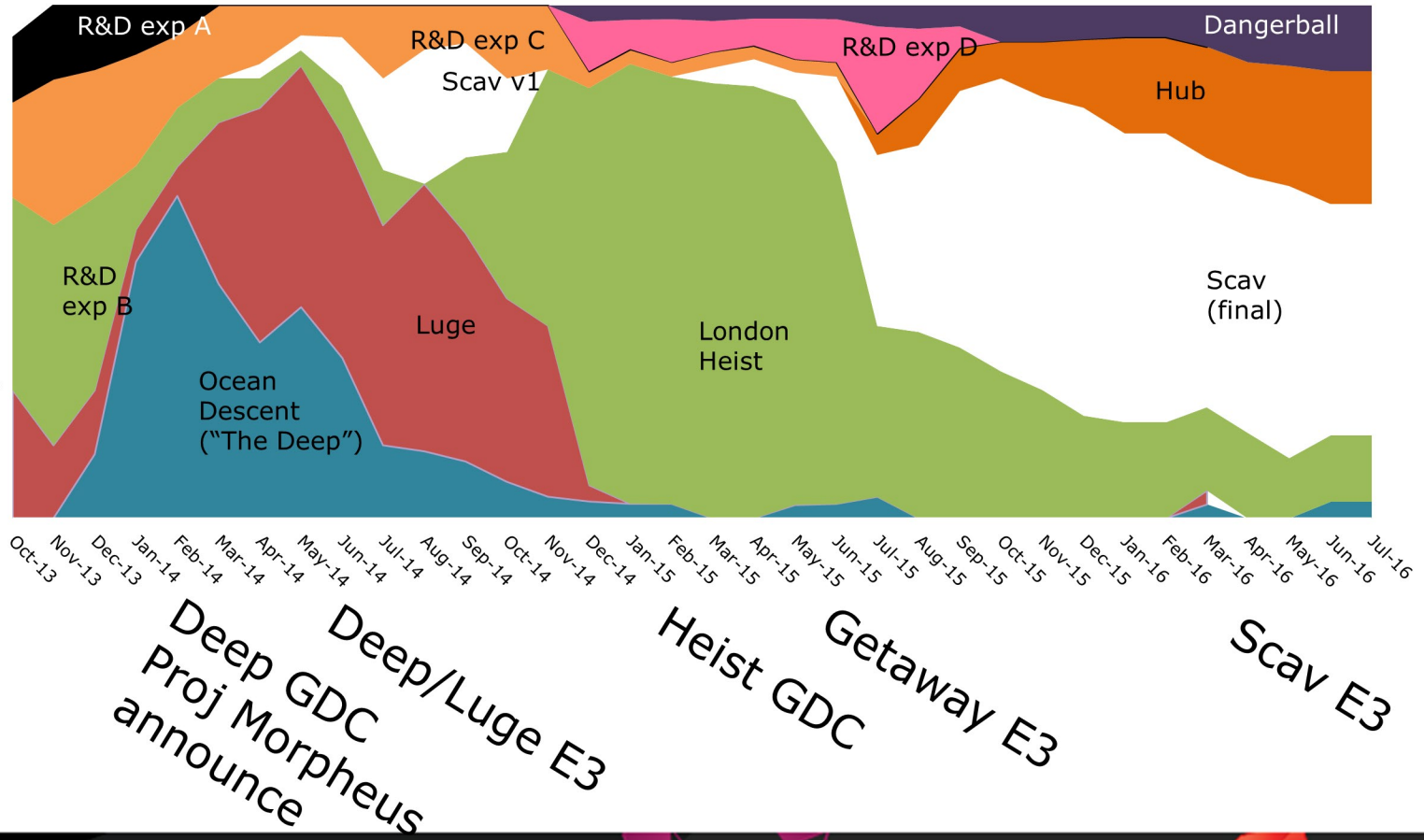
Danger Ball



Scavenger's Odyssey

The game's development cycle

- ~11 different game shows, each with different experiences shown
  - GDC, E3, Gamescom, Paris, EGX, Tokyo
- Internal demos to marketing, upper management etc



Our requirements

- Flexibility
  - Each experience would have completely different requirements
  - It could be entirely different to the last thing we made!
- Robustness
  - Each experience would combine different features in different ways
  - The interval between demo events was often small – we could not afford to spend a week debugging a crash
- Speed
  - Experiences must always be immersive.
  - 60Hz throughout development a must, with rich environments too

Agenda for today

- Renderer overview
- Texture streaming
- Bindless / SRTs
- Draw call validation
- Resolution gradient
- Source-level shader debugging
- Adaptive resolution
- Other stuff worth talking about

Renderer Overview

Forward vs Deferred

- Hybrid tiled forward / deferred
  - lighting is done in forward passes
  - but also write out G-buffer/s during prepass
    - and use for some deferred passes

Evolution

- Started off pure Tiled Forward
  - but had problems with register pressure in lighting passes
  - ⇒ were only getting occupancy of 2 wavefronts
- So started splitting lighting out into different passes
  - isolate high-register code
  - break it out into separate pass where possible
  - ⇒ only that section of code suffers the low occupancy
- eg
  - calc shadow contribution in screen-space
  - calc IBL contribution in screen-space

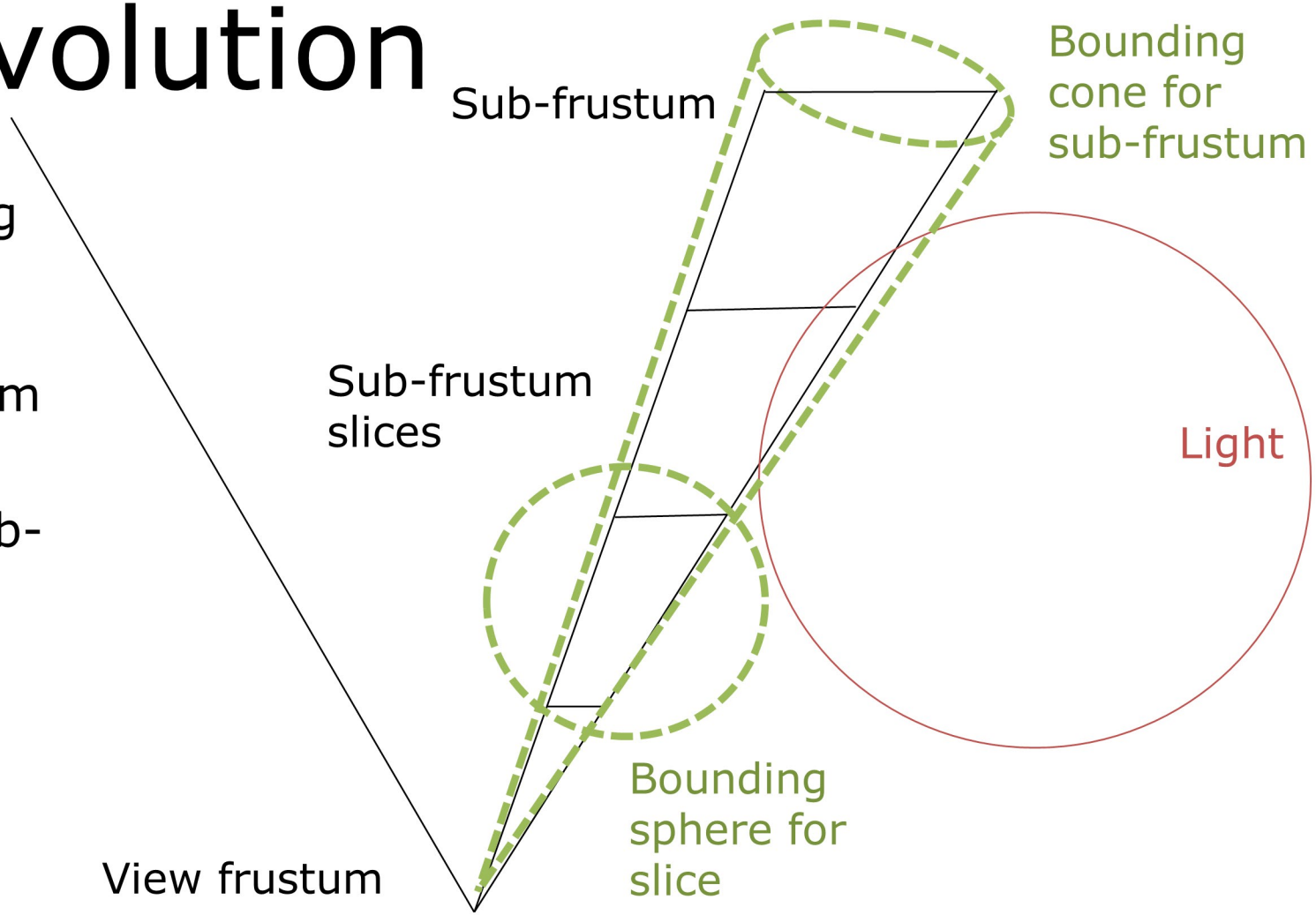
Light Tiling Evolution

- Started off with standard 2D version
  - using depth from prepass
- Moved to 3D "clustered" approach
  - introducing sub-frustum depth “slices”
  - removed dependency on prepass
  - so that we could overlap with prepass vertex work
- Completely decoupled light tiling resolution from image resolution
  - configurable per scene
  - reasonably low resolution tiling fine for many scenes
    - saving is greater than extra shading cost

Light Tiling Evolution

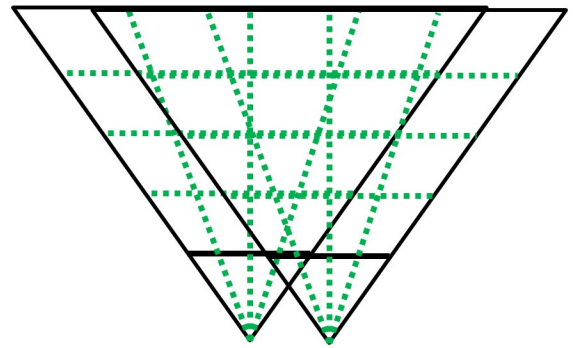
- Added quick early-outs on tiling tests
- Also to avoid false positives on plane tests when behind frustum

- Sphere-cone tests on whole sub-frustum
- Sphere-sphere tests on sub-frustum slices



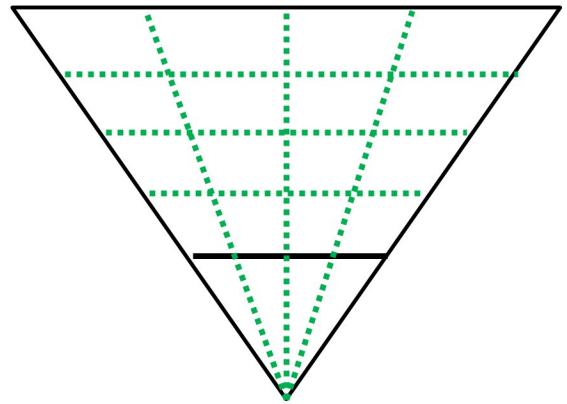
Light Tiling Evolution

- Next project:
  - since we no longer use pixel coords to do look-up
    - ie effectively binning in view space
  - ⇒ could just do once for both eyes using combined frustum
    - already do this for our decal binning



Light Tiling Evolution

- Next project:
  - since we no longer use pixel coords to do look-up
    - ie effectively binning in view space
  - ⇒ could just do once for both eyes using combined frustum
    - already do this for our decal binning



Conclusions

- Hybrid tiled forward / deferred downside
  - lost the "easy MSAA" benefit of Forward
  - and non-trivial to do the deferred approach to MSAA
  - ⇒ no MSAA in VR Worlds
- But worked out pretty well
  - gave flexibility re lighting models
  - good performance
- Decoupling light tiling resolution from image res was a big win

Texture Streaming

Became a crutch

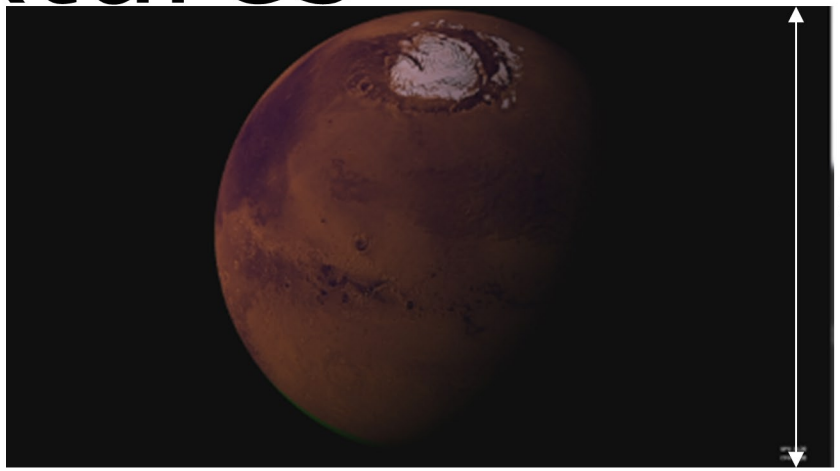
- Went in very late in development
- We all became lazy
  - We wanted to keep the download size down
  - We wanted to minimise thrashing and pop-in
  - Memory usage often view-dependent
- How do you address this?
  - We did not find a satisfactory answer
- Metrics need to be exposed to users and automated testing

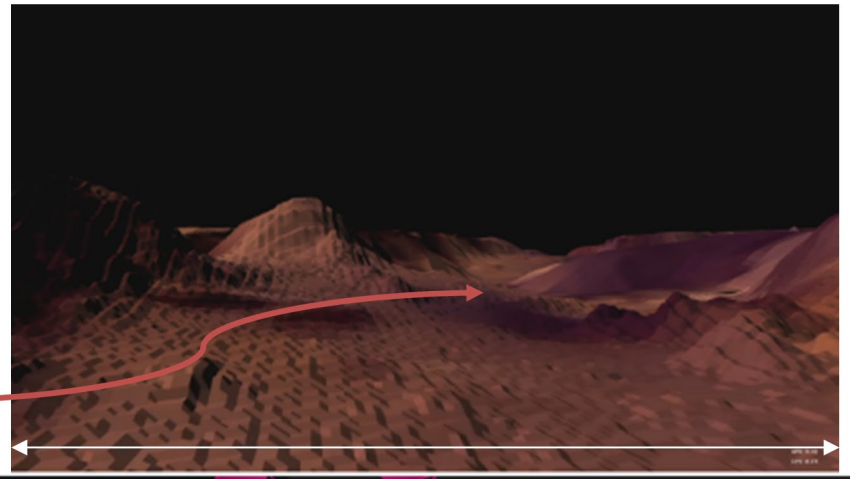
Partially-resident textures

- We spent a month considering PRT
  - The hardware provides an in-shader mechanism to inform you that you read unmapped memory
  - You can then do ‘something’ with this information
- eg you can record this failure information + UV and then bring that page in at a later date

Partially-resident textures

- Does not have to be used for just "texture maps"
  - Can be used in VS and CS too, eg for geometry
- Made a system demonstration to get a feel for the pros and cons
  - Page misses load texture and geometry
  - Streamed in 12 GB of texture data and 5 billion triangles





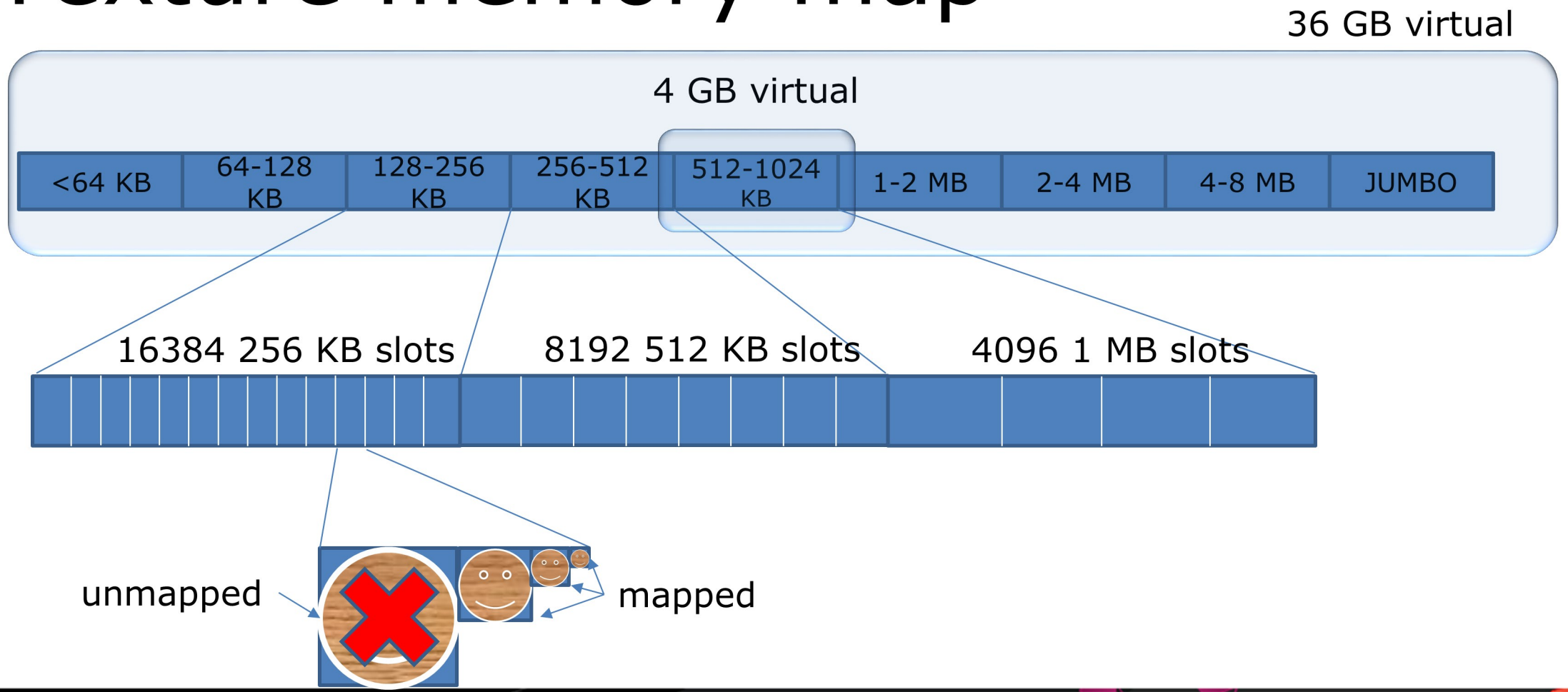
Partially-resident textures

- Translating a missed read into an actual page address is complex
  - Sampling confuses things too
- Ultimately we chose mip level-based texture streaming
  - Textures are still partially mapped but we did not use the in-shader PRT mechanism
  - Page faults are avoided by using texture's min/max mip clamp field

Virtual memory system – GDC16

- Makes great use of system presented by Aaron MacDougall
- Textures are grouped by power-of-two size in bytes
- Each power-of-two size gets its own slot allocator
- Each slot allocator is given 4 GB virtual memory
  - 36 GB used in total
- The game then provides a big bag of physical pages to use
  - Can vary the amount at run-time as needed

Texture memory map



Virtual memory system

- As textures increase in quality then map in 64KB pages
  - Tex's min/max mip fields changed immediately
  - New wavefronts will see this change this frame
- As textures decrease in quality the min/max mip fields are changed straight away
  - But pages are unmapped next frame
- No noticeable hit due to constant page mapping

Virtual memory system

- No copying or defragmentation needed
  - We do not care about virtual memory fragmentation
  - Physical fragmentation is not important
- Draw call validation of texture mappings is easy

Conclusions

- Virtual memory makes texture streaming easy!
  - and brings so many other benefits
- In-shader PRT is a nightmare
  - but mip-based PRT is very manageable instead
  - plus don’t forget PRT works in CS and VS too

Bindless / Shader Resource Tables

Background

- The Compute Units in modern GPUs are ~CPUs!
  - no specialised resource slots any more
    - slots inflexible, slow to bind / emulate and unwieldy
  - access resources "descriptors" directly from memory
  - shader can manipulate arbitrary inputs including pointers!

- New APIs have been created to expose this
  - to provide us with a way of feeding these inputs to the shaders
  - **PS4™** / **PSSL**: Shader Resource Tables ("SRTs")
  - **DX12** / **Vulkan**: Descriptors

Resource Metadata

- T#, S#, V# in AMD GCN speak
  - these are the metadata behind the resources
    - T#: texture 32 bytes
    - S#: sampler-state 16 bytes
    - V#: buffer 16 bytes
- these “sharps” are what the shaders actually deal with
  - ⇒ these **are** the things in the SRT tables
    - plus pointers and constants
  - pretty compact
- similar concepts in other HW/APIs
  - but obviously abstracted away

T# Example Contents

- base address
- dimensions
- type
- format
- tiling mode
- number of mips
- valid mips
- mip clamp
- …

Example SRT Entries

- We use 4 SRT pointers per draw-call
  - resources split depending on amount of sharing
  - prepared on initialisation
    - ⇒ minimal CPU overhead per draw
    - – just copy a few pointers

```
struct SrtData
{
        PerPassSrtEntry*                m_pPerPass;
        MaterialSrtEntry*               m_pMaterial;
        PerSubMeshSrtEntry*             m_pPerSubMesh;
        PerDrawCallSrtEntry*            m_pPerDrawCall;
};
```

Material Sub-Table

- A material sub-table looks something like this…
  - Different for each of a number of "master" material types

```
struct MaterialSrtEntry
{
        Texture2D       m_baseColourTex;
        Texture2D       m_opacityTex;
        Texture2D       m_roughnessTex
        Texture2D       m_normalTex;
        SamplerState    m_samplerState;
        //…
        float           m_saturationOffset;
        float           m_diffuseOpacityMul;
        //…
};
```

Per Submesh Sub-Table

```
struct PerSubMeshSrtEntry
{
        RegularBuffer<float3>           m_vertPositions;
        RegularBuffer<float2>           m_vertUVs;
        RegularBuffer<float2>           m_vertNormals;
        //…
        float4x4                        m_world;
        uint                            m_lightMask;
        //…
};
```

Per Draw-Call Sub-Table

```
struct PerDrawCallSrtEntry
{
        RegularBuffer<AutoInstanceData>        m_autoInstanceData;
        float4x4                               m_worldViewProj;
        float                                  m_lodFadeFactor;
        //…
};
```

Per Pass Sub-Table

```
struct PerPassSrtEntry
{
        Texture2D                       m_ssaoTex;
        Texture2D                       m_deferredShadowsTex;
        //…
        RegularBuffer<Texture2D>        m_shadowMaps;
        RegularBuffer<LightData>        m_lights;
        RegularBuffer<uint>             m_lightTileLists;
        RegularBuffer<Decal>            m_decals;
        //..
        float4x4                        View;
        float4x4                        Projection;
        ResolutionGradientParams        ResGradientParams;
        //…
};
```

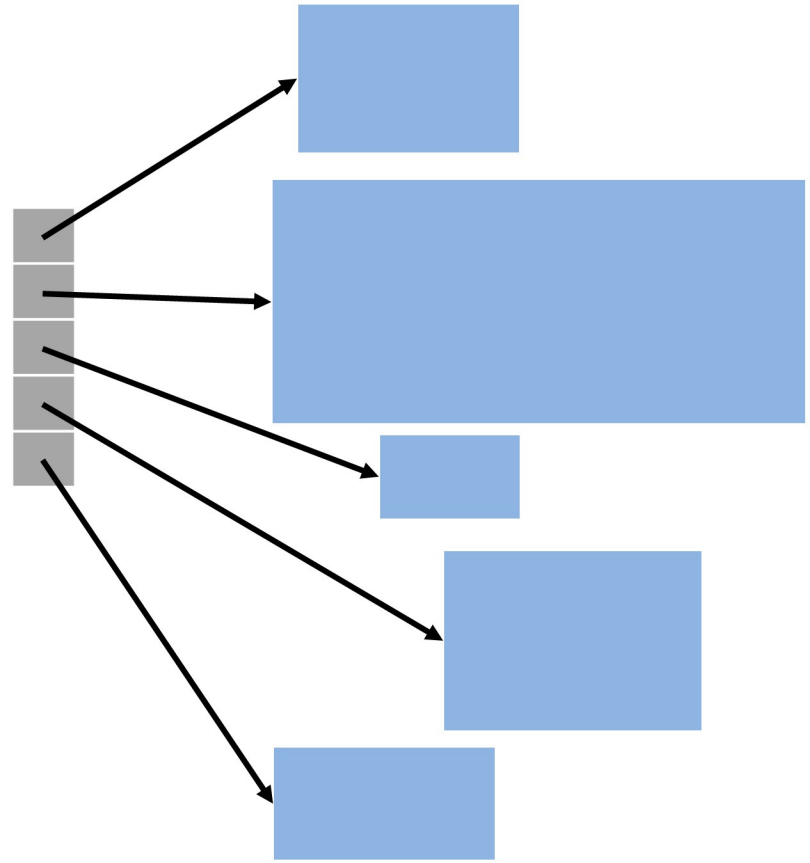
Buffer of Resources

```
RegularBuffer<Texture2D> shadowMaps;
```

- Shadow Maps in one “array of textures”
  - arbitrary number and varying dims / format / …
- Easy access to all shadow maps without:
  - using an atlas
  - using virtual memory
  - binding a fixed number and using a switch statement
  - enforcing same size and use texture array
- Can achieve the same thing in DX12 / Vulkan

```
Texture2D g_shadowMaps[] : register(t1);
```

Array of Textures



Buffer Of Objects

- With SRTs we can even have a buffer of objects
  - containing textures, buffers, constants etc

```
struct Decal
{
        float4x4        m_view;
        float           m_angleThreshold;
        float           m_opacityMul;
        //...

        Texture2D       m_opacityTex;
        Texture2D       m_baseColourTex;
        //...
};
```

Buffer Of Objects

- Just declare a buffer of them:

```
RegularBuffer<Decal>                    m_decals;
```

- **m_decals** buffer contains arbitrary number of decal entries
  - all encapsulating everything they need
  - binding it all is trivial

- Can’t encapsulate all in one struct like this with DX12 / Vulkan
  - as far as I know…?

Auto-Instancing SRT Pointer Example

- "Auto-instancing" background:
  - if N draws use the same sub-mesh
  - detect this during sorting
  - issue as one draw-call with instance count of N

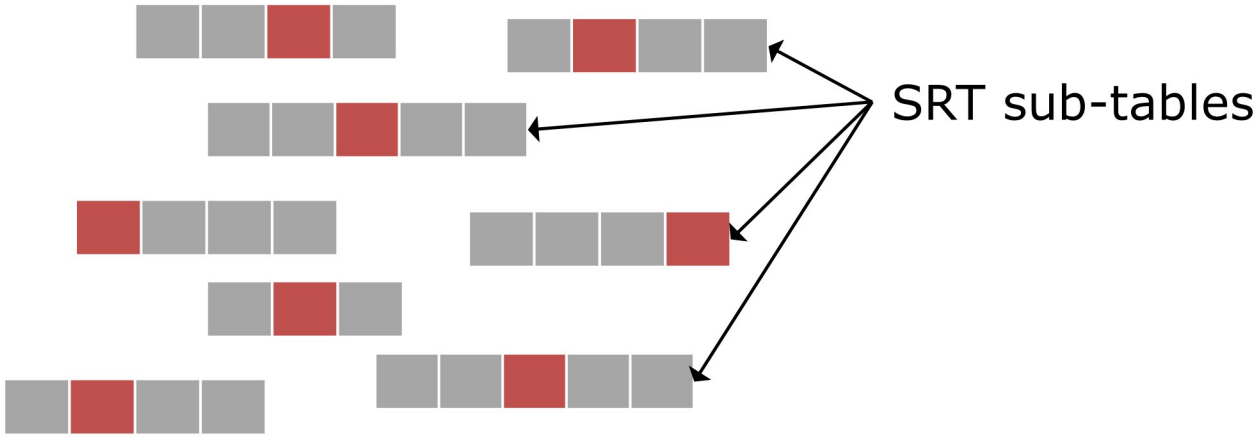
- AND it doesn't matter if the draws have different materials
  - instancing buffer just contains **pointers** to **existing** material SRT tables
    - indexed by S_INSTANCE_ID (or SV_InstanceID)
  - minimal changes to shader code

Pointers To Sharps

- The SRT can also contain pointers to sharps
  - rather than the actual sharp
  - shader code dereferences the pointer in the same way the CPU would

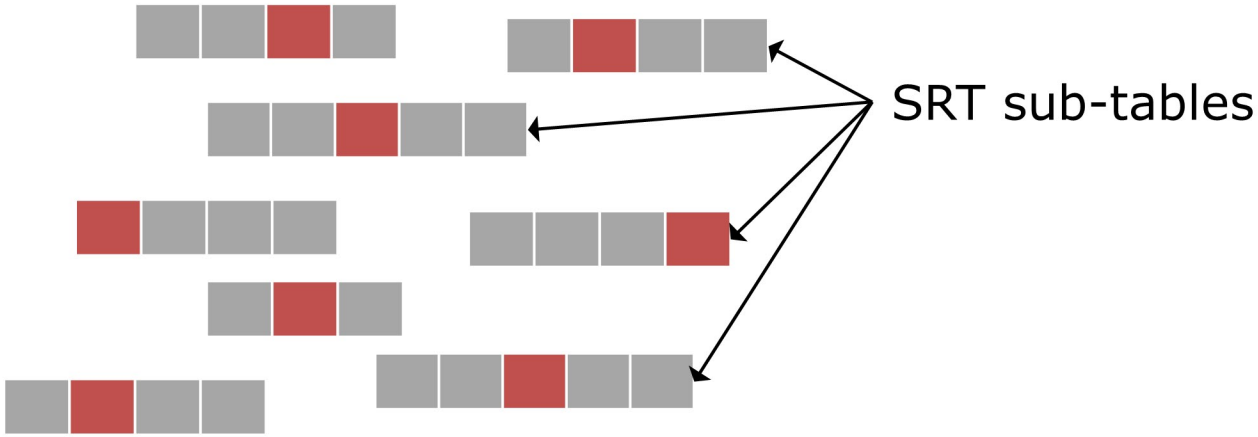
Texture Streaming Usage Example

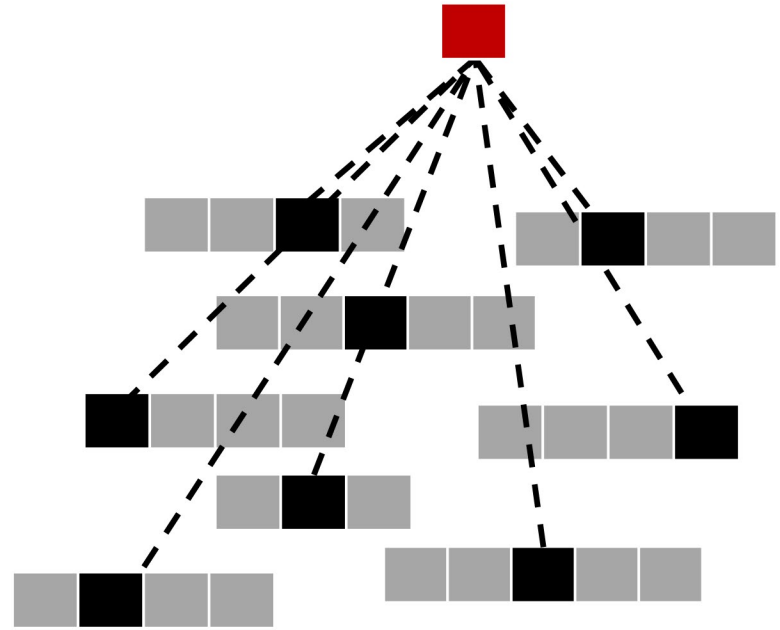
- we could copy a T# for every draw that uses a texture
  - and maintain a list of where those T#s are in memory => error-prone
  - and update them all when necessary – eg update valid mip-range



Texture Streaming Usage Example

- we could copy a T# for every draw that uses a texture
  - and maintain a list of where those T#s are in memory => error-prone
  - and update them all when necessary – eg update valid mip-range
- OR… use a **pointer** to the T#…
  - and just update that one T#





Conclusions

- Bindless / SRTs are very powerful
  - Not just to speed up draw-calls
  - Also offer great flexibility and control
  - Simplify shader / runtime communication
  - Spend less time working around limitations

Draw Call Validation

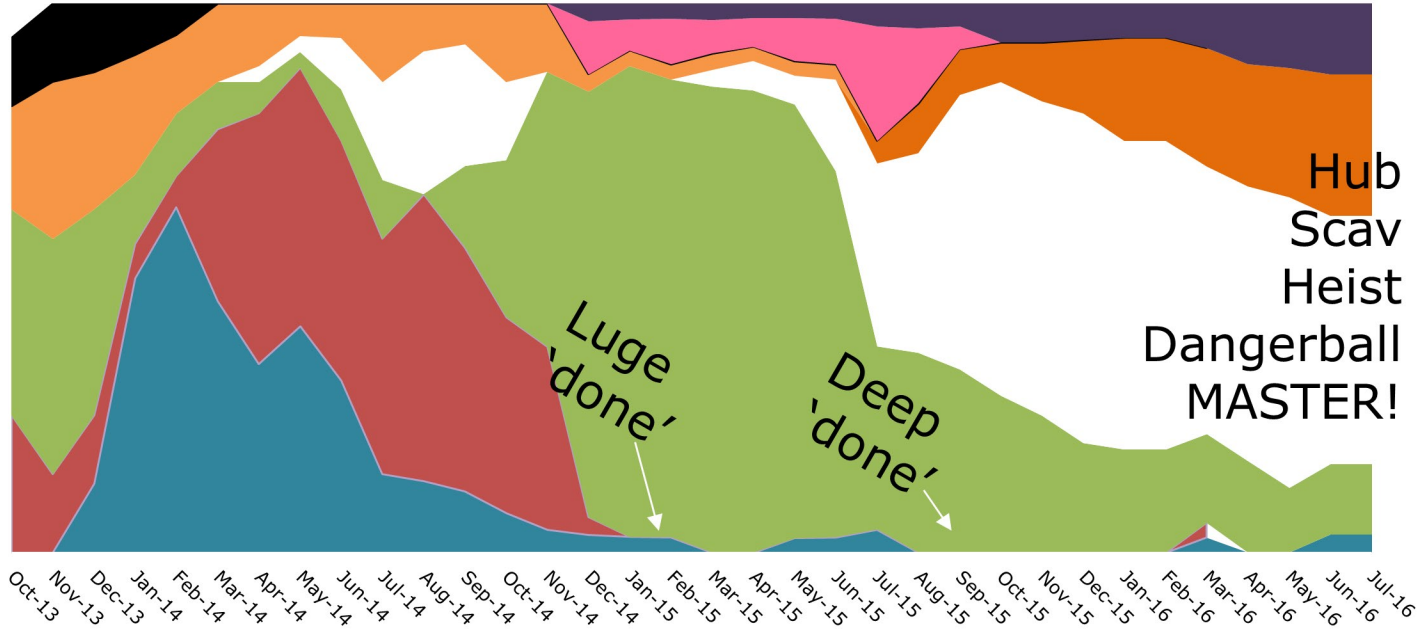
Draw call validation

- We all make mistakes
- We frequently merge thousands of files
  - Shader source, shader binaries, source code and assets can get out of sync

Being robust

- Once an experience was finished, it would not be touched until master

➢ Real risk of bit rot
  - We wanted to avoid that



Draw call validation

- We invested a fair amount of time making stuff to help us when things went inevitably wrong
- This became draw call validation

⇒ Try and catch obvious errors before issuing a draw

  - GPU crashes and timeouts are hard to debug
  - Plus not all mistakes cause such obvious, immediate errors

Let’s not waste time

- We realised early on that we spent a fair amount of time debugging when the GPU crashed
- Tools at the time made this very difficult
  - Which draw has failed? Why?
  - This took me a week to track down, minutes to fix:

GPU Protection fault. Access Read: Unmapped page access: Addr(VA) 0x000000000433e000

[Crashes now give more info]

Performing validation - SRT

- SRT makes it really easy to do validation
  - Just walk each entry and check it
  - Still possible without SRTs (just neater)
- Validation performed after all state is set
  - Do it at the end rather than as you go
- Check shader reflection against buffer cache coherency
  - Else writes may disappear

Performing validation - memory

- Confirm that all memory ranges are mapped and are allocated
  - Memory might be mapped but unallocated
- Check each address range is GPU R/O or R/W
  - A good way to catch over-fetch
- Check shader binary is read-only
  - Let’s prevent corruption after load

Good enough for nearly all cases

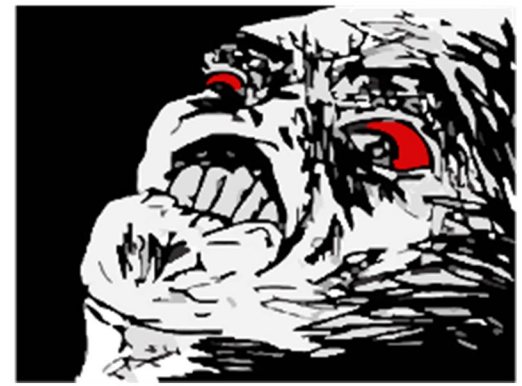
- This catches failure for us 99% of the time
- Most of these failures would be subtle (not crashes)
  - Invisible objects
  - Corrupt materials
  - Blank textures
- If you weren’t looking at the object you would miss the problem
- Or you might think the artist intended that look

So you are ready to draw

- But it’s still crashing!

  GPU Protection fault. Access Read: Unmapped page access: Addr(VA) 0x000000000433e000

  - No program counter, no shader ID, no draw ID
- State could be corrupted after command buffer is kicked
  - Memory could be unmapped
  - Sharps could be trashed
  - Code could still be corrupted



Our aspiration

- Let’s catch crashes at *the instruction they happen*
- Then less guesswork needed
  - We can inspect live state and see what’s wrong
  - Like using a normal debugger
- We need to do validation *in-shader*

Validate memory accesses

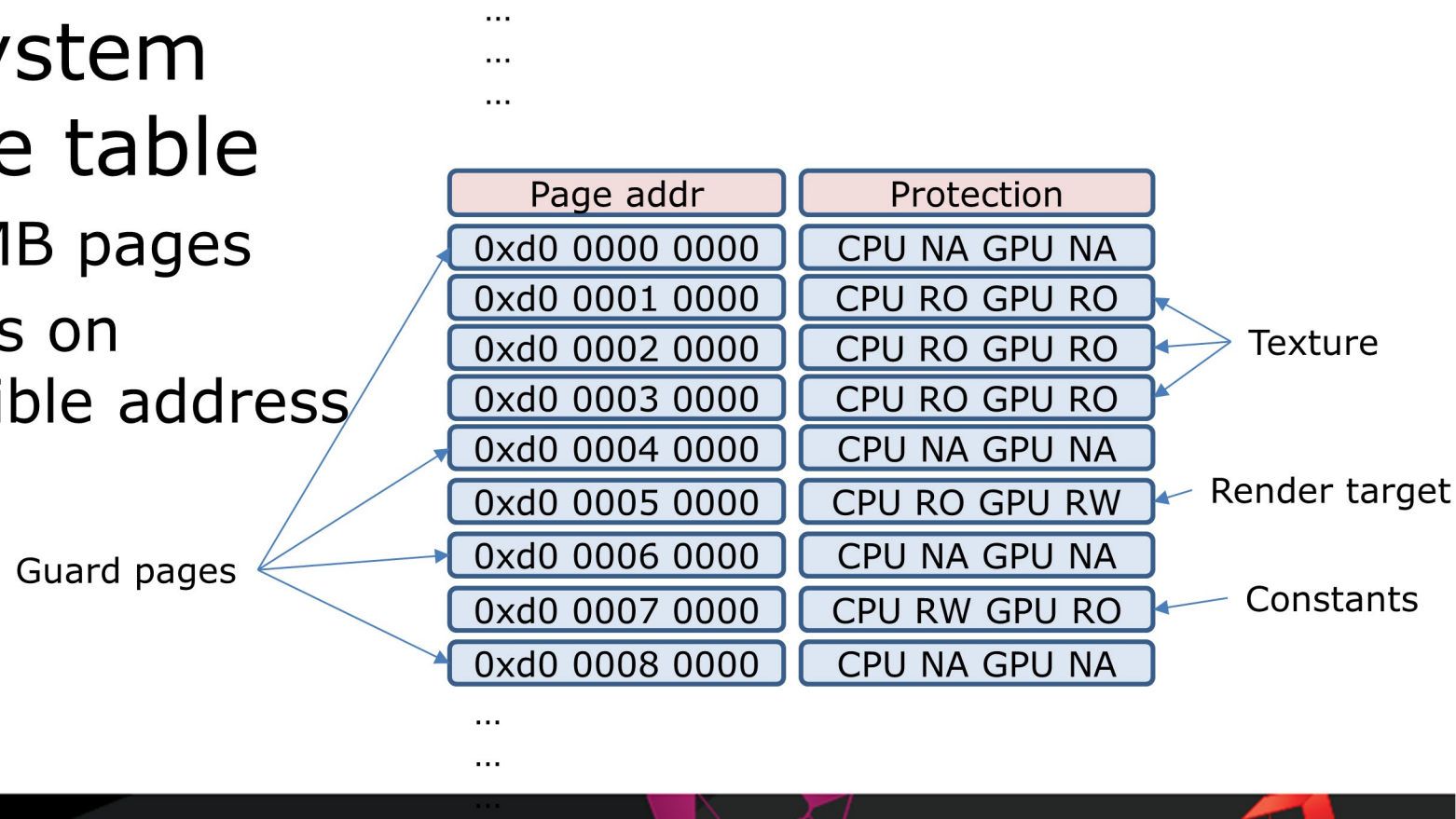
- How to get a page fault within a shader
  - Scalar load of memory address
  - Scalar load of one element addressed by a V#
  - Vector load/store of up to 64 elements addressed by a V#
  - Image sample of up to 64 elements addressed by T#

Validate memory accesses

- Replace each one with code which checks if the memory is valid
  - Assembled at run-time and injected by shader loader
  - Possible with D3D IR code manipulation, easier with machine code
- Validation not as slow as it sounds!
  - Yes we really are going to wrap every memory op
  - CS runs at full speed, PS at ~80%

Validate against what?

- Ensure memory system builds its own page table
  - Work in 64 KB or 2 MB pages
  - Adds/removes entries on allocation of GPU-visible address



Patch shader code – vector load example

- Replace vector load with jump to new code fragment

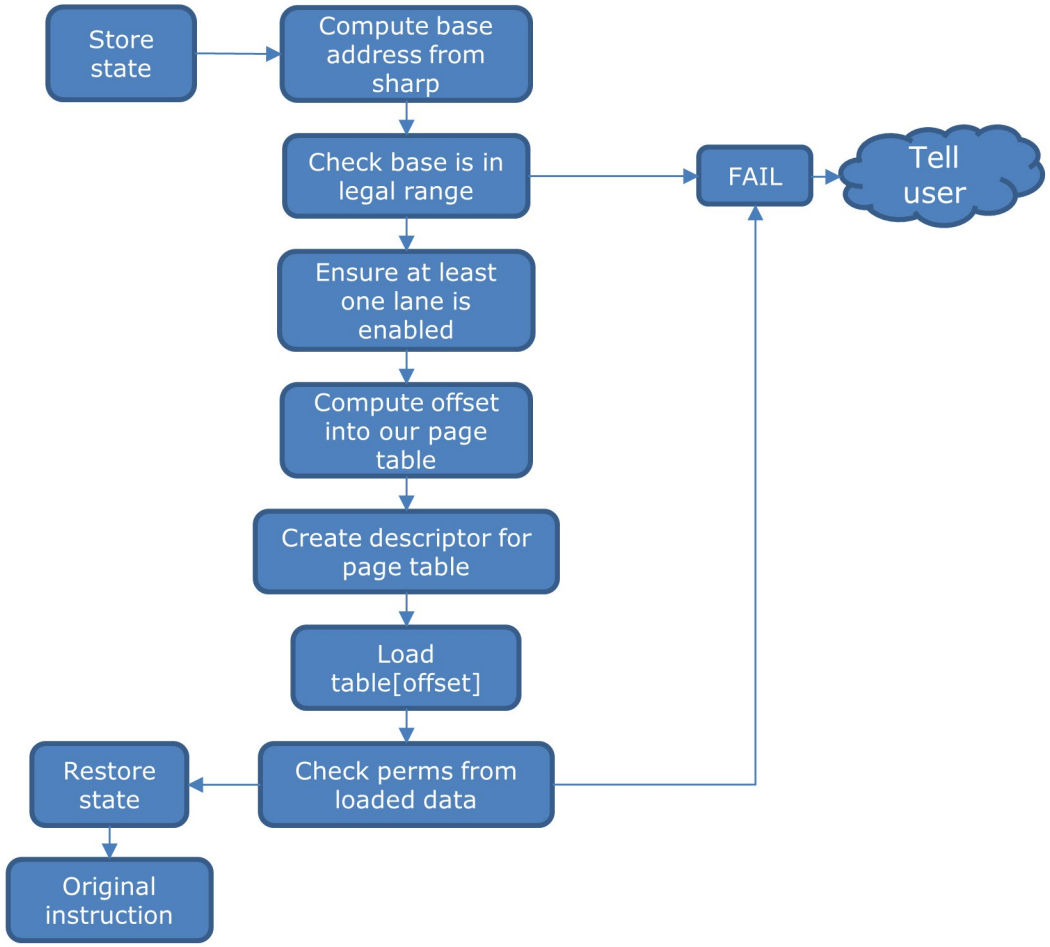
Our load instruction used to be here

```
0x000000922772fcc0  s_branch          0x000000922772d68c
0x000000922772fcc4  v_writelane_b32   v105, s21, 25
0x000000922772fcc8  v_writelane_b32   v105, s20, 24
0x000000922772fccc  s_waitcnt         lgkmcnt(0)
0x000000922772fcd0  v_mul_f32         v4, s54, v2
0x000000922772fcd4  v_mul_f32         v5, s55, v3
0x000000922772fcd8  s_branch          0x00000092277 2d6d0
0x000000922772fcdc  s_nop             0x0000
0x000000922772fce0  v_writelane_b32   v105, s87, 62
0x000000922772fce4  v_writelane_b32   v105, s86, 61
0x000000922772fce8  v_mad_f32         v7, -2.0, v5, 1.0
0x000000922772fcf0  v_mov_b32         v5, s5
0x000000922772fcf4  v_writelane_b32   v105, s85, 60
0x000000922772fcf8  v_mad_f32         v6, 2.0, v4, -1.0
0x000000922772fd00  v_mov_b32         v4, s7
```

(the s_nop is because some insns are 4 or 8 bytes in size)

New code fragment

- Compute base address and buffer size from V#
- Check range is mapped
- Check each page's permissions
  - If we're happy, run the original instruction and return
  - Else, write PC, registers to main memory and signal CPU and user



Issues with PRT

- Checking a range is completely mapped is good enough for textures which *should be* completely mapped
- Partially-mapped textures need extra validation
  - We need to validate the address for every sample that every lane will make
  - This is expensive to do
  - Address computation is non-trivial
  - Anisotropic filtering makes it even more complex
- We did not attempt to solve this

Comparison vs PS4™ SDK Tools

- PS4™ SDK now comes with many debugging and validation tools:
  - Post Mortem Crash Analysis
  - Razor GPU Capture and Replay
  - Capture On Crash
  - Validation on submission

- So if you're not familiar with them, start with those tools!

Comparison vs PS4™ SDK Tools

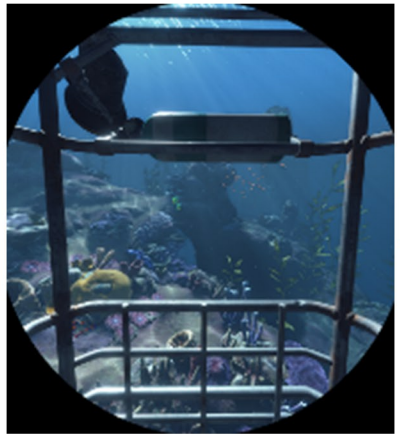
- The tools we developed augment these SDK tools
  - Based on the fact that we can do this in-engine
  - We have extra implementation info & access that the SDK tools obviously don't have
- Key benefits
  - We catch bad draw calls live as they're being "bound" ⟹ trivial to debug
  - We abandon bad draw calls and report errors so content creators can carry on unimpeded
  - In-shader, we stop on the exact wavefront & lane & instruction that causes the page fault
- DEBUG ONLY!
  - Injecting shader code like this is a TRC violation on PS4™
  - **DON'T try to ship with this code enabled!**

Conclusions

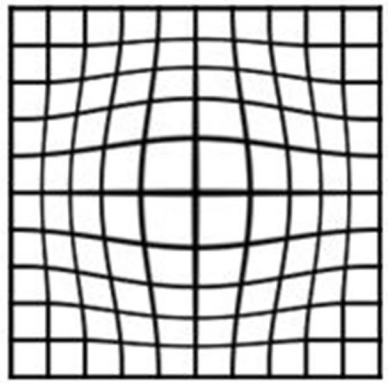
- Validation is worth doing
  - It is easy to do, and will save you time!
- Catching async page faults is tricky
  - But it’s worth developing an infrastructure before your crunch period

Resolution Gradient

Motivation

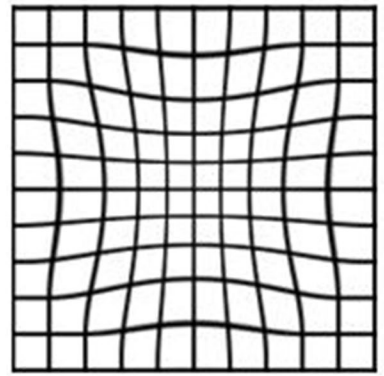


Game Output

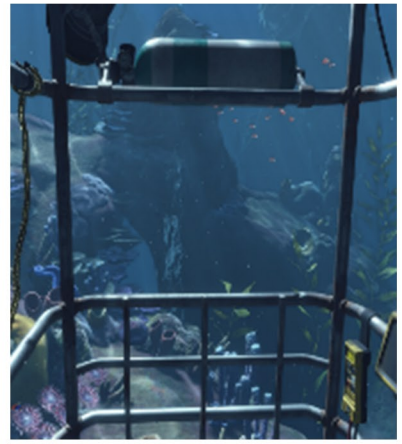


Barrel Distort

Warped Image
On Panel
960 x 1080

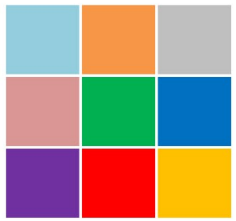


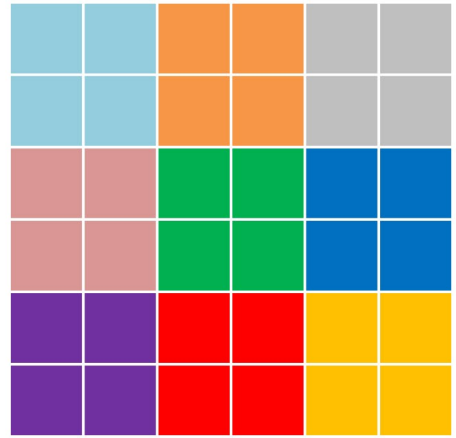
Optics Apply
Pin-Cushion Distort



As Seen By
Eye (ish)

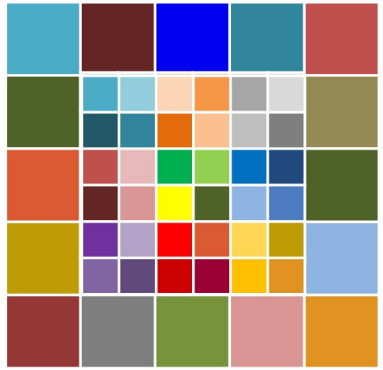
Motivation





Motivation

Higher Res



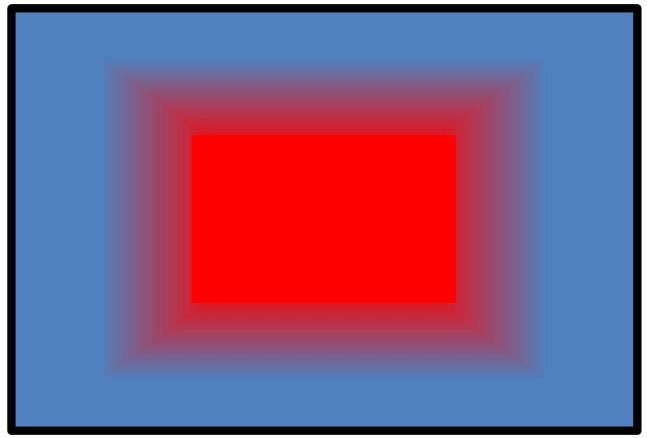
Higher Internal Resolution

- For PlayStation®VR, advised to aim for x1.4^2 panel resolution
  ⇒ ~x2 number of pixels!
- At 60Hz…
- (960 x 1.4) x (1080 x 1.4) x 2 eyes
  = 1512 x 1344 x 2
  = 4,064,256 pixels per frame
  = ~250 million pixels per second
  ⇒ same number of pixels per second as native 4k at 30Hz!

Problem

- So need to find a way to render at variable resolution
  - more pixels in the centre
  - less in the periphery
- No hardware support for this

High-Res Insert



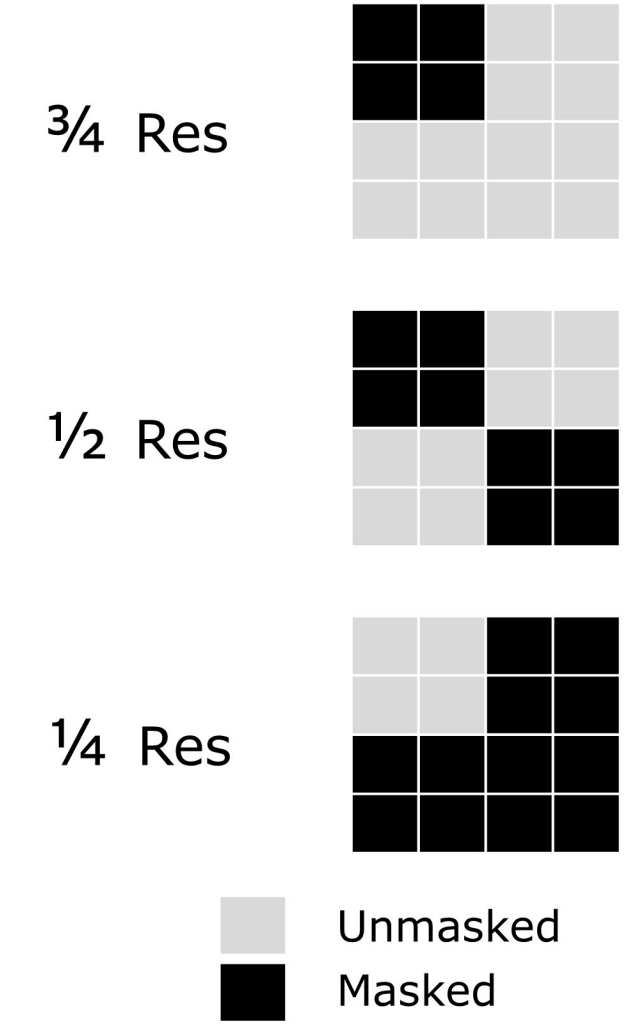
- One approach is to use two passes:
  - render central portion at high res
  - render the rest at low-res
  - combine and blend between them
- BUT…
  - would require us to use an extra geometry pass per view
  - only has two levels - high or low
  - shape not very flexible

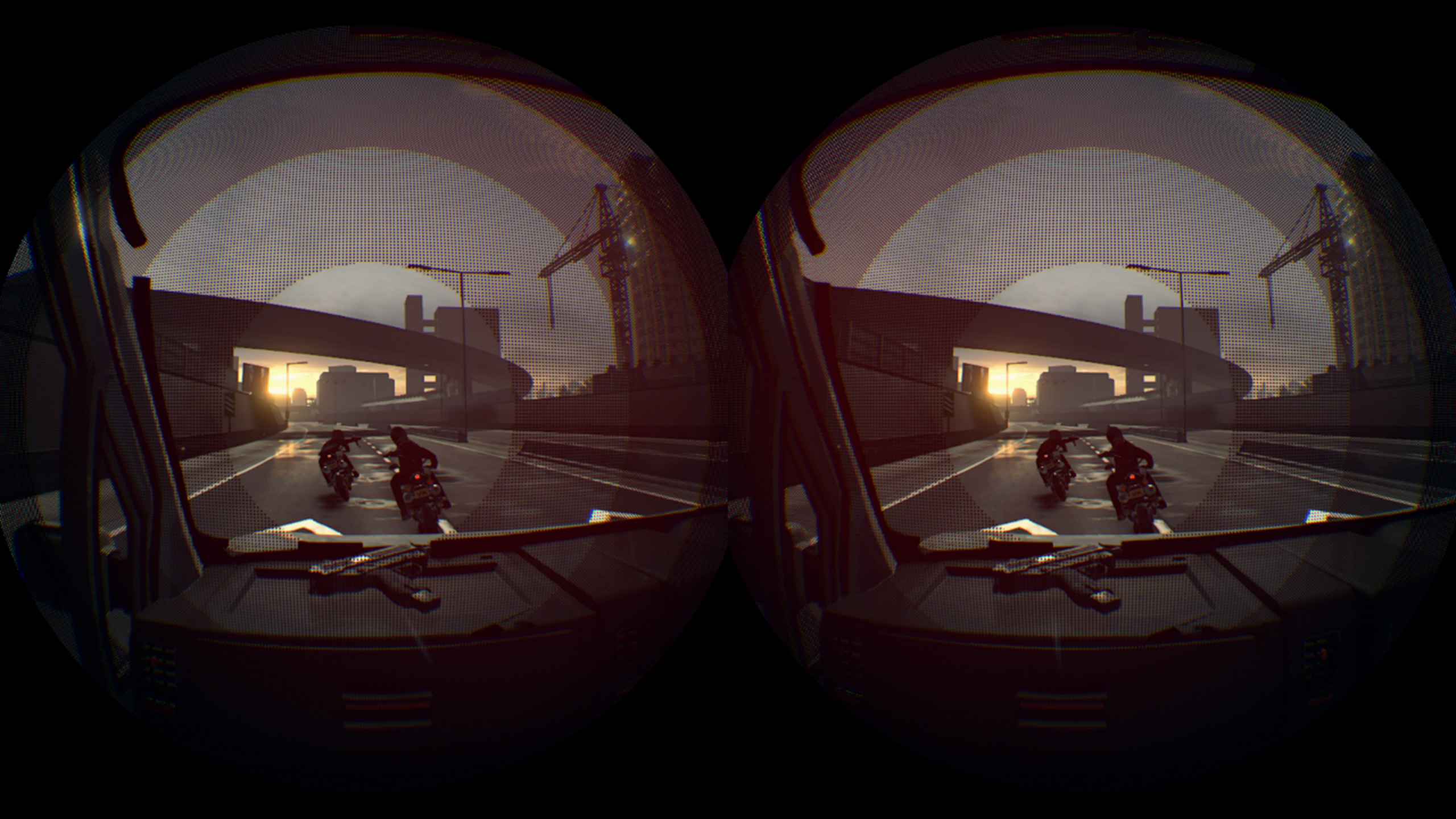
Masking

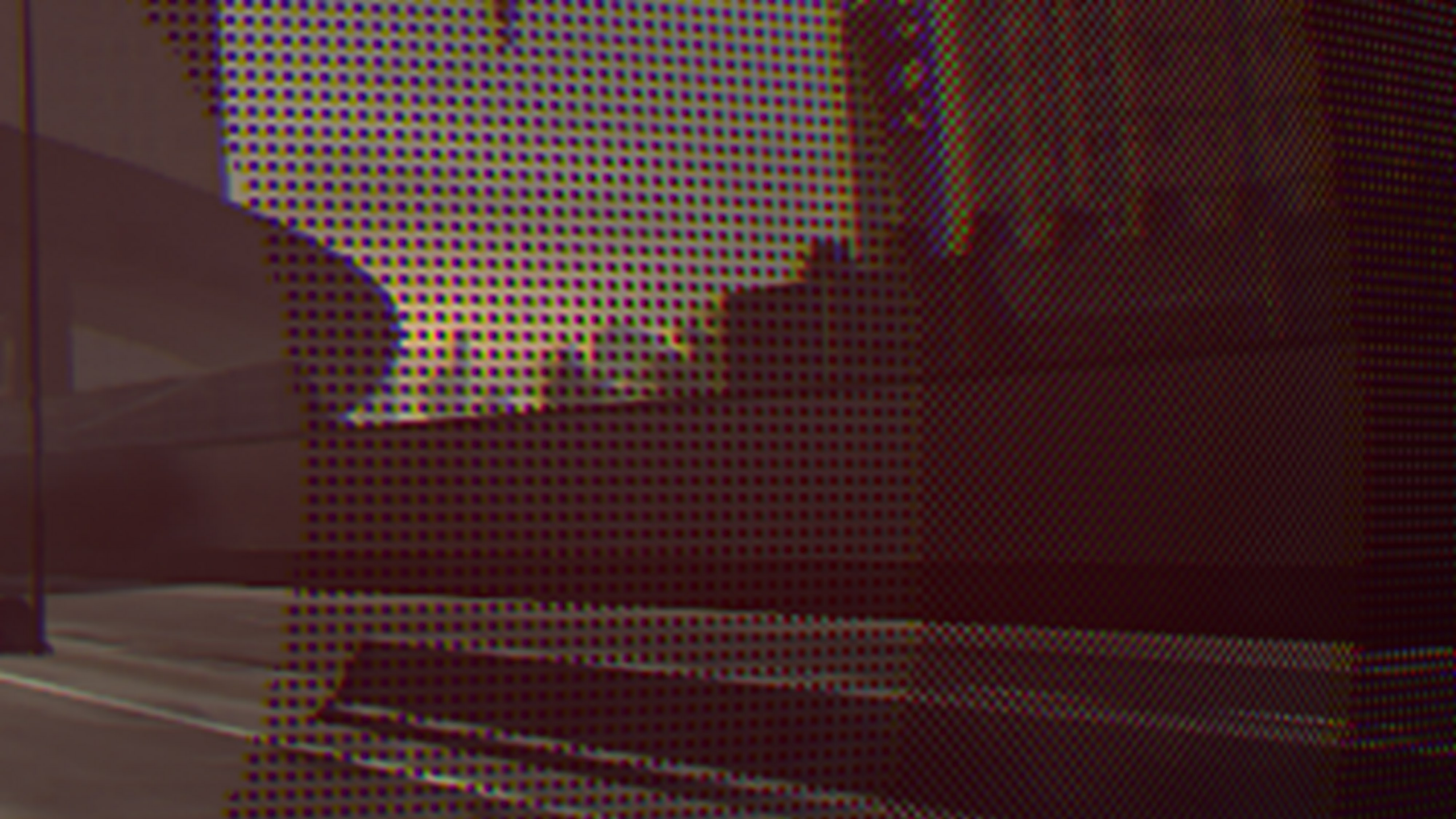
- So we came up with a plan…
- Construct dithered stencil mask to apply to the image
  - more control
  - no extra geometry pass (as we'd need for hi-res insert approach)
- First version presented by James Answer at GDC2016
  - "Fast and Flexible: Technical Art and Rendering For The Unknown"
    - http://www.gdcvault.com/play/1023184/Fast-and-Flexible-Technical-Art

Masking

- In the centre we shade as normal
- Then as we move further from the centre we gradually mask out more pixels
- At the outside of the image we're only rendering 1 in 4 pixels



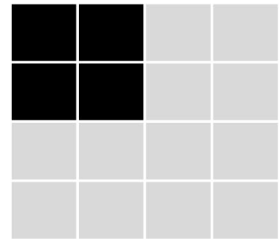




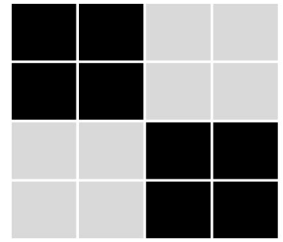
Masking

- You may have noticed we're masking out 2x2 grids of pixels…
- This is to stop quad overdraw making the whole thing pointless

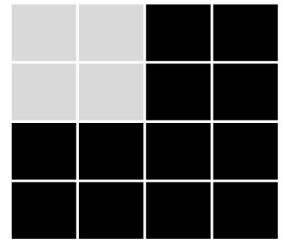
¾ Res



½ Res



¼ Res



Unmasked

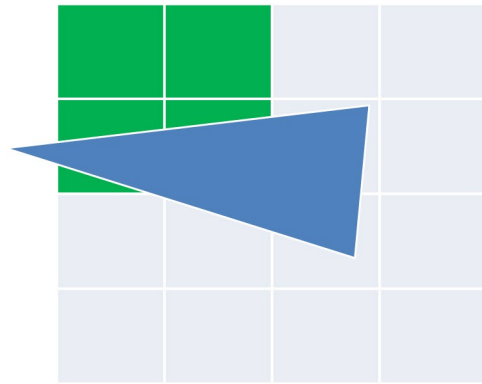

Masked

Quad Granularity Masking

- consider a 4x4 grid of pixels…

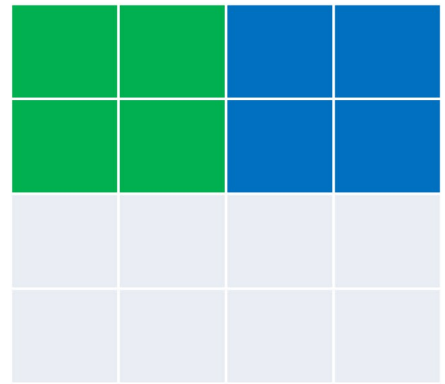
Quad Granularity Masking

- pixel are shaded in groups of 4 pixels: “quads”
  - made up of a 2x2 grid of pixels
  - the gradients between them are used for eg mip selection



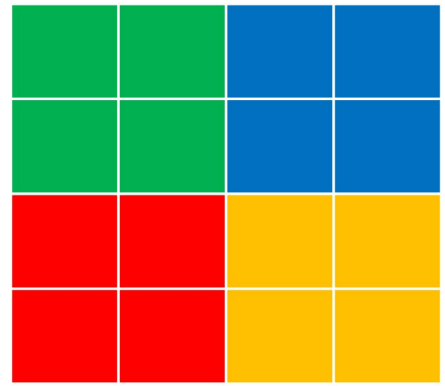
- so even though this triangle only intersects two of these pixels’ centres
  - the other pixels in the quad will be shaded – and then thrown away!

Quad Granularity Masking



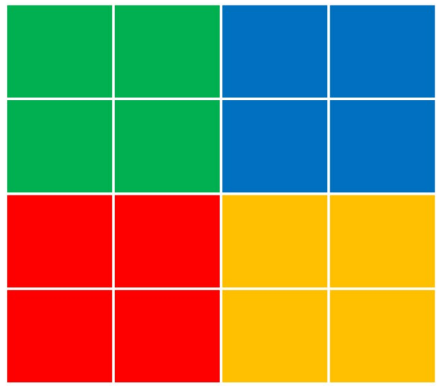
Quad Granularity Masking

Quad Granularity Masking



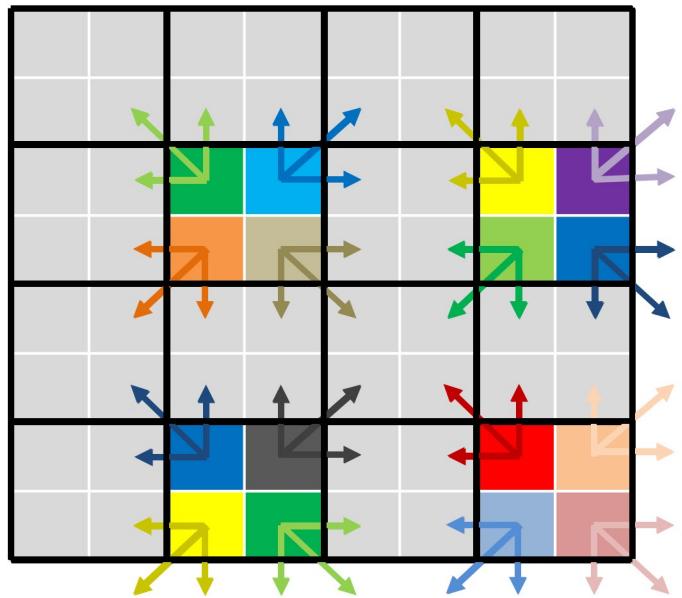
Quad Granularity Masking

- the same applies if we mask some pixels in a quad
- so to get any benefit we need to mask out **quads**, NOT **pixels**
  ⇒ quad (2x2) granularity masking



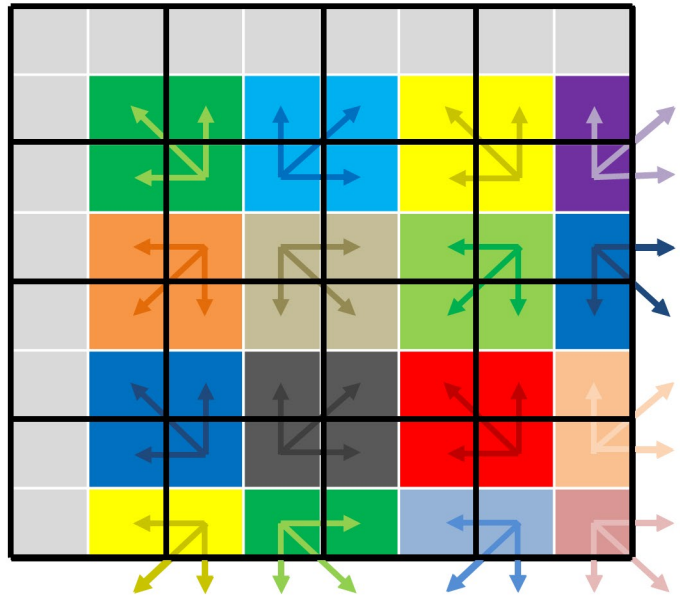
Filling In The Holes

- We then fill in the holes with a dilate pass
  - just copy over the closest shaded pixel



Filling In The Holes

- We then fill in the holes with a dilate pass
  - just copy over the closest shaded pixel



Filling In The Holes

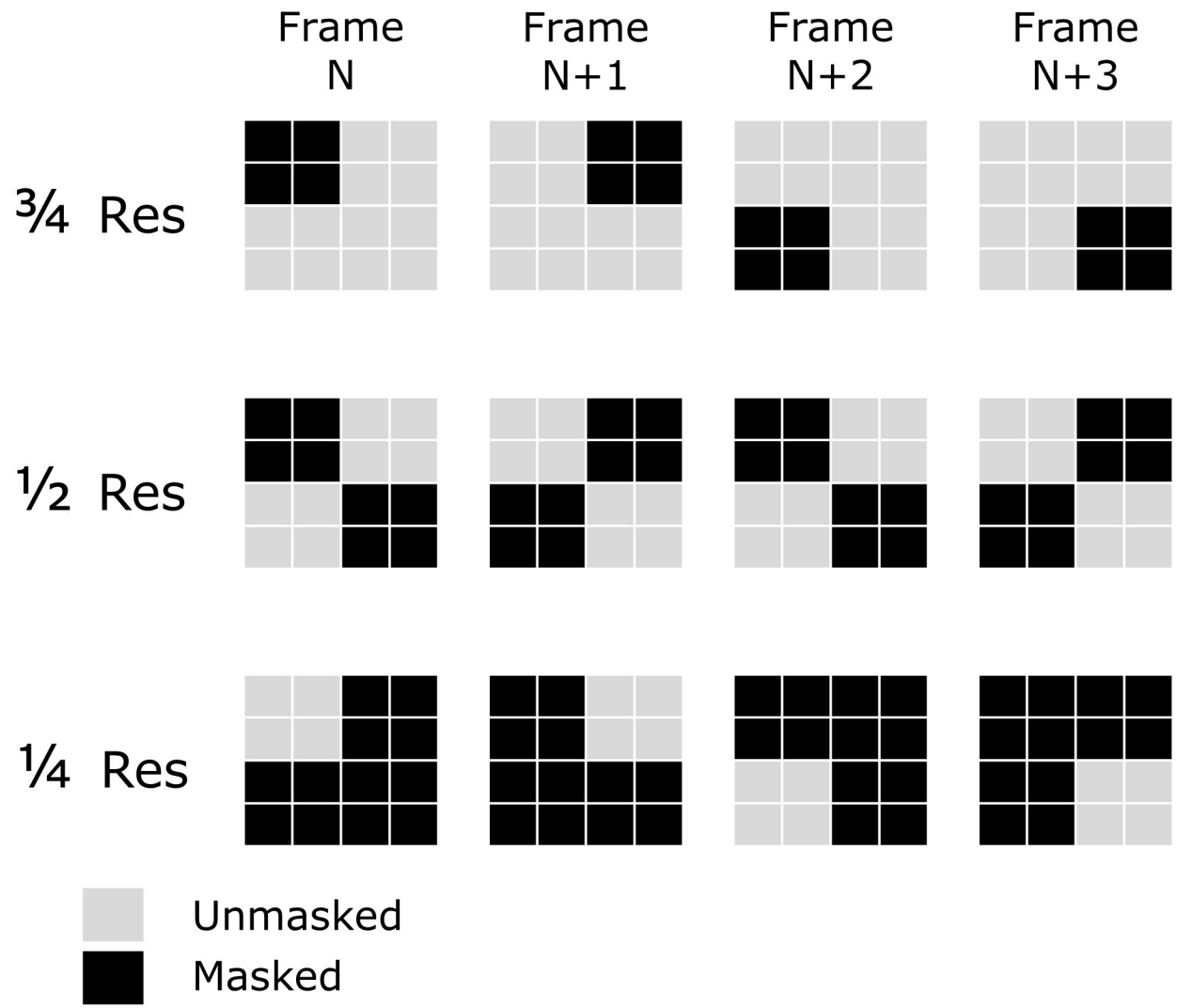
- In many cases it is quicker to fill the holes "on demand"
  - adjust sample positions in shader to select nearest **valid** pixel
- Removes need for separate dilate passes
  - extra ALU in multiple passes often cheaper than the extra reads / writes
  - Does start to infiltrate all shader code though!

Mask After Prepass

- It can sometimes be quicker to not mask in the prepass
  - depending on the geometric and depth complexity of the scene
- This makes the prepass slightly slower
  - but potentially not much, as prepass shading is light
- And avoids the need to dilate the depth and g-buffers
  - can work out as a saving

Temporal Magic

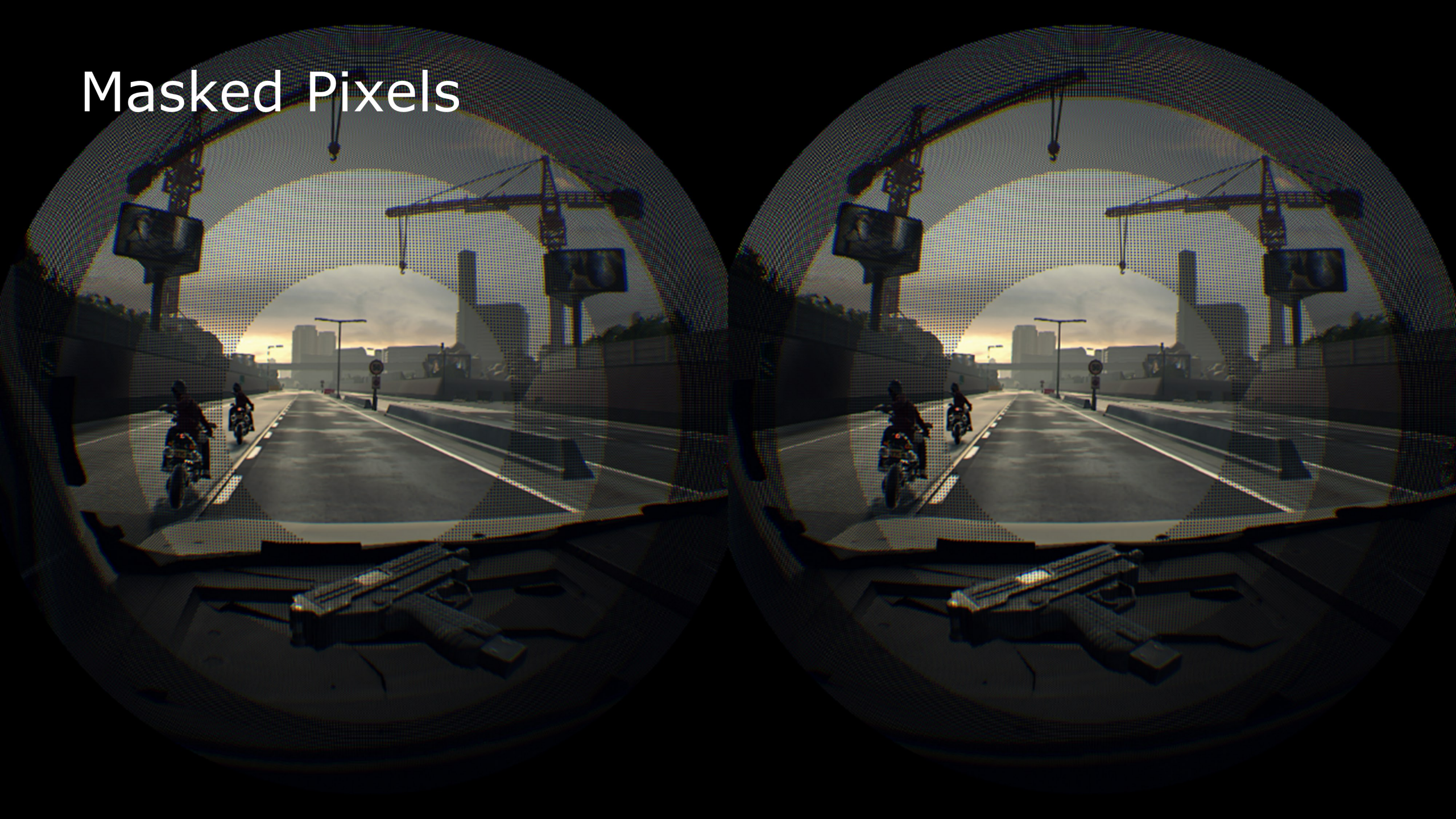
- Vary dither pattern each frame
  - TAA accumulates contributions
- Ensures contributions from all pixels
  - just at lower temporal frequency
    - ⇒ Traded temporal for spatial resolution
  - very few changes to existing TAA



Results

- Surprisingly good results
  - even in ¼ res region!
  - even though we're masking at **quad** granularity!
- Saving ~4ms on a 16.6ms frame!

Masked Pixels




Full Res

Variable Res


Full Res x 8


Variable Res x 8


Mask x 8

MSAA Abuse

- Quad granularity masking does exacerbate instability
  - in high frequency regions
  - more noticeable in motion
- But we found a way to mask at **pixel** granularity!
  - ie without the quad overdraw problem
  - thanks to an insight from Mark Cerny

MSAA Abuse

- Render at 4xMSAA at sample frequency
  - (adjust MSAA sample positions)
- And ¼ res (½ x ½)

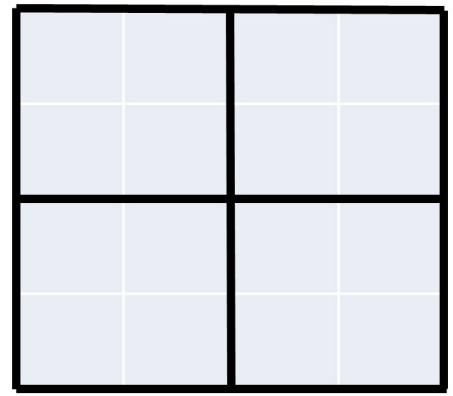
⇒ Still shading same number of samples…
  - **BUT**…
  - the samples that make up a quad are now **further apart** in the resolved image…

Pixel Granularity Masking

- consider 4x4 grid again…

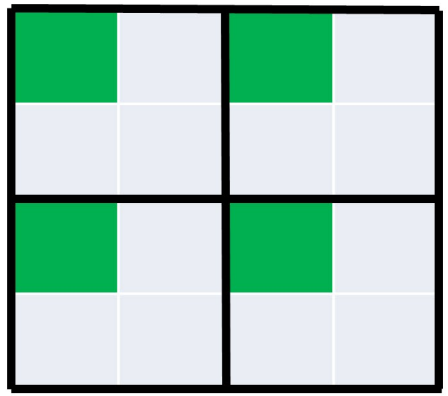
Pixel Granularity Masking

- 2x2 grid of pixels, each with 4 samples



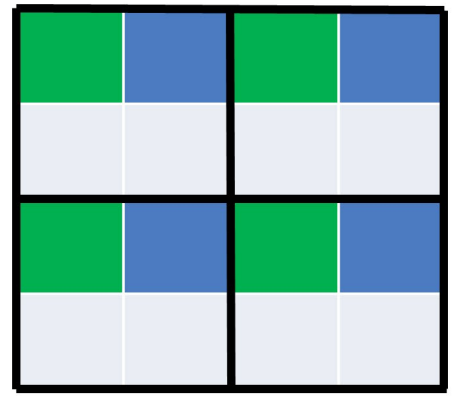
Pixel Granularity Masking

- pixel quad constructed using corresponding **sample** in each **pixel**
  - so 1st quad is made up of 1st sample from each pixel in the 2x2 pixel grid

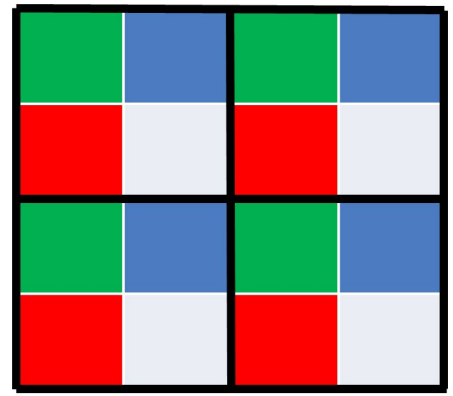


Pixel Granularity Masking

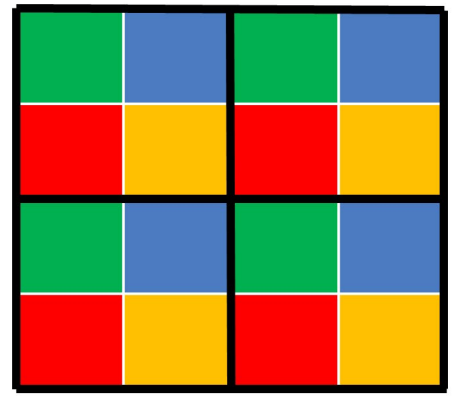
- similar for other quads



Pixel Granularity Masking

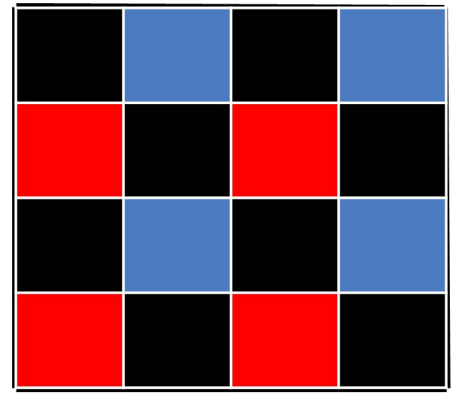


Pixel Granularity Masking



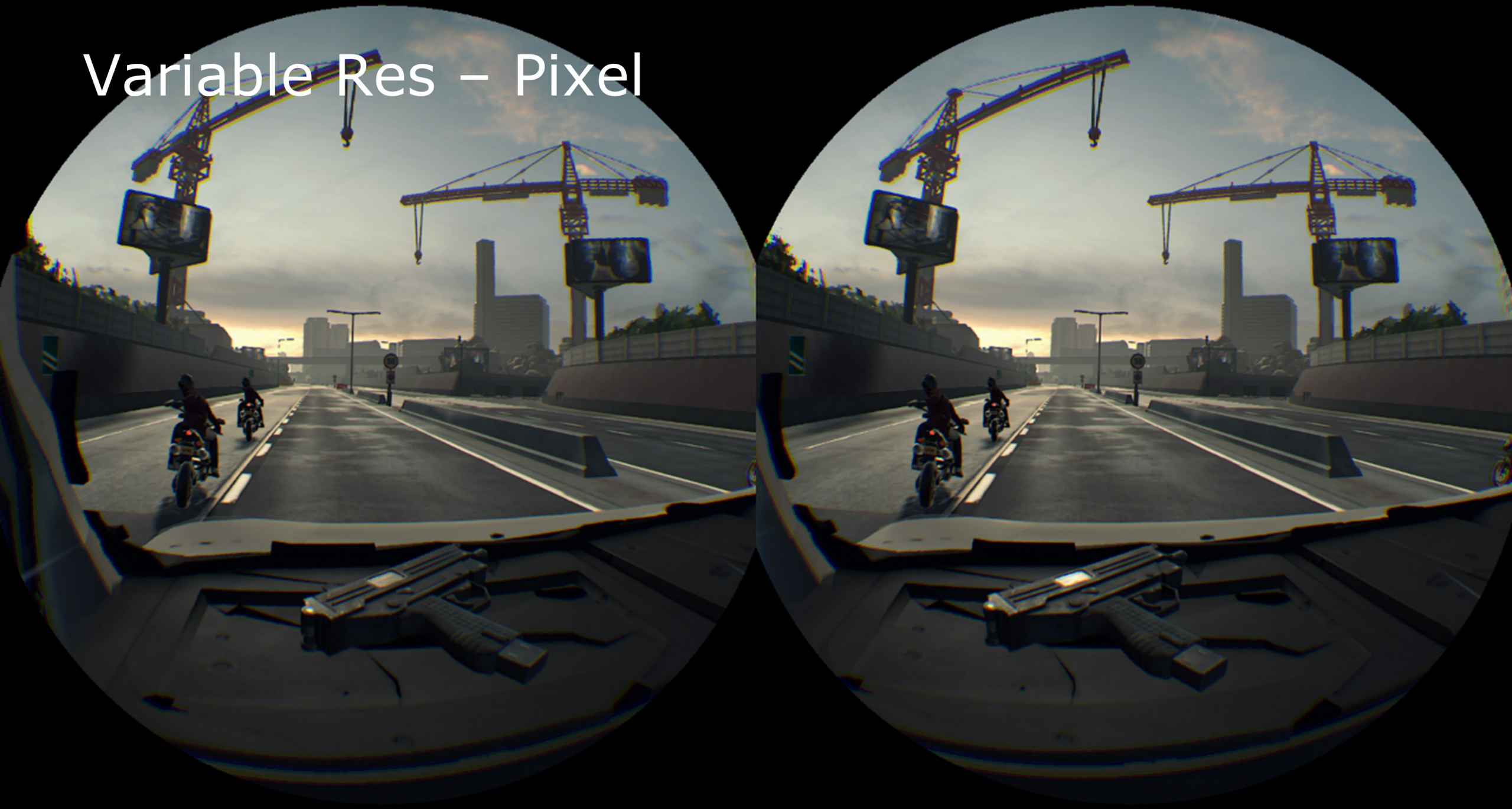
Pixel Granularity Masking

- now if we mask out the quads marked in green and yellow as before…
  - we get pixel granularity masking!



Variable Res – Quad

Variable Res – Pixel




Full Res

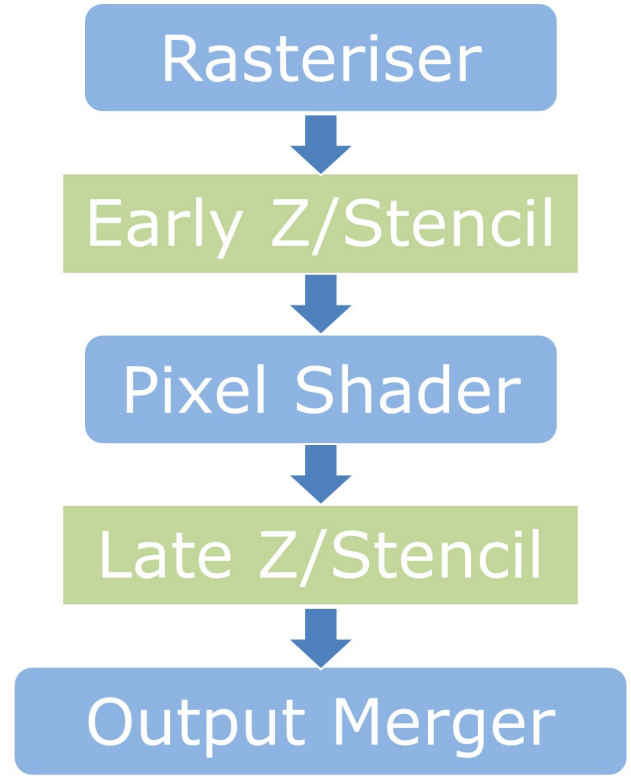

Quad x 8


Pixel x 8


Full Res x 8

Complications: Early Depth Issues

- Benefit of masking depends on **early** depth/stencil
- So if we do something to disable early depth/stencil…
  - we get **late** depth/stencil
  - we've lost all the benefit of the masking
  - masked pixels will be shaded and then rejected!
- eg Any discards
  - decals, alpha test, dithered hair, dithered LoD transitions
- Worked around this (somewhat) with eg "decal prepass"



Complications: Early Depth Issues

- Decal prepass
  - separate decal prepass lays down depth
  - so no discard required in lighting pass

- This helped but decals still considered slow by artists
  - not actually "slow"
  - just not as fast as everything else!
    - not benefitting much from variable resolution

Aside: Decals Next

- To work around this for next project:
  - decals no longer rendered in separate deferred-style "decal volumes" pass
- New system:
  1. CS to bin decals in view space
     - in the style of light tiling
  2. in lighting pass, look up which decals might affect current pixel
     - run decal code for each one
     - use decal properties to adjust inputs to lighting model for that pixel

  ⇒ no need for any discards
- Also now baking static decals into textures where possible

MSAA Trick Complications

- Code to handle it starts infiltrating all shader and runtime code
  - especially with on-demand dilation
- Graphical debugging tools harder to use
  - image is split across 4 samples
  - need to do resolve to visualise it properly

Future Plans

- Use importance of scene elements to drive high / low res regions
- Via stencil mask
  - eg character faces ⇒ high res ⇒ no masking
  - eg smoke effects ⇒ low res ⇒ mask 3 out of 4 pixels
  - …

Future Plans

- Scope for more intelligent dilate
- eg Select most appropriate sample based on depth
  - rather than just choosing closest in screen-space
  - relies on using “Mask after Prepass” so that we have valid depths for missing pixels
- eg Use object / primitive IDs on PS4™Pro…

- BUT more complex dilate
  - ⇒ less workable to apply the “dilate on demand” optimisation

Conclusions

- Resolution Gradient in general
  - makes target x1.4^2 effective resolution attainable
  - but be careful not to push it too far…
    - especially in high frequency regions
    - don’t sacrifice the image periphery just to hit 1.4^2 in the centre

- Pixel granularity
  - plenty of complications and code complexity…
  - but worth it for the large increase in quality
    - means we can push it that bit further

Source-Level Shader Debugging

SRTs can be useful for other things

- It’s an in-memory structure with names and type info
  - Not register slots
  - And it looks nice from the CPU’s point of view
  - We can read its contents without difficulty
- If only we had a CPU version of a GPU shader we could connect it directly to the SRT…
  - Then we could debug it in Visual Studio

What if we don’t use SRT?

- SRT makes it easy
  - Same structure can be interpreted the same way on both CPU/GPU
- You could shadow your real GPU bind state
  - Build an SRT which is only used for debugging
  - Then access textures/buffers through macros
- Or you put your textures/buffers in global namespace and reflect them
  - More macros

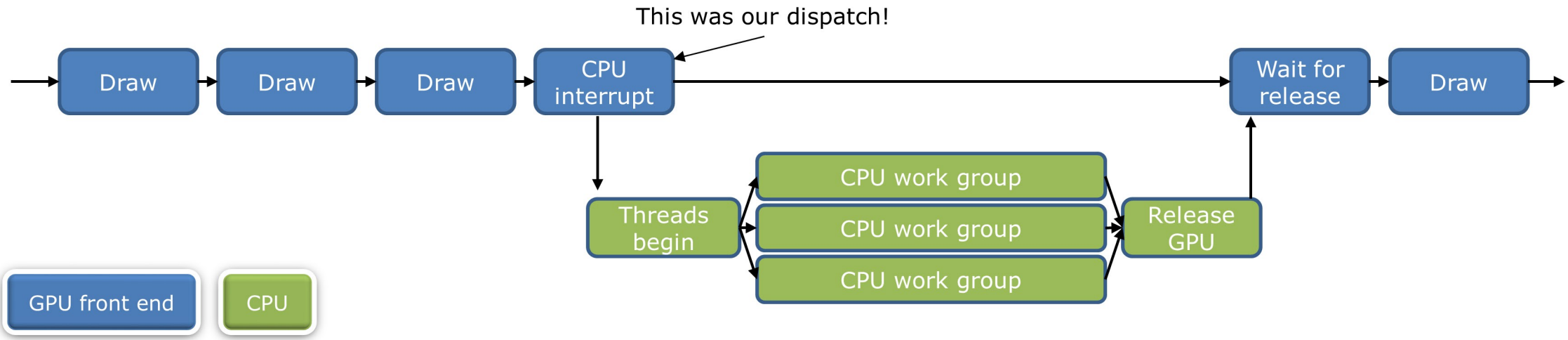
Building shaders for CPU

- Let’s compile the shader with clang
- HLSL and PSSL are similar to C/C++
  - A few templates later and vector arithmetic works
  - Add some maths functions and dot() etc work
- SRT structure is already valid C/C++ code
  - Or macro away the register slot semantic and your texture is now a global of type Texture2D

⇒ We end up with one DLL per shader

Running shaders - CS

- CS is easy; there's no hidden state to capture
- Replace dispatch() in command buffer with a CPU interrupt



Now we're on the CPU

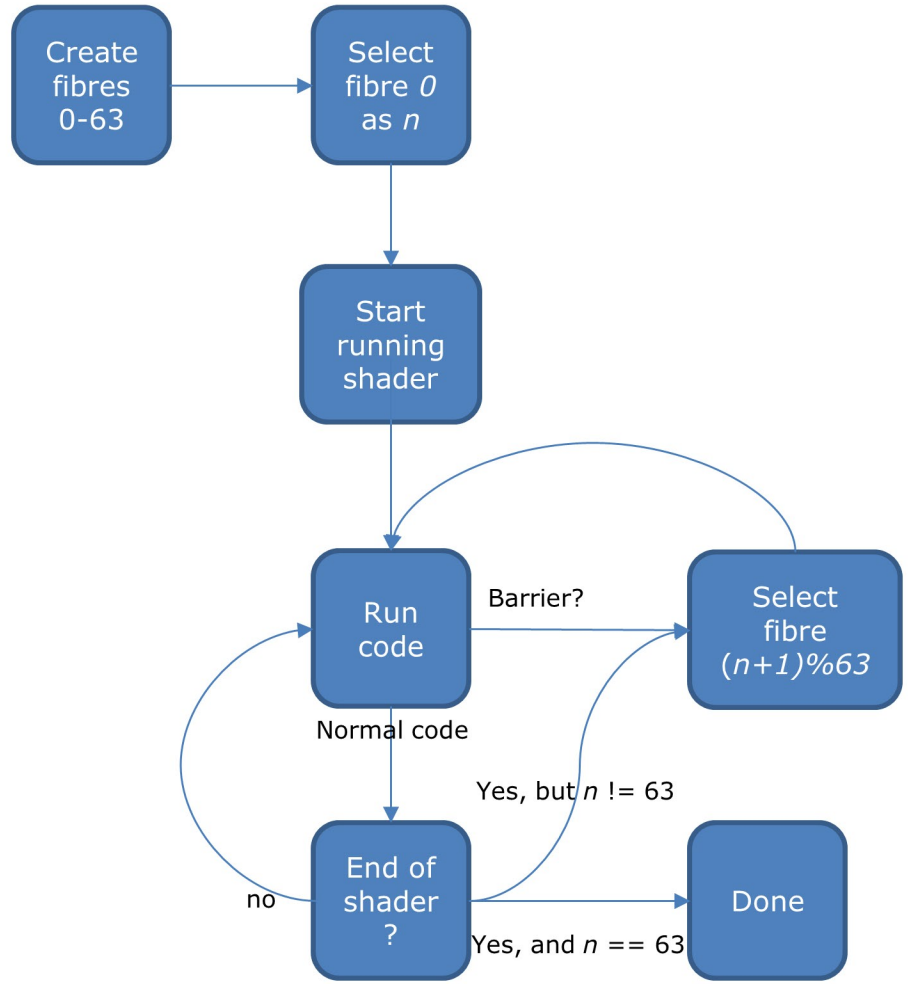
- Run function once per work group
- All work items in work group may run in parallel though
  - Barriers need to function correctly
  - We use fibres to handle barriers
- Hook GPU's SRT pointer up to DLL's SRT pointer
  - Dereference buffers and textures via the V# and T#
  - Or use stub on GPU to dereference textures for us
- Or fill each global Texture2D etc with valid data via reflection

Fibres

- A programming model for cooperative multitasking
  - Explicit context switch to another fibre
- One fibre = one SIMD work item
- One thread hosts 64 fibres
  - Only one fibre is alive at any time on the thread
  - Each fibre shares 64 KB of “LDS” via thread local storage

Fibres

- Fibre *n* will switch to *n+1*
  - At the end of the shader func
  - Or on a barrier
- When fibre 63 finishes func
  - Shader has finished
- When fibre 63 barriers
  - fibre 0 will continue



Running shaders - CS

Source code, call stack and autos work properly
SRT, globals and constants visible

Watch 1

Name	Value
srt	{ m_pPerPass=0x000000dd31b48e30 {0x000000dd31b48e
m_pPerPass	0x000000dd31b48e30 {0x000000dd31b48e30}
m_pMaterial	0x000000dd31b48e60 { m_pTextures=0x0000000000000
m_pTextures	0x0000000000000000 {0x0000000000000000}
m_pBuffers	0x000000dd31b48e90 { g_lightBoundsList={...} g_lightTi
g_lightBoundsList	{...}
g_lightTileList	{...}
g_lightList	{...}
g_probeBlendWeights	{...}
m_pCbMaterial	0x000000dd31b48c90 { ... }
mCameraWorldViewProj	{ row=0x000000dd31b48c90 {{ ={ x=-0.08384018 r=-0.08
mCameraWorldView	{ row=0x000000dd31b48cd0 {{ ={ x=-0.07545617 r=-0.07
mCameraInvProj	{ row=0x000000dd31b48d10 {{ ={ x=0.90000009 r=0.900
mCameraInvView	{ row=0x000000dd31b48d50 {{ ={ x=-0.07545623 r=-0.07
mCameraProj	{ row=0x000000dd31b48d90 {{ ={ x=1.11111104 r=1.111
mCameraNearFar	{ ={ x=0.10000000 r=0.10000000 } ={ x=0.10000000 r=0.1
mFramebufferDimensions	{ ={ x=0x00000200 r=0x00000200 } ={ x=0x00000200 r=0x
x	0x00000200
r	0x00000200
y	0x00000200
g	0x00000200
z	0x00000020
h	0x00000020

```
275         //change range from (0,1) to (-1,1) - ie NDC
276         frustum[0].xy = (frustum[0].xy * 2) - 1;
277         frustum[1].xy = (frustum[1].xy * 2) - 1;
278         frustum[2].xy = (frustum[2].xy * 2) - 1;
279         frustum[3].xy = (frustum[3].xy * 2) - 1;
280
281         //transform to view-space
282         frustum[0] = ConvertProjToView(frustum[0]);
283         frustum[1] = ConvertProjToView(frustum[1]);
284         frustum[2] = ConvertProjToView(frustum[2]);
285         frustum[3] = ConvertProjToView(frustum[3]);
286
287         //calc side planes
288         for (uint i = 0; i < 4; i++)
289             frustumPlanes[i] = CreatePlaneEquation(frustum[i], frustum[(i+1) & 3]);
290
291         //calculate the mid point of the end of this skewed frustum, and hence the radius of a circle enclosi
292         float3 skewedFrustumMiddle = (frustum[0].xyz + frustum[1].xyz + frustum[2].xyz + frustum[3].xyz) * 0.
293
294
295         //need different half dists
296         //(normalise using far clip dist)
297         normalisedSkewedFrustumFarHalfDists.x = 0.5 * length(frustum[0].xyz - frustum[1].xyz) / GET_MAT_CONST
298         normalisedSkewedFrustumFarHalfDists.y = 0.5 * length(frustum[1].xyz - frustum[2].xyz) / GET_MAT_CONST
299
300         //now normalise the frustum end points - so that this (tile) frustum is normalised and effectively no
```

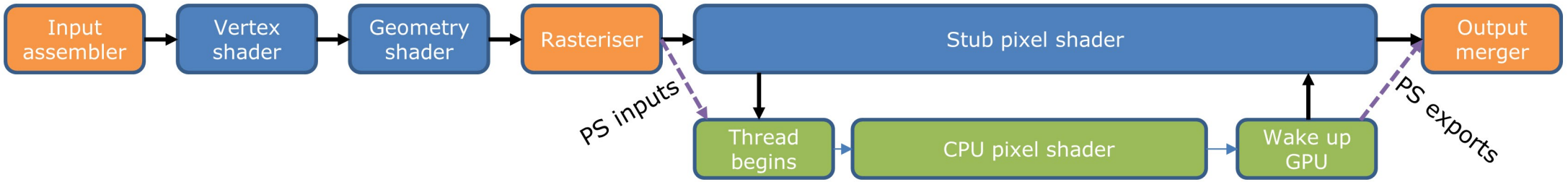
100 %

Autos

Name	Value	Type
frustum	0x00000000074ad500 {{ ={ x=-9030.79883 r=-9030.79883 } ={ x=-9030.79883 r=-903	float4[4]
frustumPlanes	0x00000000074ad540 {{ ={ x=0.00000000 r=0.00000000 } ={ x=0.00000000 r=0.0000(	float4[4]
frustumPlanes[i]	{ ={ x=0.00000000 r=0.00000000 } ={ x=0.00000000 r=0.00000000 } ={ x=0.00000000	float4
frustum[(i+1)&3]	{ ={ x=-8466.37402 r=-8466.37402 } ={ x=-8466.37402 r=-8466.37402 } ={ x=-8466.3	float4
frustum[i]	{ ={ x=-9030.79883 r=-9030.79883 } ={ x=-9030.79883 r=-9030.79883 } ={ x=-9030.7	float4
i	0x0000000000000000	uint

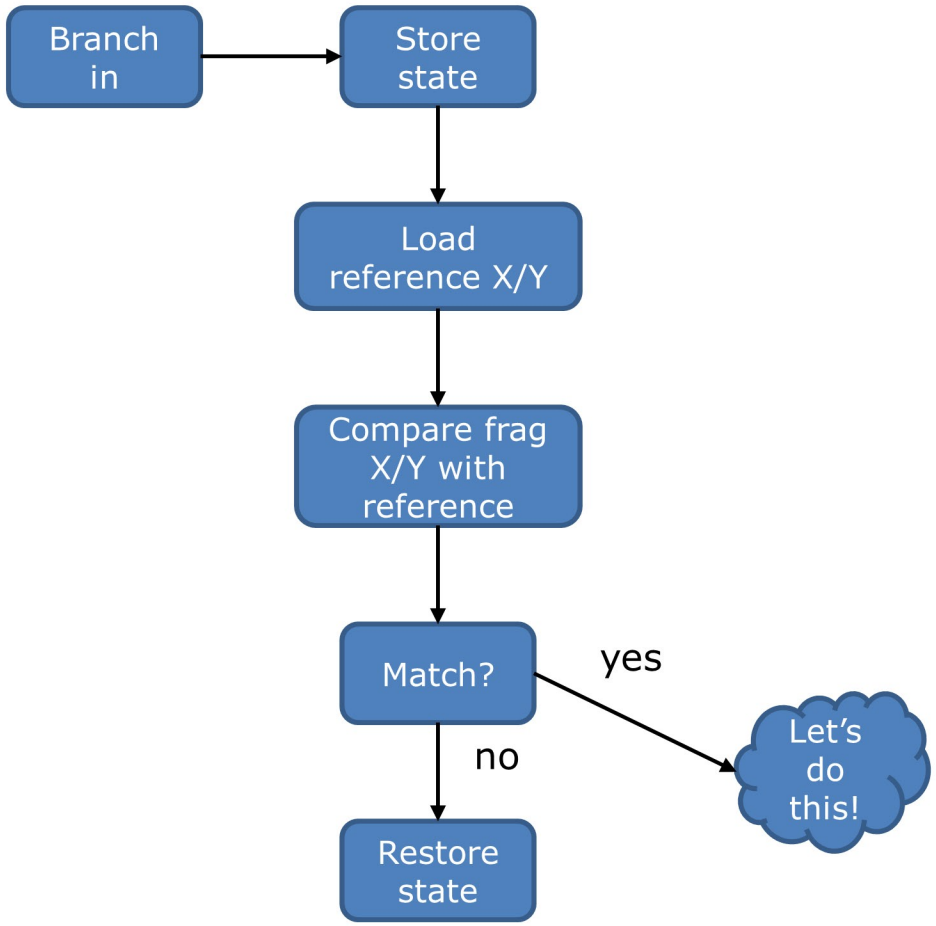
Running shaders - PS

- PS is much harder
  - We can't easily emulate what the rasteriser is doing
  - So let's use it like normal and capture the results



Stopping execution

- Select a pixel X/Y from a draw call to debug
- Replace the first few instructions of shader with something to match that X/Y against fragment's screen position
  - More ISA or D3D IR code injection!



Capturing our PS inputs

- For AMD GCN,
- Generated in-shader from LDS data and barycentric coordinates
- Could copy LDS data and evaluate on CPU
  - Where is the data in LDS? How much is there?
- Or could use v_interp_* to evaluate on GPU
  - Needs coordination with shader reflection info
- Pros and cons to each

Interpolants on GPU

- Let’s perform all the v_interps ourselves
- Place the results in a buffer for the CPU
  - Buffer matches layout of PS_INPUT
- Signal the CPU and put the GPU into a spinlock

```
v_interp_mov_f32        v6, p0, attr4.x
v_interp_mov_f32        v7, p0, attr4.y
v_interp_mov_f32        v8, p0, attr4.z
v_interp_mov_f32        v9, p0, attr4.w
buffer_store_dwordx4    v[6:9], v0, s[0:3], 0 offset:0x140 glc slc
s_waitcnt               0
v_writelane_b32         v5, s4, 4
v_writelane_b32         v5, s5, 5
v_writelane_b32         v5, s6, 6
v_writelane_b32         v5, s7, 7
s_mov_b32               exec_lo, #0x000000ff
s_mov_b32               exec_hi, 0
s_or_b32                s3, s3, #0x00800000
buffer_store_dword      v5, v0, s[0:3], 64 glc slc
s_waitcnt               0
s_mov_b64               exec, vcc
s_mov_b32               s3, #0x18027204
buffer_load_dword       v6, v0, s[0:3], 0 glc slc
s_waitcnt               0
v_cmp_eq_i32            vcc, #0x00000000, v6
s_branch                0x0000092225fde40
s_endpgm
```

interpolate

store

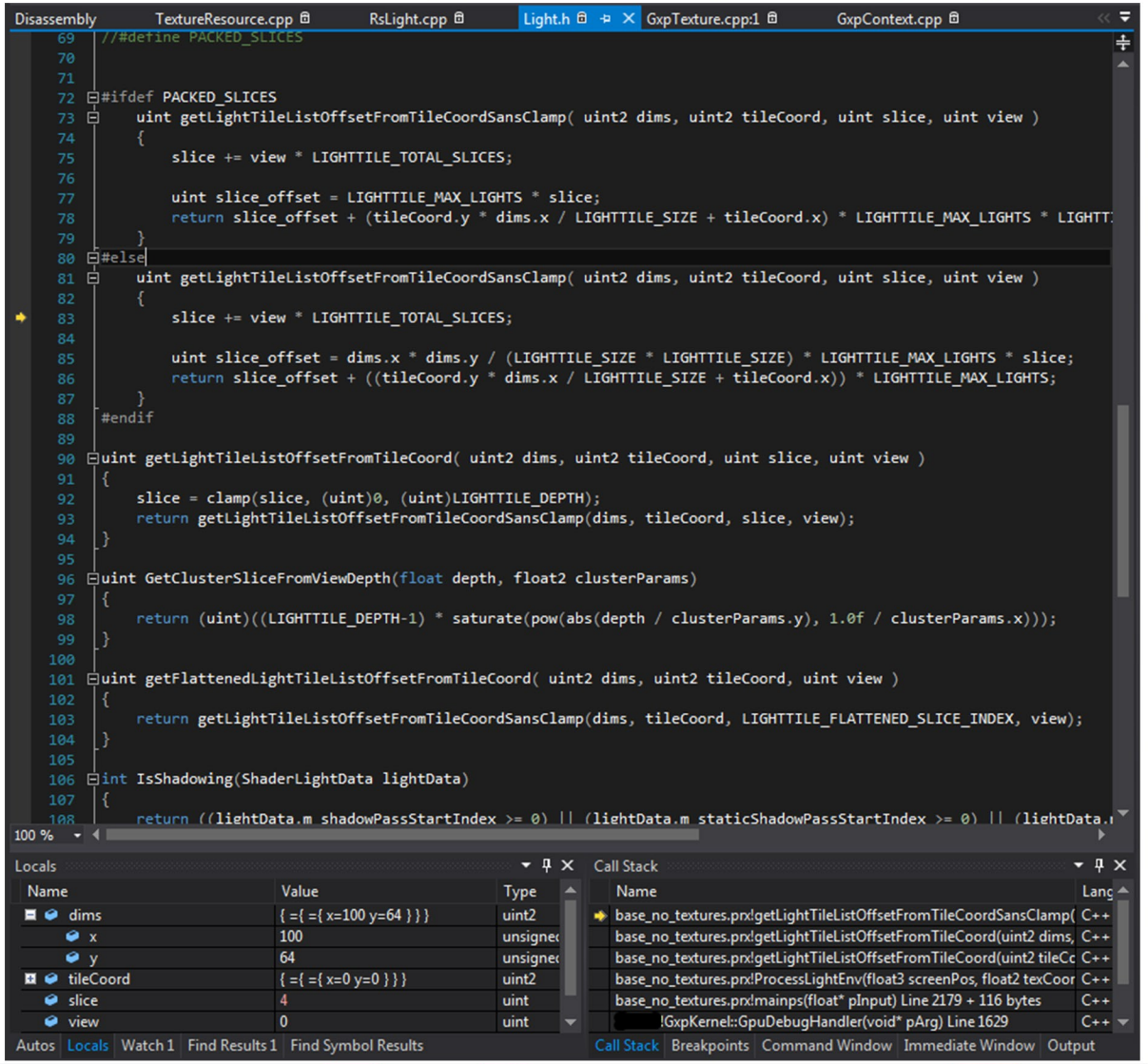
create buffer descriptor

signal CPU

spinlock

Now we're on CPU

- Connect GPU SRT pointer to DLL's SRT pointer
  - Or fill in all exported global buffers/textures with real state
- Connect PS_INPUT to the blob of interpolants
- Run the PS entry point and 'enjoy' debugging



Comparison vs PS4™ SDK Tools

- PS4™ SDK now comes with GPU Debugger
  - Which nowadays supports source-level debugging on GPU
  - So try that first!
- Our version augments this
  - Again, based on the fact that we can do this in-engine
- We can debug our shaders in a mature and familiar environment
  - with the full -O0 debugging experience
- REMEMBER: Shipping with code-injection enabled is a TRC violation!

Conclusions

- HLSL and PSSL are similar to C++
- Debugging your shaders in Visual Studio is possible with effort
  - Worth the effort as code gets more complex

Adaptive Resolution

Background

- Very hard to consistently hit 16.6ms on GPU
  - more vital than ever on VR that we really stay below that
- In addition we're aiming for 1080 x 960 $x1.4^2$ if possible
  - so we're actually looking to **increase** the resolution


25.00ms
20.00ms
16.67ms
15.00ms
10.00ms
5.00ms
0.00ms
1.50x
1.40x
1.30x
1.20x
1.10x
1.00x

Background

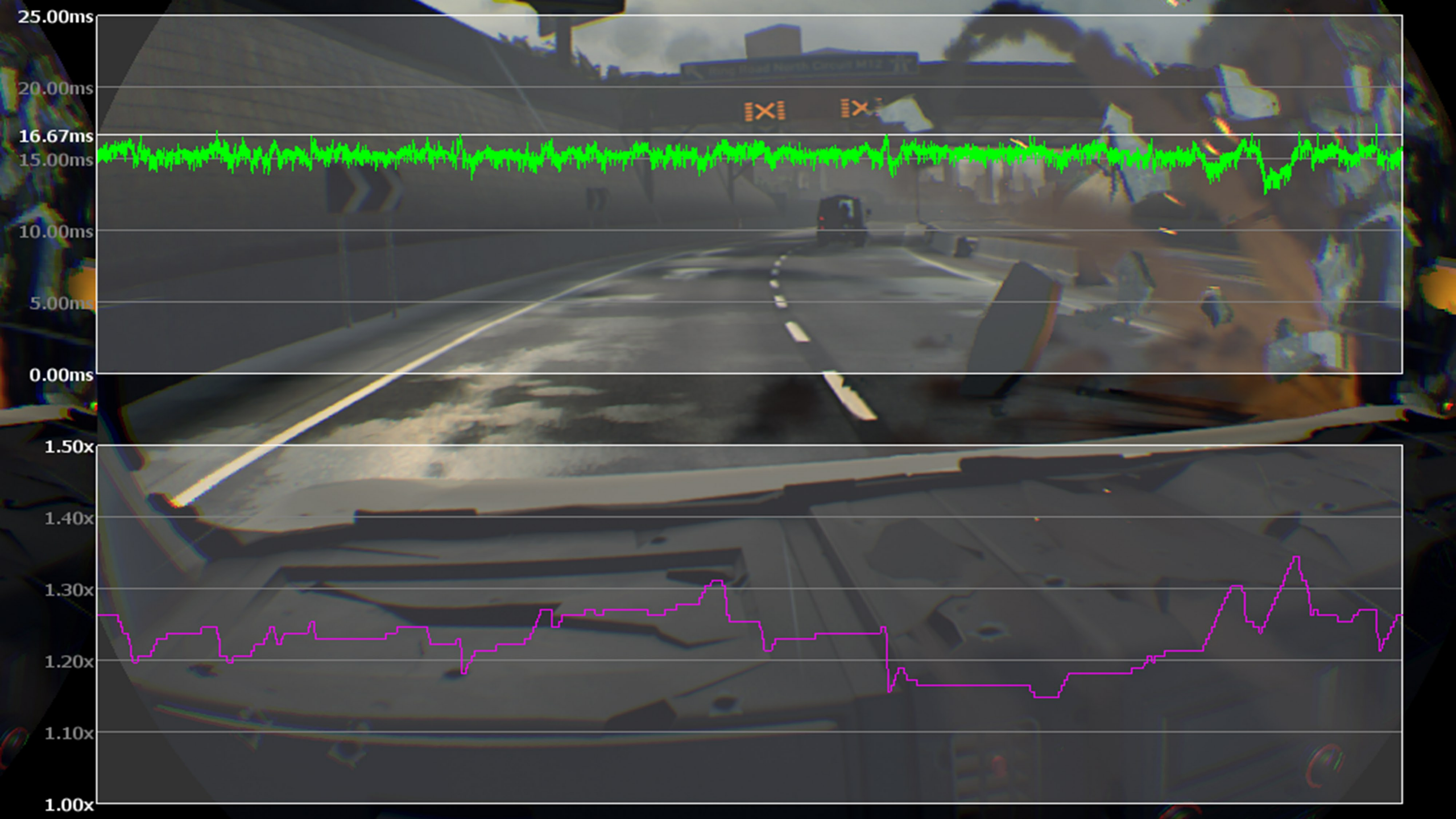
- Previous approaches to hitting 60Hz:
  - ask artists and designers to balance art / AI cost throughout level
    - but only so much they can do
    - view / number of on-screen enemies is variable
  - drop resolution throughout whole level
    - eg whole level @ $x1.1^2$
    - what a waste!
- So… we implemented Adaptive Resolution

Resource Reallocation

- Every frame
  - reallocate memory for all dynamic-res render-targets / buffers / etc
    - linear allocator ⇒ very cheap
    - virtual memory system ⇒ no fragmentation concerns
- Extra costs
  - can’t rely on any on-initialise clears any more
    - because buffers / targets are effectively re-initialised every frame
    - required the addition of some extra per-frame clears
  - stencil masks for Resolution Gradient now prepared every frame
    - couldn’t afford the memory to prepare one for every resolution up front

Heuristic

- Nothing fancy…
- Every frame:
  - is last frame’s GPU time close to high / low thresholds?
  - if so, adjust resolution up or down
    - we drop resolution much faster than increasing it
  - wait two frames after res-change before measuring new frame-time


25.00ms
20.00ms
16.67ms
15.00ms
10.00ms
5.00ms
0.00ms
1.50x
1.40x
1.30x
1.20x
1.10x
1.00x

The Good

- GPU frame spikes were effectively a thing of the past
- No longer have to account for that worst case, so:
  - ~95% of the game at higher resolution
  - Previously: locked at ~ $x1.1^2$
  - Adaptive: average > $x1.2^2$
    regularly reaches $x1.4^2$
- Bonus: PS4™Pro version automatically higher res from day 1!
  - with no code changes!
- But there were down-sides...

The Bad: Metrics

- All our artist-friendly metrics and auto-generated graphs used frame-time
  - frame-time now a useless metric!
- Changed metrics to show resolution multiplier
  - but requires a change in mind-set
    - took a while (still going, to be honest)

The Bad: Safety Net

- Adaptive Resolution is a bit of a safety net
  - content creators didn't have to be so careful any more
    - harder to spot a drop in res than a frame-out
  - became a battle to stop the resolution quietly degrading

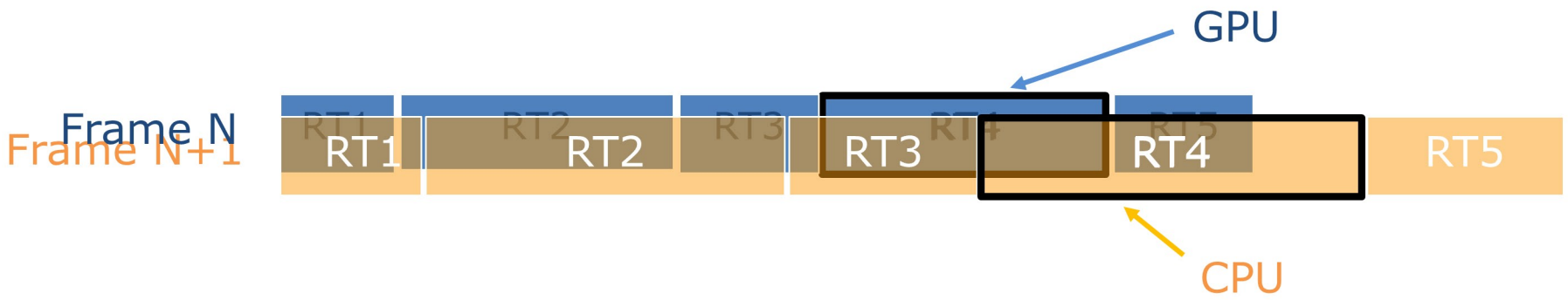
The Bad: No Optimal Resolutions

- Previously optimised for our target resolutions
  - eg hand-tuned Compute Shaders
  - was fine for a handful of resolutions
  - but we can’t realistically do this for all resolutions!

- Now our optimal resolutions are actually a bad thing
  - need performance to be predictably proportional to resolution
  - so had to disable those fast paths

The Bad: Fiddly Details

- Conceptually a little confusing
  - CPU and GPU accessing "same" resource
    - but now at **different** addresses for each on any given frame



Conclusions

- But the down-sides were well worth it…
  - it was a BIG win for average and peak resolution
  - and something of a life-saver

Stuff We Learnt Along The Way

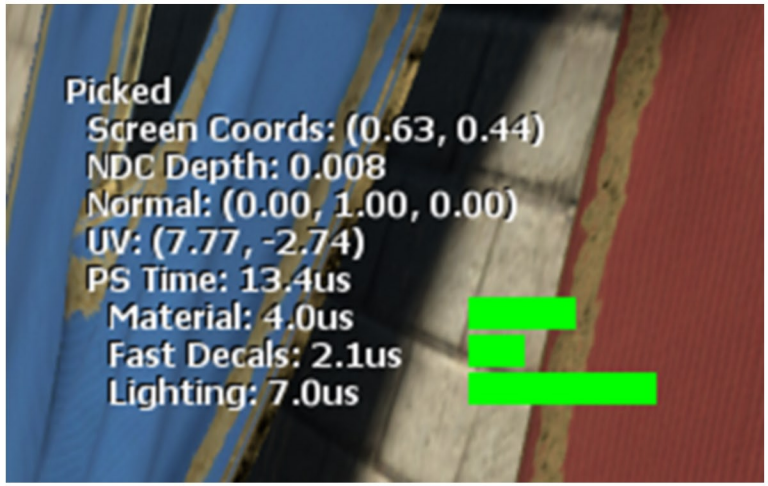
Lots learned along the way

- For every thing which worked we did five which didn’t
- Here’s a list of the more interesting things

In-shader profiling is useful

- Our shaders are huge
  - 2500+ lines
  - If CPU code was this big you'd want to instrument your functions
- So instrument your shaders with PSSL GetTimer()
- We added costs for
  - Material evaluation
  - Decal calculation
  - Different types of lighting

⇒ Empower artists to do optimisation themselves

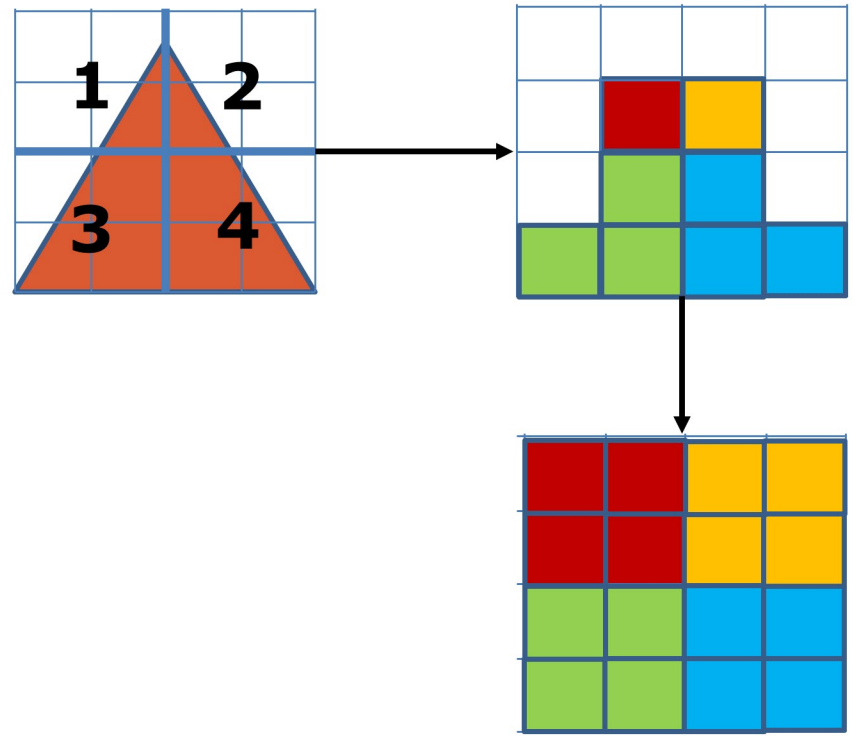


Full-screen heat map

- Timings of each draw presented on screen as a colour
- Uses label write at the end of draw with GPU core timestamp
- Can show PS, VS, normalised by pixel count
- Accuracy issues due to draw calls running in parallel

WQM “whole quad mode”

“If any bit in a group of four is set to 1, set the resulting group of four bits all to 1”



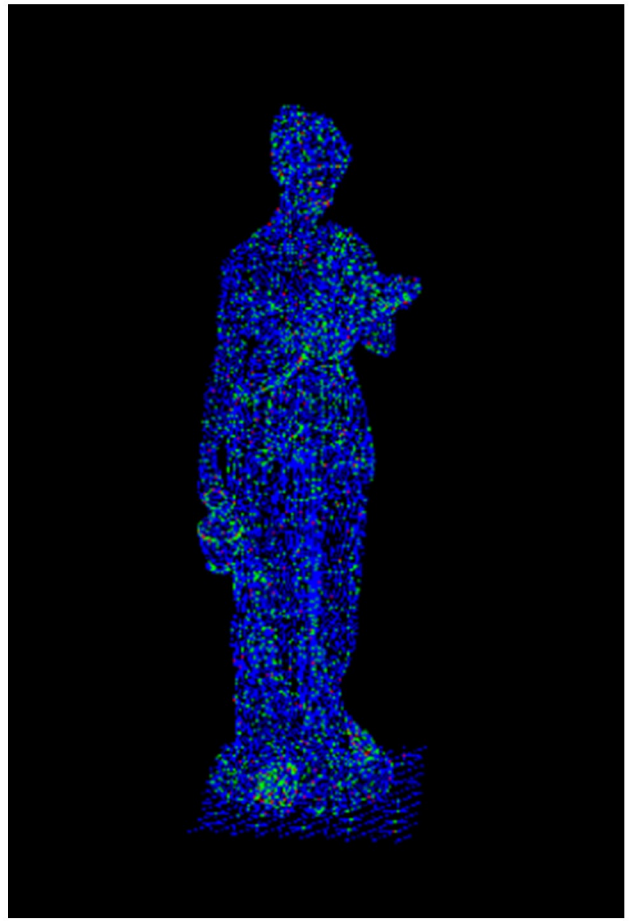
Used on EXEC it re-enables lanes which should not render

s_wqm_b64 for reference

Quad overdraw view

- Shows wasted SIMD lanes
  - eg where triangle edges are
- Multi-step render
  1. Draw scene, but use wqm instruction to inflate pixel's quad mask

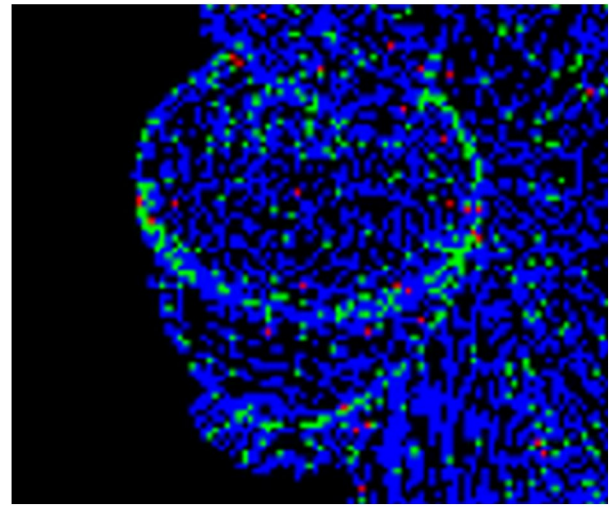
     - for each pixel in a quad atomically increment a 2D buffer at X/Y position
  2. Draw a full-screen blit of buffer as a texture and colourise the data



Quad overdraw view

- Shows wasted SIMD lanes
  - eg where triangle edges are
- Multi-step render
  1. Draw scene, but use wqm instruction to inflate pixel's quad mask

     - for each pixel in a quad atomically increment a 2D buffer at X/Y position
  2. Draw a full-screen blit of buffer as a texture and colourise the data

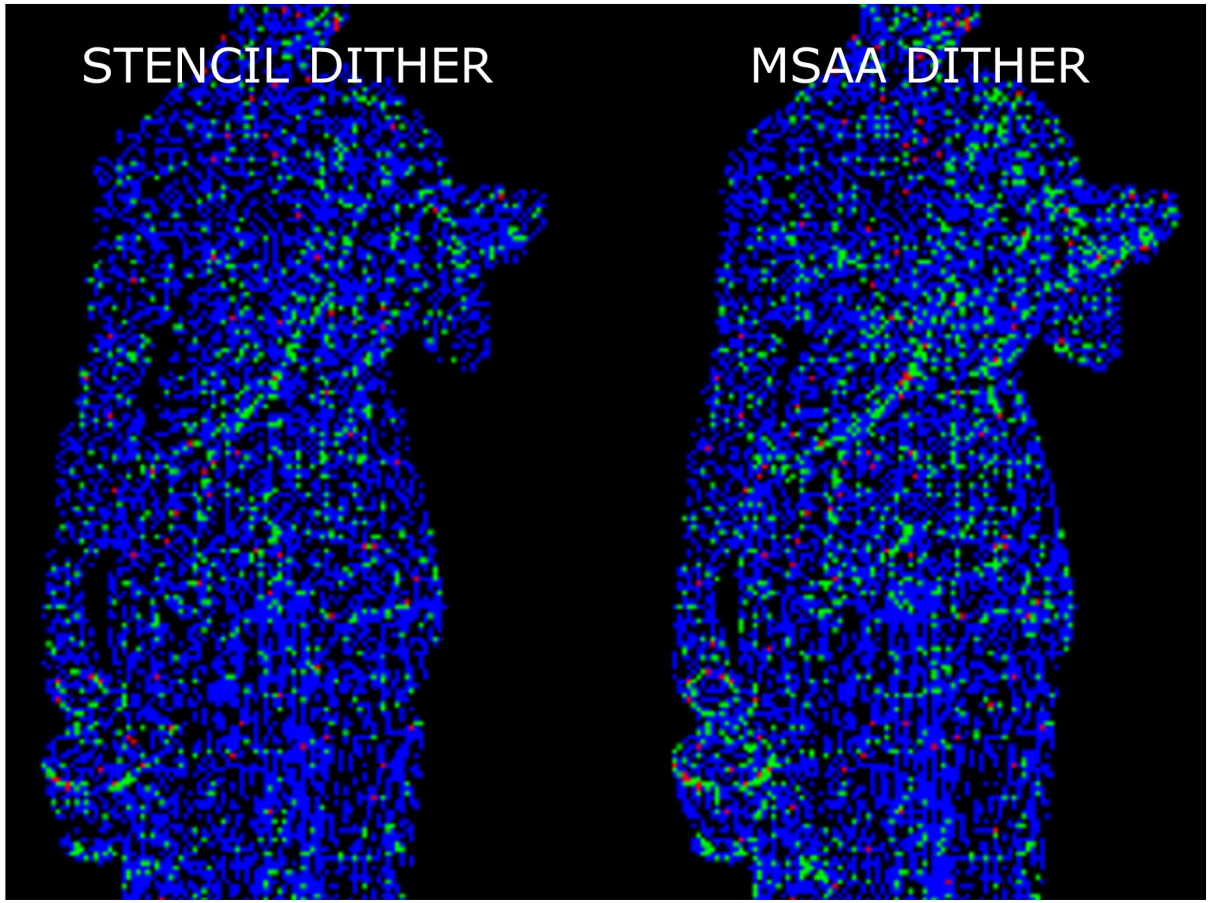


Occupancy woes

- Most shaders use near the maximum register allocation
  - Occupancy is around three wavefronts per SIMD
  - If you can’t get the number down, what *can* you do with all those registers?
  - Perhaps you can manage better internal scheduling of memory loads
- Try trading vector registers for scalars
  - Manipulate your EXEC mask!
- If bound on LDS, consider 16-bit types
  - This doubled our light tiling occupancy
  - Though check how bank conflicts will affect performance

MSAA pixel granularity

- It’s slower than doing it normally
- The quad pattern is coarser now, so overdraw increases

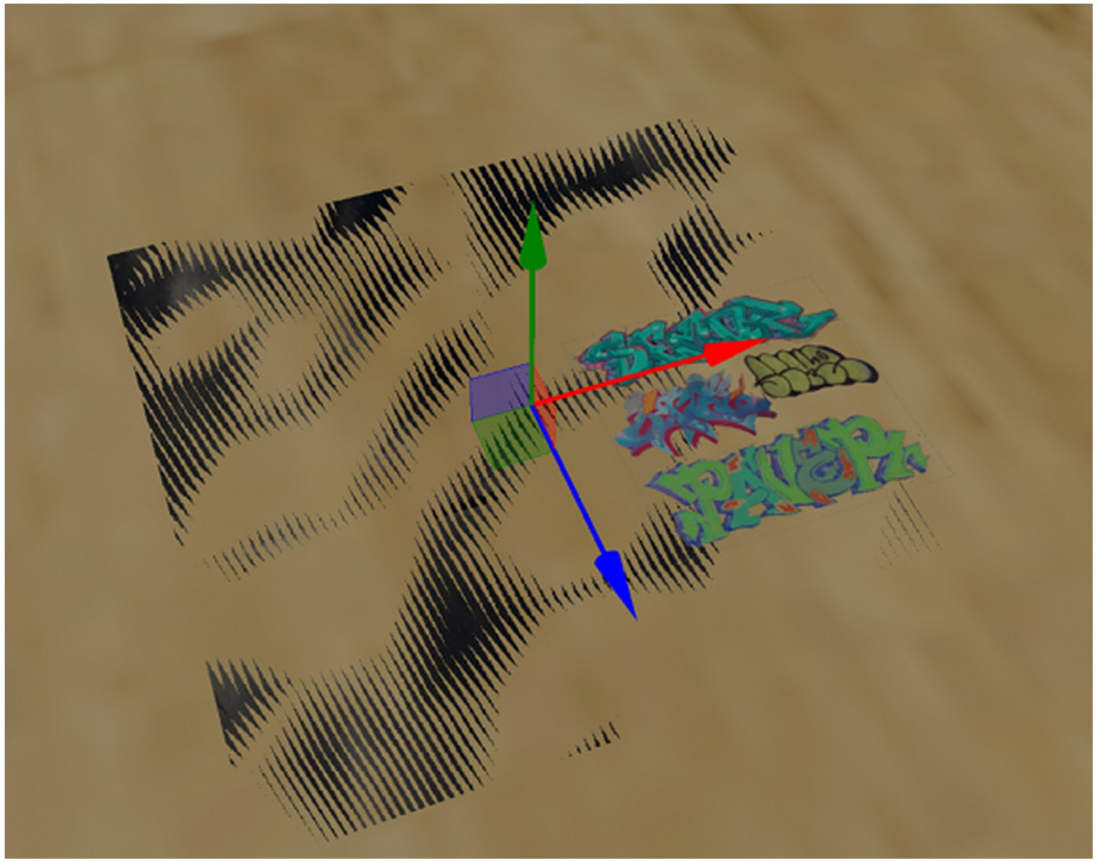


MSAA pixel granularity

- Z metadata is one quarter of the size
- Worse fast Z performance ☹

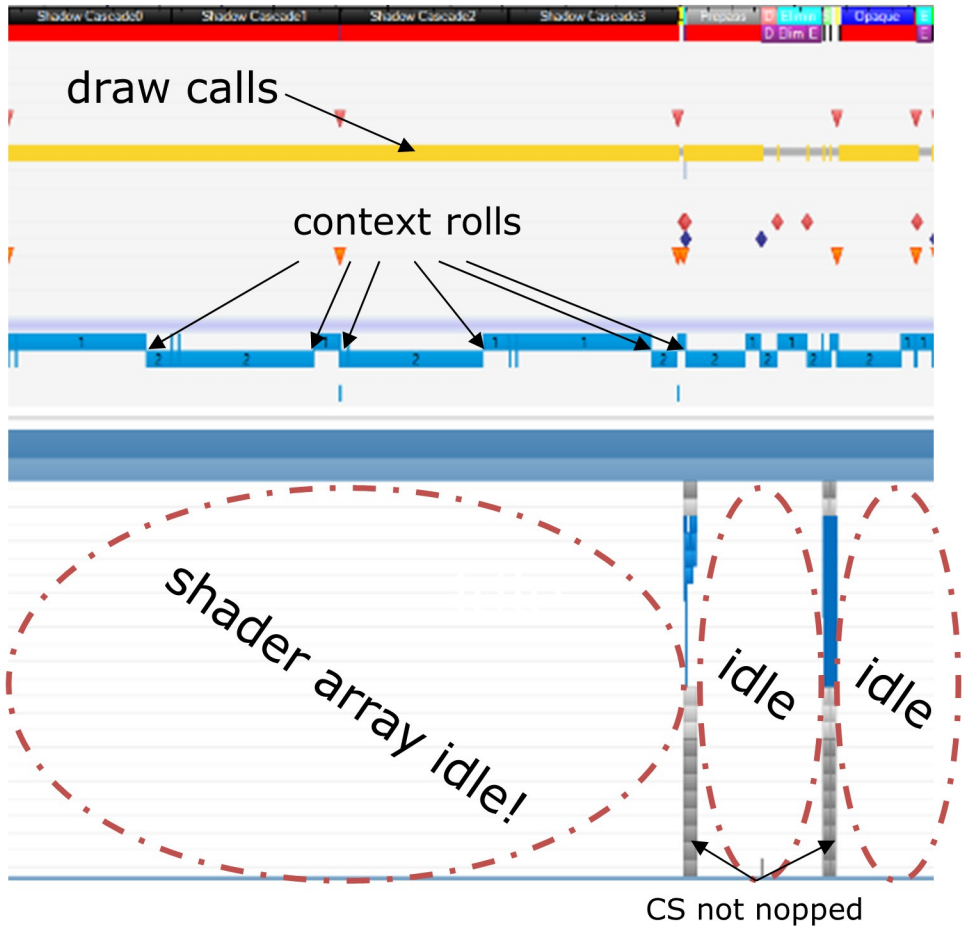
MSAA pixel granularity

- Rasterization results are not identical compared with non-sample rate version
- This prevents us sharing Z-writing depth pass with sample rate colour passes
- Tile modes also do not allow aliasing of quarter-res 4xMSAA as non-MSAA full-res target



Empty draw calls are not free

- Avoid redundant state setting
  - Significant win early on in project
- Avoid non-contributing draw calls
  - Try making your draw calls draw zero prims then profile to measure your GPU command processing time
- 14000 empty draw calls still take 7.5ms



Draw call predication

- Use Z-only pre-pass pixel counts to omit draws in opaque pass
- Rarely a win
  - Complications with auto-instancing
  - Empty draw calls are not free!
  - Frees up shader resource but it is left idle

Draw call predication

- We attempted to use command patching by CS to address this, using predication stats
- Still high GPU command processing time, blocking shader array resource
  - Other pre-draw set-up consumes significant time

Stuff We Learnt Along The Way: Async Compute

Async compute scheduling

- We spent a significant amount of time working with async compute
- Used three ways,
  1. “I must have this by 8.3 ms into the frame”
  2. “I must have this by the time we light the image”
  3. “I’d like to have this by the beginning of next frame”
- All modes presented a challenge
- We just want to fill in any unused resource – we don’t want to slow graphics down!

Async light tiling example

- "I must have this by the time we light the image"
  - Start time for shading lit pixels depends on scene, shadow complexity
- Scheduling the kick is tricky
  - Run once per eye, ALU heavy, uses lots of LDS.
- Do we do both eyes at once in the shadow pass?
  - Still slows down bandwidth-sensitive stuff.
  - What if shadows take less time than two eye's worth of light tiling?
- Or do we try and keep each eye symmetric?

More scheduling woes

- "I'd like to have this by the beginning of next frame"
- For one shader optimising CS dims early in project caused problems later on
- Shader had hand-coded versions for different resolutions including slow generic fall-back
  - When we introduced dynamic resolution this became a bottleneck
  - Generic version always selected, 100x too slow
  - So I wrote a tool to permute group dims of shader based on resolution 😐
  - This led to inconsistent performance

More scheduling woes

- Wavefront limits and CU masks for management of async CS
  - Fiddly and still no joy whenever we used this
- Shader needed a complete re-write
- It required large amounts of wavefronts and LDS, which did not fit well with our PS at master time
  - Yet it ran well when the shader was originally written
  - The shipping “any resolution” version had high latency and lower throughput but fit much better with PS – giving overall higher frame rate
- In future I would write CS to use minimal resource, at the expense of performance

Async compute: go sync instead?

- Use of async compute is not *required*
  - It is another tool in your arsenal
  - Use when appropriate
  - Experiment with scheduling options on various complex scenes
- Ensure your job interface allows easy switching from sync<->async
  - Programming model is pretty similar
  - Don’t get locked in to one mode: you may want to change your mind

Semi-synchronous

- If we did it all again, we would probably not go async for tightly-coupled graphics tasks
- Instead kick CS synchronously from the graphics ring
- But have asynchronous waits instead
  - You don’t have to sync on a CS job straight after its dispatch
  - Easier programming model
  - Perhaps will absorb unused cycles more efficiently

THE END

Special thanks to:

Bruno Ribeiro

James Answer

Aaron MacDougall

THE END

simon.hall@sony.com
joe.milner-moore@sony.com

QUESTIONS?