Generating Geometry Silhouettes With Mesh Shaders For Use in Stencil Shadows

Ravi Bhatt

University of Michigan CS LSA Undergraduate Thesis
September 2023

1 Abstract

Stencil shadows remain one of the viable realtime shadow algorithms which preserve surface detail. However, existing approaches for generating the mesh silhouette polygons necessary for computing shadow areas suffer from scalability issues because they rely on non-optimal GPU features like geometry shaders, or are forced to use excessive memory when leveraging compute shaders. We present a proof-of-concept method for generating shadow volumes in real-time rasterization environments by taking advantage of *Mesh Shaders* in modern graphics APIs.

2 Motivation

Lighting and shadows are considered among the most important aspects of look development for communicating the appearance and feel of a scene, but realtime shadow rendering remains a difficult and computationally-expensive part of the rendering process. As of today, game developers mainly employ two methods: depth mapping and path tracing. Depth mapping is the faster of the two and involves rendering a scene multiple times from the perspective of each light to a texture, then sampling from that texture to determine what pixels are in shadow, and for transparent objects, the shadow color. While reasonably efficient, this method scales poorly with large numbers of lights and is limited by texture resolution and texture memory access speeds, resulting in low-quality shadows and loss of detail. Ray tracing is an alternative method which involves simulating photon interactions within a scene. While it is more accurate than depth mapping, and can account for bounce lighting and refractions, it is not

practical to implement in real time aside from environments with access to the most advanced and most recent desktop graphics hardware.

This project focuses on a less popular method of shadow rendering: depth stencil shadows, also known as shadow volumes. This method was first introduced in 1977 by Franklin Crow [1]. In real time implementations of shadow volumes, the render engine generates a polygon which represents the region of the scene blocked by the casting object with respect to a given light, and then uses stencil texture operations to determine which pixels on surfaces are in shadow. It has the benefit of producing pixel-perfect shadows, and scales better towards higher light counts than depth mapping. However, most implementations require geometry shaders (a feature typically not present on mobile) and do not handle transparency or edge softness. This technique gained popularity in the early 2000s but fell out of use in favor of depth mapping, and later, ray tracing.

Many graphics API vendors have recently introduced a new shader stage named *mesh shaders* to address the limitations of geometry and tessellation shaders [13]. Mesh shaders replace the vertex processing stages with two new stages: Object (also called Amplification or Task) and Mesh. The object stage determines the number of mesh invocations and their parameter data, and the mesh shader takes that data and emits meshlets, which are groups of primitives. In contrast, vertex shaders lack programmable dispatch, and each vertex invocation can only process one vertex in a vacuum. In 2022, Apple added mesh shaders to Metal 3, making this API the only implementation of mesh shaders on mobile devices with broad support, at present.

3 Context and Past Work

At present, the most popular method of rendering shadows in real-time applications is depth mapping. The steps of this method for a deferred-lighting [2] render engine are listed in Algorithm 1. Depth mapping has a few advantages. First, it is simple to implement. Second, it is trivial to add transparent shadow casters, by including a color attachment to the depth-only pass that includes the color information for the scene. However, it also has many disadvantages. Its quality is limited by the resolution of the offscreen textures. If the resolution is too low, smaller geometry details are lost. Increasing the resolution also increases the VRAM allocation requirements, so properly capturing all geometry details becomes quite difficult. In addition, floating point precision loss in the render target textures causes a phenomenon where speckles appear on surfaces due to self-intersections (colloquially known as “shadow acne”). The issue can be reduced in a couple different ways. One method involves adding a bias to the sampling, effectively elevating all samples off their surfaces by a small amount. Another method inverts the culling mode when executing the depth-only pass, eliminating acne on surfaces facing the

light, but reintroducing it on oblique and narrow shadowed surfaces. Too much bias causes a different phenomenon for both artifact reduction strategies known as *peter-panning*, where shadows appear detached from their casters. Getting shadow maps to look correct requires extensive artist involvement.

Algorithm 1 - Depth Map Shadows

1. For each light l in scene:
    a. Let g be the depth geometry buffer from the deferred data collection pass.
    b. Render a depth-only pass to an offscreen texture t from the perspective of l
    c. Render an additive-blending color pass from the camera's perspective. For each fragment f in the render target:
        i. Transform f into the space of l
        ii. Sample depth value d_l from t
        iii. Sample depth value d_g from the depth geometry buffer
        iv. If $d_g < d_l$, execute the lighting calculation, otherwise, return vector 0

In addition to the quality issues with depth mapping, there are a myriad of performance issues. Shadow map passes are difficult to batch, resulting in expensive state changes per light due to the need to switch between shader programs multiple times. This issue can be alleviated by allocating more shadow map textures, at the downside of increased VRAM requirements.

The shortcomings of shadow mapping pose serious issues to look-development in real time applications. There have been attempts to address these shortcomings, with a variety of these already shipping in market-leading products. One example, named *Virtual Shadow Maps*, addresses the resolution limitation by emulating an infinite-size shadow map on a sparse texture through a paging system. The mechanism bears similarity to how modern operating systems provide programs with the entire address space while having less physical memory present. The shadow system then renders tiles to these sparse textures which are loaded in as-needed, allowing for virtually infinite resolution at the cost of extra processing time. Unreal Engine 5 uses virtual shadow maps [3] on directional lights.

Other methods try to mask the resolution issues, for example *Percentage-Closer Filtering* (PCF), which involves sampling multiple pixels around the target pixel and calculating a coverage percentage instead of a binary covered/not-covered value, creating a feathered edge at shadow borders. Unfortunately, PCF incurs a higher bandwidth cost through its extra sampling, which is mitigated somewhat on some GPUs with hardware support for PCF [7] though hardware PCF is usually not configurable. In addition, PCF loses detail, which negatively affects small geometry such as facial features.

Cascaded Shadow Mapping attempts to make better use of shadow map resolution by allocating multiple light target depth textures, one at full resolution, one at half, one at one-fourth, and so on. The render engine then renders to each target depth texture, zooming out the projection matrix each time, so that objects closest to the camera consume the highest resolution depth texture while further away objects use the lower resolution textures. The result is better quality shadows up close, at a cost of worse shadows for objects further away. Cascade Shadow Map systems are currently shipping in both Unity [5] and Unreal [6].

Shadow Volumes are a markedly different approach than shadow maps. Instead of rendering the geometry to offscreen lightmap textures, the renderer analyzes the geometry directly to determine the regions of the scene, or volumes, that are not accessible to the light. The stencil shadow method determines the silhouette of every object, then extends these silhouettes along the light vector and renders them to a depth-stencil only pass with face culling disabled and a specific set of stencil write operations enabled. When the rasterizer draws a front face, it *increment*-wraps that pixel in the stencil texture by one. When the rasterizer draws a back face, it *decrement*-wraps the pixel in the stencil texture by one. Then during rendering the light pass, the system applies a stencil test that rejects every pixel with a stencil value other than 0. The benefits of this method are (1) no need to allocate additional framebuffers, (2) it can provide pixel-perfect shadow edges, and (3) it extensively leverages fixed-function GPU hardware rather than running in software in a shader.

Unfortunately, shadow volumes are not without weakness. Before GPGPU, the render engine needed to calculate the volumes on the CPU, uploading the new mesh buffers to the GPU every time the volumes changed, which could be every frame depending on how the scene changes. The host-side software approach is also memory-inefficient, because a copy of all shadow caster geometry must reside in system memory. In GPU-driven implementations, the engine either uses geometry shaders or a compute shader to generate the volumes. Even on platforms with hardware support for geometry shaders, hardware vendors discourage their use because they are slow [4] compared to compute shaders.

The specific problem with stencil shadows this paper aims to address is that calculating the volumes on the host CPU or in geometry shaders is an expensive operation. Even when geometry shaders are supported, they incur significant overhead, and when using a compute shader, one must allocate potentially large buffers in VRAM to store intermediate results. This paper aims to implement a proof-of-concept geometry volume generation system using mesh shaders.

Geometry shaders [8] are an optional shader stage used for *mesh amplification* that are passed a primitive (a vertex, line strip, or line loop) and can return zero or more new primitives up to a user-specified limit. Newly-generated primitives are treated as part of the input mesh since geometry shaders run as a stage along the traditional rasterization pipeline. The application has relatively little control over geometry shader dispatch, so geometry shaders can also be

instanced, meaning run multiple times on the same primitive. Pseudocode for a geometry shader is provided in Code Block 1.

Code Block 1 - Example Geometry Shader

```
On input vertex v, uniform l, output primitive line_strip:
    if dot(v, l) < 0:
        return;
    EmitVertex(v + 1)
    EmitVertex(v + 2)
```

In Code Block 1, mesh amplification has occurred because the number of output vertices is greater than the number of input vertices. Note that this shader can terminate early resulting in zero generated primitives, so geometry shaders can also be used for vertex culling.

In contrast to geometry shaders, Mesh shaders share many similarities with the Map-Reduce algorithm [9]. Execution is divided into two phases, known as Object and Mesh, with the programmer providing one shader for each phase. The Object shader decides what to draw, and the Mesh shader decides how to draw it. The Object shader receives arbitrary data from the application and decides the number of Mesh shader dispatches to generate and their thread-group size, along with the data to provide to each invocation in a fixed-size struct known as the *object payload*. The Mesh shader reads its parameter data from the object payload and produces triangle information in another fixed-size struct known as the *mesh payload,* which is passed directly to the rasterizer.

The traditional raster pipeline guarantees that triangles will appear on-screen as though they were rendered in monotonically increasing index order. In practice, geometry shaders are quite slow due to the synchronization points and excessive memory writes they cause in order to satisfy this ordering guarantee, because additional vertices are treated as part of the original input geometry. Unlike geometry shaders, mesh shaders only guarantee ordering within the results of the output of a given mesh dispatch [10], therefore, they do not have the same memory overhead or stalling due to synchronization.

A pseudocode implementation of a naïve hair renderer with the traditional vertex-fragment and compute shader pipelines is shown in Algorithm 2.

Algorithm 2 - Example Compute Shader Hair Renderer

1. Render the source mesh using its vertex and fragment shaders
2. Allocate a VRAM buffer containing the worst case number of generated hair Bézier curves for every generated hair.
3. Allocate a VRAM buffer containing the worst case number of triangles needed to store every generated hair.
4. DispatchCompute, given a list of coarse hair curves, where each thread writes to a given slot in the hair curve buffer:
    a. Determine if the given hair should exist. If not, exit (or return)bail, and write NAN into the assigned VRAM buffer slot.
    b. Write Bézier curve points for the generated hair in the assigned VRAM buffer slot.
5. DispatchCompute, given the generated hair buffer, where each thread writes into a given slot in the hair triangle buffer:.
    a. If the data is NAN, return. Fill the buffer assignment slot with NAN.
    b. Generate triangles for the given Bézier curve.
    c. Write NAN to any extra cells in the buffer assignment, because NAN vertices will be culled by the rasterizer.
6. DrawPrimitives, given the hair triangle buffer

Algorithm 3 shows the corresponding pseudocode implementation of the renderer with mesh shaders. Compared to Algorithm 2, Algorithm 3 gives us additional flexibility and a reduction of unnecessary reads and writes in the mesh stage. Unlike with the compute shader approach, allocations of large buffers as seen in steps 2 and 3 of Algorithm 2 are not required. One can easily alter the mesh before drawing it, as shown in step 2-b-ii of Algorithm 3. By culling invisible meshlets before rendering them (Step 2-a of Algorithm 3), one can reduce the workload for the rasterizer, increasing efficiency. The mesh implementation can run in O(k) memory, where k is the number of concurrent mesh dispatches, whereas the compute implementation runs in O(n) memory, where n is the size of the input data. The graphics driver is also free to schedule the dispatch of each object-mesh pair as is most efficient in the given moment. The compute shader implementation can be made more efficient by creating smaller buffers and doing a larger number of total dispatches, however, such implementations end up approaching an emulation of what mesh shaders already provide.

Algorithm 3 - Example Mesh Shader Hair Renderer

1. Offline, generate meshlets of the source mesh. A meshlet is a small section of the source mesh whose data is contained in a contiguous memory block, and each block is the same size. Meshlet blocks are auxiliary buffers that reference the original vertex and index buffers.
2. DispatchMeshThreadgroups on the number of meshlets
    a. Object stage:
        i. Determine the orientation of the meshlet. If all triangles are facing away from the camera, exit.
        ii. Call dispatchMesh, passing the meshlet buffer references via the *object payload*
    b. Mesh stage
        i. Retrieve the mesh data using the stage parameter data
        ii. Assemble the meshlet into a vertex and index buffer inside the *mesh payload* by reading the auxiliary buffer. One can run a polygon reduction or augmentation algorithm at this stage if a lower or higher resolution mesh is desired.
        iii. Inform the API of the number of triangles written.
    c. Fragment stage
        i. This stage is the same as with the traditional pipeline.
3. DispatchMeshThreadgroups on the number of meshlets (this time for *hair*), passing in Bézier curve information encoding hair direction for each meshlet.
    a. Object stage
        i. Determine if the meshlet hair is visible. One can use the depth buffer from the previous dispatch and a meshlet bounding box to coarsely determine this. If it is occluded, exit.
        ii. Determine the LOD for the given meshlet. One can use distance to the camera as the deciding factor.
        iii. Based on the LOD, fill a buffer within the *object payload* containing Bézier curve points for each hair to be placed on the source meshlet. Call dispatchMesh, passing in the Bézier curve data
    b. Mesh stage
        i. Given the Bézier curve data, for each curve, generate triangle data for each hair and write to vertex and index buffers in the *mesh payload*.
        ii. Inform the API of the number of triangles written.
    c. Fragment stage
        i. This stage is the same as with the traditional pipeline.

4 Implementation

Our implementation is divided into three stages, one offline during mesh loading, and two online during rendering. At mesh load, along with creating the vertex and index buffers, the implementation creates a third buffer, known as the *edge buffer*. This buffer contains the vertices that compose each edge, along with the normal vectors of the triangles that share the edge. The purpose of the edge buffer is to make determining the mesh silhouette on the GPU efficient.

The second two stages, the Object and Mesh stages, run during the lighting pass after the renderer writes to the deferred render targets. The lighting pass uses Algorithm 4.

Algorithm 4 - CPU-Side Mesh Shader Dispatch

1. Let L = the set of unique lights in the scene
2. Let M = the set of all unique meshes in the scene
3. For $l \in L$:
    a. For $m = \{m_{vert}, m_{ind}, m_{edge}\} \in M$:
        i. $n_t = |m_{edge}|$
        ii. dispatchMeshThreadGroups with n_t threads on m
        iii. Execute fragment shader for l, using results of stencil buffer as shadow mask (a.k.a. the *fragment-stencil* stage)

As shown in Algorithm 5, given a mesh *m* and an edge, the Object stage dispatches the Mesh stage if the current edge is part of the silhouette, writing the relevant data into the *object payload*. The Object stage does not produce a complete list of edges, instead the graphics driver is free to dispatch the relevant Mesh shader with no constraints in ordering, because stencil shadows do not depend on the order in which the faces of a volume are rendered. In Algorithm 5, there is no *else* stage for the final *if* statement, because we do not want to include edges that are not part of the silhouette. If the casting mesh is such that no silhouette edges exist, then no Mesh dispatches occur and no shadow is rendered.

Algorithm 5 - Object Shader Stage

1. Let E = the set of all Edges $\in m$
2. Let V = the set of all Vertices $\in m$
3. For $(v_1, v_2), (n_1, n_2) \in E$:
    a. Where n_1, n_2 are the normal vectors of the triangles sharing v_1, v_2 respectively
    b. Let vectors t_1, t_2 be the vectors from v_1, v_2 to l, with respect to the ordering.
    c. Transform v_1, v_2 to world space by applying the model matrix
    d. Let $d_1 = n_1 \bullet t_1$
    e. Let $d_2 = n_2 \bullet t_2$
    f. if $\text{sign}(d_1) \neq \text{sign}(d_2)$
        i. Let $r = n_1$ if $d_1 > d_2$ else n_2
            1. r is the *reference normal*, which is the surface normal of the triangle in shadow
        ii. Execute Mesh Stage on (v_1, v_2, r)

The Mesh stage applies Algorithm 6 on the output from the Object stage and l from the lighting pass. The use of the reference normal in Algorithm 6 allows the input mesh to have arbitrary tessellation, not limited to a triangle strip, in contrast to many previous implementations of stencil shadows. Instead of requiring a specific layout in memory for the winding order, having a reference normal allows for the computation of the correct triangle orientation on the fly. This is an important advantage because it allows the implementation to handle meshes exported from artist content creation tools which encode geometry as triangle lists. In contrast, implementations that require triangle strips only work correctly on meshes whose data has been carefully arranged in memory, often manually because the optimal triangle strip algorithm is NP-Complete [14].

The *fragment-stencil* stage implements the Z-Pass stencil technique [12]. After this stage completes, all pixels with a stencil value other than zero are not accessible to the current light. Subsequently, the lighting pass runs as usual, with the stencil mask set to only execute on pixels that have a stencil value of 0.

Because the graphics driver is responsible for scheduling the dispatch of each workgroup within a *drawMeshThreadgroups* call, and the memory that each dispatch uses is known, the implementation does not need to allocate a worst-case sized buffer, unlike with compute shader

implementations. Instead, the *payload* structures only need to have enough space for one generated plane.

Algorithm 6 - Mesh Shader Stage

1. Let t_1, t_2 = vectors from v_1, v_2 to l
2. Let k_{inf} be a sufficiently large real number for extending volume vectors
3. On input vertices (v_1, v_2) and reference normal r
    a. Create new position vectors
        i. $q_1 = v_1 + (\text{normalize}(-t_1) * k_{inf})$
        ii. $q_2 = v_2 + (\text{normalize}(-t_2) * k_{inf})$
    b. Determine the winding order of the new triangles using r
        i. Let $n' = (q_1 - v_1) \times (q_1 - v_2)$
        ii. Let $w =$ clockwise if $n' \bullet r > 0$ else counterclockwise
    c. Transform v_1, v_2, q_1, q_2 into clip space by applying the view and projection matrices.
    d. Create triangles x_1, x_2 from vertices (q_1, q_2, v_1) and (q_1, v_2, v_1), writing to the *mesh payload.*
        i. The winding order is set following the right-hand rule. Use w to determine the index order.

5 Results

Our implementation results in accurate pixel-perfect shadows by leveraging mesh shaders to generate geometry volumes. It provides higher shadow edge quality than shadow mapped implementations in existing products, such as Unity, can provide. It also offloads all of the shadow processing during frame generation to the GPU, freeing up the CPU for domain-specific tasks. In this section, we present figures and more detailed information regarding our results.

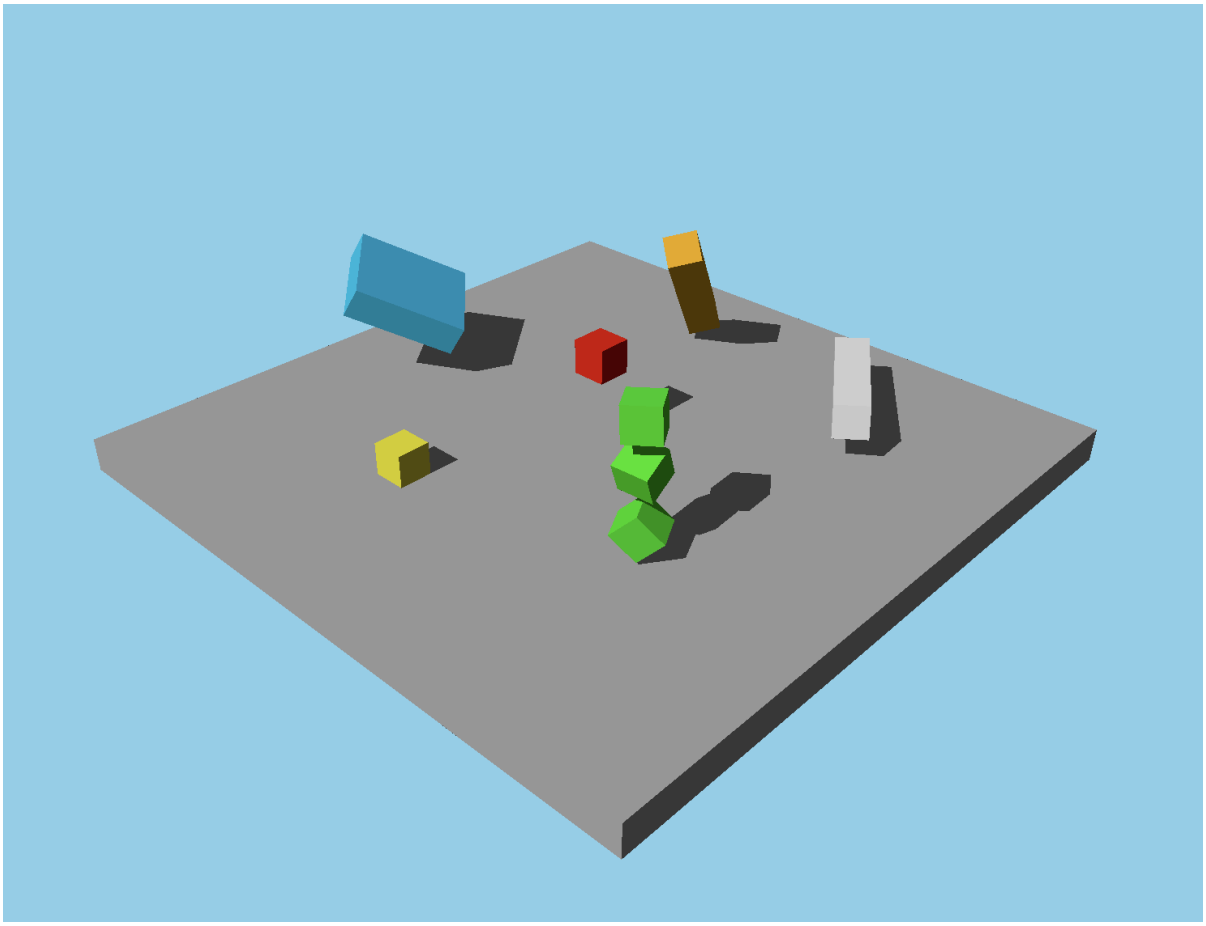

Figure 1: The arbitrarily transformed cubes in the scene have accurate pixel-perfect shadows.

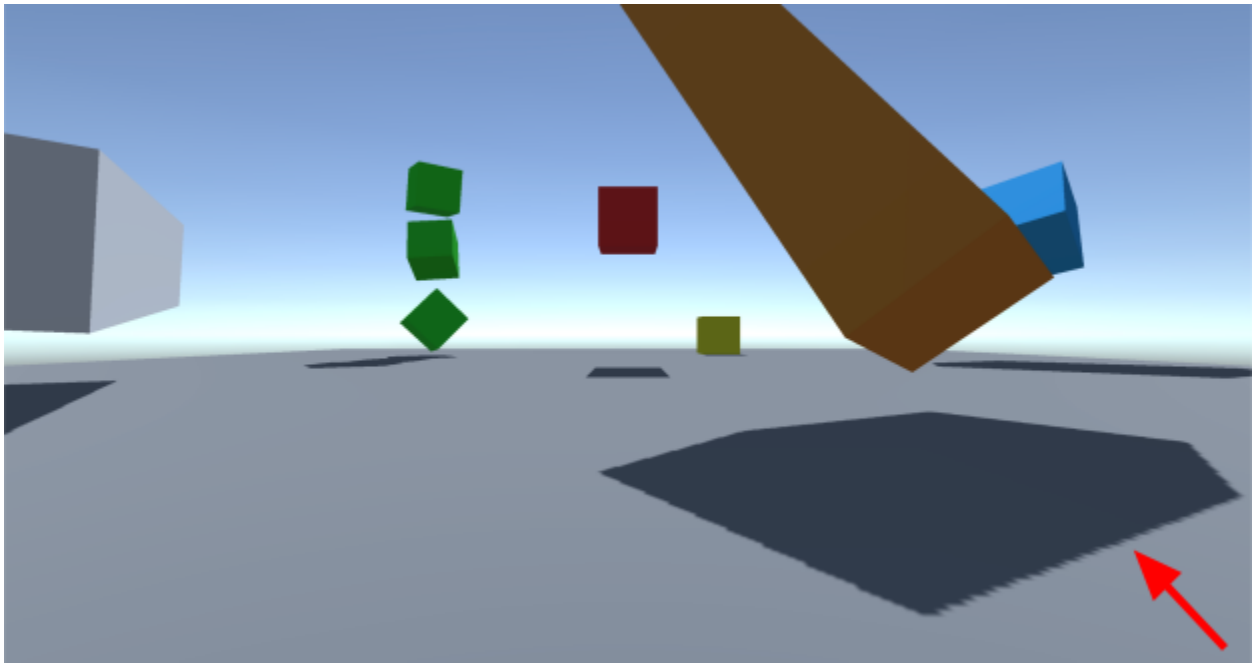

Figure 2: The same scene rendered with Unity's default shadow mapped implementation at low quality (the setting typically used on mobile), showing artifacts on the edge of the shadow.

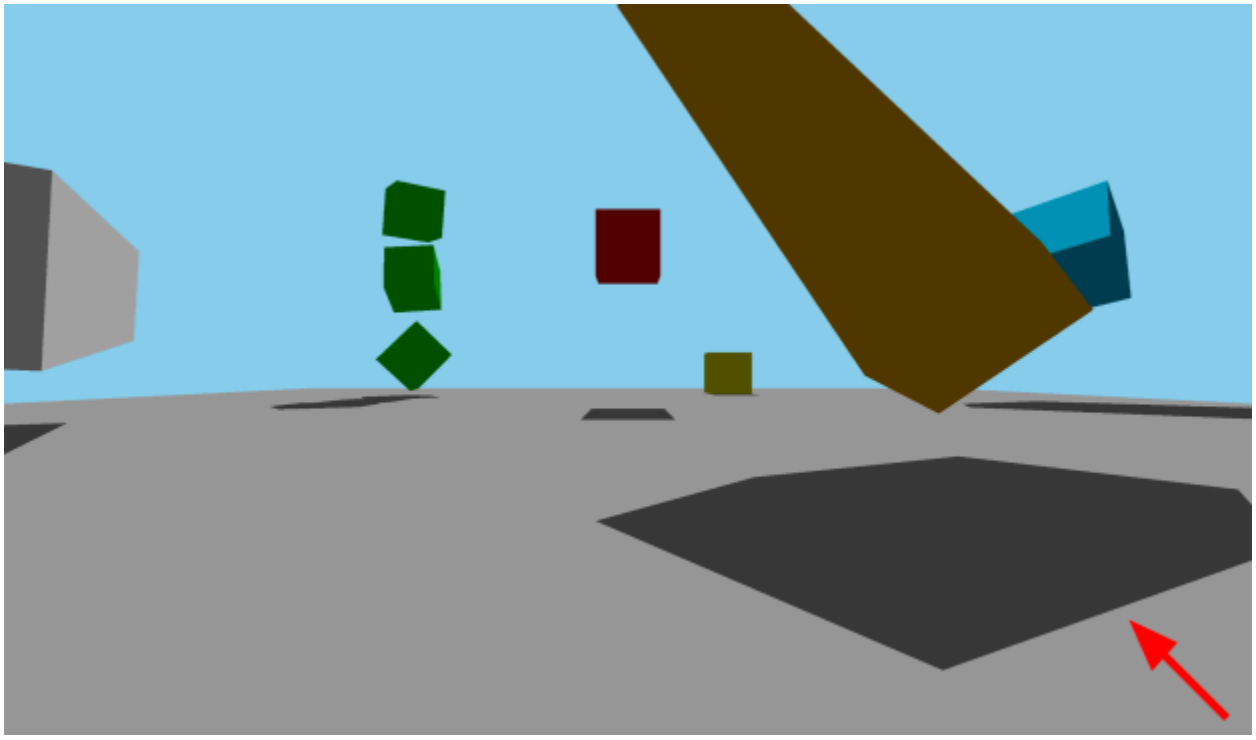

Figure 3: The same point-of-view rendered using the proposed mesh-shader based renderer. Note the difference in shadow quality for the close-up object.

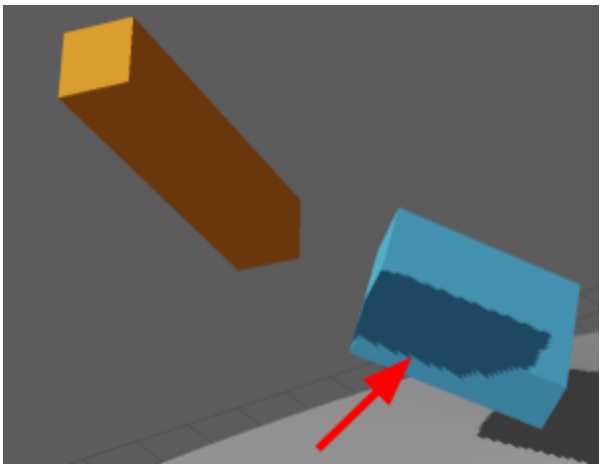

Figure 4:
An oblique caster rendered with the viewport lighting system in Autodesk Maya. Note that on more extreme edges, the shadow resolution worsens.

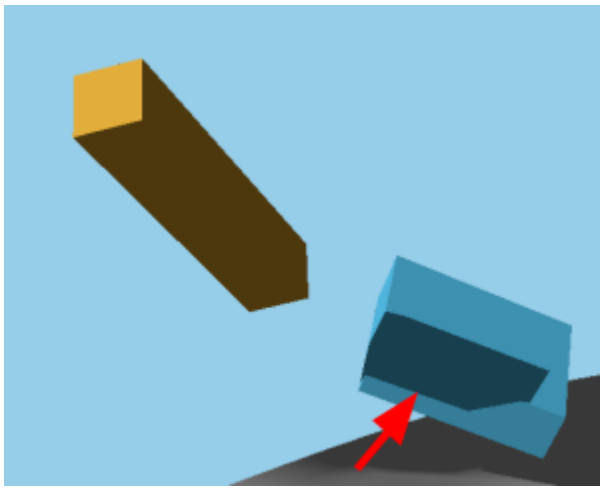

Figure 5:
On oblique receivers, the shadows in our mesh-shader based stencil shadow renderer do not lose resolution.

We compare the average total frame time of our mesh shader implementation with a direct port of our mesh shader implementation that calculates the volumes on the CPU. Due to current tooling limitations, we could not get more granular information than a total frame time.

Scene	Mesh Shader	CPU
8 cubes	1.9 ms	1.2 ms
20 cubes	2.9 ms	1.2 ms
50 cubes	5.5 ms	1.5 ms
100 cubes	11.3 ms	1.9 ms
200 cubes	19.4 ms	4.3 ms

Hardware: Macbook Pro with Intel i9-9880H and AMD Radeon Pro 5500M
Resolution: 1600x1200

Our proof-of-concept implementation is fairly generic. Our performance results are not particularly surprising to us because we did not tailor the algorithm to the specialized graphics hardware that it was running on. In particular, calls to *Dispatch*(1,1,1), which we make in our Object shader, are quite inefficient. A more optimized algorithm would use a smaller number of total mesh dispatches from the Object shader that each generate multiple quads in the resulting volume. Such a modification would more efficiently utilize GPU resources. Nevertheless, we have achieved the goal of the thesis, to demonstrate that it is possible to use mesh shaders to generate geometry volumes for use in stencil shadows.

6 Future Work

We were only able to get the algorithm working on arbitrarily transformed cube meshes. However, previous implementations of stencil shadows have been demonstrated to work with arbitrary meshes. The programmable nature of mesh shaders should simplify such efforts, but this is left as future work.

Many naïve approaches to generating the volumes also require the source geometry to be watertight [11], a requirement that is often not satisfied in interactive environments. With mesh shaders, geometry analysis can be easily moved to the GPU, freeing up CPU time for domain-related computation, but this is left as future work.

As shown in the data table above, our implementation, while functional, performs poorly compared to a CPU implementation. We believe our implementation parallelizes poorly on the

GPU, and a smarter mesh dispatch with larger workgroup sizes should improve performance significantly, but this is left as future work.

7 Glossary

This section introduces terminology used throughout this document. It is assumed that the reader has a basic understanding of graphics programming.

Host, Device, and Memory

The *host* is the main computer consisting of a CPU and random-access memory, also referred to as the *system*. The *device* is the GPU. In desktop computers, the host and the device typically have separate memory because the device is often an add-in card, and on the device, this memory is known as video memory or VRAM. To move data between these memory pools, one must explicitly copy the data across the interconnect bus (typically PCI-Express). On devices with a System-on-a-Chip (SoC), like mobile devices, it is common for the host and device to use the same physical memory. In this case, both the host and the device can read or write to any allocated address in the address space, so there is no need to perform an expensive copy to make host data accessible to the device and vice-versa.

Pipeline

The GPU executs shaders in a predefined order, known as a *pipeline*. Some stages are programmable, meaning the user can supply custom code to run, or they are fixed-function, meaning that the behavior of the stage is predefined and can only be influenced by configuration options. There are multiple types of pipeline. *Render Pipelines* include stages for geometry and pixel processing. *Compute Pipelines* contain a single arbitrary shader that can read or write any data. Modern graphics APIs like Metal, Vulkan, and DirectX12 consolidate all pipeline configuration into Pipeline State Objects, or PSOs. In modern graphics APIs the programmer explicitly creates and manages pipelines, while in OpenGL they are created implicitly.

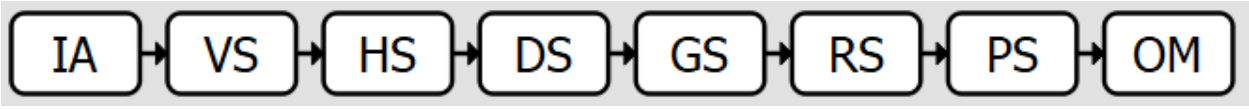

Figure 6: A diagram from the Renderdoc graphics debugger, showing the stages and their order in the traditional rasterization pipeline. They are:

- IA: Input Assembly (fixed-function)
- VS: Vertex Shader
- HS: Hull Shader (or Tessellation Control Shader)
- DS: Domain Shader (or Tessellation Evaluation Shader)
- GS: Geometry Shader

- RS: Rasterizer (fixed-function)
- PS: Pixel (or Fragment) Shader
- OM: Framebuffer output (fixed-function)

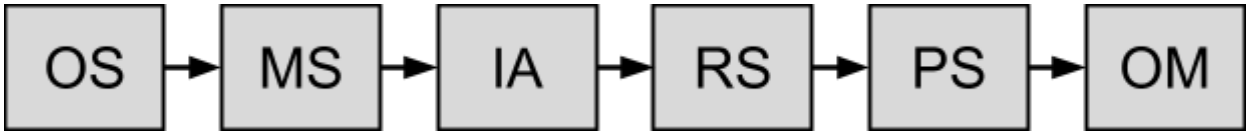

Figure 7: A diagram of a mesh shader pipeline. Mesh shader rasterization pipelines are a variant of the traditional rasterization pipeline shown in Figure 6. Note that the VS→HS→DS→GS stages from the traditional render pipeline have been removed, and the OS (Object Shader) and MS (Mesh Shader) stages have been prepended to the input assembly stage. The specifics of these stages are explained in the Context and Past Work section of this document.

Samplers, Depth Texture, Stencil Texture and Operations

GPUs store image data in *textures*. The result of pixel-processing shaders is written to *render target* textures. When a shader wants to read a texture, it uses a *sampler,* providing a unitized coordinate for the pixel to read. To determine depth sorting of geometry, the user can add a depth render target, which includes the normalized distance of a written pixel from the camera. Modern GPUs also offer a special 8-bit texture known as the *Stencil* texture, which is typically not directly writable from a shader but can have configurable operations applied to it. For example, the user can configure the stencil texture to increment when a front face is rasterized, but only if the depth test has failed. The user can also configure the PSO to not run the fragment shader on a given pixel if the value in the stencil buffer is greater than a provided value, making the stencil texture a convenient way to implement shape masking. These tests are typically implemented directly in hardware, and so are often faster than manually implementing such behavior with samplers and the `discard` instruction.

Forward and Deferred rendering

It is common for render engines to make multiple render targets to store relevant data. Textures that are not displayed to the user directly are known as *offscreen* textures. Offscreen render target textures that contain geometry information, such as the normal vector of a given triangle, are known as *geometry buffers* or *gbuffers*. The reference render engine in this document implements an approach known as *deferred lighting*, whereby the render engine first collects information about the geometry without lighting into *gbuffers*, then performs the lighting calculations in a separate step. This is in contrast to *forward* rendering, where all lighting and shading for an object happens in a single step. We chose deferred lighting for this implementation to make the render engine easier to understand, because there is a clear separation between lighting calculations and the rest of the rendering process.

References

[1] Franklin C. Crow. 1977. Shadow algorithms for computer graphics. In Proceedings of the 4th annual conference on Computer graphics and interactive techniques (SIGGRAPH '77). Association for Computing Machinery, New York, NY, USA, 242–248. https://doi.org/10.1145/563858.563901

[2] *Real-Time Rendering · Deferred lighting approaches*. (2009, June 2). Real-Time Rendering. Retrieved March, 2023, from https://www.realtimerendering.com/blog/deferred-lighting-approaches/

[3] Unreal Engine, "Virtual Shadow Maps," Unreal Engine 5.1 Documentation, URL: https://docs.unrealengine.com/5.1/en-US/virtual-shadow-maps-in-unreal-engine/, retrieved January 16, 2023.

[4] ARM. (n.d.). *Geometry Shading*. Arm GPU Best Practices Developer Guide. Retrieved July, 2023, from https://developer.arm.com/documentation/101897/0301/Tessellation--geometry-shading--and-tiling/Geometry-shading?lang=en

[5] Unity Technologies. (n.d.). *Shadow Cascades*. Unity - Manual. Retrieved January 16, 2023, from https://docs.unity3d.com/Manual/shadow-cascades.html

[6] Epic Games. (n.d.). Use Cascaded Shadows. Unreal Engine Documentation. Retrieved September 4, 2023, from https://docs.unrealengine.com/4.27/en-US/SharingAndReleasing/Mobile/Lighting/HowTo/CascadedShadow/

[7] Michael Bunnell and Fabio Pellacini, “Chapter 11. Shadow Map Antialiasing,” GPU Gems Online, URL: https://developer.nvidia.com/gpugems/gpugems/part-ii-lighting-and-shadows/chapter-11-shadow-map-antialiasing, retrieved January 28, 2023.

[8] Khronos. (2022, November 23). Geometry Shader. OpenGL Wiki. Retrieved August, 2023, from https://www.khronos.org/opengl/wiki/Geometry_Shader

[9] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51, 1 (January 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[10] Microsoft. (n.d.). Mesh Shader | DirectX-Specs. Microsoft Open Source. Retrieved July, 2023, from https://microsoft.github.io/DirectX-Specs/d3d/MeshShader.html

[11] Stich, M., Wächter, C., & Keller, A. (n.d.). Chapter 11. Efficient and Robust Shadow Volumes Using Hierarchical Occlusion Culling and Geometry Shaders. NVIDIA Developer. Retrieved February, 2023, from https://developer.nvidia.com/gpugems/gpugems3/part-ii-light-and-shadows/chapter-11-efficient-and-robust-shadow-volumes-using

[12] Bilodeau, W. & Songy, M. (2002, May 7). Method For Rendering Shadows Using a Shadow Volume and a Stencil Buffer. US6384822

[13] Peddie, J. (2021, April 6). Mesh Shaders Release the Intrinsic Power of a GPU. ACM SIGGRAPH Blog. Retrieved September 5, 2023, from https://blog.siggraph.org/2021/04/mesh-shaders-release-the-intrinsic-power-of-a-gpu.html/

[14] Estkowski, R., Mitchell, J., & Xiang, X. (n.d.). Optimal Decomposition of Polygonal Models into Triangle Strips. Retrieved September 5, 2023, from http://www.ams.sunysb.edu/~jsbm/papers/p151-mitchell.pdf