Eurographics Symposium on Rendering (2022)
A. Ghosh and L.-Y. Wei (Guest Editors)

Generalized Decoupled and Object Space Shading System

D. Baker and M. Jarzynski

Oxide Games, USA

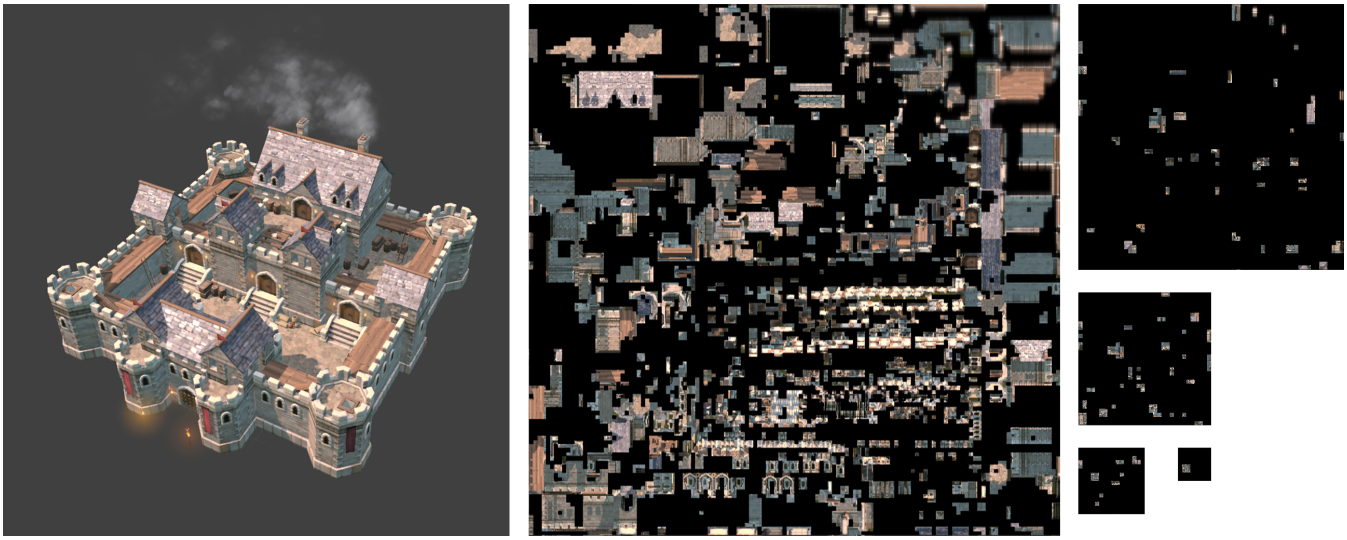

Figure 1: *Rendered castle model (left) and the associated virtualized shadel sheet (right) with level of detail system (similar to a mipmap).*

Abstract

We present a generalized decoupled and object space shading system. This system includes a new layer material system, a dynamic hierarchical sparse shade space allocation system, a GPU shade work dispatcher, and a multi-frame shade work distribution system. Together, these new systems create a generalized solution to decoupled shading, solving both the visibility problem, where shading samples which are not visible are shaded; the overshading and under shading problem, where parts of the scene are shaded at higher or lower number of samples than needed; and the shade allocation problem, where shade samples must be efficiently stored in GPU memory. The generalized decoupled shading system introduced shades and stores only the samples which are actually needed for the rendering a frame, with minimal overshading and no undershading. It does so with minimal overhead, and overall performance is competitive with other rendering techniques on a modern GPU.

CCS Concepts

• ***Computing methodologies*** → ***Rasterization;***

1. Introduction

Decoupled shading, also known as object space shading, is a method of rendering specifically designed to address well known weaknesses present in both forward and deferred renderers. Decoupled shading helps address mathematical stability problems caused by evaluating shade samples at different locations each frame, as well as the practical problems posed by compositing samples, fragment wastage and randomized memory access.

The advantages of decoupling shading are well known. Until recent advances in GPU architecture, decoupling shading remained impractical due to both performance consideration and lack of generalized GPU architectures. In 2016, the Nitrous game engine demonstrated that limited decoupled and object space rendering

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

was practical for D3D12 and Vulkan class desktop GPU. The first Nitrous title, Ashes of the Singularity, [Bak16] demonstrated that despite wastage in overshading and general overhead of decoupled shading, the increases in efficiency for shading in a decoupled manner overcame these issues.

However, Nitrous 1.0 and Ashes of the Singularity did not represent a generalized solution for object space and decoupled shading. Because the game is a top-down strategy game, the general problem of samples which were not visible being shaded was not overly severe, the projected texel density was relatively constant for most objects, and the terrain system was built with a limited variable rate stitch map with shading controlled by the CPU.

Furthermore, artists were able to tune art assets to circumvent object space and Nitrous 1.0 issues, with techniques such as not texture mapping the underside of units, or mapping it with limited texel density. This introduced additional cost and time to art assets relative to deferred and forward rendering architectures. Furthermore, decoupled and object space shading produced other problems; the need to overshade required substantial amount of shade buffer space, potentially more storage space even than a deferred render, though the total bandwidth used was typically far less.

Though this solution worked well for Ashes of the Singularity, it was not a generalized solution for decoupled and object space shading. The same rendering technique would perform suboptimally where there is both significant occlusion and large variation in shading samples across the domain of a single object.

In earlier versions of Nitrous, we attempted to solve the overshading and large texel variation by breaking up objects into smaller chunks. While this approach had merits and mitigated some of the aforementioned problems, it still required more memory and performance was less than was desirable. Additionally, the burden of building art assets broken into small chunks represented a nontrivial cost to production. Part of this cost was related to art assets needing to be built with a methodology which was considerably different from conventional art pipelines and tool support.

For Nitrous 2.0 we therefore desired, based on our experience and the experience of our partners, a solution with a number of constraints and goals:

- Ability to consume art assets with minimal variation from existing tools and art processes. Most art assets have a normal map, albedo map, a color map, and various attribute maps, sometimes stored in vertices. The only extra burden for creation of assets which is acceptable is the creation of a unique UV chart. We note that the creation of the UV chart can be automated in many cases.
- Ability to amortize shading (or parts of the shading) across multiple frames, decoupled temporally from rasterization, allowable on a material by material basis.
- Have near perfect shade sample and shading coverage. Pixels on the screen should be neither overshaded nor undershaded, nor should back facing or occluded shading samples necessarily be rendered.
- Maintain the filtering advantages of decoupled and object space shading. One advantage of object space shading is that most aliasing occurs only once during the capture of the shading attributes and subsequent shading occurs on these captured attribute buffers. This distinct advantage over other rendering technologies, such as forward, deferred, or even the Reyes architecture, is among one of the fundamental reasons to use decoupled shading.
- Efficiently store in GPU memory shade samples. Nitrous 1.0's shade packing scheme required significant wastage due to gutter spaces and other problems. Though the memory was never read or written to, it occupied valuable space in the high performance memory for the GPU.
- Competitive with other real time rendering architectures such as deferred, forward, and forward+ on a modern GPU. We define competitive to be within 10% performance of a similar scene implemented via forward or deferred.
- Allow arbitrary complex materials with relatively strong robustness. Materials should render with high quality without massive effort from shader authors. Capabilities such as complex material layering should be supported. Decoupled shading materials should be a super set of any material authorable in other rendering architectures.
- Be compatible with a variety of rendering technologies such as rasterization or ray-tracing.
- Have simple, easy to understand, and real time adjustable performance controls.

With these constraints in mind, we believe that the generalized decoupled shading system solves all these problems and can be used to render most types of scenes created for real time systems today.

2. Related Work

The Reyes Rendering Architecture [CCC87] implemented a technique called *object-based shading*. Reyes subdivides surfaces into micropolygons, that are approximately the size of a pixel, which are inserted into a jittered grid – a super-sampled z-buffer, to introduce noise and avoid aliasing. In object-based shading, shading happens before rasterization, as opposed to *forward shading* where shading takes place during rasterization in screen space, and *deferred shading* [DWS*88] takes place after rasterization. Since shading occurs before occlusion testing in the z-buffer, overdrawing can occur.

Burns et al. [BFM10] built upon the Reyes by making two major improvements: no longer requiring surfaces to be subdivided into micropolygon size and shades 2×2 blocks of pixels on demand after performing visibility testing, reducing the number of samples that are shaded but not used. Fascione, et al. [FHL*18] created Makuna, a modern implementation of Reyes. While not developed for real time rendering, it did make many improvements to Reyes including decoupling shading and path sampling.

Ragan-Kelley et al. [RLC*11] proposed a hardware extension based on decoupled sampling, sampling visibility and shading separately, and applying to depth of field and motion blur. Liktor and Dachsbacher [LD12] used a deferred shading system where shading samples are cached when computed, to speed up the rendering of stochastic supersampling, depth of field, and motion blur. Clarberg et al. [CTH*14] proposed hardware extensions for computing shading in texture space, reducing the overshading problem and allowed for bilinear filtering.

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

Andersson, et al. [AHTA14] used an approach called *texture-space shading* where each triangle is tested for culling and shades its surface, while a geometry shader computes each visible triangle's size to determine where in a structure similar to a mipmap to insert the triangle.

Hillesland and Yang [HY16] combined texture-space shading and caching concepts and used a compute shader in D3D11 to generate a mipmap-like structure of object-based shading results. Saving triangle IDs in a visibility buffer [BH13], so that the vertex attributes can be later accessed for interpolation.

Nitrous 1.0 and *Ashes of the Singularity* had a non-generalized solution for *decoupled shading* [Bak16]. Baker used *object-space shading* where objects can have any number of materials by utilizing masks. Several large master textures were allocated and an objects estimated area was assigned a proportion of the master texture, scaled to make all the shade requests fit. Texture-based shading was performed in a compute shader, where each material is accumulated in the assigned master texture. Mipmap levels where computed for the master textures. Finally, objects where rasterized with the master textures used to shade them.

Mueller, et al. [MVD*18] created a system to atlas a large scene, broken into parts, with CPU coarse culling, and streaming the shading at a lower FPS than the rasterization rate. We implemented a similar system, including streaming the shading with a 100ms simulated delay, and shared shading between viewports for VR. Unfortunately, our analysis showed memory requirements and shading visibility was far too coarse to meet our goals. Our conclusion at the time was that such an approach was not currently feasible for production. Thus, this effort was abandoned in favor of our current approach.

3. Architecture

The generalized decoupled shading engine consists of several complex sub-systems which interact to render a provided scene. Because of the general complexity and latency of CPU and GPU interaction, the systems and control flow remain largely on the GPU. The primary responsibility of the generalized decoupled shading engine is the allocation, generation, processing, and management of shade elements, which we call shadels.

The process works by running two primary loops which may run at different frequencies. For our upcoming title, Ara: History Untold, we always run them at the same temporal frequency. The first loop is the raster frame, which corresponds roughly to what is typically thought of in deferred or forward renderers' entire rendering pipeline. The second loop is the shade frame, which performs the actual shading. In the typical configuration, the shade frame runs at no more than 30 times per second, while the raster frame runs at a much higher rate, ideally as fast as the display device can display which may be upwards of 200 times per second.

The raster frame and the shade frame have very different computational needs. The raster frame involves mostly the use of rasterization hardware such as triangle rasterization and depth buffer, where the shade frame requires only the use of compute shaders. Because they can be run in parallel, in one configuration the shade frame can co-execute on a compute queue in the D3D12 or Vulkan API, running in parallel to the raster frame.

Figure 2: *A pre-alpha screenshot of our upcoming title: Ara: History Untold*

The raster frame process begins by collecting the scene for the GPU rendering. It then dispatches this scene the shadel mark prepass step. This step's purpose is not to rasterize, but to mark which shadels will be needed for the rasterization of the scene. Once this is complete, this data is sent to the shade frame loop.

The raster frame then may either proceed to rasterize the scene or wait for the shade frame to complete its work. If the raster frame proceeds to render, it may optionally compare its current shadels to the shadels which were marked in the shadel mark prepass. Shadels which are missing are shaded immediately, so that there will be no cracks, holes, or undersampling of the scene. Once the shadels are shaded, the scene can then be rasterized.

In the shade frame loop, the process begins by performing a layer space support expander to allocate shadels which are not directly visible but which may contribute to the scene in indirect ways. Once this is done, the shadels themselves are allocated. During the allocation, data is collected for the actual use of shadels and a global adjustment factor is computed to adjust the shading rate to fit into the desired shade space to not exceed the number of shadels allocated.

The next step is for the work queues to be generated, allocated, and filled. Shadels must be dispatched for every object, for each material instance on that object, and for each layer on that object. Once shadels have been processed, they are sent back to the render frame. Figure 3 illustrates these steps and the raster frame interaction with the shade frame.

3.1. Objects, Shade requests and Materials

In Nitrous 2.0, the following things which are rendered through the decoupled shading system are: objects which consist of triangles, a material instance which goes with the object, dynamic and constant data, which is generated by the application, and a group of arbitrary resources, usually collections of textures.

All objects together represent a scene and anything which may contribute to the final rendering of the screen should be in this

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

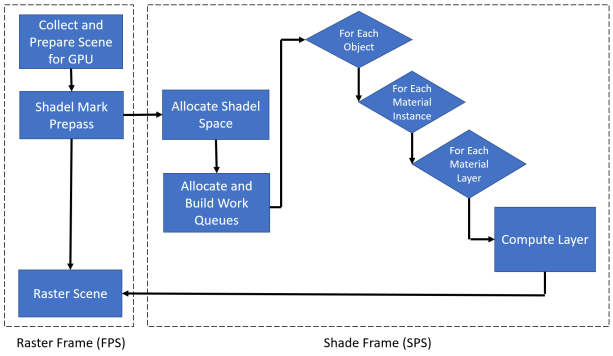

Figure 3: *The Raster and Shade frames. The raster frame collects the scene and prepares it for the GPU, then dispatches it to the shadel mark prepass. The shadel mark prepass marks which shadels need to be rasterized and sends this data to the shade frame loop. Once the shadels have been processed in the shade frame loop they are sent back to the render frame for rasterization.*

Figure 4: *A pre-alpha screenshot of Ara: History Untold. Everything in this screenshot are rendering through the decoupled shading system, except for the trees, people, and special effects such as fire.*

scene. This scene is created implicitly by tracking all objects which have been requested to be rendered. Scene culling and object refinement occur before the decoupled shading system begins its processing. Figure 4 is a scene from Ara: History Untold.

Objects have two distinct stages of existing. They may be instantiated, which means that they exist somewhere in the world and will have an object instance ID, and they may be also requested for rendering - which means they will be rendered into the scene if they are visible. Multiple objects may share material instances, and virtually any buffer or GPU resource with the exception that they will never share shadel space.

3.2. Attribute Processing

Triangulated meshes which we use typically have some attributes stored in the vertices including: normal, positions, texture coordinates, specular power, albedo, etc., which are interpolated across a triangle for each pixel. In the decoupled shading system and in object space rendering triangles are not used for shading, rather each shadel has a collection of corresponding input attributes.

In our first implementation, triangulated meshes were converted to have shadel input attribute textures. This process involved rendering the model from the 2D texture parameter space into a buffer (thereby capturing the rendered attributes) either repeating this process for each shadel level (analogous to a mip level) or performing a downsampling filter. In our Nitrous 2.0, rather than capturing each individual attribute, we follow the approach of Hillesland and Yang [HY16] and only capture the triangle ID. When the shading occurs, the mesh index buffer and vertex buffers are bound as inputs to the shadel shader. The triangle ID is used to look up the vertices, which are used to interpolate the input attributes from the vertex data in nearly the exactly same way as would occur during a forward or deferred renderer. This has the advantage of using far less data, and also kept attributes more consistent with any corresponding values that might come from rasterized buffers such as shadow maps.

To capture the triangle ID, we rasterize the triangles in texture space, with the output being only the triangle ID. The resolution of this texture is captured at is controlled by a setting for each individual asset. This is an asset cooking step and does not occur while the game is running. This triangle ID is captured for each mip level of the triangle ID texture. One problem that arose was that sometimes triangle or section of triangles on the mesh resulted in no triangle ID being captured, due to triangles falling in between coverage rules for rasterization. This can later result in geometry being rendered and having no shadels which represent it.

To solve these problems, we make some important modifications to other approaches to capture triangle IDs. After the attributes are first generated as previously described, we capture them again, but this time switch the rasterization mode to conservative rasterization [AA05][HAO05]. This changes the coverage rule such that all triangles will emit an attribute to any sample they touch. This process is repeated similarly to the aforementioned process.

Next, the two triangle ID maps are merged. Non-conservative rasterization is preferred, however if a sample exists in the conservative rasterization where no sample exists in the non-conservative version, the merged version uses the conservative rasterization sample. This process means that there is no chance that a triangle when applied to rasterization, does not have any captured triangle IDs. See Figure 5.

We have found that these improvements greatly increase the robustness of triangulated meshes such that in general the Nitrous engine can consume most assets created for forward or deferred renderers. In addition, by saving only the triangle ID, the only additional memory for an asset is the creation of a triangle ID texture.

3.3. Virtualized shadel allocation

When objects are instantiated and could possibly render (but may not be actually requested to render for a particular frame), they are allocated shade space inside the virtualized shade space system. We call them shadels as they are distinct from both texels and pixels, because they could be implemented in a variety of ways depending

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

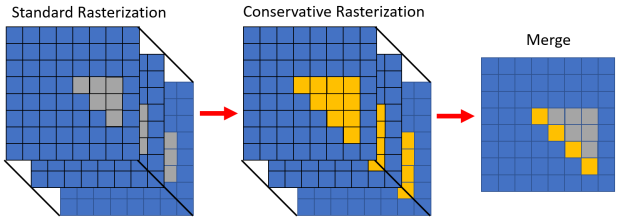

Figure 5: *Merging triangle ID maps between non-conservative and conservative rasterization. Preferring conservative rasterization where samples exist and falling back to non-conservative where conservative samples do not exist.*

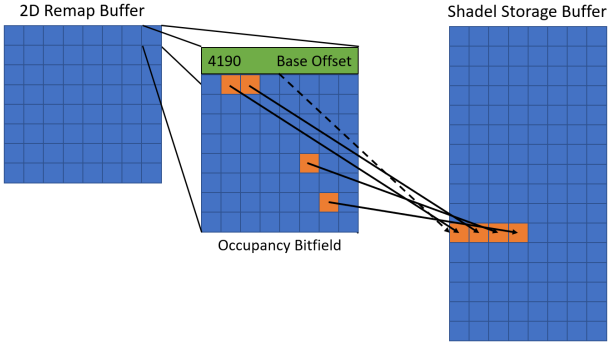

Figure 6: *Virtualized shadel allocation. 8×8 chunks of shadels from a 2D shadel remap buffer are stored in a shadel storage buffer.*

on the specific hardware being used. On some platforms, for example, it might be more efficient to store shadels in a typeless buffer rather than a texture. Currently, we use textures to store them but have experimented with other storage methods.

The shadels are stored inside a large virtualized 2D space, where virtualized 2D dimensions have been tested as large as 512K×512K. Each object is allocated with the maximum possible shadel density which could be needed for any subsection of the object, including detail which might be introduced via a material shade.

This massive virtual shadel sheet creates enough space such that objects that are instantiated in the scene need not bound their maximum scale other than a very coarse maximum. Though the virtualized texture is beyond the scope of what could practically be on a GPU now or in the near future, the working set of shade samples which could actually be touched is a tiny percent of this space.

The virtualized shadel sheet is created as a two level hierarchy, but implemented in such a way as to require only one level of dependent reads. The shadels themselves are stored in 2D blocks, which we implemented as 8×8 chunks of shadels. The virtualized shadel space consists of two memory buffers, implemented for convenience as two 2D textures: a 2D remap buffer and a shadel storage buffer. The remap buffer contain three values: a 32-bit shadel block start offset which marks the beginning index location of the shadel storage buffer, the object instance ID which represents which object the shadels belong to, and the occupancy field, a 64-bit value which represents 1 bit for each of the chunks of shadels if it is occupied or not occupied. Therefore, each entry in the 2D remap buffer represents 8×8 chunks of shadels, if a shadel chunk is also 8×8 then each entry in the remap buffer represents 64×64 contiguous shadels.

To address a particular shadel, the 2D shadel location is indexed to the remap buffer which corresponds to it, and the sub index of the shadel chunk is also calculated, which will corresponds to 1 of the 64 bits in the remap buffer. The address of the shadel chunk is then calculated by Formula 1. This gives the location of the shadel chunk, where each shadel can easily be sub-indexed. The shadel allocation ID number provides an index value which will be used for the dispatching of work by the GPU.

$$ShadelChunkStartIndex + countbits(\sim(ShadelSubIndex - 1)\ \&\ OccupancyField) \quad (1)$$

For more extreme scenes, we experimented with an additional level of indirection. In this scenario, we introduced a secondary remap buffer and perform a similar allocation step. This configuration introduces an additional dependent texture read, but increases practical virtualized shade space dimensions to the 4M×4M range at a small decrease in performance. Our experience was that this additional shade space was unnecessary. However, it is possible to use virtual resident textures to map pages in/out as necessary which should allow these same resolutions with no additional passes and minimal additional overhead.

The shadel storage buffer also has a level of detail system similar to a mipmap in a texture. However, even though the remap buffer may be stored in a 2D texture, the mip hardware and texture addressing capabilities are not used. This is due to unordered access view (UAV) mapping challenges and alignment requirements which require specific rules for mip block alignment. Specifically, because a shadel chunk is set to 8×8 and each entry the remap buffer therefore corresponds to 64 shadels, single allocations smaller than 64 shadels can't be made in the shade space, even for smaller mip levels.

Figure 6 shows an actual breakdown of the 2D remap buffer along with the shadel detail levels. Note that past the first few entries, each level of detail level takes the same amount of space. We determined that for our own use, we did not need to allocate shadel chunks smaller than 256 shadels in our shadel space. This observation meant that lower mip levels would all align to the same size, which effectively means that some occupancy bits can never be set on the edges of the lower mip level and some entries in the 2D shadel remap buffer are never written to nor read from.

3.4. Shadel Mark Prepass

The shadel remap buffer allows locations in the virtual shadel storage buffer to be physically mapped. Before the shadels can be mapped, the renderer must mark the actual shadels which are

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

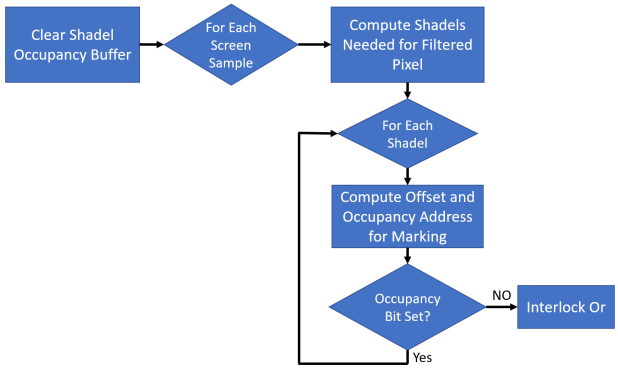

Figure 7: *The marking process where an occupancy bit is set using 'interlocked or' corresponding to a 8×8 chunk of shadels.*

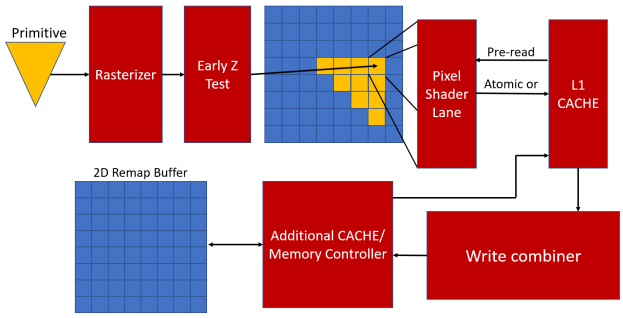

Figure 8: *Pipeline of our pixel shader. It takes advantage of the fixed function rasterization, atomic operators, early z depth, write combiners, and efficient caches.*

needed. This process is called the shadel mark prepass. After this prepass is complete, every shadel chunk which is needed for the rasterization will be marked.

This process starts by clearing all bits from the bitfield element of shadel remap buffer. If we chose to use a texture for the shadel remap buffer, this clear is essentially free on most GPUs. Next, we perform a proxy render of the scene. For Nitrous 2.0, this means running the corresponding shader stages which render the objects on the screen. This can include any type of shader, such as vertex shader, hull shader, domain shader, a mesh shader, etc. Since GPUs have complex and efficient pixel shader and rasterization hardware, we want to repurpose the hardware.

To repurpose the pixel shader, the goal is to use the pixel shader similar to a simple object rasterization. Each object has an associated start location and dimension entry given to it when it allocates space in the virtualized shadel storage buffer. Because each object has a unique UV chart, this chart is used to index shadels similarly to a texture.

Because it is not a texture, an equivalent trilinear or anisotropic filter need to be implemented manually. During the mark prepass, the actual shadels will not be loaded but rather the shadels which would be accessed during a filter need to be marked (8 in the case of trilinear). This approach works for any type of filtering desired, with old techniques such as bilinear able to give modest performance gains on lower end hardware.

The marking process, illustrated in Figure 7, consists of the setting an occupancy bit corresponding to a chunk of shadels (which we usually set to 8×8). Every chunk of shadels allocated in the virtualized shadel space has a single bit which corresponds to it. Setting a single bit could be problematic with hardware since the bit is co-located with other bits. We note that bits only need to be set, and only 1 bit would ever be set for each shadel with most shadels mapping to the same occupancy bit.

We use 'interlocked or' to set the bits for each shadel, with each pixel on screen possibly setting 8 bits. Since most pixels in a GPU are processed in the same unit as other nearby pixels, we take advantage of the write combiners on modern hardware such that the actual amount of writes to system memory are on the order of only 2 to 4 bits per shadel chunk. If the shadel chunk size is 8×8, a 4K screen writes only about 1 MB of actual data.

As fast as 'interlocked or' operations are, the overhead is still too high. Because all bits were initialized to 0 and only 1 bit at a time is set during each operation, a simple optimization is to read the bit field first, and only perform the interlocked operation if the bit needs to be set. This results in significant performance increases since the bitfields are so small as to usually be in the L1 or L2 cache on the GPU.

Another important performance optimization is to use the `earlydepthstencil` pixel shader attribute to allow the hardware to early exit work of shadels which cannot be visible. By using this attribute, only shadels which will actually end up contributing to the scene will be marked.

Figure 8 illustrates how we set up a pixel shader pipeline to take advantage of fixed function rasterization, atomic operators, early z depth, write combiners, and efficient caches. These aforementioned hardware units have already been implemented on modern GPU architectures and combined with the locality of pixel processing effectively repurpose the GPU to make the pixel shader efficient as marking the shadels with only one prepass of the geometry.

3.5. Shade Space Allocator

Once the shadel chunks are marked, the shadels must be allocated in the shadel storage buffer. This process is performed by examining each bitfield which corresponds up to 64 shadel chunks. The number of chunks needed for that shadel chunk cluster is the number of bits set in the bitfield.

This can be performed efficiently by first subdividing the shade texel storage buffer into N subdivisions. The chunks can then be allocated in parallel performing an atomic exchange increment on one of the subdivisions by using any number of simple hash functions to map a shadel group to a subdivision. The actual location of an individual shadel chunk is computed by examining the set bits of the shadel remap texture. Although this requires slightly more reading of data for addressing, we note that it reduces the opera-

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

Table 1: *Memory requirements to store shadels per resolution with 4x MSAA.*

Resolution	Back Buffer Size	Total Shadel Memory
1920×1080	16 MB	64 + 50 MB
2560×1600	32 MB	128 + 50 MB
3840×2160	64 MB	256 + 50 MB

tion to only one dependent read and the remap locations small size means they are typically inside the L1 cache of the GPU.

3.6. Global Shade Space Adjuster

A typical scene produces large numbers of shadel chunks to shade. For reasonable efficiently mapped objects, we have noted that as a rule of thumb we need twice the number of shadels as we do pixels in the screen resolution to be able to guarantee perfect shade coverage as defined by similar texel coverage in a deferred or forward renderer. This is given a certain amount of overlap due to transparency and alpha blending. We note that this comparison is somewhat unfair to decoupled shading since a given pixel on the screen has computed both it's maximum LOD and at least one LOD less to provide smooth shadel downfiltering. Thus, the number of samples used for a frame has a lower bound of about 25% more than a forward or deferred renderer.

Because scenes can vary in complexity, it is possible for a given scene to exceed the number of shaded samples in the shade allocation buffer. This would cause incorrect rendering as the shade samples have no memory to be stored. To mitigate this potential problem, the current number of shadels is continuously uploaded to the host CPU. As it approaches certain thresholds the system increases a global mip bias value, which can fractionally adjust itself to keep the number of shading samples to fit within the shade allocation buffer. Because very small amounts of mip biasing decrease the number of samples dramatically, we found that in the very rare event the shade space hit our 2x rule, the perceived quality difference was imperceptible as the fractional mip bias never exceeded the 0.1 mark.

Table 1 refers to the amount of memory needed to store the shadels. Experimentally, a virtualized 2D map of 256K×256K worked well for most scenes, so long as some effort was taken to bind maximum shadel density and reallocate it as objects move further and closer to the screen

3.7. Layer Shading

As part of the material system, Nitrous supports the concept of layered materials. Material layers exist as a series of layer images, with a layer program executing on each layer image. Each layer image is evaluated before the next layer can process, and layers can read and write from arbitrary previous layers.

This is useful for any operation which requires neighborhood information. For example, a height map can be converted into a normal map by a material creating a height map in one layer, and then computing a normal from that layer by processing the heights. In another example, a skin shader can use successive passes to perform a blur on the shaded layer. Both of these cases are used by materials in Ara: History Untold. Figure 9 is an example of the layering process.

Figure 9: *Pre-alpha render of Sappho, one of the leaders in Ara: History Untold. The skin shader uses multiple layers and successive passes.*

To support layers, generalized decoupled shading uses the same allocation location to correspond and to address multiple image layer sample planes, which can store these possible intermediate image layers. However, because a layer may need support for a kernel operation (e.g. a wider neighborhood to compute a normal), this could create problems where these samples were never evaluated because the shadel of that location would never be used for the final composition in this scene. In practice, however, we never found issues, the support needed for a kernel didn't exceed the naturally shown visible shadels. If this become a problem, a pass causing visibility expansion could be added.

Layer images can also store values across frames. In this use case, we buffer the remapping buffers for the previous frame and allow access to the previous layer images directly from any mate-

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

rial. Because any material can read and write to any layer image as it desires, it is up to the individual material for how it might reuse any intermediate value.

The retrieval of previous frames layer image samples works by making a function call in the material which uses the previous frame shadel's allocation and storage. If a sample exists, it is returned, otherwise, the function returns false so that the material can decided how to proceed. This feature is exploited extensively in our terrain material, where we cache the expensive blended attributes that need to be generated.

3.8. Work Dispatch Aggregator and Allocator

Once shadel storage has been allocated and the system knows which shadels need to be processed, it still needs to generate the actual GPU commands to process the shadels. In Nitrous 2.0, all shadels are processed via a compute shader, but shadels can be processed easily by any capable computing device.

A naïve method is to evaluate shade samples is to evaluate the entire shadel remap buffer for each object which has been submitted and for each layer. Early testing of this indicated that for small workloads, this is sufficient however the larger the virtualized shadel space, the more prohibitive this becomes since GPUs are efficient at terminating work early, this quickly becomes an $O(n^2)$ operation.

Instead, the entire shadel remap buffer is examined, and a work list is generated for each object which is to be rendered. This work list consists of the virtual remap location of each shadel chunk and the associated shadel detail level.

The work queue buffer is shared among all objects, and since allocated into segments depending on how many work items are needed for each object, each work item being effectively a shadel chunk. Some objects are not visible at all, due to occlusion, frustum or other conditions which might mean that their samples are not touched. These objects will have a work count of 0 and take no space in the work queue buffer.

When the work queue buffer is populated, a work dispatch parameter buffer is also populated with the start location of the work queue to read and the number of items which will require shading.

Each entry in the work dispatch parameter buffer is referenced via a dispatch indirect call, which corresponds to all the state required for the shading and processing of shadels for a particular object with one indirect dispatch per material layer per object.

3.9. Frame Render

The frame rasterization occurs as a similar process to the shadel mark prepass, however this time instead of marking the shadel chunks which will be used, the shadels are read from the shadel storage buffer using the already populated shadel remap buffer.

The frame render and shade mark preprocess should ideally run at the same resolution and exactly replicate one another to guarantee watertightness. Practically speaking if layer edge expansion is enabled and because shadels are marked in chunks, the shadel mark preprocess can be run at lower resolution often without noticeable issues. Problems can arise from certain types of charting on some high triangle count meshes, but can be mitigated by LOD choices and carefulness in art preprocess. In our current implementation, we chose to keep the resolutions the same, because we had a need to have a screen resolution Z prepass for other rendering needs.

Our data indicates, that the shadel mark preprocess step is often less than 1ms on modern GPUs even at high resolutions, therefore we typically run the process at the full resolution to guarantee watertightness with lower resolutions reserved for lower performance systems.

4. Results

4.1. Frame Analysis

One critical requirement for decoupled rendering is that to be usable, it must be both general purpose, and also be comparable to other rendering architectures in terms of performance and memory. This remains the key obstacle to using decoupled shading.

While the memory requirement is straightforward to determine, the performance comparison is difficult. To be useful, this metric should be applied to a full game scene. Unfortunately, it is not feasible to implement the scene of our current title with an entirely different rendering architecture (some of our materials cannot be used in a forward or deferred renderer), therefore we use some other analysis to indicate if we met our goal with being competitive with other rendering architectures.

Because our decoupled rendering architecture is fully compatible with forward rendering, there is some reasonable analysis which we can perform to get a good estimate of a direct performance comparison. By taking a frame trace of a typical scene, we can isolate the particular parts of the frame which are specific to decoupled shading and the parts which would be identical to or are similar to a forward renderer. The ratio of these two numbers is a reasonable estimate of the cost differential between the two rendering types, with some caveats to be discussed.

To perform this analysis, we created a typical scene in our game. We ran this scene on an Nvidia GeForce 1080 Ti at 1080p, with 4x MSAA, and it had an average frame time of around 24ms. This scene has 8 million triangles, 6500 draw calls, 6GB of GPU Memory, 8MB CPU to GPU data upload per frame, and around a 5% variance of frame time.

One important note about the scene, is that our foliage does not run through decoupled shading. This is because the material for our foliage is a simple, mostly lambertian material, meaning that there is limited visual benefit for decoupled shading. Our foliage is almost entirely fillrate/front end bound, and uses very little shader compute resources.

In the timing for the scene, the total time directly required for decoupled shading is about 2ms. This represents around 10% of the total frame time. The shading represents around 40% of the frame. At first glance, this would appear to be directly related to decoupled shading, however, in a forward renderer this shading would occur during the rasterization pass, so this represents a transfer of work rather than overhead due to decoupled shading.

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

Table 2: *A frame analysis of a typical frame in our Decoupled Shading Engine (DSE) from Ara: History Untold, which featured 8 million triangles and 6,500 draw calls.*

Operation	Description	Timing (ms)	DSE Only
DSE Mark buffer clear	DSE Setup	0.25	Y
Terrain Overlay	Terrain UI	0.5	N
Shadow maps	Shadow Depth Raster	1	N
Rasterization	Z Prepass	0.75	N
Rasterization	DSE Shade Marking	1	Y
DSE Work Dispatch	Control logic for DSE	0.75	Y
Shading	Material Shading	10	N
Rasterization	Final Scene Composite	1.25	N
Rasterization	Foilage	4	N
Volumue Rendering	Volumetric Effects	1	N
UI	UI Effects	1	N
DOF	Post Process	0.75	N
Color Curve	Post Process	0.25	N

The major difficulty with the shading section is that the number of samples evaluated from a forward renderer and a decoupled is not the same. A decoupled renderer will typically process more samples, though this is not always the case since small triangles will over shade in a forward renderer due to the fact pixels must evaluate on a quad. Whereas in a decoupled renderer such as ours, the shading granularity is higher and the renderer takes more samples on anisotropic edges.

However, we can use a couple of observations to estimate they are likely approximately the same. We have also observed higher GPU occupancy shading in decoupled shading than with forward rendering due to better cache use and dispatching, with occupancy often being near perfect if GPU register usage isn't too high. Therefore, while decoupled shading is likely processing more samples, it can evaluate samples somewhat faster. Additionally, we observe that 30-50% of a frame time spent shading is typical for many games, based a variety of frame captures we have both down on our own prior titles and the types of captures we have seen from other games and engines. Therefore, 40% of our time spent in shading is roughly in-line with our expectations.

We also ran timings on a similar scene on an AMD Radeon RX 6800 GPU. According to UserBench, this GPU is nearly identical performance to the Nvidia GeForce GTX 1080 Ti. However, we noted that while most of the timings were nearly identical to the 1080 Ti as expected, the shading time was less than half at around 4ms, with the entire frame time around 18ms. In this GPU, shading would represent only around 25% at most of the frame time. We are still investigating the reason for the performance difference. It could be due to a newer, more efficient architecture for compute shading, or perhaps some missed optimizations in the Nvidia Driver. On this GPU architecture, it seems unlikely that any forward rendering of our scene could be significantly faster than our current approach.

There are a few more considerations worth discussing that would further benefit decoupled shading. One is that due to intrinsic temporal and shader anti-aliasing, our renderer does not require temporal anti-aliasing. The second observation is that a detailed analysis revealed that there is likely a large performance gain simultaneously rendering foliage with shading. This is because foliage is opaque and using alpha to coverage, it may render before anything which needs shading. Since the foliage uses little compute resources and mostly uses rasterization resources, on an architecture which allows asynchronous compute it should be possible to co-execute direct rendered opaque objects while shading is occurring. This could yield a performance increase of 2-4ms.

Overall, our analysis indicates that performance should be withing 10% of other rendering architectures, and if other mentioned factors are considered it may be able to outperform other rendering architectures in some cases.

4.2. Quality Feature Improvements

We have already noted that decoupled shading is competitive with other rendering architectures, but performance is only one factor to consider when building a rendering system.

Regardless of performance advantages, decoupled shading has intrinsic quality improvements. One of the biggest problems with shading techniques is shading aliasing and shading instability to interpolation of non-linear data. A common example of this is shimmering results when a high frequency normal map is used with a low roughness factor. This results in high frequency highlights which shimmer frame to frame as the micro variations in the samples can cause widely different shading results.

There are many techniques designed to help mitigate this problem such as LEAN Mapping [OB10], but there exists no generalized solution to this problem for forward or deferred, nor can there be since real world materials are often too complex to fully solve these issues. Part of our design requirements was shading robustness, where any shader will have some degree of anti-aliasing regardless of the attention spent to anti-alias it.

While decoupled shading still aliases, the samples used per

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

frame are invariant. This is distinct even from architectures such as Reyes where triangles are generated relative to the current frame. The advantage of this is that while decoupled shading may be incorrect, it is incorrect in the same way each frame thereby removing one class of rendering artifacts.

Another feature decoupled shading brings is the aforementioned concept of material layers. Simply put, shadels can have precise access to their complete neighborhood, as well as precise access to previous temporal samples. Additionally, similar to forward rendering, decoupled shading has no real limit to the number of different material and material instances which can be used.

Finally, it is a trivial matter in decoupled shading to adjust the sampling rate for specific objects, areas of objects, materials, etc. Shading sampling and super sampling is controlled by a single number in the shader. Foveated rendering can be implemented with a handful of shader code. Performance can also be easily adjusted by varying total shading samples across the scene independent of resolution.

5. Further Work

Decoupled Shading can integrate with ray tracing hardware in a variety of ways. To ray trace inside a material, the scene is also updated and maintained in one or more bounding volume hierarchies (BVH), as is typical for real time ray tracing. At this point, any shadel can request a ray trace in the same manner as a pixel shader could, allowing full integration with ray tracing.

Ray tracing can be integrated more deeply into a decoupled shading system. If various surface properties are collected into different layers, traced rays can look up their values into the populated shadel remap and storage buffer, marking the shadels in the remap buffer so that they become available in future frames.

Additionally, decoupled shading allows for additional interesting modes of operation. Rather than trace rays directly in the material, the ray origin and direction can be stored into one or more layers. This layer is dispatched to ray tracing hardware which populates another layer with the results of the ray trace shader.

By dispatching large clusters of rays at once, the decoupled shading engine can sort and group the rays for a much faster trace through the scene, avoiding costly shading during the hit shaders.

6. Conclusion

The described process produces scenes which render quickly and efficiently on modern hardware, despite the fact that the hardware tested was not designed for this type of rendering. This rendering architecture can replace many forward rendering architectures for a production title at scale, with approximately the same performance characteristics, and is competitive in overall performance with other rendering techniques. Our current title has over 5,000 assets, ranging from characters, to terrain, to buildings compatible with this decoupled shading architecture.

In addition, this rendering architecture can co-exist with forward rendering, and is generally a superset. Many of our materials now rely on some of these additional features, especially materials with multi-pass layers, such as when normals are generated from composited heights and human skin.

The primary hardware features which make this feasible for efficient rendering are decent speed interlocked bitwise atomics, good L1 caches, and good write-combiners. On newer GPU architectures such as RDNA 2 the rendering is particularly efficient, with shading accounting for only 25% of a typical frame time.

The estimated cost over a forward renderer of similar complexity is around 10%, with the hope that further optimizations will make the performance nearly the same or better. The primary overhead for decoupled shading is twofold. First is we require an additional prepass on the geometry of the scene, and the second is that there is some amount of fixed overhead aggregating the results of this prepass to dynamically dispatch GPU work.

The Nitrous 2.0 rendering system provides the first, comprehensive, and complete decoupled rendering solution. Decoupled shading is practical and efficient on modern GPUs. Decoupled shading can be used with many rendering techniques including, ray tracing, point rendering, triangle rasterization, tile rendering, MSAA, alpha blending, and many other GPU features. Additionally, decoupled shading has intrinsic quality and flexibility benefits that cannot be matched by forward or deferred rendering architectures [Bak16]. Finally, we believe with additional GPU modifications, decoupled shading could exceed forward and deferred rendering in almost all work loads

7. Acknowledgements

Ara: History Untold is published by Xbox Game Studios.

References

[AA05] AKENINE-MÖLLER, TOMAS and AILA, TIMO. "Conservative and Tiled Rasterization Using a Modified Triangle Set-Up". *J. Graphics Tools* 10 (Jan. 2005), 1–8. DOI: 10.1080/2151237X.2005.10129198 4.

[AHTA14] ANDERSSON, M., HASSELGREN, J., TOTH, R., and AKENINE-MÖLLER, T. "Adaptive Texture Space Shading for Stochastic Rendering". *Comput. Graph. Forum* 33.2 (May 2014), 341–350. ISSN: 0167-7055. DOI: 10.1111/cgf.12303 3.

[Bak16] BAKER, DAN. "Object Space Lighting". Mar. 2016. URL: https://www.gdcvault.com/play/1023511/Advanced-Graphics-Techniques-Tutorial-Day 2, 3, 10.

[BFM10] BURNS, CHRISTOPHER, FATAHALIAN, KAYVON, and MARK, WILLIAM. "A Lazy Object-Space Shading Architecture With Decoupled Sampling". Jan. 2010, 19–28 2.

[BH13] BURNS, CHRISTOPHER A. and HUNT, WARREN A. "The Visibility Buffer: A Cache-Friendly Approach to Deferred Shading". *Journal of Computer Graphics Techniques (JCGT)* 2.2 (Aug. 2013), 55–69. ISSN: 2331-7418. URL: http://jcgt.org/published/0002/02/04/ 3.

[CCC87] COOK, ROBERT L., CARPENTER, LOREN, and CATMULL, EDWIN. "The Reyes Image Rendering Architecture". *SIGGRAPH Comput. Graph.* 21.4 (Aug. 1987), 95–102. ISSN: 0097-8930. DOI: 10.1145/37402.37414 2.

[CTH*14] CLARBERG, PETRIK, TOTH, ROBERT, HASSELGREN, JON, et al. "AMFS: Adaptive Multi-Frequency Shading for Future Graphics Processors". *ACM Trans. Graph.* 33.4 (July 2014). ISSN: 0730-0301. DOI: 10.1145/2601097.2601214 2.

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

[DWS*88] DEERING, MICHAEL, WINNER, STEPHANIE, SCHEDIWY, BIC, et al. "The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics". *SIGGRAPH Comput. Graph.* 22.4 (June 1988), 21–30. ISSN: 0097-8930. DOI: 10.1145/378456.378468 2.

[FHL*18] FASCIONE, LUCA, HANIKA, JOHANNES, LEONE, MARK, et al. "Manuka: A Batch-Shading Architecture for Spectral Path Tracing in Movie Production". *ACM Trans. Graph.* 37.3 (Aug. 2018). ISSN: 0730-0301. DOI: 10.1145/3182161. URL: https://doi.org/10.1145/3182161 2.

[HAO05] HASSELGREN, JON, AKENINE-MÖLLER, TOMAS, and OHLSSON, L. "Conservative Rasterization". *GPU Gems 2*. Ed. by PHARR, MATT. 2005, 677–690. URL: https://developer.nvidia.com/gpugems/GPUGems2/gpugems2_chapter42.html 4.

[HY16] HILLESLAND, K. E. and YANG, J. C. "Texel Shading". *Proceedings of the 37th Annual Conference of the European Association for Computer Graphics: Short Papers*. EG '16. Lisbon, Portugal: Eurographics Association, 2016, 73–76 3, 4.

[LD12] LIKTOR, GÁBOR and DACHSBACHER, CARSTEN. "Decoupled Deferred Shading for Hardware Rasterization". *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '12. Costa Mesa, California: Association for Computing Machinery, 2012, 143–150. ISBN: 9781450311946. DOI: 10.1145/2159616.2159640 2.

[MVD*18] MUELLER, JOERG H., VOGLREITER, PHILIP, DOKTER, MARK, et al. "Shading Atlas Streaming". *ACM Trans. Graph.* 37.6 (Dec. 2018). ISSN: 0730-0301. DOI: 10.1145/3272127.3275087. URL: https://doi.org/10.1145/3272127.3275087 3.

[OB10] OLANO, MARC and BAKER, DAN. "LEAN Mapping". *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '10. Washington, D.C.: Association for Computing Machinery, 2010, 181–188. ISBN: 9781605589398. DOI: 10.1145/1730804.1730834 9.

[RLC*11] RAGAN-KELLEY, JONATHAN, LEHTINEN, JAAKKO, CHEN, JIAWEN, et al. "Decoupled Sampling for Graphics Pipelines". *ACM Trans. Graph.* 30.3 (May 2011). ISSN: 0730-0301. DOI: 10.1145/1966394.1966396 2.

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.