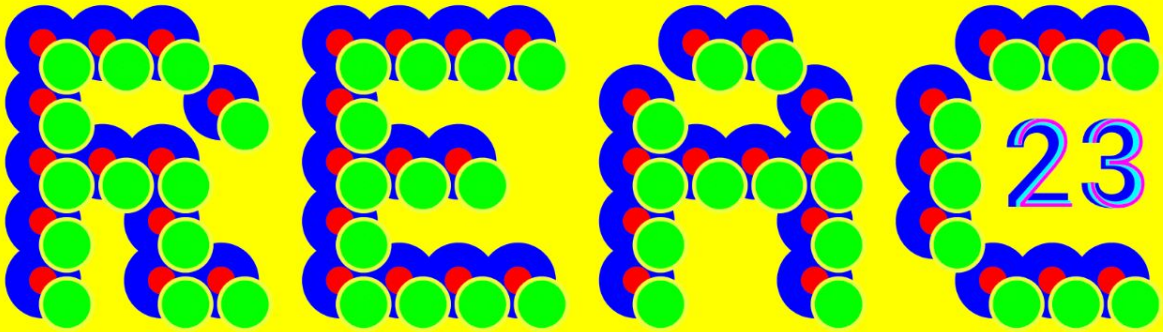
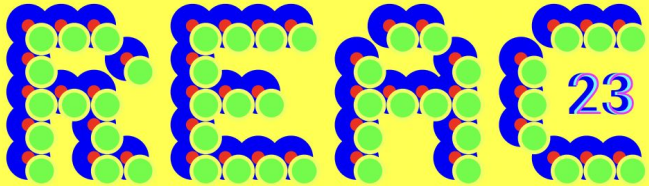


Rendering
Engine
Architecture
Conference

Dan Baker, dan.baker@oxidegames.com

Generalized Decoupled Shading

Oxide Games

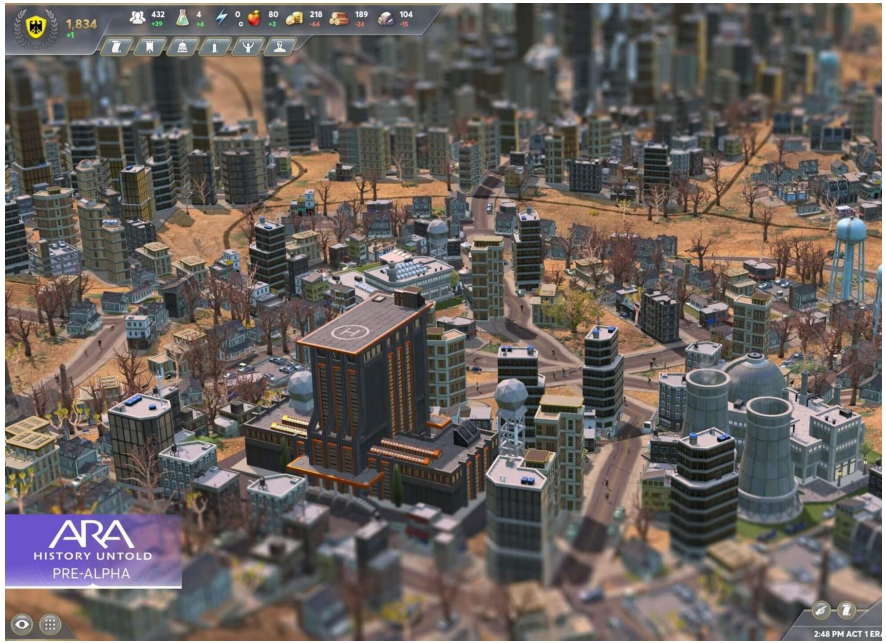


Rendering Engine
Architecture
Conference

Generalized Decoupled Shading

What is decoupled Shading?

- Shading is decoupled from the screen sampling – We shade first, then rasterize
- Similar to concept of REYES
- Lots of advantages
  - Shader Anti Aliasing
  - Better BRDF accuracy (distinct from AA)
  - Variable shading
- But no major game/engine built on it in a general way
  - Ashes of the Singularity had rudimentary solution
  - Mega texturing from Doom might have been a path to get there



What is decoupled shading

- Shade the object somewhere first
- Raster the already shaded object onto the screen
- Hopefully only shading only the visible parts and appropriate LODs

EUROGRAPHICS 2022 / A. Ghosh and L.-Y. Wei
(Guest Editors)

Volume 41 (2022), Number 4

Generalized Decoupled and Object Space Shading System

D. Baker and M. Jarzynski

Oxide Games, USA

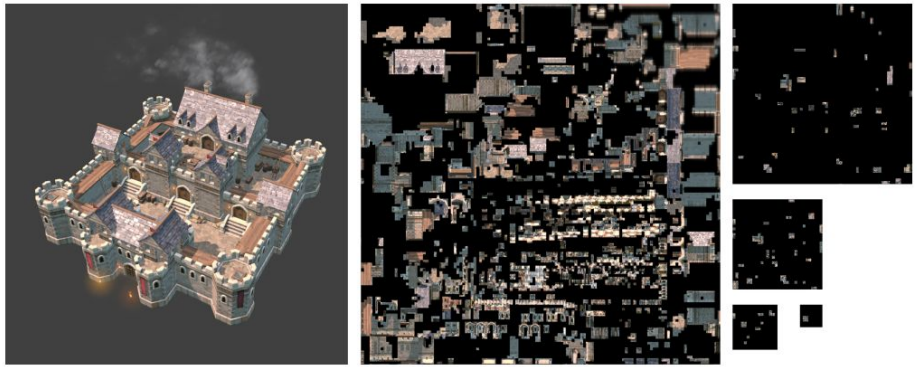


Figure 1: *Rendered castle model (left) and the associated virtualized shadel sheet (right) with level of detail system (similar to a mipmap).*

Obstacles for making a decoupled renderer

- Art assets need different process, more memory, difficult to make robust
- **Too much memory needed for sample storage (also cited as a reason for abandonment of Mega textures)**
- Performance overhead too high compared to other rendering methods
- Visibility not fine grained enough, too many shade samples computed which aren't visible (also exasperates memory pressure)
- Material integration – need to build a virtual shader stage
- Too many problems to build a major game

Prior Engine version

Ashes of the Singularity

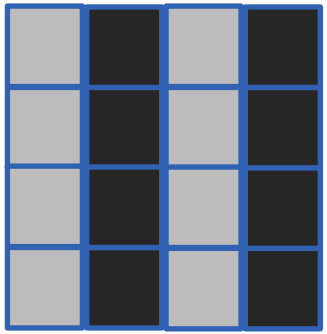
- Rendered every object into its own section of a texture
- CPU size estimation, MIP processed at highest level, filtered down
- Terrain used own custom stitching system
- Worked because very specific game

- Not enough Bullets – VR demo
  - Similar to Muellar et al, broke models into small chunks, did visibility estimation, streaming, network simulation
  - Shared shading, shade delay
  - Too many drawbacks

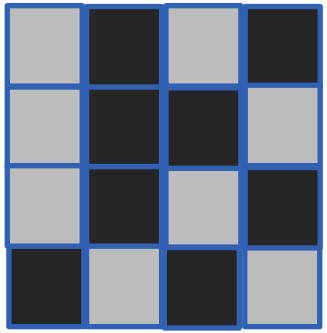
Anti-Aliasing

A simple shader such as:
Return IntTexCoord.x % 2 == 0 ? 1.0f : 0.0f

will alias instantly



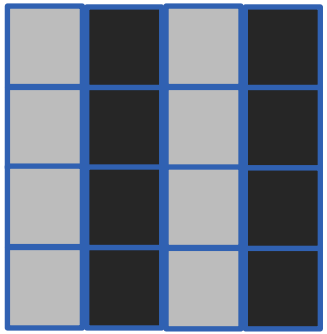
In texture space might look perfect



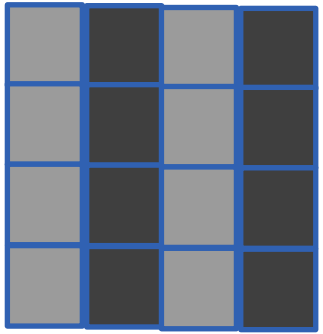
Will get near noise on rendering

Anti-Aliasing

Decoupled Shading, samples are evaluated then filtered, results will be typical linear blur

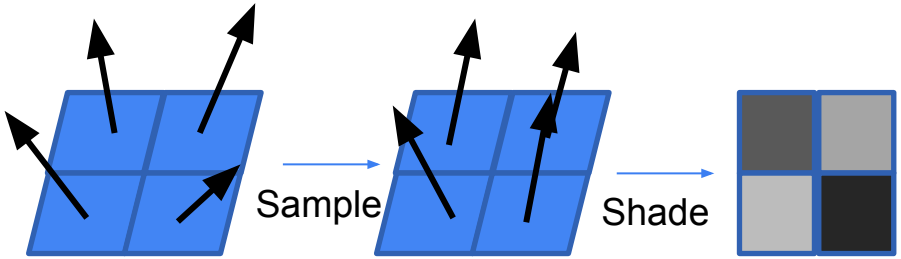


In shade space it’s exactly what we want



Get blurrier via implicit AA, but not noise

Temporal Stability

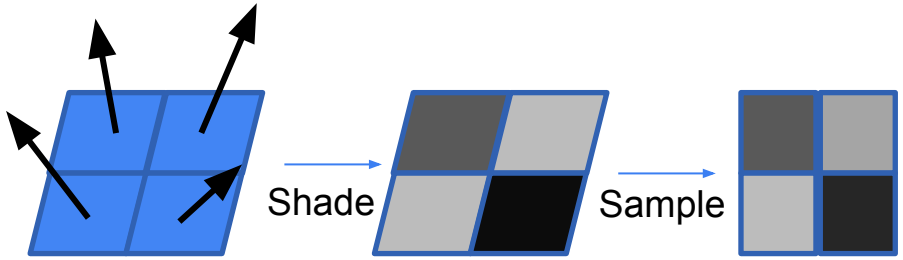


Sampling introduce tiny differences in things like normals between every frame – due to sub pixel sampling differences. Causes aliasing. However, in addition to aliasing, any filtering of samples alters most BRDF formulations

A few techniques to help solve both these issues (e.g. LEAN mapping), not always practical

The screen spaced aliasing component causes shimmering and is a deep intrinsic problem with deferred and forward rendering for games. Altering the BRDF more of a problem for film

Anti-Aliasing / Temporal Stability



Decoupled shading, input samples are always the same, then filtered. This helps anti-alias screen space aliasing.

However, can't filter inputs to BRDF and not alter shape of a BRDF. With decoupled shading, if BRDF samples are aligned to shade samples, (e.g. exact point sampling), can preserve shape of the BRDF. Toksvig factor works quite well if need to handle MIPs and don't want to precompute

Overview

- Main issue is always the same thing, we shade too much
- Is there a way to only shade what is visible? Must be fine grained enough to not overshade much
- Is there a way to shade only what is visible at the right MIP equivalent level(s)?
- Is there a way to use the visibility information to dynamically allocate per frame what memory is needed?

Underside of ships is shaded, even though not visible

Overview – The concept of a Shadel Chunk

- Lots of approaches to problems
  - Breaking objects apart
  - Allocating texture coordinates based on screen projection (defeats point of Decoupled Shading)
  - Shading triangles based on there visibility
- New approach
  - GPUs are fast. GPUS are programmable. Can’t we determine in a pixel shader what is visible at a fine grained level?
  - The core concept of our approach
    - Shadel (Shaded element)
    - Shadel Chunk (a small cluster of Shaded elements)
  - We shade our objects with shadels, then draw rasterize them with the shaded results
  - We always shade a chunk of shadels, not a single shadel

Shadels

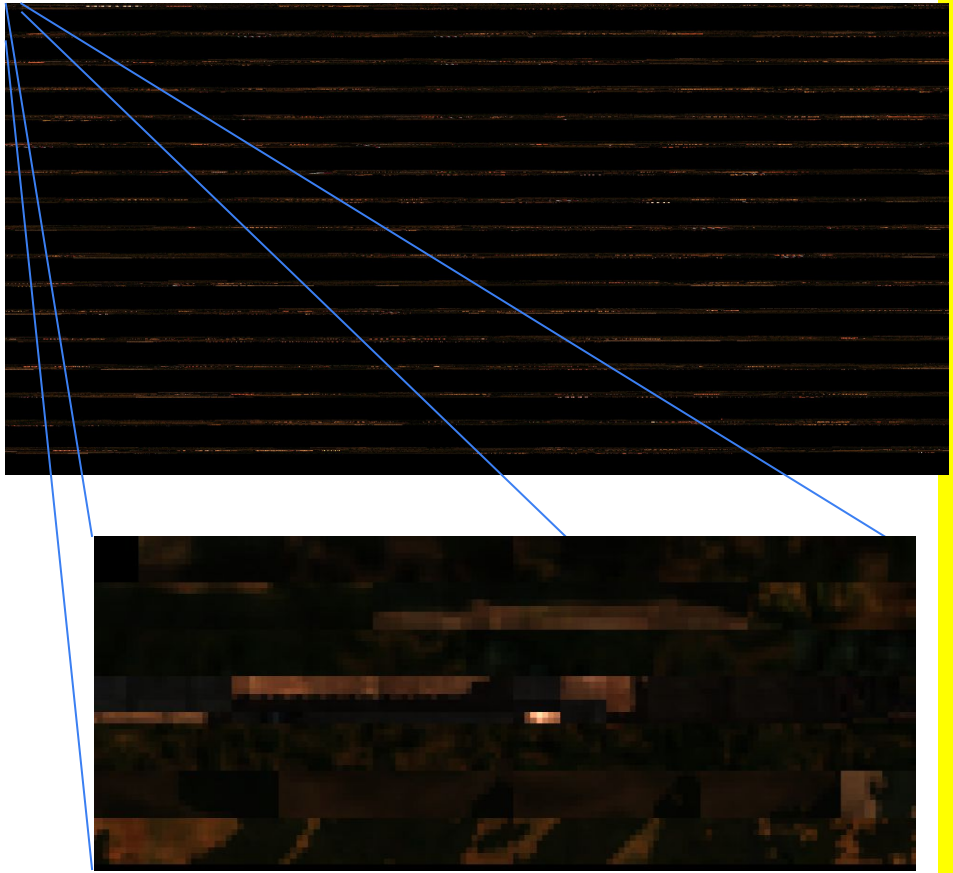
- How small is small? Our solution uses 8x8 chunks of shadels. 8x8 because
  - Small enough to not get significant overdraw
  - Big enough (64 samples) to run well in a hardware thread group, (typically 32)
  - Good memory compromise(ends up being ~1.5 bit per shadel)
  - Object has one set of texture coordinates just for shadels, similar to rendering to texture, but without a real texture to render to. This gives objects a mapping of shadels

Overall strategy

- Bind some UAVs to the pixel shader and use them to mark which shadel chunks are visible – fancy way of saying we are going to output data other than raster data from pixel shader
- Use the visibility info in the UAVs to allocate storage space in a custom virtualized space
- Use the visibility info to create work queues to shade the shadels
- Run materials with compute shaders to shade the shadels
  - In compute shader, determine attributes that would have come via geometry by using a triangle ID map and using the model's geometry
  - Can get derivatives in compute shaders now, yay!
- Raster the scene with simple shaders with everything already shaded
  - Can mix in forward rendering stuff at this time with ease

Virtualizing our shadels

- All objects, (even chunks of world), must have texture coordinates on them
- Create one very large master ‘shadel’ texture storage. By large, in our case it is 256kx256k, also sections of this will represent the MIP chain
- Actual storage is ~4kx4k, depending on settings
- When an object is potentially visible, CPU side max space estimator will allocate a chunk out of the large shadel chunk
  - Only has to be a very coarse allocation – resolution of alloc dependent on maximum possible ‘MIP’ map
  - No worries if it isn’t visible or over allocated, it will have minimal perf overhead
- When object is no longer possibly visible, remove from storage space

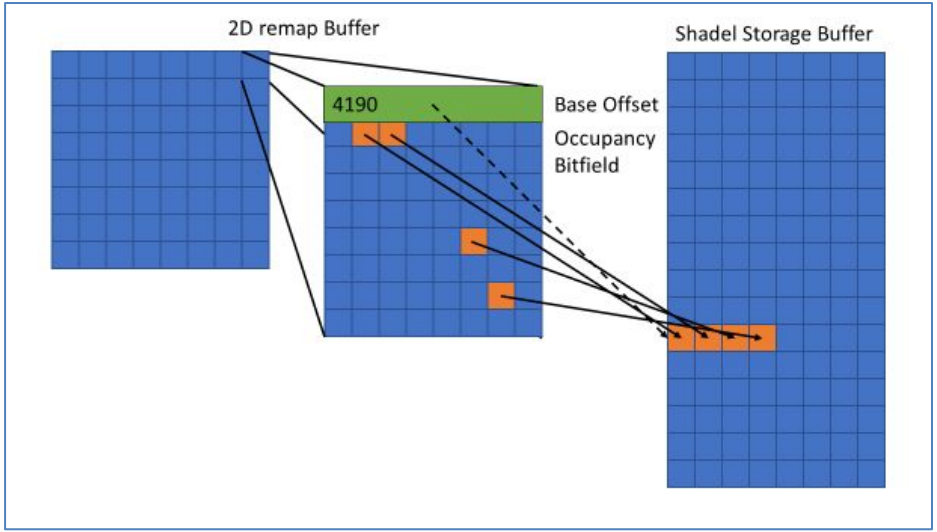


Our virtualized sheet is very big

- Virtualized sheet will always be mostly empty
- Only small sections of each MIP section will be shaded

Virtualizing our shadels

- Shadel chunks allocations into 8x8 chunks of 8x8 shadel chunks
  - Yes a chunk of chunks!
  - 64x64 shadels all together
- 2 textures
  - Offset texture – pointer into actual shadel storage
  - Occupancy texture, for each shadel chunk, 1 bit associated if shadel chunk is occupied or not
  - Counting bits lets us know the suboffset for a chunk of shadels
- Custom MIP calculation, detail levels sit in a specific part of the texture
  - Some complexity here when lower detail MIP levels become smaller then the block sizes. Solution is to snap allocations for smaller sizes



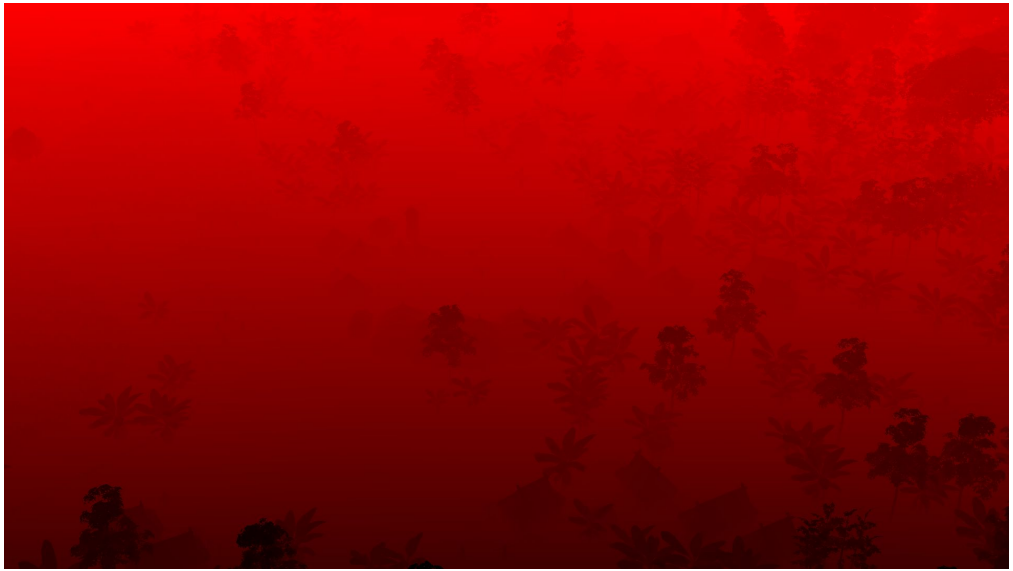
```
float2 BaseCoord = In.fTexPos * cast<float2>(Dynamics.Level0Dimensions.xy * Dynamics.BlockSize) / Dynamics.MipOffets[Dynamics.MipLevel].w;
int2 TexDims = cast<int2>(BaseCoord);
TexDims += cast<int2>(Dynamics.MipOffets[Dynamics.MipLevel].xy);

uint Level1Remap = RBatch0.Level0Map.Load(int3(TexDims, 0));
uint Bits0       = RBatch0.MarkTexture0.Load(int3(TexDims, 0));
uint Bits1       = RBatch0.MarkTexture1.Load(int3(TexDims, 0));

//Not mapped
if (Bits0 == 0 && Bits1 == 0)
    return float4(0, 0, 0, 1);

float2 MIPNTexCoords = In.fTexPos.xy *  cast<float2>(Dynamics.VirtualDims.xy) / pow(2, Dynamics.MipLevel);////(BaseCoord.xy *  (float)g_VirtualDims.xy) / pow(2, g_uMipLevel);
uint2  SubOffset = cast<uint2>(frac(MIPNTexCoords / (BlockSizeLevel0 * Dynamics.Level1Dimensions.zz)) * cast<float>(BlockSizeLevel0));

bool bSectionMapped = false;
int BitIndex = cast<int>(SubOffset.x + (SubOffset.y) * BlockSizeLevel0);

if( BitIndex < 32 && Bits0 & ( 1U << BitIndex))
    bSectionMapped = true;
if (BitIndex >= 32 && Bits1 & (1U << (BitIndex-32)))
    bSectionMapped = true;
if(!bSectionMapped)
    return float4(0, 0, 0, 1);

//find the sub sample location in the cell
uint uCellIndex = SubOffset.x + SubOffset.y *   BlockSizeLevel0;

uint uCount = 0;
if(uCellIndex < 32)
{
    uint uBitPattern =  (1U << uCellIndex) - 1;
    uCount       = countbits(uBitPattern & Bits0);
}
else
{
    uint uBitPattern = ( 1U << (uCellIndex-32))- 1;
    uCount           = countbits(uBitPattern & Bits1) + countbits(Bits0);
}

uint uSampleBlock = Level1Remap + uCount;

uint SampleChunkDimension = Dynamics.SampleDimensions.x / Dynamics.Level1Dimensions.z;
uint2 SampleBlockBaseIndex = uint2(uSampleBlock % SampleChunkDimension, uSampleBlock / SampleChunkDimension) * Dynamics.Level1Dimensions.zz;

uint2 SampleOffset = cast<uint2>(frac(MIPNTexCoords / Dynamics.Level1Dimensions.zz) * cast<float2>(Dynamics.Level1Dimensions.zz));
uint2 SampleIndex = SampleBlockBaseIndex + SampleOffset;

return RBatch0.ShadeStorage.Load(int3(cast<int2>(SampleIndex),0));
```

Implementation

- Z Prepass
- Shadel Marking pass
- Work Aggregation
- Shading
  - Materials have multiple layers
  - Happens in Compute Shaders
  - Heavy use of DispatchIndirect
  - Processed on shadel chunks
- Final Rasterization

Z-Prepass

- Strictly speaking, optional
- More impact than a typical z-prepass
  - Because over shaded objects will involve more than just performance, will allocate extra memory



Shadel Marking pass

- A chunk of shadel chunks is an 8x8 grid of 8x8 chunks, so 64x64 total shadels
- With this arrangement, the 8x8 grid is a single 64 bit value that can double as a visibility field
- First, clear the visibility field/shadel allocation field.
  - Biggest performance issue when having an overly large virtualized space
- When the pixel shader runs during the marking pass, it writes no output to color buffer (which is not even bound), instead it manipulates the visibility field/allocation by using atomic or operators
  - Faster to not do atomic operation or operation with an if statement if the input is 0 (seems like this could be a hardware optimization…)
  - Need to use the [earlydepthstencil] attribute!
- Note - Parallel to the offset texture, there is an allocID texture – a hash/ID of which object the allocation belongs to, not used during this pass

Shadel allocation pass

- Occurs entirely on GPU
- Allocation is normally serial, to get some parallelization, perform N allocations in parallel to N different section of the virtualized space
- Frames will have different allocation even if nothing changes, not deterministic

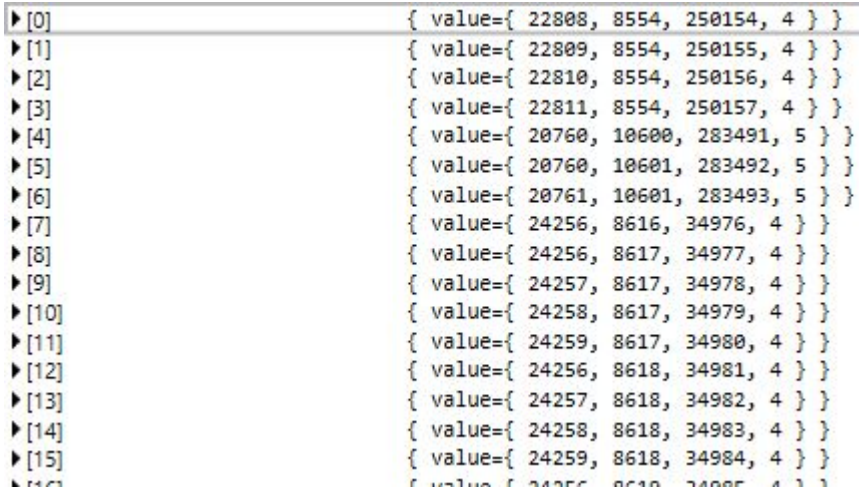
```
//Counters 0 to eAllocCounters - 1 are shade counters
void AllocateLevels(uint3 DTid : SV_DISPATCHTHREADID)
{
    uint2 BlockLocation = DTid.xy;
    uint Bits0 = RBatch0.MarkTexture0.Load(uint3(BlockLocation, 0));
    uint Bits1 = RBatch0.MarkTexture1.Load(uint3(BlockLocation, 0));

    if (Bits0 || Bits1)
    {
        uint ShadeBlocks = countbits(Bits0) + countbits(Bits1);

        uint uShadeBase;
        uint iCounterIndex = (BlockLocation.x + (BlockLocation.y*Dynamics.VirtualDims.x)) % ZNOSchemas.DSE.eDCEEnums.eAllocCounters;
        InterlockedAdd(inout UAV.AllocCounters.Lookup(iCounterIndex), ShadeBlocks, out uShadeBase);
        UAV.Level0Map.Store(BlockLocation, uShadeBase + iCounterIndex * Dynamics.AllocsPerCounterLevel1 + 1);
    }
}
```

Building work queues

- For each shadel chunk marked, need to process the material for it
  - Part of the rational for being 8x8 chunks to be a good size for a compute shader to process
- Each object rendered has N blank ComputeIndirect calls
  - Why N? Because our materials can have multiple layers (discussed later)
- References One buffer which contains the header for the computeindirect, for any object which has been instanced (even if it wasn't requested to render for the frame)

What is in work queues

- A work queue contains a list of shadel chunks to process
  - Shadel offset
  - Object ID, MIP map used
  - Number of shadel chunks that are visible



Building work queues

- Count the amount of work for every allocated object
- Allocate space for the work queues
  - Similar alloc problem to shadel space, do in parallel
- Create the work queue and put a reference to it in the indirect reference buffer
- Blank commands can be generated. An object can be submitted but turns out to have no coverage, therefore can be DispatchIndirect with 0 work items in it, generated by scanning the region of the virtual texture for marked bits
- Future work – be able to create different dispatch indirects for each layer of material

Allocation Pass/Collection

- Iterate through the occupancy texture, using the allocation information for each object rendering
- Counting bits, each bit represents a shadel chunk
- To get some parallel execution, we break the shadel storage into 16 parts, using a hash function to pick one of the sections to allocate. This prevents serialization on the atomic increments
- Along side allocation, add an entry into shadel chunk execution buffer containing
  - Shadel offset
  - Object ID, MIP map used
  - Number of shadel chunks that are visible
- Allocation stats uploaded back to CPU, used to do global shading adjustment if we get close to our maximum shadel storage. Sort of a more detailed occlusion query

Allocation Pass/Collection


```
//-------------------------------------------------------------------------------------------------
//Now store the work items into the queue and setup the params for the DispatchIndirect call
void CreateWorkQueue(uint3  DTid : SV_DISPATCHTHREADID)
{
    uint2 BlockLocation = DTid.xy;
    uint uAllocID   = RBatch0.ShaderAllocIDLevel0.Load(uint3(BlockLocation, 0)).x;
    uint uShadeBase = UAV.Level0Map.Load(BlockLocation);

    uint Bits0 = RBatch0.MarkTexture0.Load(uint3(BlockLocation, 0));
    uint Bits1 = RBatch0.MarkTexture1.Load(uint3(BlockLocation, 0));

    if(!Bits0 && !Bits1)
        return;

    uint uPlace = uShade
    uint uCounterPlace;

    {
    [nobound]
    for(uint y = 0;y < B
    {
        [nobound]
        for(uint x = 0;x <
        {
            uint uBitToTest = 1U << (x + y*BlockSizeLevel0);
            if(uBitToTest & Bits0)
            {
                InterlockedAdd(inout UAV.ShadeWorkCounts.Lookup(uAllocID.x), 1u, out uCounterPlace);
                uint WorkQueueOffset = UAV.ShadeWorkOffsets.Load(uAllocID.x) + uCounterPlace;

                uint iMIP = 0;
                [nobound]
                for (; iMIP < 16; iMIP++)
                {
                    if (all(BlockLocation.xy >= cast<uint2>(Dynamics.MipOffsets[iMIP].xy)) && all(BlockLocation.xy < cast<uint2>((Dynamics.MipOffsets[iMIP].xy + Dynamics.MipOffsets[iMIP].zz))))
                        break;
                }

                //Store the block location (as x,y pair of block of texels in virtual texture), the location of the shade store samples, and the MIP level being read
                UAV.WorkQueue.Store(WorkQueueOffset, uint4(BlockLocation * BlockSizeLevel0 + uint2(x,y) , uPlace, iMIP));
                uPlace++;
            }
        }
    }
```

Happens in parallel across many compute cores, the order is not the same for a work entry frame to frame

Get the object ID of whatever is occupying this part of the Shadel Storage Texture

64 bits spread across 2 textures, a bit set means that shadel chunk is active and we have allocated space

Running out of space…

- 1 Pixel means 1 shadel right? No…
  - We process MIP levels, so 1.25 no matter what (unless brilinear!)
  - Shadel chunks are 8x8, so some wastage
  - Anisotropic filters- heavy edge on objects need more samples. Better quality, but needs more space
  - And Transparency? Did we say transparency? Works just fine, but will need more space
  - ~ Rule of them 2-3X shadels to pixels w/o transparency, 3-4x if there is some transparency
- But still no guarantee we don’t use too many! All sorts of corruption if we do

Biasing all MIP levels

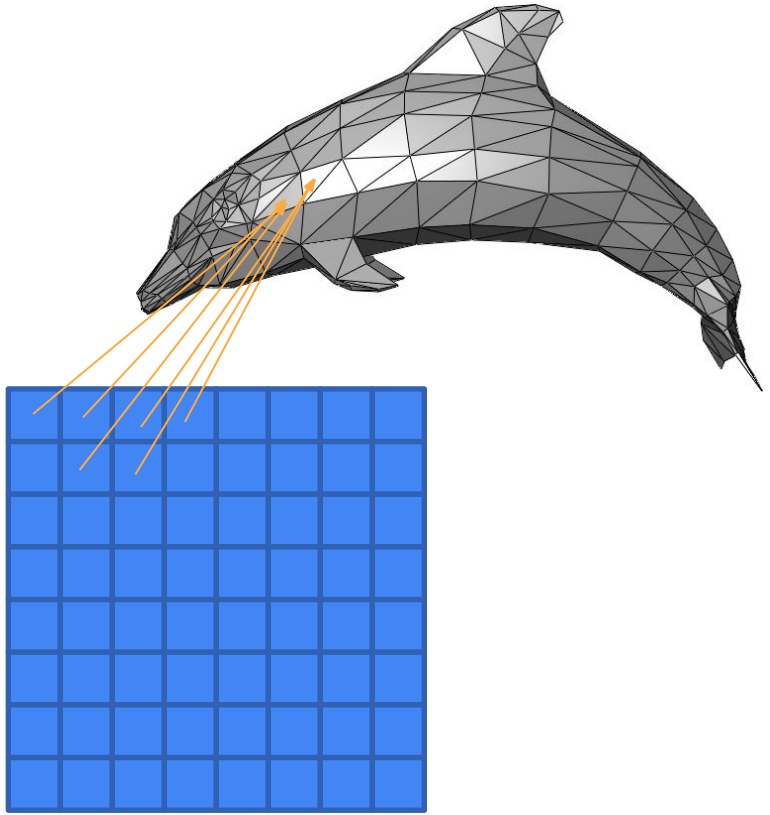
- Similar to foveated render, can bias the amount of shadels we process
- Simple concept – it's effectively a MIP bias, so can use fractional values to great effect (e.g. can undershade by .25 or so) – could hook into DOF to get more perf
- Each frame, upload our count buffer back to the CPU. CPU now has perfect statistics of last frames usage, down to what sections of the MIP were processed for any object
  - Lots could be done with this for content streaming… and we do none of that yet…
- The engine has thresholds that cause a bias to happen across all objects, then decrease shading bias if we go back to under capacity
  - In rare cases – a 'spike' is still possible for a frame but usually caused by overall aggressive shading settings, if bias relaxation is slow not been a problem

ID map pre-processing

- Pixel shaders as input take a barycentric attributes from triangles from mesh
- In decoupled shading, invoked via compute shader. No direct correlation to geometry
- For Ashes, captured relevant attributes like positions into texture
- Lots of issues, mainly this took too much memory

ID map pre-processing

- Attribute maps – rather than store the input data for a shaded sample, only store the triangle ID from the mesh
- Compute shader can now load the vertices and perform a similar interpolation as a pixel shader might have done
- Only extra data required is an ID map, a 16 bit uint
- Requires quite a bit of memory, largest drawback for this method – likely there exist good compression for this
- Index buffer of mesh and geometry attributes bound into compute shader as a buffer



How to generate ID maps?

- Raster the scene from the perspective of the texture coordinates
  - Texture space renders could invoke a pixel shader every frame , what Nitrous 1.0 did, but no fine grain culling, other issue
- Then do some small flood expansion and/or shader math to make sure we get a valid sample, ok to have a sample that falls outside of triangle withs some appropriate cipping
- But… edge cases! Literally… Some triangles might end up with no pixel coverage. Can end up with no samples or seams

Conservative rasterization

- Some technique render at really high res,
  - But no guarantee here
- Our solution, render the triangle ID map twice
- First pass renders it normally
- Second pass renders with conservative rasterization
- The results of the 2 passes are then merged
  - If a sample is uninitialized in the first pass, then the conservative rast pass is examined and if a non initialized value is there then this value is used
- This ensures that ALL triangles which will be shaded will always have at least 1 triangle ID location

Geometry Attributes


```
uint Index0 = TriangleIndicesBuffer.Load(TriIndex * 3);
uint Index1 = TriangleIndicesBuffer.Load(TriIndex * 3 + 1);
uint Index2 = TriangleIndicesBuffer.Load(TriIndex * 3 + 2);


float2 ShadeTexCoord0 = ShadeCoordsBuffer.Load(Index0).xy;
float2 ShadeTexCoord1 = ShadeCoordsBuffer.Load(Index1).xy;
float2 ShadeTexCoord2 = ShadeCoordsBuffer.Load(Index2).xy;


float3 BaryBase, Baryddx, Baryddy;
BaryBase = ComputeBarycentric(SourceCoord, ShadeTexCoord0, ShadeTexCoord1, ShadeTexCoord2 );
Baryddx  = ComputeBarycentric(SourceCoord + DevX, ShadeTexCoord0, ShadeTexCoord1, ShadeTexCoord2 );
Baryddy  = ComputeBarycentric(SourceCoord + DevY, ShadeTexCoord0, ShadeTexCoord1, ShadeTexCoord2 );
Out.BaryBase = BaryBase;
Out.Baryddx = Baryddx;
Out.Baryddy = Baryddy;
Out.VertexIndex0 = Index0;
Out.VertexIndex1 = Index1;
Out.VertexIndex2 = Index2;

float4 Position0 = PositionBuffer.Load(Index0);
float4 Position1 = PositionBuffer.Load(Index1);
float4 Position2 = PositionBuffer.Load(Index2);

Out.Position    = Position0 * BaryBase.x + Position1 * BaryBase.y + Position2 * BaryBase.z;
Out.PositionDDX = -Out.Position.xyz + Position0.xyz * Baryddx.x + Position1.xyz * Baryddx.y + Position2.xyz * Baryddx.z;
Out.PositionDDY = -Out.Position.xyz + Position0.xyz * Baryddy.x + Position1.xyz * Baryddy.y + Position2.xyz * Baryddy.z;
```

Geometry Attributes


```
float3 ComputeBarycentric(float2 p, float2 a, float2 b, float2 c)
{
    float3 BaryCoords;
    float2 v0 = b - a;
    float2 v1 = c - a;
    float2 v2 = p - a;
    float den = v0.x * v1.y - v1.x * v0.y;
    {
        float fEpsilon = 1.0f;//.2f;
        BaryCoords.y = clamp( (v2.x * v1.y - v1.x * v2.y) / den, - fEpsilon, 1 + fEpsilon);
        BaryCoords.z = clamp( (v0.x * v2.y - v2.x * v0.y) / den, - fEpsilon, 1 + fEpsilon);
        BaryCoords.x = 1.0f - BaryCoords.y - BaryCoords.z;
    }

    return BaryCoords;
}
```

Clamping epsilon is important, if we clamp to barycentric cords within triangle (e.g. 0 to 1) will create edge artifacts as samples will naturally fall outside of range. Very similar to centroid/not centroid with MSAA enabled. Indeed, this is related to small triangles that have minimal projection and ‘shimmer’ under normal rendering.

Material System

- OMG, what have we done? We've invented 2 new shader stages!
- Marking pixel shader
  - Just used to allocate shade space
- Shadel Shader, (Shadelader?)
- Very specific shader code preamble for a layer shader
  - Find section of virtualized storage to store/load
  - Interpolate mesh geometry
- A lot of boilerplate setup and other integration to our system

```
92
93      InstanceConsts := ZNOSchemas.TerrainSystem.Dynamics;
94      Textures       := ZNOSchemas.TerrainSystem.Textures;
95      VertexData     := ZNOSchemas.ShaderSpec.NoVertexData;
96
97      DirectRenderVS := TerrainVS.TerrainDirectRenderVS;
98      DirectRenderPS := TerrainDirectRenderPS;
99      Layers         := { TerrainMaterial.LayerNewSample,
00                          TerrainMaterial.LayerComputeNormal,
01                          TerrainMaterial.LayerShade };
02
```

Oxide's shading language

- Enter ZNO, e.g. Zinc Oxide
- Oxide's full shading language replacement
  - Handles all shading syntax
  - Handles shader specialization, ala a 'collection' concept similar to hygienic macros
  - Some general cleanup of HLSL Syntax
  - Handles shader constant frequency, data binding, etc
  - Added features like 'function types', 'function pointers'
  - Fast compiled C++ Data structures
  - A lot more… really an all day talk by itself
- ZNO has a material program, which has several types of shader stages
  - Vertex Shader/PixelShader pair
  - Layer Shadel Shader
- Layer Shadel shader stage is a virtualized shader stage, all setup code already been generated, mostly looks like a pixel shader to a tech artist
- New derivative functions in SM 6.4 allow us to simulate texture fetches in the same way
- But a ton of interesting new features – compute shader so you can get local shared memory and other crazy things are possible.

Oxide's shading language

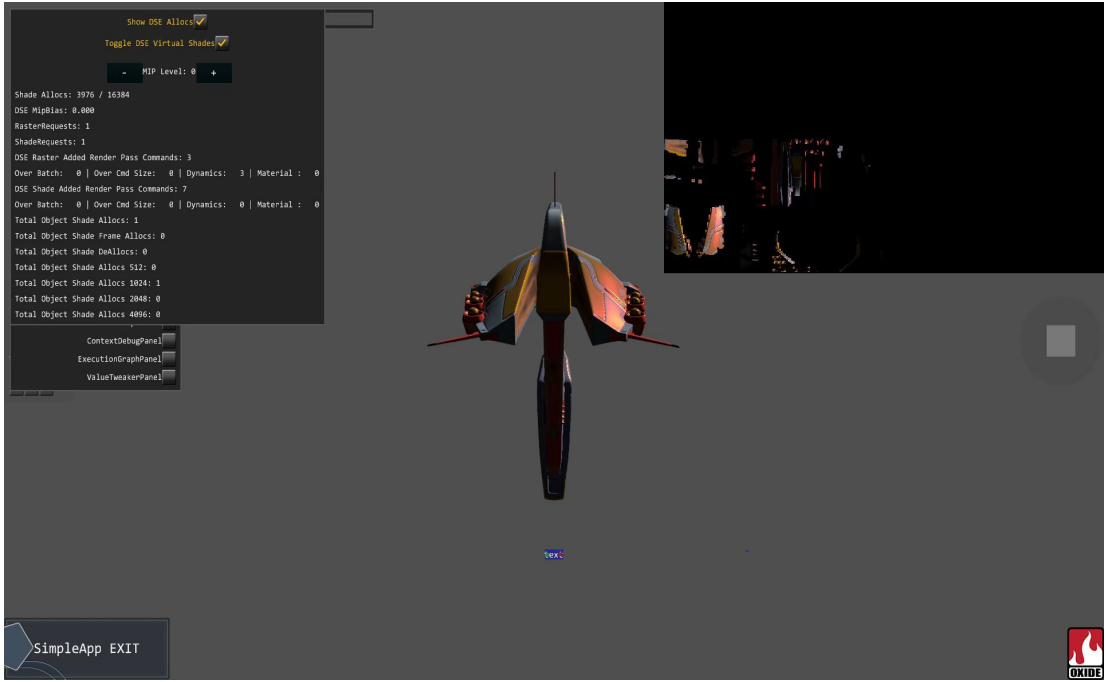
- Layers
  - Nitrous does not have just one place to store shaded values, instead it is configurable
  - Ara is set up with 5 layers of different bit depths and different number of channels
  - Similar to deferred renderer… a little bit, but any layer can hold any data, entirely at the discretion of the material
- Some of the layers can be marked as preserved,
  - At the expense of memory, can preserve calculated values from previous frame
  - Function in ZNO language called GetIndexPrevSampleVirtualCoords, returns the sample from previous frame if it exists, false if it doesn't
  - 100% accurate, not a guess, if it was computed last frame its there
  - Mostly used for caching invariant things for now, but obviously has big implications for future work

Scene Render

- This is the simple part – objects have now been shaded, with there shading samples sitting in the shadel storage space with the virtualized buffers pointing to them
- Rerun the sample shader but this time actually pull the samples out of the shading – reconstruction shader
- Run any forward shaders and forward objects
- No hard rule on what runs forward/decoupled. Object materials can blend between forward and decoupled elements as desired. Each section is just a shader stage

Demo

- Single asset from Ashes using decoupled
- Useful to visualize the process
- Green/blue are visualizing marking buffers, green means any bit is hit, blue means a section is entirely visible
- Different sections of the model are visible on different MIP levels



Basic hand-wavy cost analysis

- Full performance comparison between decoupled and other rendering techniques is prohibitive, since need different materials and optimizations to get valid comparison
  - Small demo/limited scenes are not useful or indicative of real world situations
- However, because decoupled is a superset of forward, can get a solid estimate of performance cost from a roughly similar implementation if using forward only
- Overhead is about 2 ms, around 10%
- Additional shadel processing appears to be offset by more efficient processing during shading

*Taken on a 1080Ti, on a Radeon 6800 timings were similar except shading was twice as fast

Operation	Time(ms)	Decoupled specific
Clear DSE Mark Buffer	.25	Y
Terrain Overlay	.5	N
Shadow Maps	1	N
Rasterization Z prepass	.75	N
Rasterization Shadel Marking	1	Y
DSE work dispatch	.75	Y
Shading	10*	N
Rasterization Final Scene	1.25	N
Rasterization Foilage	4	N
Volumetric effects	1	N
UI	1	N
Post Process	1	N

But things aren't so rosy…

- Decoupled Shading gets too expensive for micro-geometry that is not well charted (e.g. micro charted)
- Small triangles might have 1 sample, but rounded up to 128 (2 MIPs, 64x64). Also fixed overhead for sample
- ~4x more expensive
- Individual samples vary frame to frame so hard to temporally reuse
- Currently, small objects fall back to a direct only render
- Does look slightly better, decoupled shading making an effort to clean up samples, but not worth the perf

Possible Solutions

- For small charts,shade one sample (or perhaps 4 to get derivatives)
- Use a hybrid shader – shader can blend forward/decoupled at a per pixel level
- Really an LOD problem – can fix the LODs, problem would be mitigated.
- Not generally true that forward is faster, however. For some shaders, decoupled outperforms if it takes advantage of cached intermediate values. So… it depends…

Conclusion, Decouple Shading works

- Decoupled Shading works in the general case, on current hardware
- Minimal overhead (~10%)
- Limited memory overhead
- Compatible with ‘off-the-shelf’ assets and tooling
  - Ara has over 5000 assets, many from outsourcing studios
  - Over course of project, some assets such as leaders switched from forward to decoupled because desire to use advanced material features
- Some engine level book-keeping for virtualized shading space – 256x256k is a lot, but for bigger streaming world some management of active set is required, larger virtualized support possible with some more overhead