Rendering Surgery Simulation with Vulkan

Nicholas Milef, Di Qi, and Suvranu De

1.1 Introduction

While surgical simulation requires much of the same rendering functionality as games, critical differences necessitate simulation-specific optimizations and engine design decisions that aren't commonly needed or provided in rendering engines for games. Given our unique use cases, we take advantage of the explicitness of the Vulkan API (as compared to OpenGL) to develop a rendering engine for surgical simulation. In this article, we explain how we tailored our rendering engine design around surgery simulation including how higher level design decisions propagate to lower-level usage of Vulkan. To achieve this goal, our rendering architecture is designed to be flexible, maintainable and efficient. In surgical use cases, soft tissues are modeled by deformable meshes which are specially handled by our efficient memory system. We show how performance scales with our memory system. Later, we present a case study using our renderer in a virtual cricothyroidotomy (CCT) 3D simulator.

1.2 Overview

Virtual surgery simulators present some unique computational and development challenges that are less common in other applications such as games. Rendering is particularly important because the appearance of the simulator must be convincingly realistic to properly train surgeons for real-life surgery scenarios.

General-purpose game engines often have limited soft-body physics and haptics support. Platforms such as the Software Framework for Multimodal Interactive Simulation (SoFMIS) [Halic et al. 11], the Simulation Open Framework Architecture (SOFA) [SOF 18], OpenSurgSim [Ope 17], and the Interactive Medical Simulation Toolkit (iMSTK) [iMS 18] seek to fill this gap. Our rendering engine in particular is part of the larger framework of iMSTK. In addition, newer APIs such as Vulkan [The Khronos Group 18] provide more capabilities to make better use of computing resources and allow for more predictable performance compared to older graphics APIs such as OpenGL.

1.2.1 Current Rendering Challenges for Surgery Simulation

One of the main challenges with surgery simulation is the rendering of difficult materials such as skin, tissue, and organs. Unlike in games, difficult rendering scenarios cannot be avoided by artwork or level design. It's necessary to render intricate details such as marks on organs and skin while making the rendering look photorealistic so that users can become familiar with the real surgical procedure.

Another challenge we face is the handling of dynamic meshes to prepare for rendering. Since a major component of useful surgery simulation is the simulation of the physical deformation of tissue and organs and their interactions with the virtual environment, the renderer needs to be able to efficiently and correctly handle the updating of these geometries. Although soft bodies exist in video games, they often have limited interactions with other objects (e.g., cloth simulation), so the simulation and rendering can be often be done exclusively on the GPU. However, surgery simulation also relies on haptic devices, so forces must be sent to the haptic device drivers. GPU computation isn't as desirable option because the computation results must be send to the haptic devices as soon as possible and many of the physics computations use serialized constraint solving, making these algorithms difficult to parallelize for the GPU. In order to render the deformable meshes, we needed to develop an architecture to quickly update data.

Finally, although the Vulkan API is more explicit than OpenGL, many API usage choices depend on the target hardware. In other words, it's not always clear what acceptable parameters are to guarantee satisfactory performance. Memory management is one of the areas where the optimal use cases are particularly ambiguous.

In this chapter, we first present our approach towards achieving realistic rendering with our rendering architecture. Next, we present methods for handling the computation and transfer of dynamic meshes. Finally, we share our approach to memory management for our rendering system.

1.3 Render Pass Architecture

Rendering architectures construct their render passes in a way to minimize rendering artifacts, reach performance goals, and allow for advanced rendering algorithms. For a specific application, it is critical to balance these tradeoffs. Additionally, the increased complexity of these rendering engines

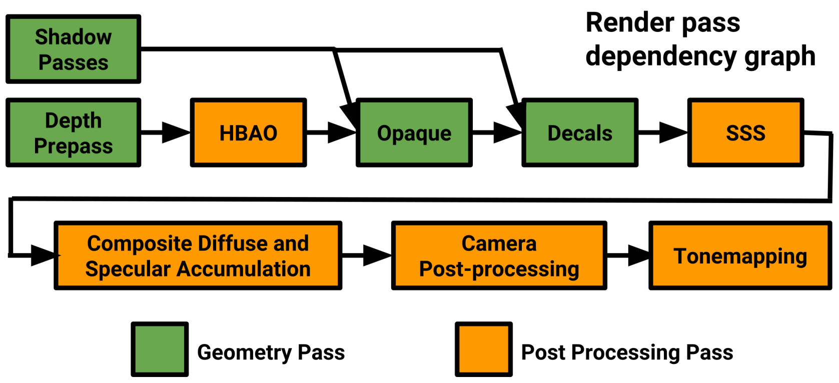


Figure 1.1. the render pass architecture

necessitates more complex shader code, which can be difficult for users and engine developers to maintain.

1.3.1 Render Pass Stages

Our rendering architecture is divided into individual render passes (Figure 1.1). Surgical simulation requires the rendering of diverse materials which some lighting techniques, such as deferred rendering, have difficulty expressing efficiently since lighting is decoupled from the geometry rendering. In deferred rendering, the geometry is rendering to a geometry buffer (G-buffer) that separate lighting passes read to perform lighting calculations for each fragment on the screen. Classical deferred renderers can be more efficient for rendering many lights, but surgery simulation generally only requires a small amount of lights (usually less than a dozen). While branching can be used to allow for some material variety [Garawany 16], it ultimately still limits the number and types of materials.

In contrast, in forward rendering, each material is evaluated independently, allowing for completely arbitrary material evaluations in each fragment shader for a material. While we chose forward rendering for this reason, we still incorporated a thinner G-buffer so that we could later use its data for post-processing such as screen-space subsurface scattering.

Each render pass is explained in detail in the following sections.

Shadow Mapping In some surgery simulation scenarios, many shadow-casting lights are necessary for helping users to judge depth perception of their instruments in the virtual environment. The first pass of each frame includes rendering shadows for each directional light into shadow maps, which we place into a texture array. Rendering shadow maps into texture arrays is a common approach [Pettineo 15] as it allows for the binding of many shadows maps during a single draw, which is necessary for forward rendering architectures that evaluate multiple lights in one draw call. We save on material permutations by reusing the same shadow pipelines for each shadow pass (since the render passes are compatible). We pass the index of the shadow current shadow map to the shadow shader using a push constant that accesses an array of light inverse matrices.

Depth Prepass and Ambient Occlusion Forward rendering is inefficient for scenes with significant overdraw. In order to minimize this, we implemented a depth-prepass as the next pass to minimize the lighting computation. We follow the depth-prepass with a horizon-based ambient occlusion (HBAO) pass [Bavoil et al. 08]. By calculating ambient occlusion (AO) before the lighting passes, the AO becomes available for use in the lighting pass. We calculate the HBAO at quarter resolution by first downscaling using a min-depth operator, and then we upscale the AO using a Gaussian-based bilateral depth-aware filter, as opposed to just a 2-pass Gaussian filter, to prevent AO bleeding.

Physically-Based Rendering (PBR) Lighting The lighting passes for both opaque geometry and decals follow. During this pass, lighting is calculated for both specular and diffuse and written into separate accumulation HDR 16-bit buffers (Table 1.1), similar to what was done in CryEngine [Sousa et al. 12]. In the case of the opaque geometry pass, we populate another buffer with 4 8-bit color channels for world-space normals and a subsurface scattering constant. The light types include both classical computer graphics light sources such as directional and point lights, and global illumination approximation, which in our case, is image-based lighting (IBL) with split-sum approximation [Karis 13]. Because we already calculated the AO, the IBL can be used with both decals and the underlying opaque geometry. Baked AO textures for rigid objects are read to supplement the screen-space AO, and the maximum of these two AO values is taken to determine the total AO. For the lighting pass, we support a roughness/metalness workflow for PBR content.

Buffer Type	**R**	**G**	**B**	**A**
Diffuse Accumulation	16-bit per channel color			
Specular Accumulation	16-bit per channel color			
Normal/SSS	x	y	z	SSS strength
Depth	32-bit depth			

Table 1.1. G-buffer layout

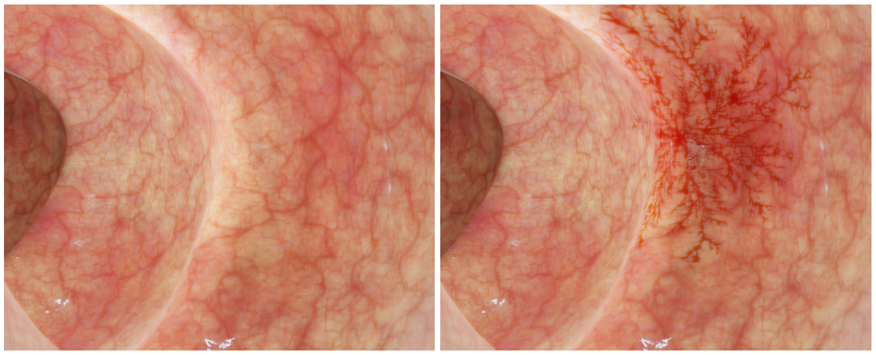


Figure 1.2. (left) a mesh without a decal, (right) a mesh with a decal affected by subsurface scattering

Deferred Decals Deferred decals are a low-cost yet flexible method for adding details to underlying geometry. The decal pass differs from the opaque rendering in that it doesn't write to the normal or subsurface scattering (SSS) buffer but rather reads from it. One of the benefits of using the underlying SSS buffer is that it allows the decal to blend in with the surface underneath during the SSS render pass, so any marks on the skin for instance will look more naturally integrated. Since the decals are deferred, they need a normal, and we chose to use the underlying normal and the underlying SSS constant (Figure 1.2). Many use cases of the decals such as bleeding or marking only make small changes to the surface normal, so the underlying surface normal can provide an acceptable approximation.

Post Processing Surgery simulations include organic geometry such as skin and organs, which requires subsurface scattering (SSS) to display accurately (Figure 1.3). We chose to implement a screen-space SSS as opposed to a texture-space implementation for several reasons. First, screen-space avoids overdraw which becomes a problem when inside the body. Although a depth prepass mitigates this, there's still the need to perform extra texture lookups as compared to screen space methods. Secondly, the diffusion profile is relatively similar since most of the materials have blood and/or fat under the surface. Third, it samples across different draw calls. This becomes particularly important when a mesh is split into smaller meshes in order to avoid unnecessary physics computations. For example, if a section of an organ undergoes operation, then it must be deformable, but the surrounding organ can be rigid. With screen-space SSS, the SSS can sample from across both meshes, creating a seamless rendering.

After the lighting pass, separable screen-space SSS [Jimenez et al. 15] is applied to only the diffuse buffer. We keep a pool of 3 HDR buffers (one

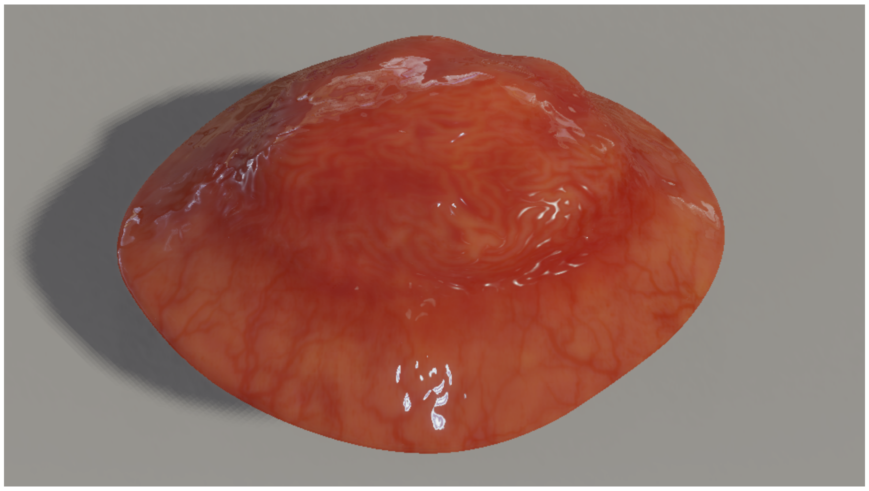


Figure 1.3. a rendering of a polyp with subsurface scattering

for the specular render target, one for the diffuse, and one free) in order to ping-pong the diffuse buffer during the two passes. After the SSS, the specular buffer and diffuse buffer are composited into the free buffer. These buffers are reused for later passes.

Bloom is then calculated in two passes at quarter resolution and then composited with the previous result. A filmic tone mapping pass [Hable 10] follows the bloom pass to map down to a 32-bit sRGB buffer. In the early stages of implementing SSS, we found that using a non-filmic tone mapper such as Reinhard [Reinhard et al. 02] hide the effect of the SSS by desaturating the effect of the diffusion profile.

1.3.2 Uber-Shader Approach and Material System

One of the challenges with forward rendering is the possible combinatorial explosion of different shader options.

In Vulkan, all draw state is compiled into a single *VkPipeline* object. We build materials for each object; each material contains a *VkPipeline* object and associated descriptor sets. A single object may have multiple materials (and thus *VkPipeline* objects) to account for multiple render passes. For example, a single object could have a material for shadow map passes and another one for the lighting pass.

One of the challenges with forward rendering is the need for software developers to maintain a large number of possible shader permutations. This becomes especially problematic in our framework because we allow arbitrary combinations of textures to be attached to a material as well as different draw modes such as for debug rendering. We chose to use an Uber-shader approach to reduce maintenance; we use a large shader that contains all possible shader code combinations and block out irrelevant parts during the shader compilation process. This generally works well for our applications since our applications are made to be photorealistic, so the shaders generally try to model lighting physics rather closely. Separate shader code paths can be made available for different lighting conditions such as debug rendering without introducing branching.

Shaders are compiled as a build step for compilation of the C++ code, so only SPIR-V binary files are read into the engine at runtime. This creates some problems with texture resource management, however, since each material can provide a different number texture resources. We solve this problem by creating small placeholder textures, but the lookup of these textures are restricted by specialization constants. Specialization constants allow the driver's shader compiler to optimize away shader code that gets set during pipeline creation. Other expensive operations, such as PBR lighting code, can be optimized away for situations that don't require it. One drawback of using specialization is that the pipeline objects can become incompatible for similar materials. In this case, they cannot be shared across draw calls.

1.4 Handling Deformable Meshes

Deformable meshes, which we define as meshes that include per frame vertex and index updates, require data transfer to the GPU, mesh recomputation, and efficient memory usage.

1.4.1 Efficient Data Transfer to the GPU

Deformable meshes require frequent and large-scale updates to the whole mesh, and transferring this to the GPU can be complex due to needed synchronization and slow if the data is large enough. Unlike OpenGL, Vulkan gives explicit control over the location of data, but the correct locations (e.g., which meshes should be on the GPU) are often difficult to assess for a given use case. Furthermore, different hardware vendors even have different recommended memory types that aren't universally available.

We specifically chose a solution that could work on a wider range of hardware and, through experimentation, gave acceptable results. We tested on NVidia's GTX 1080 GPU. We chose to use a single queue, the same queue used for rendering, to do the transfer operations required to update the meshes to avoid the need for inter-queue synchronization. Although some graphics drivers expose transfer-only queues that could potentially allow for higher bandwidth, this is not universally available across all major vendors [Willems 18].

With using a single queue, there are two effective ways to update data for rendering: 1) using a CPU-accessible staging buffer with a mirrored GPU buffer and running transfer operations or 2) using a CPU-accessible buffer. In comparing these two approaches, we found that for large meshes (i.e., larger than 100k triangles), the differences in performance were negligible between the two methods. Even with multiple render passes for the same mesh data that could cause redundant PCI-E bus usage, the performance was similar. On the other hand, using system memory (CPU-accessible buffer) allows us to avoid managing memory barriers to ensure the data is uploaded to the GPU in time.

Multi-Buffering Our renderer includes a pool of command buffers for geometry passes, and each frame, a command buffer is recycled and rewritten to. When we finish writing to a command buffer, we submit it to the driver, and start recording another command buffer. The command buffer will run asynchronously to our render loop. We needed to avoid read-write hazards but didn't want to stall the render loop, so we implemented multi-buffering for vertex and index buffers (Figure 1.4). We buffer the data the same amount as the number of back buffers presented in the swap chain; if the application renders with triple-buffering, then the mesh data also uses triple-buffering. This makes tracking the region to update simple as the remainder of the frame number can just be passed to the update functions.

Our multi-buffering implementation is similar to unsynchronized multi-buffering in OpenGL with persistent data mapping [Hrabcak and Masserann 12], but we have more control over the memory management and synchronization.

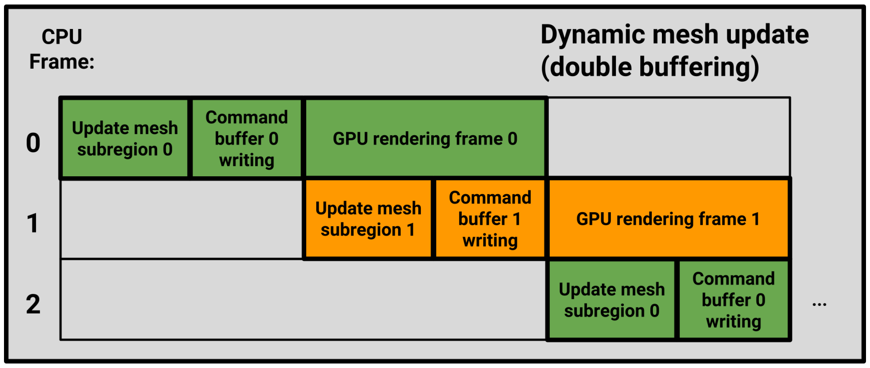


Figure 1.4. an example of dynamic mesh updating using double-buffering

1.4.2 Normal Calculations per Frame with Smoothing Groups

Because the deformable meshes can have both topology changes and individual vertex displacements, the normals and tangents must be able to be recomputed each frame. Additionally, many of the geometries in surgery simulation tend to have organic shapes, leading to vertex seams along the edges of the UV maps. These seams cause lighting discontinuities which make surfaces incorrectly appear to have hard edges.

To handle these problems, on file import, we create a mapping on vertices that belong to the same smoothing group. With recomputation of the normals, these mappings are preserved. The final normal for each vertex in the group is calculated from each neighboring triangle's normal and each vertex that belongs to the group. The tangents are calculated separately, however, because they are aligned to each vertex's UV coordinate, which differs for each vertex in the group.

One problem with this approach is that the tangents can diverge from the normals since tangents depend on the UV coordinates which are likely unique to each vertex, whereas normals can be shared across vertices. This produced shading artifacts that were highlighted by our BRDFs, but a simple fix was to orthogonalize the tangent-bitangent-normal basis in the vertex shader through the Gram-Schmidt process.

1.5 Memory Management System

Unlike OpenGL, Vulkan gives users explicit control on where they can store data, but the correct locations are often difficult to assess. In addition, memory backings for resources such as images and buffers are not automatically allocated. In contrast, behind the scenes, OpenGL drivers do additional work such as memory defragmentation and suballocation. OpenGL abstracts the physically locations of all resources (such as in system RAM or VRAM), although this hides differences between different GPUs and drivers. While this requires less development from the application side, this can lead to inconsistent performance across platforms. In order to fill this gap lying in the Vulkan API, we implemented our own memory allocator.

1.5.1 Custom Memory Allocator

The performance implications of different allocation strategies are not always obvious and can differ depending on the vendor. Furthermore, certain allocations strategies can be tailored to specific applications. For instance, in surgery simulation, it's uncommon for new geometry and resources to be added during the simulation. It is generally known beforehand what

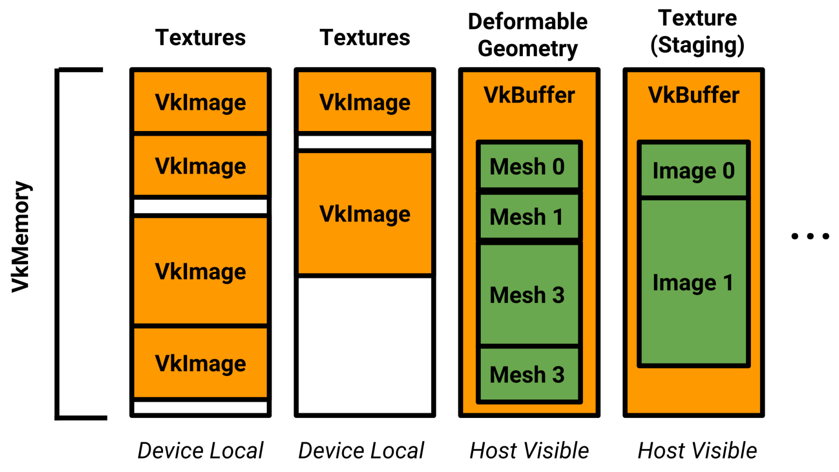


Figure 1.5. an overview of the memory manager

resources are needed for a specific application. This allows us to avoid implementing performance-sensitive features such as memory defragmentation.

Our memory manager separates different resources into different memory allocations (Figure 1.5). For instance, uniform buffers occupy a different memory allocation than textures. The advantage of this approach is that the certain resource types need to be in certain areas of memory for optimal performance. For instance, staging buffers need to be in host visible memory, whereas images (e.g., textures) need to stay on the GPU, so they reside in device-local memory.

Certain resources such as images, uniform buffers, and storage buffers have alignment requirements. This allows mesh data to be more compact, which is useful since mesh data might need to be transferred each frame for dynamic objects.

With the exception of uniform buffers, a single *VkBuffer* occupies the whole underlying memory allocation. Internally, our memory manager uses lightweight abstract buffer objects that point to regions within the *VkBuffer* object. With uniform buffers and *VkImages*, however, multiple uniform buffers and images can fill a single allocation. Uniform buffers are treated this way since the minimum uniform buffer size guaranteed by the specification is 64 KiB.

The initial allocation size we chose was 16 MiB, and this allocation is used until it runs out of space and then a new allocation is made. For images and buffers larger than the allocation size, the allocation is expanded to

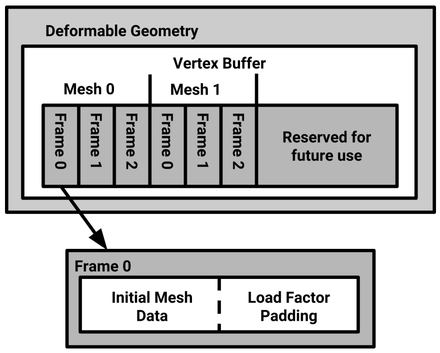


Figure 1.6. a buffer layout allowing for per-frame updates

account for these, so very large resources can potentially have their own allocation.

Mesh Data Mesh data is handled differently from the other resource types. Deformable meshes reside in host visible memory, whereas other meshes (e.g., rigid objects) are in device local memory through staging. Other game engines such as Source 2 have followed a similar approach for static/dynamic resources [McDonald 16].

Deformable meshes also take up more space than rigid meshes need as they are multi-buffered. When the mesh is initially allocated, this extra buffering space is also allocated in the same location. For host visible memory, this works out well since the transfer to the GPU is implicitly done, avoided overhead in calling transfer functions.

In some operations, such as mesh cutting operations, additional triangles or vertices can be added or removed to the mesh. Adding new vertices or triangles would require more buffer space. Because operations such as cutting are a high-frequency operation, we needed a way to expand geometry all of the time without allocating new memory and deallocating old memory, which could be costly and cause memory fragmentation. To solve this problem, we allow users to specify a load factor that sets a maximum size of the geometry relative to the original mesh size. We allocate additional space within each buffer subsection for each frame (Figure 1.6).

1.6 Performance and results

To analyze the performance of our memory management system within our renderer, we devised a benchmark that tests the role of different default memory allocation sizes. A size of zero MB represents a naive approach of

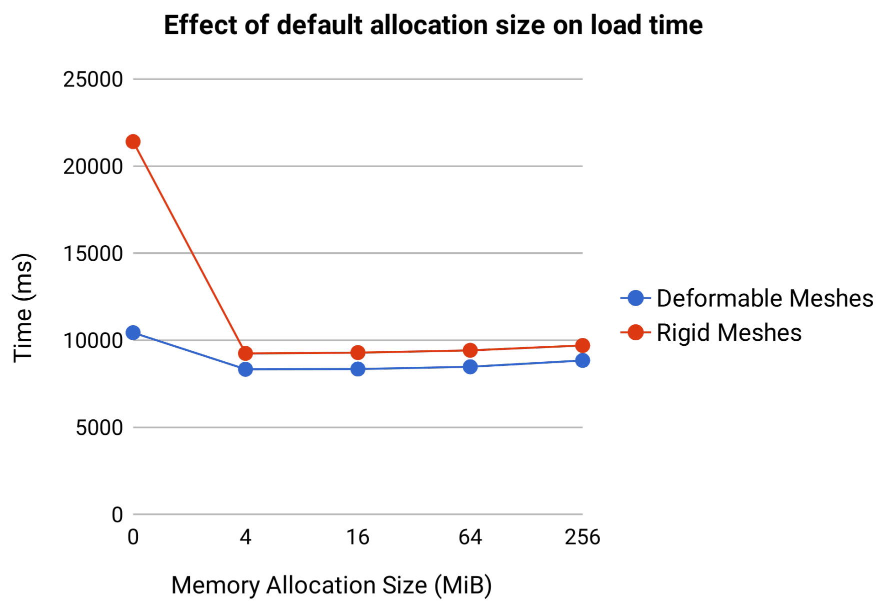
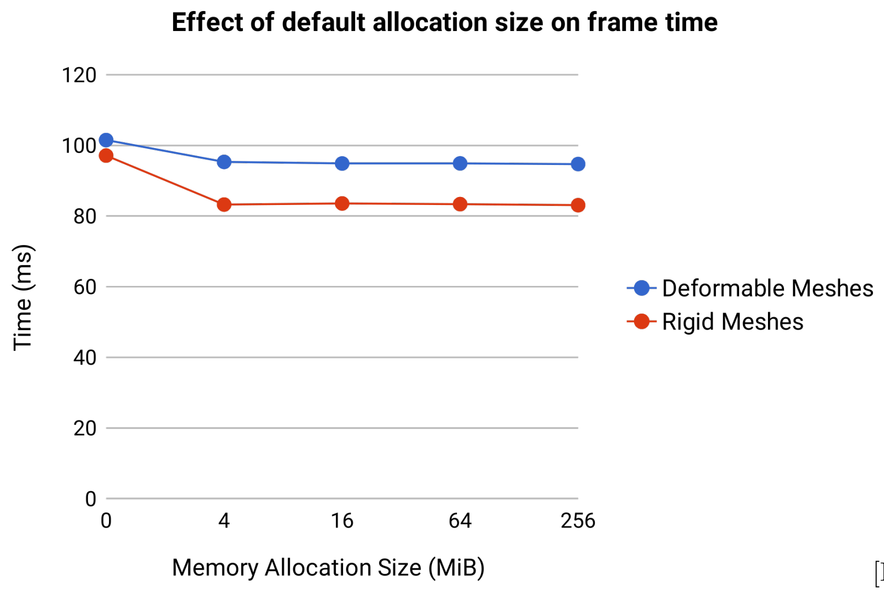


Figure 1.7. 10,000 meshes, each made up of 100 lines



[H]

Figure 1.8. 10,000 meshes, each made up of 100 lines creating one memory allocation per vertex and index buffer. We found two areas that showed improvement based on our test scenario: application load time (Figure 1.7) and frame time (Figure 1.8). We tested on a Windows computer with an Intel i7 6850k CPU and an NVidia GTX 1080.

There are optimal allocation sizes that meet both of these metrics. For example, larger allocations (such as 256 MiB) have slightly longer load times while they have comparable performance to smaller allocations (4 MiB). Different default allocation sizes have been proposed by vendors such as 8MB for mesh data and 128 MB for textures [McDonald 16] or 256 MiB [Sawicki 18]. This depends on the hardware and application resources sizes to some extent, but there should be a fairly large range of acceptable default allocation sizes, as demonstrated by our data. Unless very large resources are used (e.g. 10s of MB per resource), there's a point of diminishing returns for runtime performance with larger buffer sizes. However, larger allocation sizes slightly increase load time and will waste more memory in cases when they aren't saturated.

For our tests, we compared rigid mesh data with deformable meshes. Rigid meshes require two buffers, a staging (CPU) and a device local (GPU) buffer. Meanwhile, the deformable meshes require larger buffers to account for multi-buffering, but they remain on the CPU, resulting in much lower the allocation times, particularly for more allocations. We experienced a larger decrease in load time performance when using the naive allocation strategy for rigid meshes as compared to deformable meshes, which indicates that making many GPU allocations can be much slower than CPU allocations. For the runtimes, the dynamic meshes require rewriting of the data and transfer to the GPU, which slows down performance compared to rigid meshes.

1.7 Case Study: CCT

Surgeons perform the cricothyroidotomy (CCT) procedure as an emergency procedure when patients have a restricted airway. The steps involved in the procedure are as follows:

1. Palpating the neck region to identify the locations of the thyroid and cricoid cartilages which are the landmarks anatomies in this procedure.

2. Making an incision along the midline of the neck through the skin and the fat tissue to uncover the cricothyroid membrane underneath.

3. Making an incision along the membrane to open an entrance to the trachea.

4. Inserting an endotracheal tube inside the trachea through the new incision.

There are two main problems with the CCT simulation from a rendering perspective: efficiently updating the large geometric models representing the fat and membrane tissues and rendering the surface of each cut.

1.7.1 Dynamic Mesh Update

The CCT procedure requires at least two incisions to allow for intubation. This is accomplished on the CPU side through a mesh cutting algorithm which causes the mesh to regenerate in order to incorporate topological changes. In addition, the mesh is deformed during each physics step, which causes the vertex positions to be displaced and the normals and tangents to be recalculated. All of this data must be reuploaded to the GPU each frame.

Initially, we had difficulty with transfer speeds for this use case as we used staging and GPU buffers to handle all transfers. We wrote to a host visible buffer and performed a transfer operation, and we operated on a single queue. We quickly found this actually substantially sped up performance, so we switched to using host-visible memory.

1.7.2 Surface Cut Rendering

The rendering of the cut mesh introduced a few challenges with our current framework. First, the outer surface, the skin for example, must be rendered with a different material than the inner surface, which in our case is the fat tissue. On top of this, the UV coordinates are generated during runtime as the new surface mesh is recreated.

We wanted to reduce the number of surface meshes to be updated each frame, but our renderer doesn't currently support multiple materials per mesh. Both the skin and the fat tissue use the same shading logic, but they differ in texture sets. In order to circumvent our single material per mesh limitation, we project the sides of the mesh onto different regions of a texture atlas.

When performing the cut, we needed a visual cue (e.g., bleeding) for the progress of the cut. We opted to use a pool of blood decals to display the cutting path while it was being performed by the user. The decals automatically recycle after the pool hits a certain maximum number so they can be reused for multiple cuts.

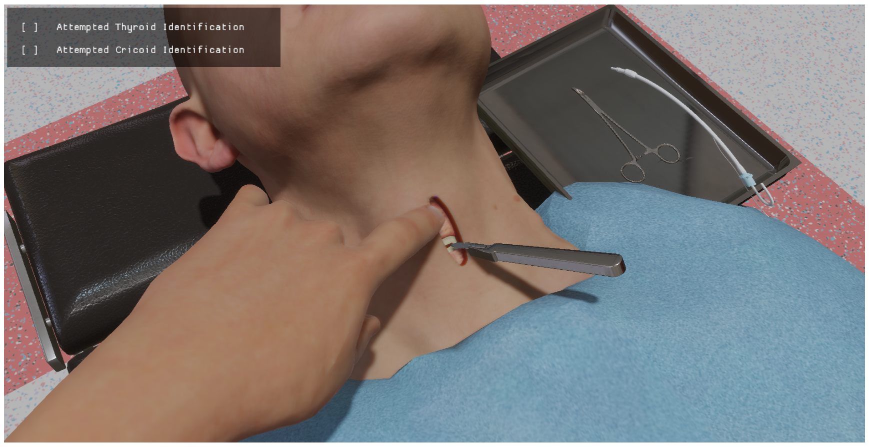

[ ] Attempted Thyroid Identification
[ ] Attempted Cricoid Identification

Figure 1.9. the CCT case study demonstrating cutting

1.8 Conclusion and Future Work

Rendering for surgery simulation is critical for creating a realistic immersive experience for training surgeons. The Vulkan API provided more flexibility, but also some challenges with determining the optimal implementation strategies given our use cases.

We were able to create a rendering architecture that reduced the amount of shader maintenance needed in the future and would give us more accurate visualization. Our handling of deformable meshes allows us to efficiently render the output of various CPU-based algorithms. Finally, our memory management system allows us to scale our applications without worrying about introducing substantial overhead, and it can easily be extended support new types of resources.

In the future, we hope to expand these subsystems to further improve performance and rendering capabilities. One area we would like to explore is using asynchronous buffer transfers with multiple queues for mesh updates, as this could potentially decrease the transfer times by using more bandwidth. Another area that could see performance improvements would be the normal/tangents recalculations, possibly through compute shaders as this takes a considerable amount of time each frame depending on the topology and number of triangles of the mesh. Finally, we would like to expand our material system to allow for more complex and expressive materials.

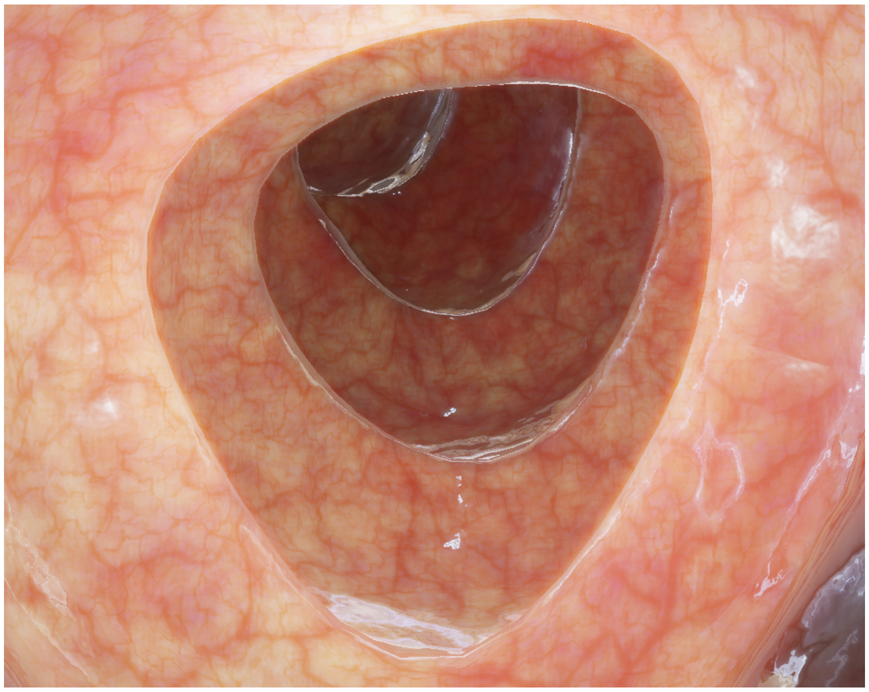


Figure 1.10. a rendering of an internal organ using our renderer

1.9 Source Code

The source code is available as part of iMSTK: https://www.imstk.org

1.10 Acknowledgements

Research reported in this publication was supported by the National Institute of Biomedical Imaging and Bioengineering (NIBIB) of the National Institutes of Health (NIH) under Award Number 2R01EB005807, 5R01EB010037, 1R01EB009362, 1R01EB014305; National Heart, Lung, and Blood Institute (NHLBI) of NIH under Award Number 5R01HL119248; National Cancer Institute (NCI) of NIH under Award Number 1R01CA197491 and NIH under Award Number R44OD018334.

Bibliography

[Bavoil et al. 08] Louis Bavoil, Miguel Sainz, and Rouslan Dimitrov. "Image-space horizon-based ambient occlusion." In *ACM SIGGRAPH 2008 talks*, p. 22. ACM, 2008.

[Garawany 16] Ramy El Garawany. "Deferred Lighting in Uncharted 4." In *Advances in Real-Time Rendering in Games: Part I*. ACM, 2016.

[Hable 10] John Hable. "Uncharted 2: HDR lighting." In *Game Developers Conference*, 2010.

[Halic et al. 11] Tansel Halic, Sreekanth A Venkata, Ganesh Sankaranarayanan, Zhonghua Lu, Woojin Ahn, and Suvranu De. "A software framework for multimodal interactive simulations (SoFMIS)." In *MMVR*, pp. 213–217, 2011.

[Hrabcak and Masserann 12] Ladislav Hrabcak and Arnaud Masserann. *Asynchronous Buffer Transfer*. CRC Press, 2012.

[iMS 18] "iMSTK." http://www.imstk.org/, 2018.

[Jimenez et al. 15] Jorge Jimenez, Károly Zsolnai, Adrian Jarabo, Christian Freude, Thomas Auzinger, Xian-Chun Wu, Javier von der Pahlen, Michael Wimmer, and Diego Gutierrez. "Separable Subsurface Scattering." *Computer Graphics Forum* 34:6 (2015), 188–197.

[Karis 13] Brian Karis. "Real shading in Unreal Engine 4." In *Proc. Physically Based Shading Theory Practice*, pp. 621–635, 2013.

[McDonald 16] John McDonald. "High Performance Vulkan: Lessons Learned from Source 2." In *GPU Technology Conference 2016 (GTC)*, 2016.

[Ope 17] "OpenSurgSim." https://www.sofa-framework.org/, 2017.

[Pettineo 15] Matt Pettineo. "Rendering the alternate history in The Order 1886." In *Advances in Real-Time Rendering in Games: Part II*. ACM, 2015.

[Reinhard et al. 02] Erik Reinhard, Michael Stark, Peter Shirley, and James Ferwerda. "Photographic tone reproduction for digital images." *ACM transactions on graphics (TOG)* 21:3 (2002), 267–276.

[Sawicki 18] Adam Sawicki. "Memory Management in Vulkan DX12." In *Game Developers Conference*, 2018.

[SOF 18] "SOFA." https://www.sofa-framework.org/, 2018.

[Sousa et al. 12] Tiago Sousa, Nickolay Kasyan, and Nicolas Schulz. *CryENGINE 3: Three Years of Work in Review.* CRC Press, 2012.

[The Khronos Group 18] The Khronos Group. “Vulkan® 1.1.81 - A Specification.”, 2018.

[Willems 18] Sascha Willems. “GPUInfo.” https://www.gpuinfo.org/, 2018. Accessed: 2018-07-18.