Simulating Instruments on the GPU

Daniel Sussmann - c21029111 - SussmannD@cardiff.ac.uk
Supervisor: Kirill Sidorov - SidorovK@cardiff.ac.uk

Introduction

Project Background and Motivation

Physical modeling synthesis represents an exciting frontier in music production and digital audio. Unlike traditional multisampling-based approaches, which deliver realism and variety at the cost of extensive storage, physics-based modelling offers the potential to recreate the complex sounds of acoustic instruments through computational simulation rather than recording and replaying. This opens up the possibility to generate highly dynamic, realistic and responsive sounds by simulating the physical mechanisms responsible for generating sounds from their respective instruments. Among these, drums present a particularly interesting challenge due to the complex interactions between vibrating membranes and resonant bodies, and as a drummer myself, their simulation will be the focus of this project. However, the computational demands of accurate physical modeling have generally limited its application, especially for real-time interactive use. The simulation of a vibrating membrane alone requires solving complex differential equations across thousands of spatial points and time steps, and this computational barrier has historically meant that high-fidelity physics-based drum synthesis has remained primarily in the domain of academic research or offline rendering.

One NESS [1] project demonstrated a simulation of a timpani [2], where both the membrane and shell were modeled using physics-based techniques. CUDA [3] acceleration was used to improve efficiency. This approach provided accurate sound synthesis while reducing computational cost, making it a promising method for realistic percussion modeling; however, despite these optimizations, the computation time remained significant, with one example requiring four minutes to simulate just three seconds of sound. Inspired by this work, the main goal of this project has been to tackle this computational challenge by utilising two complementary approaches; leveraging the massive parallelism offered by GPU-based computation, and developing approximations of the physical mechanisms that preserve accuracy to an acceptable level whilst further reducing computational complexity.

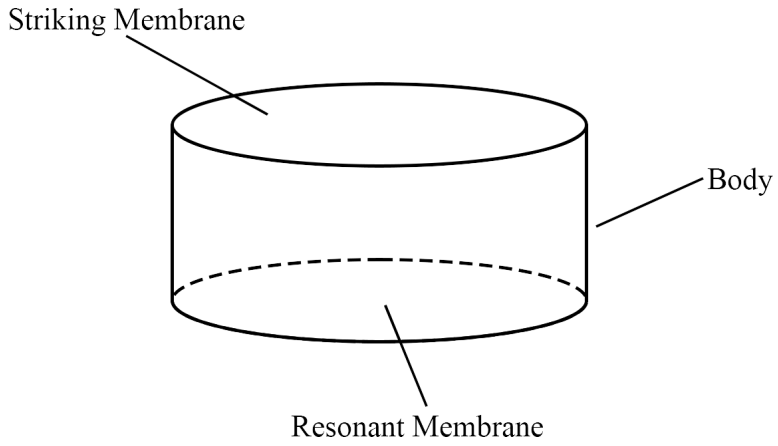
A secondary objective of this project is to explore the evolving role of Large Language Models (LLMs) as a collaborative tool for development on complex projects. This project wouldn't have been achievable without the help of Anthropic’s Claude [4], which served as both a knowledge base and implementation partner. While GPU programming and physical modeling synthesis would typically be considered advanced topics for undergraduate work, this project aims to demonstrate how LLMs can help to bridge knowledge gaps and potentially enable students in the future to be able to tackle more ambitious technical challenges earlier on in their academic career.

Background Knowledge

Drum Acoustics

The most standard drum can be thought of as three (core) components: the striking membrane (the head of the drum), the body (or shell) and the resonant membrane.

Figure 1: Simplified Diagram of a Drum



**Note: [5] Provides a good discussion of a more complete drum model, including additional components and their functions.*

The striking membrane is the primary source of a drum's sound, undergoing complex vibrations when struck. These vibrations define the instrument's tonal characteristics and decay patterns, causing a resonating response in the body, as well as pressure changes in the internal air cavity of the drum which result in the resonant membrane vibrating. To capture these dynamics, this project implements high-fidelity physics-based modeling to simulate vibrations on the striking membrane. However, for the sake of maintaining a manageable project scope, the resonant membrane is not simulated - a reasonable simplification given that many drum types do not include this component.

Beyond the striking membrane (just membrane from here on out), the drum body influences the overall sound by acting as a resonating structure, modifying both the tonal quality and sustain of the membrane's vibrations. The size and shape of the body significantly affect the sound characteristics—larger diameter or deeper shells produce thicker and heavier tones, while smaller or shallower shells create brighter and lighter tones [6]. Additionally, different body materials transmit sound waves with varying speeds, affecting the timbre and tone of the instrument. For instance, wooden shells typically offer a warmer and more resonant sound due to their ability to absorb and reflect sound waves in a particular way [7]. In principle a high-fidelity physics-based model of the shell would yield the most accurate results, however, again for the sake of maintaining a manageable project scope and to reduce computational complexity, a key question arises–can this influence be effectively modeled without a full simulation? To investigate this, the drum body's response is simulated as a simple resonator rather than fully modeled using physics-based methods. This approach has already been successfully applied to string instruments, where high-fidelity string vibrations were modeled separately from the body which was modelled as a simple resonator [8].

DrumForge

DrumForge is the result of this project and the program responsible for carrying out the drum simulations. I thought the best way of showcasing the functionality and demonstrating some of the simulated sounds was through a video, which you can watch by clicking the link or scanning the QR code below. I recommend watching the video prior to reading the rest of the report.

Video 1: DrumForge Demonstration

https://youtu.be/7DiiWhnudG8?si=nMzkJmIg54GejosC

Before talking about the implementation details of DrumForge, the following sections discuss the physical equations and modelling techniques behind the simulation, as well as a bit of background on GPU programming and how it differs from traditional programming.

Physical Equations and Modelling Techniques

The physical-model of the membrane in DrumForge is based on the 2D wave equation, which describes the transverse vibrations of a thin elastic membrane under tension. For the simulation, the following equation is used which also allows for damping and tension parameters:

$$\frac{\partial^2 u}{\partial t^2} + 2\beta \frac{\partial u}{\partial t} = c^2 \nabla^2 u$$

Where:

$u(x, y, t)$ is the displacement of the membrane at position (x, y) and time t

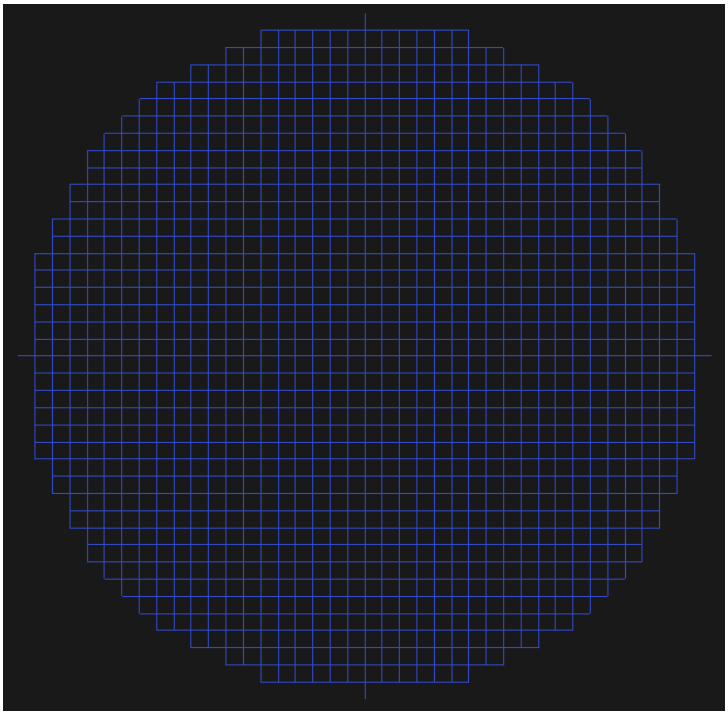
β is the damping coefficient

c is the wave propagation speed derived from tension as $c = \sqrt{\frac{T}{\rho}}$

$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$ is the Laplacian operator

To solve this equation numerically, the Finite Difference Time Domain (FDTD) method is used, a standard approach in physics simulations due to its intuitive nature and robust accuracy. FDTD operates by replacing continuous differential equations with discrete approximations across both space and time. To put it into plain words, the membrane is broken down into individually simulated grid points, each connected to their neighbouring grid points, like a mesh. As the simulation progresses, the position of each node (grid point) at each point in time is recalculated based on the position of its neighbouring nodes. The following is a simulation from DrumForge showing how the membrane can be represented as a grid:

Figure 2: Discretized Membrane from DrumForge



The model also incorporates a circular boundary condition to ensure that points lying on the edge of the membrane are not displaced, in the same way that the part of the membrane connected to the rim on a real drum is held in place. This boundary condition is responsible for the way that a wave travelling through the membrane reflects when it reaches the rim, as well as how different wave patterns are observed when playing near the rim (as seen from 1:40 in the demo video).

The body of the drum is modelled using a technique called modal synthesis, which represents the acoustic response of the body as a bank of second-order resonators. In physical acoustics, a "mode" refers to a natural pattern of vibration in which the entire structure vibrates at a specific frequency. For a drum shell, these modes correspond to different ways the material can vibrate - for example, one mode might involve expansion and contraction around the circumference, while another might involve vibration along the height of the shell. Each mode produces a characteristic partial tone that contributes to the overall timbre of the drum.

Modal synthesis captures each specific vibrational mode of the shell with a dedicated resonator, and collectively these resonators approximate the overall acoustic response. The mathematical model used for each resonant mode of the body follows a damped harmonic oscillator equation:

$$\frac{d^2x}{dt^2} + 2\zeta\omega_0\frac{dx}{dt} + \omega_0^2 x = F(t)$$

Where:

x is the mode's displacement

$\omega_0 = 2\pi f$ is the angular frequency of the mode (with f being the resonant frequency in Hz)

ζ is the damping ratio (derived from the decay time)

$F(t)$ is the external force (input from the membrane)

During the simulation, when the membrane receives an impulse, it transmits this information to the body. The body then processes this impulse information, calculating how strongly each mode should be excited based on the impulse's strength and position, and subsequently carries out the excitations.

GPU Computing

Graphics Processing Units (GPUs) were originally designed to accelerate the rendering of images and video, but their architecture makes them exceptionally well-suited for a wide range of computationally intensive tasks beyond graphics. Unlike Central Processing Units (CPUs), which are optimized for sequential processing with a few powerful cores, GPUs contain thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously. This architectural difference is the foundation of GPU parallel computing. Traditional CPU-based computing typically processes instructions sequentially—one after another—which is efficient for general-purpose computing tasks that involve complex branching logic and irregular memory access patterns. However, this approach becomes a bottleneck for computations where the same operation needs to be performed on large sets of data.

GPU computing addresses this limitation through massive parallelism, utilizing thousands of lightweight execution units, referred to as "threads." In GPU computing, a thread represents the smallest unit of execution that performs a specific computation on a single data element. Unlike CPU threads which are relatively heavyweight and limited in number, GPU threads are designed to be extremely lightweight, allowing thousands or even millions to execute concurrently.

Parallel GPU computing works excellently with the FDTD method used for simulating the membrane in DrumForge, as each node can be assigned to and continuously updated by its own thread. This allows the entire membrane to update concurrently, as opposed to updating each node one by one. The following link is an animation I've generated with Claude that demonstrates how the membrane updates when running sequentially on the CPU versus parallelly on the GPU:

Video 2: CPU vs GPU Animation

https://youtu.be/jsWG2l2VHAw?si=rEK1yWpLx7s7A1jf

System Overview

System Architecture

DrumForge is implemented in C++, and follows a layered design with clear separation of concerns:

1. **Core Simulation Layer:** Handles the physics simulation and component management
2. **GPU Acceleration Layer:** Provides CUDA-based parallelization and memory management
3. **User Interface Layer:** Manages visualization, user input, and control interfaces
4. **Audio Processing Layer:** Handles audio sampling, processing, and export

The two main architectural choices in DrumForge are the use of Manager Classes, each of which has its own individual responsibilities, and the Component Classes, each of which is a simulated physical component of the drum (membrane and body). The following sections break down each of these architectural choices further.

Note: You are able to navigate back and forth between the report and relevant code excerpts by clicking on the [a] tags and the Return to report links in the appendix.

Manager Classes

The architecture is centered around six manager classes that coordinate different aspects of the simulation system. Each manager operates as a singleton, providing global access to its functionality while maintaining internal state separation:

1. **SimulationManager:** Coordinates the simulation lifecycle and component interactions
    - Maintains global simulation parameters
    - Manages component registration, life cycle and coupling
    - Controls simulation timesteps and advancement [a1]
    - Handles grid space allocation for components
2. **CudaMemoryManager:** Manages GPU memory and CUDA operations [a2]
    - Allocates and tracks GPU memory resources
    - Provides buffer abstractions for CUDA operations
    - Detects device capabilities and adapts accordingly
3. **VisualizationManager:** Controls the visual representation [a3]
    - Manages OpenGL context and rendering pipeline
    - Provides rendering utilities for components
    - Handles camera and view transformations
    - Coordinates frame rendering and synchronization
4. **GuiManager:** Manages user interface [a4]
    - Integrates Dear ImGui [9] for interactive controls
    - Handles application state transitions
    - Provides parameter controls for components
    - Manages audio recording interface
5. **InputHandler:** Processes user interactions
    - Captures keyboard and mouse events
    - Provides ray casting for 3D interaction
    - Handles user interactions with the membrane [a5]
6. **AudioManager:** Handles audio processing [a6]
    - Records samples from simulation components
    - Manages multiple audio channels
    - Provides interpolation between simulation and audio rates
    - Exports audio to WAV files

The central coordination happens in the main simulation loop [a7], which orchestrates the flow of data between components and managers.

Component Based Approach

DrumForge uses a component-based design to model different aspects of the drum's physical structure. This approach allows for:

1. **Modularity:** Each component can be developed, tested, and optimized independently
2. **Flexibility:** Different physical models can be swapped in and out
3. **Scalability:** New components can be added to model additional aspects without requiring complete redesign of preexisting components

DrumForge implements two primary components:

1. **MembraneComponent:** Models the drum membrane [a8]
    - Implements 2D wave equation using FDTD method
    - Processes physics in parallel on the GPU
    - Provides virtual microphones for audio sampling
    - Applies impulses on the membrane
2. **BodyComponent:** Models the drum shell [a9]
    - Uses modal synthesis with resonant filters
    - Provides different material presets
    - Resonates in response to membrane impacts
    - Contributes to the final audio output

The component system is built around the **ComponentInterface** [a10] abstract base class, which defines a common API for all simulation components. Components interact through a standardized physics coupling mechanism, the **CouplingData** [a11] structure, serving as a platform for data exchange between components. This approach allows components to communicate with each other without tight code dependencies. In DrumForge, the primary coupling occurs between the membrane and the body, where, as mentioned earlier in the Modelling Techniques section, impact events from the membrane are transmitted to the body component which then responds with appropriate resonance. The coupling is implemented through the **getInterfaceData()** and **setCouplingData()** [a12] methods of the **ComponentInterface**.

Simulation Pipeline

The simulation pipeline in DrumForge defines how the various components interact and advance over time. This section describes the control flow and timing mechanisms that drive the simulation.

As mentioned above, the core of DrumForge's operation is the main simulation loop [a7], which coordinates all aspects of the application. This loop structure ensures that each iteration:

1. Processes user input
2. Advances the physics simulation by one step
3. Updates the visualization
4. Renders the user interface
5. Maintains a stable frame rate

Ons of the critical aspects of the simulation is managing time steps correctly. The **SimulationManager** is responsible for advancing the simulation with fixed-size time steps [a1], and this approach ensures simulation stability, while still allowing the visualization to run at a variable frame rate. The stable time step is calculated based on the requirements of all components [a13]. At this stage as the membrane component is the only physically simulated one, it is determined by the CFL (Courant-Friedrichs-Lewy) condition [10] for the membrane simulation.

The **SimulationManager** also serves as a spatial coordinator, managing the global grid that components use to allocate space for their simulations [a14]. For future iterations of DrumForge, this grid management system would allow multiple components to coexist in a shared spatial context, enabling more complex simulations with interactions between components.

The simulation initialization process follows a specific sequence to ensure proper setup:

1. **Global Configuration:** The simulation parameters are configured
2. **Component Creation:** Individual components are created with their specific parameters
3. **Component Registration:** Components are registered with the **SimulationManager**
4. **Grid Allocation:** Each component requests a region of the global grid
5. **Component Initialization:** Components initialize their internal state
6. **Coupling Setup:** Couplings between components are established

This is implemented in the main application code [a15] before the main simulation loop, and ensures that all components have the resources they need and are properly connected before the simulation begins.

CUDA Implementation

The CUDA implementation in DrumForge forms the foundation of the system's performance, enabling real-time physics-based simulation of the drum membrane. This section explains how GPU parallelism is leveraged to achieve the computational efficiency necessary for interactive use.

The core of the CUDA implementation is contained in **membrane_kernels.cu**, which implements several specialized kernels that handle different aspects of the membrane simulation. The membrane is modeled as a circular region within a rectangular bounding box placed in the world grid of the simulation. The **initializeCircleMaskKernel()** [a16] kernel initializes a mask that identifies which grid points are inside the circular membrane region. This approach ensures that we only simulate physics for points within the actual membrane, avoiding unnecessary computation for grid points outside the circular boundary.

The main physics simulation is implemented in the **updateMembraneKernel()** [a17] kernel which, as mentioned earlier, applies the FDTD method to solve the 2D wave equation. This kernel implements the leapfrog integration scheme, which is second-order accurate in time and well-suited for wave equation simulations. The damping term models energy loss in the membrane, controlling the decay rate of vibrations.

When the user strikes the drum via a mouse click, the **applyImpulseKernel()** [a18] kernel is used to introduce an impulse on the membrane at a specific position. The impulse is applied as a Gaussian-shaped displacement, creating a semi-realistic drum strike that propagates as a wave across the membrane, as seen in the demo video.

As seen above in the Manager section, DrumForge uses a dedicated **CudaMemoryManager** to handle GPU memory allocation and management. This class also provides a type-safe interface for buffer allocation and automatic resource cleanup [a19]. This class simplifies memory management and reduces the risk of errors in CUDA code, providing clean abstractions for common operations.

Finally, to maximize GPU utilization, DrumForge organizes threads into 2D blocks that match the structure of the membrane grid [a20]. This adaptive approach ensures efficient execution for different membrane sizes, maximizing occupancy and compute throughput.

Graphics

The graphics system in DrumForge provides a real-time visualization of the simulated drum components, allowing users to observe the vibration patterns as they interact with the membrane. This section describes how the visualization system is implemented using OpenGL [11] with GLFW [12] and GLEW [13]. DrumForge employs a component-based visualization architecture, where each simulation component can provide its own rendering implementation, facilitated by the aforementioned **VisualizationManager**. This architecture allows for flexible extension with new visualization techniques without modifying the core rendering system.

DrumForge uses modern OpenGL 3.3 with the core profile, providing a forward-compatible rendering pipeline. The initialization process sets up the OpenGL context through GLFW [a21], and after the context is created, GLEW is initialised to provide access to OpenGL extensions [a22]. This setup ensures that all required OpenGL functionality is available and compatible with the host system.

Custom GLSL shaders are used to render the drum components. The shaders are defined as string literals and compiled at runtime [a23]. The vertex shader includes a colouring algorithm that visually represents the membrane's displacement - positive displacements (upward) are coloured red, while negative displacements (downward) are coloured blue. The intensity of the colour corresponds to the magnitude of the displacement, providing an intuitive visual representation of the wave propagation.

The membrane component is visualized just as it is simulated, as a grid of nodes that update in real-time based on the simulation. The membrane is initialized as a flat grid of vertices [a24], which then have their z-coordinates updated during the simulation based on the current membrane height values [a25]. The implementation supports both optimized GPU-GPU updates through CUDA-OpenGL interoperability, and a more compatible CPU-based approach. Unfortunately, as CUDA-OpenGL interop is not yet supported on Windows Subsystem for Linux (WSL) [14], which is what I use as my development environment, I was unable to test the functionality of this part of the implementation, so relied on the CPU-based rendering alone (which still worked seamlessly for real time visualization). The

drum body is visualized as a cylindrical wireframe using line primitives. This provides a clear representation of the shell without overwhelming the visualization [a26].

Audio Generation

The audio generation system in DrumForge translates the physical simulation of the membrane and body into audible sounds that represent the simulated drum. This section describes how the audio sampling, processing and export are implemented.

As mentioned above, the audio system is built around the AudioManager class, which provides audio samping from the simulation, multi-channel audio support for virtual microphones, sample rate conversion and interpolation, and WAV file export capabilities. This design ensures centralized audio handling across the application [a27], and provides a flexible audio processing pipeline that can adapt to different membrane configurations and sample rates.

A key implementation in DrumForge is the virtual microphone system, where membrane height displacement values are sampled to generate audio. This also allows for multiple sampling points on the membrane to capture a more thorough sound. Each microphone has properties that govern its placement and functionality with the membrane [a28]. The membrane component maintains a collection of these virtual microphones, allowing for multi-microphone setups [a29]. This approach allows users to place microphones at different positions to capture various aspects of the membrane's vibration, i.e. a central microphone to capture the fundamental tone, or multiple microphones for a more balanced sound.

A significant challenge with the audio generation was bridging the gap between the simulation rate (~60-120 Hz) and the audio sample rate (44.1 kHz). This was solved through linear interpolation, ensuring smooth audio transitions between simulation steps, and preventing artifacts that would otherwise occur from the much lower simulation rate [a30]. The audio generation also combines the signals from both the membrane and body components to produce the complete drum sound, balancing the direct membrane sound with the resonant characteristics of the body [a31].

Finally, the audio data can be exported to a standard WAV file for listening and for further post processing [a32]. This also includes automatic normalization to ensure optimal volume levels in the exported file.

Results

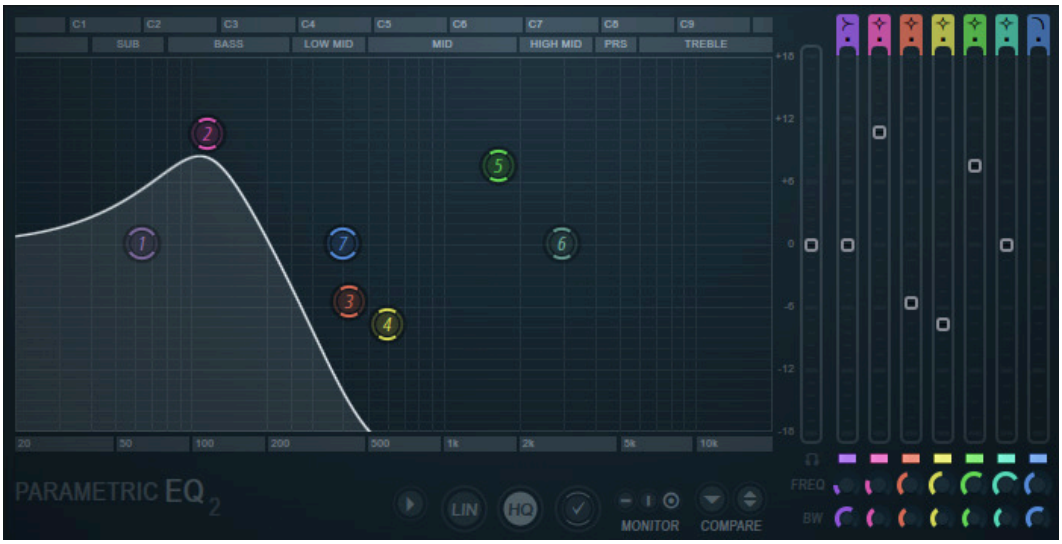
The following section is a discussion on the sounds generated by DrumForge and their accuracy, and the performance of the program. All of the simulated sounds have been recorded using the "Quad" virtual microphone preset.

As demonstrated in the video, the initial sounds generated by the simulation can sound quite abstract (robotic, static etc.) due to an abundance of high frequencies, I believe this could be due to a couple of reasons, however would require further investigation to get to the root cause:

1. Discretization of the membrane
2. The membrane equation is somewhat ideal - lacking some of the non-linear physical characteristics that might dampen higher frequencies
3. Linear interpolation of generated audio due to difference in simulation and sampling rate
4. Complexity of the audio generation pipeline leading to high frequency artifacts

After applying equalization, while still electronic, the sound is noticeably more similar to that of a real drum. The following figure shows the equalizer preset applied to the sounds in the demonstration video, through the Fruity Parametric EQ 2 [15]:

Figure 3: Equalizer Preset Used for Post Processing



Note: The following video contains the sounds of the simulated drums and real drums that they were modeled off of played in sequence:

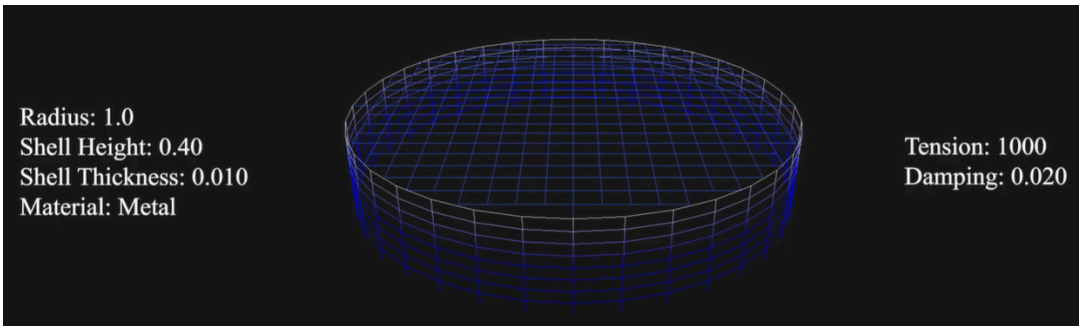
Video 3: Comparison of Real and Simulated Drums

https://youtu.be/PpYj2dtdBBU?si=qPxbpMvhkm3e_pFd

The first sound evaluated was generated from a shallow, high-tension drum with medium level dampening and a metal body. This configuration was modeled after a timbale, a shallow single-headed drum with a metal shell most commonly used in Latin percussion.

Figure 3: Timbale Drum

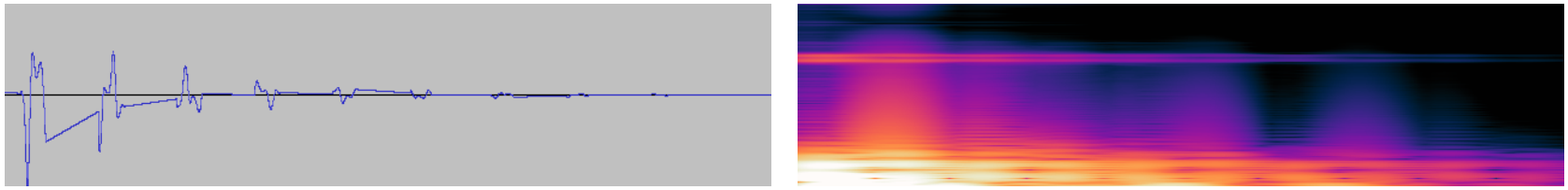
Figure 4: DrumForge Timbale Model



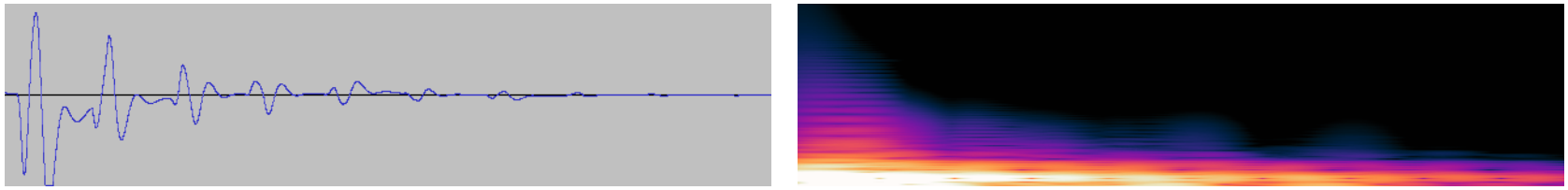
When comparing the simulated timbale sound to recordings of real timbales, the simulation lacks the sharp attack and punch characteristic of the actual instrument. Additionally, the simulated sound exhibits a notably lower pitch than a real timbale, though this limitation could maybe have been addressed by modifying the configuration GUI to accept a wider range of tension values, allowing for the higher frequencies typically produced by tightly-tuned timbales. Looking at the waveforms, we can also see some clear differences. The overall waveform shape is the same in terms of attack and decay, however the real timbale appears to oscillate at a much higher frequency than the simulated sound, further confirming that the tension value or radius parameter was not optimal. When examining the simulated waveform pre-equalization, clear straight-line sections between the waves are visible, seemingly revealing the linear interpolation due to the mismatch in simulation and audio rates - this is likely the source of the high frequency noise in the sound. The corresponding spectrogram shows a distinct high frequency component cutting through the sound. After applying equalization, the post-eq waveform appears smoother and less jagged, similar to how a low-pass filter blurs an image. The high frequency outlier previously visible in the spectrogram has been eliminated. When comparing the post-eq simulation spectrogram to the spectrogram of the real timbale, they actually share quite a similar shape, however the real instrument clearly contains a wider range of mid and some high frequencies that likely contribute to the attack and timbral richness that the simulation lacks.

Figure 5: Comparison of Real And Simulated Timbale Waveforms and Spectrograms

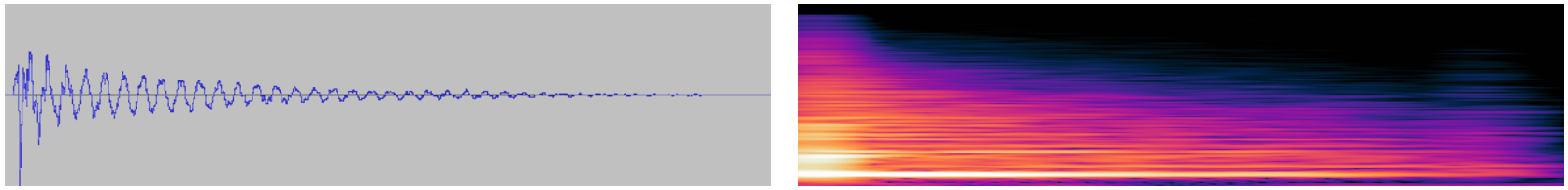
Simulated Timbale Waveform and Spectrogram Before Equalization



Simulated Timbale Waveform and Spectrogram After Equalization



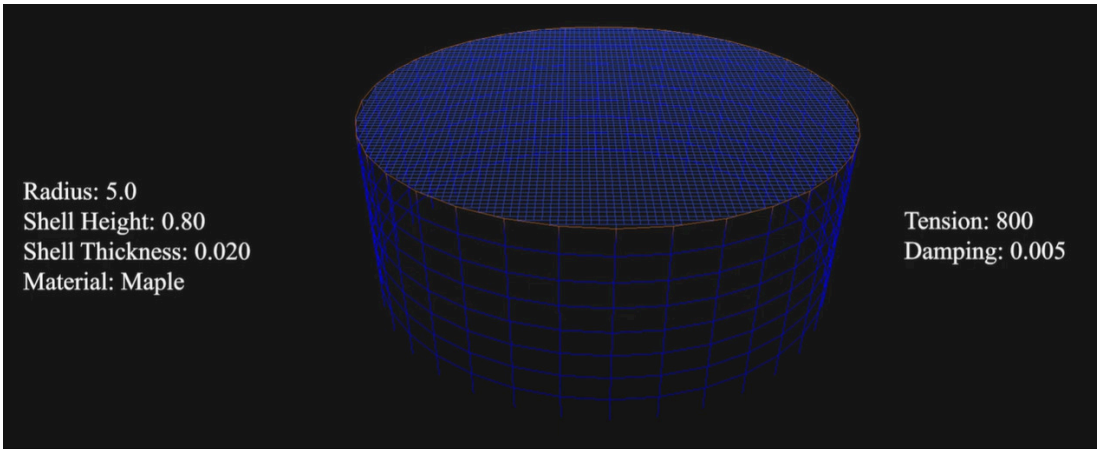
Real Timbale Waveform and Spectrogram [16]



The second sound evaluated was generated from a deep, low-tension drum with low level dampening and a maple body. This configuration was modeled after a concert bass drum, a deep double headed dum with a usually wooden shell most commonly used in orchestral pieces.

Figure 6: Concert Bass Drum

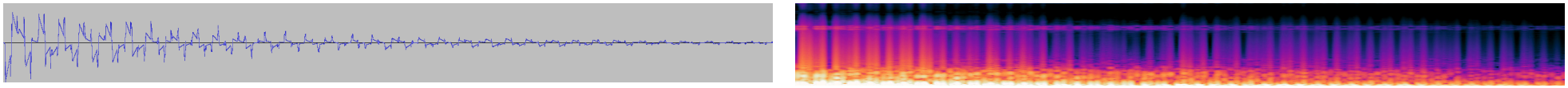
Figure 7: DrumForge Concert Bass Drum



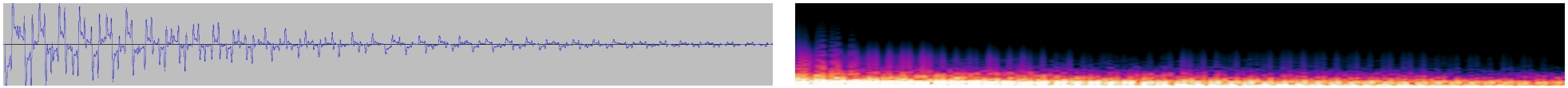
When comparing the simulated concert bass drum to recordings of real concert bass drums, the simulation again lacks the sharp attack and punch characteristic of the actual instrument. However, the simulated sound does successfully capture the deep, resonant tone and sustained ringing that follows the initial strike, which is a defining characteristic of concert bass drums. This indicates that while the simulation effectively models the low-frequency resonance and decay patterns, it struggles to reproduce the complex initial transients that give acoustic drums their distinctive attack. Again when looking at the waveform and spectrogram, we can see the same trends: the pre-eq waveform is more jagged, and the pre-eq spectrogram has a clear outlying high frequency. In these spectrograms where a bigger drum was simulated, we can also see what seems to be a limitation of using only four virtual microphones, with subtle gaps being present in the spectrogram along the horizontal axis. The shape of the post-eq spectrogram, if you ignore these gaps, is also very similar to that of the real drum. The real drum recording does show some subtle noise in the high frequencies, however this can be attributed to the recording quality rather than the actual instrument characteristics and can be ignored for comparison purposes.

Figure 8: Comparison of Real and Simulated Concert Bass Drum Waveforms and Spectrograms

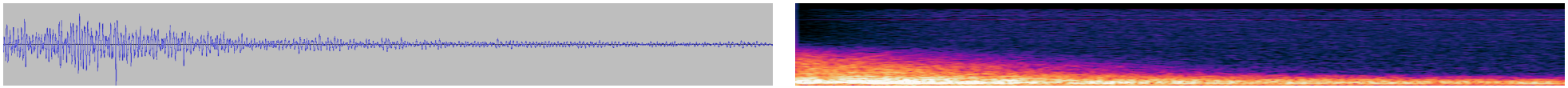
Simulated Concert Bass Drum Waveform and Spectrogram Before Equalization



Simulated Concert Bass Drum Waveform and Spectrogram After Equalization



Real Concert Bass Drum Waveform and Spectrogram [17]



Performance

While formal benchmarking was not conducted, DrumForge demonstrated excellent real-time performance throughout testing. The membrane simulation maintained a consistent 60 frames per second across all tested configurations, even with large radius membranes on high-resolution world grids (up to 256×256), highlighting the effectiveness of the CUDA implementation. This performance was achieved using an NVIDIA RTX 4050 Mobile GPU, which proved more than capable of handling the parallel computation demands of the physics simulation.

Audio generation and export was also instantaneous, with no noticeable delay between stopping a recording and having the processed audio available. The combination of GPU-accelerated membrane simulation and efficient CPU-based audio processing created a seamlessly responsive experience throughout testing.

The only performance limitation encountered was during experimental development of a more physically complex body component model. This advanced model required adherence to a much stricter CFL condition, resulting in an extremely small stable timestep. Consequently, the simulation attempted to update at an unsustainably high frequency, causing performance degradation.

Future Extendibility

DrumForge was designed with future extension in mind, providing a solid foundation for adding new components and enhancing existing functionality. The following section talks about some directions for future development.

Implementing real-time audio output capabilities would be a reasonable next step, as well as improving the overall quality of life when using the program (allowing drum reconstruction without having to restart the program, updating the gui, improving camera controls etc.). This could also potentially allow users to use DrumForge as a VST for music production or electric drum kits. Additionally, the audio quality could be substantially improved by using a different method for bridging the gap between simulation and audio sampling rate, and by using a physically simulated microphone in the 3D space rather than virtual microphones on the membrane.

The component-based architecture of DrumForge provides a clear path for integrating additional drum elements to enhance acoustic realism. A bottom membrane component could be implemented to model the resonant head found in many drums, responding dynamically to both the striking membrane's vibrations and pressure changes within the drum. The addition of an air cavity component could capture the acoustic coupling between membranes through a simulation tracking pressure waves inside the shell. For specific drum types like snare drums, a specialized component modeling wires vibrating against the resonant membrane could introduce characteristic buzzing effects through additional contact dynamics. Enhancing the physical models behind the existing components would also be a valuable improvement, such as implementing non-linearities in the membrane physics, and improving the material models of the

body. These extensions would leverage the existing component interface, requiring minimal changes to the core architecture while significantly expanding the range of drum sounds the system can produce.

LLM Usage and Development Process

As mentioned in the introduction, Anthropic's Claude was an important tool for the successful completion of DrumForge. The following section contains a discussion on how Claude (mainly Sonnet 3.7) was used as a collaborative tool in learning, designing and developing.

First of all, to provide a bit of context, prior to this project I did not have any experience in physical simulation, GPU programming or even coding in C++. I did not study physics past the AS level, and so, diving into multiple heavily physics-based research papers with complex equation after complex equation during the beginning of this project was fairly overwhelming, and made me question whether or not I had selected a project suitable for completion. I initially turned to Claude to help digest these research papers, and after being inspired with a bit more confidence, decided to use it for the development process. However, instead of just "asking AI to do it for me", I was determined to use it in a collaborative and constructive way, meaning that I would have a say in every stage of the design, as well being able to learn about how and why it would make its development choices.

Initially, to build some familiarity with developing in C++ and physical modelling, a simple CPU based prototype of DrumForge was made. The development of this prototype also helped to solidify implementation choices (physical modelling techniques, libraries etc.), as well as to somewhat rectify the project scope (initially the 3D modelling was going to use external 3D modelling programs) [a33]. This initial stage also gave me some realisations on some of the pitfalls to be wary of when developing with LLMs, as well as helped me to develop techniques to avoid these pitfalls when going on to work on the GPU accelerated system.

By default, Claude (and in my experience pretty much every LLM) was very "trigger happy" when it came to writing code, which it would do in as much detail, with as many features as it could, as early as possible, very different to traditional development, which is a much slower, gradual iterative process. Claude provides access to a feature called Projects, where you can provide specific project instructions, instructions that shape how Claude operates, and knowledge, information that will be utilised in the context of each conversation had within said project (when speaking in terms of LLMs context refers to the information that has been discussed with the model after its initial baseline state). Both of these features became paramount for development. After some time developing with Claude, continually asking it to slow down and take things one step at a time and asking it to discuss things thoroughly before starting an implementation [a34], I had it analyse my preferred working style and generate the following project instructions:

Here are the key aspects of our working style that we'll keep in mind:

- *Educational approach: we appreciate learning about the technical decisions and trade-offs at each step.*
- *Purposeful planning: we like to discuss the next steps and confirm the direction before writing new code.*
- *Modular design: we prefer building systems with discrete, interchangeable components rather than monolithic designs.*
- *Architectural clarity: we value clear separation of concerns and well-defined interfaces between components.*
- *Incremental implementation: we prefer to build one component at a time, ensuring each part works before moving to the next.*
- *Thorough understanding: we like detailed explanations of each part of the code before implementation, understanding the purpose and functionality of each section.*
- *Code walkthrough: we benefit from line-by-line explanations of complex implementations to ensure complete understanding.*
- *Systematic verification: After implementing each component, we test to verify it works correctly before proceeding.*

When collaborating, you'll continue to provide explanations for each component we develop, discuss implementation strategies before coding, and ensure we verify each step works correctly before moving forward.

A friend of mine likes to describe LLMs as "toddlers that know everything" (which I think is a very fitting description), and after applying these instructions to Claude, our conversations became much more "adult" - more focused and thoroughly discussed. This made design and development much more meaningful, and also enabled me to learn a huge amount in a relatively short period of time. Our development cycle began to follow a routine:

1. Start work on feature/requirement
2. Discuss how the feature fits within the system architecture
3. Learn about conventional methods to implement the feature
4. Discuss Claude's ideas for the implementation plan
5. Provide feedback/suggest a more suitable implementation plan
6. Generate and review high level pseudocode for implementation
7. Generate code for the implementation
8. Review code/provide feedback
9. Make sure implementation compiles
10. Test implementation and fix bugs until implementation is correct
11. Move on to next feature

In this development cycle, it was almost as if I took on the role of a senior developer, designing and guiding the development, while Claude was the junior developer actually carrying out the implementation [a35].

Another issue I ran into with development was being rate limited - not being able to use Claude for a period of time after using it too much. This became a problem as our conversations got longer and longer, due to the following reasons:

1. **Conversation context**: Claude needs to process the entire conversation history with each new message, which consumes more computational resources as the conversation grows
2. **Token accumulation**: Each message adds to the total token count of the conversation, and longer conversations require more tokens to process
3. **Response generation**: Generating responses to complex conversations with extensive context requires more computational resources

This creates a compounding effect - as the conversation gets longer, each new interaction "costs" more against your rate limits, causing you to reach them faster than you would with multiple shorter conversations. The problem with having multiple shorter conversations is that the context that you have built up in the previous conversation is forgotten about each time you start a new one - a problem when trying to develop a complex yet cohesive system. I managed to find a work around to this issue using a simple chrome extension [17] that would let me export entire conversations to txt files, which I could then provide to Claude at the start of a new conversation, effectively treating an entire conversation as a single message. This in combination with the ability to provide a Github repository as project knowledge allowed me to seamlessly bridge the gap between each conversation, continuing development right from where we left off [a36]. Therefore I was able to use multiple shorter conversations, allowing me to work on DrumForge for extended periods of time without being rate limited.

Overall, I think this is a good demonstration of how LLMs can be valuable educational tools when approached with the right methodology, and being able to go from having zero knowledge to building and understanding complex systems in a matter of a couple of weeks/months is certainly a valuable asset.

Critical Reflection and Conclusion

DrumForge successfully implements GPU-accelerated drum simulation with real-time visualization and audio generation. The membrane physics model accurately captures wave propagation as seen in real drums, and the body resonator adds tonal characteristics based on different materials. However, the following technical limitations should be acknowledged. While the audio output captures fundamental drum characteristics, the generated sounds aren't yet at the standard of those achieved by recording real acoustic drums. The membrane model, though physically accurate for the basic wave equation, omits real-world nonlinearities that affect authentic drum sounds. Similarly, the modal synthesis approach for the body offers a reasonable approximation but lacks the precision that a full physical simulation might provide.

The system architecture proved to be highly extensible, with the component-based design facilitating independent development and testing of different drum elements. This design enables future enhancements should they be implemented, such as additional components like the resonant membrane and air cavity.

From a performance perspective, DrumForge achieved the goal of real-time simulation, maintaining consistent frame rates even with higher-resolution grids. This validates the choice of GPU acceleration and the CUDA implementation approach. The FDTD method for membrane simulation proved to be well-suited for parallel computation.

From a development perspective, collaborating with Claude presented a unique balance of advantages and challenges. The LLM enabled rapid knowledge acquisition across multiple specialized domains (physical modeling, GPU programming, C++, OpenGL), compressing a learning timeline that would typically require months or years. However, Claude's tendency to rush to implementation provided some early hurdles in the development process. Establishing clear project instructions and maintaining a structured development workflow proved crucial for effective collaboration. The effectiveness of this approach demonstrates both the potential of AI as a technical collaborator and the importance of maintaining human direction in the development process.

This project has been transformative for my development as both a programmer and a researcher. Prior to undertaking DrumForge, I had no experience with physical simulation, GPU programming, or C++. The progression from feeling overwhelmed by physics-heavy research papers to successfully implementing a complex GPU-accelerated simulation system represents significant growth in my technical capabilities. However, I believe the most valuable skill I've developed is architectural thinking. Working through the design and implementation of DrumForge's component-based system forced me to consider how different parts of a complex application should interact, establishing clear interfaces while maintaining a separation of concerns. This approach to software design—creating modular, reusable components with well-defined responsibilities—is something I'll carry forward into future projects.

Looking ahead, DrumForge provides a solid foundation for future work in physical modeling synthesis. The ultimate goal would be creating a comprehensive drum simulation system that produces sounds indistinguishable from recorded acoustic drums while offering the flexibility and dynamic response that only physics-based models can provide.

In conclusion, this project demonstrates that GPU-accelerated physical modeling could indeed make previously impractical simulations viable for interactive use. By balancing the level of simulation fidelity with computational efficiency, DrumForge achieves real-time performance while producing somewhat recognizable drum sounds. While there is vast room for improvement in audio quality, the current implementation successfully demonstrates the core concept and provides a platform for continued development. Another key takeaway from this project for me is the demonstration of how LLMs, when used effectively, can accelerate technical learning and complex software development, enabling a much wider range of people to tackle ambitious projects that would have been out of reach just a few years ago.

As a final note, if you'd like to run DrumForge yourself, or view the entire codebase, the repository can be cloned from https://github.com/danielsuss/Drumforge. Instructions can be found in the README regarding setup.

References

[1] The University of Edinburgh
NESS / Next Generation Sound Synthesis [online]
Available at: https://www.ness.music.ed.ac.uk/

[2] Stefan Bilbao, Craig Webb (2012)
Timpani Drum Synthesis in 3D on GPGPUS [online]
Available at: https://www.ness.music.ed.ac.uk/wp-content/uploads/2012/06/timpani.pdf

[3] NVIDIA
CUDA [online]
Available at: https://developer.nvidia.com/cuda-toolkit

[4] Anthropic
Claude [online]
Available at: https://claude.ai/

[5] Anders Gärder (2005)
Physical Modeling of Percussion Instruments [online]
Available at: https://publications.lib.chalmers.se/records/fulltext/5768.pdf

[6] Yamaha
Musical Instrument Guide - The Structure of the Drum [online]
Available at: Yamaha Musical Instrument Guide

[7] TAKDRUM (2023)
The Physics of Percussion: Understanding Sound Generation and Resonance [online]
Available at: https://takdrumhandpan.com

[8] Enda Zhang, Gopal Gupta, Charles Greif, Andrew Paplinski (2019)
An Efficient Modal-Based Approach Towards Guzheng Sound Synthesis [online]
Available at: https://arxiv.org/pdf/1910.05447

[9] Omar Cornut
Dear ImGui [online]
Available at: https://github.com/ocornut/imgui

[10] R. Courant, K. Friedrichs, H. Lewy (1928)
On the Partial Difference Equations of Mathematical Physics [online]
Available at: https://archive.org/details/onpartialdiffere00cour/mode/2up?view=theater

[11] KHRONOS GROUP
OpenGL [online]
Available at: https://www.opengl.org/

[12] GLFW
GLFW [online]
Available at: https://github.com/glfw/glfw

[13] Milan Ikets, Marcello Magalon
GLEW [online]
Available at: https://github.com/nigels-com/glew

[14] Microsoft
Windows Subsystem for Linux [online]
Available at: https://learn.microsoft.com/en-us/windows/wsl/about

[15] Image Line
Fruity Parametric EQ 2 [online]
Available at: https://www.image-line.com/fl-studio-learning/fl-studio-online-manual

[16] Giovanni Hidalgo, Latin Percussion (YouTube) (2013)
In the Studio with Giovanni Hidalgo (Timbale Solo #1) [online]
Available at: https://youtu.be/ths6YHITLTU?si=QAFYHNywrEZFJBLi

[17] epubsreadsoft
Claude to PDF [online]
Available at: https://chromewebstore.google.com/detail/claude-to-pdf/

Appendix

[a1] **SimulationManager's advance()** method *Return to report*

```
void SimulationManager::advance(float deltaTime) {
    if (params.pauseSimulation) {
        return;
    }

    // Apply time scaling
    float scaledDeltaTime =  deltaTime * params.timeScale;

    // Accumulate time for fixed-step updates
    accumulatedTime +=   scaledDeltaTime;

    // Calculate a stable timestep
    float stableTimestep =  calculateStableTimestep();

    // Update components with fixed time steps
    while (accumulatedTime >=   stableTimestep) {
        // First update all components
        for (auto& component : components) {
            component->update(stableTimestep);
        }

        // Then process all couplings
        for (auto& coupling : couplings) {
            // Exchange data between coupled components
            CouplingData data =  coupling.source->getInterfaceData();
            coupling.target->setCouplingData(data);
        }

        // Update audio channel values for all components
        for (auto& component : components) {
            if (component->hasAudio())  {
                component->updateAudio(stableTimestep);
            }
        }

        // Process mixed audio through AudioManager
        AudioManager& audioManager =  AudioManager::getInstance();
        if (audioManager.getIsRecording() && audioManager.getUseChannelMixing()) {
            audioManager.processMixedAudioStep(stableTimestep);
        }

        // Update simulation time
        currentTime +=   stableTimestep;
        accumulatedTime -=  stableTimestep;
    }
}
```

[a2] **CudaMemoryManager** allocating GPU memory buffers for the membrane *Return to report*

```
// In MembraneComponent::initialize()
void MembraneComponent::initialize() {
    // Get reference to memory manager
    CudaMemoryManager& memoryManager =  CudaMemoryManager::getInstance();

    // Allocate all required CUDA buffers
    int totalElements =  membraneWidth * membraneHeight;
    d_heights =  memoryManager.allocateBuffer<float> (totalElements);
    d_prevHeights =  memoryManager.allocateBuffer<float> (totalElements);
    d_velocities =  memoryManager.allocateBuffer<float> (totalElements);
    d_circleMask =  memoryManager.allocateBuffer<int> (totalElements);

    // Initialize the circular mask
    initializeCircleMask(d_circleMask-> get(), *kernelParams);

    // Reset the membrane to its initial flat state
    resetMembrane(d_heights-> get(), d_prevHeights-> get(), d_velocities-> get(),
                  d_circleMask-> get(), *kernelParams);

    // Set up OpenGL interop for visualization if available
    if (memoryManager.isGLInteropSupported()) {
        // Setup for visualization...
    }
}
```

[a3] **renderComponents()** method from **VisualizationManager** *Return to report*

```
void VisualizationManager::renderComponents(SimulationManager& simManager) {
    // Get all components from the simulation manager
    const auto& components =  simManager.getComponents();

    // Render each visualizable component
    for (const auto& component : components) {
        if (component && component-> isVisualizable()) {
            // Initialize visualization for this component if needed
            initializeComponentVisualization(component);

            // Let the component prepare its visualization data
            component-> prepareForVisualization();

            // Let the component visualize itself using this manager
            component-> visualize(*this);
        }
    }
}
```

[a4] **renderMainMenu()** method from **GuiManager** *Return to report*

```
void GUIManager::renderMainMenu() {
    // Set up centered window
    ImGuiIO& io =  ImGui::GetIO();
    ImVec2  windowSize =  ImVec2 (5,05 0);
    ImVec2  windowPos =  ImVec2 ((io.DisplaySize.x - windowSize.x) * 0 .5,f
                            (io.DisplaySize.y - windowSize.y) * 0 .5);f

    ImGui::SetNextWindowPos(windowPos, ImGuiCond_Always);
    ImGui::SetNextWindowSize(windowSize, ImGuiCond_Always);

    ImGui::Begin(" DrumForge - Setup", nullptr,
                ImGuiWindowFlags_NoResize | ImGuiWindowFlags_NoMove |
                ImGuiWindowFlags_NoCollapse |
ImGuiWindowFlags_NoBringToFrontOnFocus);

    // Title and description
    ImGui::TextColored(ImVec4 (1 .0,f0 .8,f0 .0,f1 .0);f " DrumForge Initial
Configuration");
    ImGui::Spacing();
    ImGui::Spacing();

    // Grid Configuration Section
    ImGui::TextColored(ImVec4 (0 .4,f0 .8,f1 .0,f1 .0);f " Grid Configuration");
    ImGui::Separator();

    // Grid Size Controls
    ImGui::SliderInt(" Grid Size X", &configState.gridSizeX, 6 4, 2 5);6
    ImGui::SliderInt(" Grid Size Y", &configState.gridSizeY, 6 4, 2 5);6
    ImGui::SliderInt(" Grid Size Z", &configState.gridSizeZ, 6 4, 2 5);6

    // Cell Size Control
    ImGui::SliderFloat(" Cell Size", &configState.cellSize, 0 .0 5, 0 .2,f" %.2);f"

    ImGui::Spacing();
    ImGui::Spacing();

    // Membrane Configuration Section
    ImGui::TextColored(ImVec4 (0 .4,f1 .0,f0 .8,f1 .0);f " Membrane Configuration");
    ImGui::Separator();
    // Method continues...
```

[a5] **processMembraneClick()** method from **InputHandler** *Return to report*

```
// Process mouse click on membrane
void InputHandler::processMembraneClick(double mouseX, double mouseY) {
    // Check if camera and membrane are connected
    if (!camera) {
        return;
    }

    auto membrane =  membraneComponent.lock();
    if (!membrane) {
        return;
    }

    // Calculate ray from camera through mouse point
    auto [rayOrigin, rayDirection] =  screenToRay(mouseX, mouseY);

    // Calculate membrane plane (at z = 0)
    glm::vec3 planeNormal(0 .0,f0 .0,f1 .0)f;
    glm::vec3 planePoint(0 .0,f0 .0,f0 .0)f; // Center of the membrane

    // Check if ray is parallel to the plane
    float denom =  glm::dot(rayDirection, planeNormal);
    if (std::abs(denom) < 0 .0 0 0) 1{ f
        return; // Ray is parallel to the plane
    }

    // Calculate intersection
    float t =  glm::dot(planePoint - rayOrigin, planeNormal)/ denom;
    if (t < =0 .0)f{
        return; // Intersection is behind the camera
    }

    // Calculate intersection point
    glm::vec3  intersectionPoint =  rayOrigin +  t *ayDirection;

    // Convert to normalized membrane coordinates (0 to 1)
    float membraneRadius =  membrane-> getRadius();
    float clickX =  (intersectionPoint.x +  membraneRadius) /(2 .0 * membraneRadius);
    float clickY =  (intersectionPoint.y +  membraneRadius) /(2 .0 * membraneRadius);

    // Check if click is within the membrane radius
    float centerX = 0 .5;f
    float centerY = 0 .5;f
    float dx =  clickX - centerX;
    float dy =  clickY - centerY;
    float distSquared =  dx * dx +  dy * dy;

    if (distSquared < =0 .2 5) { // 0.5 * 0.5 = 0.25 is the squared radius in normalized coordinates
        // Apply impulse with a fixed strength
        membrane-> applyImpulse(clickX, clickY,1 .0)f;
        std::cout < <" Applied impulse at membrane coordinates ("< <  clickX <"<, "< <  clickY <"<)"< <
std::endl;
    }
}
```

[a6] **processAudioStep()** method from **AudioManager** *Return to report*

```
void AudioManager::processAudioStep(float simulationTimeStep, float
currentSampleValue) {
    if (!isRecording) return;

    // Update interpolation state with new value
    interpolationState.lastValue =  interpolationState.currentValue;
    interpolationState.currentValue =  currentSampleValue;

    // Update time tracking
    double previousTime =  accumulatedTime;
    accumulatedTime + =  simulationTimeStep;

    // For the first sample, just store the value without interpolation
    if (recordBuffer.empty()) {
        recordBuffer.push_back(currentSampleValue);
        interpolationState.lastSampleTime = 0 .0;
        return;
    }

    interpolationState.lastSampleTime =  previousTime;

    // Calculate how many audio samples we need for this time step
    double nextSampleTime =  (recordBuffer.size() * sampleInterval);

    // Generate all audio samples needed to cover this simulation step
    while (nextSampleTime < =  accumulatedTime) {
        // Get interpolated sample at exactly the right time point
        float sample =  getInterpolatedSample(nextSampleTime);

        // Add to record buffer
        recordBuffer.push_back(sample);

        // Move to next sample time
        nextSampleTime =  (recordBuffer.size() * sampleInterval);
    }
}
```

[a7] Main Simulation Loop Code Return to report

```
// Main simulation loop
while (!visManager.shouldClose()) {
    // Process user input
    inputHandler.processInput(deltaTime);

    // Advance simulation
    simManager.advance(deltaTime);

    // Render visualization
    visManager.beginFrame();
    visManager.renderComponents(simManager);

    // Update GUI
    guiManager.beginFrame();
    guiManager.renderGUI(simManager, membrane);
    guiManager.renderFrame();

    // Finish frame
    visManager.endFrame();
}
```

[a8] **MembraneComponent** Constructor Return to report

```
MembraneComponent::MembraneComponent(const std::string& name, float radius, float tension,
float damping)
    : memoryManager(CudaMemoryManager::getInstance())
    , simulationManager(SimulationManager::getInstance())
    , tension(tension)
    , damping(damping)
    , radius(radius)
    , membraneWidth(0)
    , membraneHeight(0)
    , cellSize(0.0f)
    , vaoId(0)
    , eboId(0)
    , name(name)
    , kernelParams(new MembraneKernelParams())
    , audioSamplePoint(0.5f, 0.5f)  // Legacy - default to center
    , audioGain(1.0f)
    , masterGain(1.0f)
    , useMixedOutput(true)
    , pendingImpulse{0.0f, 0.0f, 0.0f, false} {

    std::cout << "MembraneComponent  '"<<   name <<"'    created"<< std::endl;

    // Setup default microphone at the center
    setupSingleCenterMicrophone();
}
```

[a9] **BodyComponent** Constructor *Return to report*

```
BodyComponent::BodyComponent(
    const std::string& name,
    float radius,
    float height,
    float thickness,
    const std::string& material)
    : simulationManager(SimulationManager::getInstance())
    , radius(radius)
    , height(height)
    , thickness(thickness)
    , damping(0 .0 5 ) f// Default damping
    , masterGain(0 .6)f // Default master gain
    , material(material)
    , name(name) {

    // Initialize material presets
    initializeMaterialPresets();

    // Set up filters for the selected material
    setupFiltersForMaterial(material);

    // Adjust filters based on dimensions
    adjustFiltersForDimensions();

    std::cout < <" BodyComponent '  ‘< <  name <"<'   created‘< <std::endl;
    std::cout < <"   Dimensions: radius= ‘< <  radius
              < <" , height= ‘< <  height
              < <" , thickness= ‘< <  thickness <std::endl;
    std::cout < <"   Material: "< <  material <std::endl;
}
```

[a10] **ComponentInterface** Abstract Base Class *Return to report*

```
class ComponentInterface {
public:
    virtual ~ComponentInterface() = default;

    // Core simulation methods
    virtual void initialize() = 0;
    virtual void update(float timestep) = 0;

    // Component coupling
    virtual CouplingData getInterfaceData() = 0;
    virtual void setCouplingData(const CouplingData& data) = 0;

    // Visualization methods
    virtual bool isVisualizable() const { return false; }
    virtual void prepareForVisualization() = 0;
    virtual void visualize(VisualizationManager& visManager) = 0;

    // Audio methods
    virtual bool hasAudio() const { return false; }
    virtual void updateAudio(float timestep) = 0;

    // Utility methods
    virtual std::string getName() const = 0;
    virtual float calculateStableTimestep() const = 0;
};
```

[a11] **CouplingData** Structure *Return to report*

```
struct CouplingData {
    // Impact data
    float impactStrength = 0 .0;f
    glm::vec2  impactPosition =  glm::vec2 (5,f0 .5);f
    bool hasImpact = false;

    // Other coupling data
    std::vector<float>  resonanceValues;
};
```

[a12] **getInterfaceData()** and **setCouplingData**() Methods *Return to report*

```
// In MembraneComponent
CouplingData MembraneComponent::getInterfaceData() {
    CouplingData data;
    data.hasImpact =  couplingImpulsePending;
    data.impactStrength =  pendingImpulse.strength;
    data.impactPosition =  glm::vec2 (pendingImpulse.x, pendingImpulse.y);

    // Reset the coupling flag after sending the data
    if (data.hasImpact) {
        couplingImpulsePending = false;
    }

    return data;
}

// In BodyComponent
void BodyComponent::setCouplingData(const CouplingData& data) {
    if (data.hasImpact) {
        // Store the impact strength for audio processing
        lastImpactStrength =  data.impactStrength;
        impactOccurred = true;
        impactDecay = 0 .1;f // 100ms decay time

        // Temporarily increase resonance based on impact
        float resonanceBoost =  data.impactStrength *0 .1;f
        for (auto& filter : filters) {
            filter.resonance = std::min(
                filter.resonance +  resonanceBoost,0 .9 9)9 f
        }
    }
}
```

[a13] **calculateStableTimestep()** method from the **SimulationManager** *Return to report*

```
float SimulationManager::calculateStableTimestep() const {
    float minTimestep = std::numeric_limits<float> ::max();

    // Find the smallest stable timestep among all components
    for (const auto& component : components) {
        float componentTimestep =  component-> calculateStableTimestep();
        minTimestep = std::min(minTimestep, componentTimestep);
    }

    return minTimestep;
}
```

[a14] **allocateGridRegion()** method from the **SimulationManager** *Return to report*

```
// Allocate a region of the global grid for a component
GridRegion SimulationManager::allocateGridRegion(
    std::shared_ptr<ComponentInterface>   component,
    int startX, int startY, int startZ,
    int sizeX, int sizeY, int sizeZ) {

    // Validate input
    if (startX < 0 || startY < 0 || startZ < 0 ||
        startX +  sizeX >  params.gridSizeX ||
        startY +  sizeY >  params.gridSizeY ||
        startZ +  sizeZ >  params.gridSizeZ) {
        std::cerr << "Error:  Requested grid region is outside global grid bounds"<<
std::endl;
        return GridRegion(); // Return empty region on error
    }

    // Check for overlap with existing regions
    for (const auto& region : allocatedRegions) {
        // Simple check: if this new region's bounding box overlaps with an existing region
        bool overlaps =  !(
            startX +  sizeX <=   region.startX ||
            startY +  sizeY <=   region.startY ||
            startZ +  sizeZ <=   region.startZ ||
            startX >=   region.startX +  region.sizeX ||
            startY >=   region.startY +  region.sizeY ||
            startZ >=   region.startZ +  region.sizeZ
        );

        if (overlaps) {
            std::cerr << "Error:  Requested grid region overlaps with an existing region"<<
std::endl;
            return GridRegion(); // Return empty region on error
        }
    }

    // Create the new region
    GridRegion newRegion;
    newRegion.startX =  startX;
    newRegion.startY =  startY;
    newRegion.startZ =  startZ;
    newRegion.sizeX =  sizeX;
    newRegion.sizeY =  sizeY;
    newRegion.sizeZ =  sizeZ;
    newRegion.owner =  component;

    // Add to allocated regions
    allocatedRegions.push_back(newRegion);

    return newRegion;
}
```

[a15] Simulation Initialization Process

Return to report

```
// Get configuration parameters from GUI
int gridSizeX =  guiManager.getConfigGridSizeX();
int gridSizeY =  guiManager.getConfigGridSizeY();
int gridSizeZ =  guiManager.getConfigGridSizeZ();
float cellSize =  guiManager.getConfigCellSize();
float membraneRadius =  guiManager.getConfigRadius();
float membraneTension =  guiManager.getConfigTension();
float membraneDamping =  guiManager.getConfigDamping();
float timeScale =  guiManager.getConfigTimeScale();

// Initialize the simulation
simManager.initialize();

// Apply configuration parameters
drumforge::SimulationParameters params =  simManager.getParameters();
params.timeScale =  timeScale;
params.gridSizeX =  gridSizeX;
params.gridSizeY =  gridSizeY;
params.gridSizeZ =  gridSizeZ;
params.cellSize =  cellSize;
simManager.updateParameters(params);

// Create the membrane component
membrane = std::make_shared<drumforge::MembraneComponent>(
    "Drumhead",
    membraneRadius,   // Radius
    membraneTension,  // Tension
    membraneDamping   // Damping
);

// Add membrane to simulation
simManager.addComponent(membrane);

// Initialize the membrane component
membrane->initialize();

// Create the body resonator
float bodyRadius =  membrane->getRadius();
float bodyHeight =  bodyRadius * guiManager.getConfigBodyHeight();
float bodyThickness =  bodyRadius * guiManager.getConfigBodyThickness();
std::string bodyMaterial =  guiManager.getConfigBodyMaterial();

bodyResonator = std::make_shared<drumforge::BodyComponent>(
    "DrumShell",
    bodyRadius,    // Same radius as membrane
    bodyHeight,    // Height from config
    bodyThickness, // Thickness from config
    bodyMaterial   // Material from config
);

// Add body resonator to simulation
simManager.addComponent(bodyResonator);

// Initialize the body resonator
bodyResonator->initialize();

// Set up coupling from membrane to body
simManager.setupCoupling(membrane, bodyResonator);
```

[a16] **initializeCircleMaskKernel()** kernel *Return to report*

```
__global__ void initializeCircleMaskKernel(int* mask, const MembraneKernelParams
params) {
    // Calculate thread's position in the membrane grid
    int x =  blockIdx.x * blockDim.x +  threadIdx.x;
    int y =  blockIdx.y * blockDim.y +  threadIdx.y;

    // Check bounds
    if (x > =  params.membraneWidth || y > =  params.membraneHeight) {
        return;
    }

    // Calculate distance from center
    float dx =  x - params.centerX;
    float dy =  y - params.centerY;
    float distSquared =  dx * dx +  dy * dy;

    // Set mask value (1 if inside circle, 0 if outside)
    mask[getIndex(x, y, params.membraneWidth)] =
        (distSquared < =  params.radiusSquared) ? 1 : 0;
}
```

[a17] **updateMembraneKernel()** kernel *Return to report*

```
__global__ void updateMembraneKernel(float* heights, float* prevHeights, float* velocities,
                                     const int* mask, const MembraneKernelParams params,
                                     float timestep) {
    // Calculate thread's position in the membrane grid
    int x =  blockIdx.x * blockDim.x +  threadIdx.x;
    int y =  blockIdx.y * blockDim.y +  threadIdx.y;

    // Check bounds and ensure we're at least one cell away from the boundary
    if (x < 1 || x > =  params.membraneWidth 1 ||
        y < 1 || y > =  params.membraneHeight 1) {
        return;
    }

    // Get linear index into arrays
    int idx =  getIndex(x, y, params.membraneWidth);

    // Only process points within the circular membrane
    if (mask[idx]) {
        // Get the current height
        float current =  heights[idx];
        float prev =  prevHeights[idx];

        // Get neighboring heights
        float left =  heights[getIndex(x1, y, params.membraneWidth)];
        float right =  heights[getIndex(x1, y, params.membraneWidth)];
        float up =  heights[getIndex(x, y1, params.membraneWidth)];
        float down =  heights[getIndex(x, y1, params.membraneWidth)];

        // Calculate the Laplacian (discretized ∇²u term)
        float laplacian =  left +  right +  up +  down4 .0 f current;

        // Calculate the coefficient for the wave equation
        float c =  params.waveSpeed;
        float coef =  c * c * timestep * timestep / (dx * dx);

        // Standard leapfrog update for wave equation:
        // u(t+dt) = 2*u(t) - u(t-dt) + c²*(dt/dx)²*∇²u(t)
        float new_height = 2 .0 f current - prev +  coef * laplacian;

        // Apply damping term proportional to first time derivative of u
        float velocity_approx =  (current - prev) / timestep;
        new_height -=  params.damping * velocity_appox * timestep;

        // Update the velocity using central difference approximation
        velocities[idx] =  (new_height - prev) / 2 .0 f timestep);

        // Store the new height in the output array
        prevHeights[idx] =  current;
        heights[idx] =  new_height;
    }
}
```

[a18] **applyImpulseKernel()** kernel *Return to report*

```
__global__ void applyImpulseKernel(float* heights, const int* mask,
                                   const MembraneKernelParams params,
                                   float x, float y, float strength) {
    // Calculate thread's position in the membrane grid
    int ix =  blockIdx.x * blockDim.x +  threadIdx.x;
    int iy =  blockIdx.y * blockDim.y +  threadIdx.y;

    // Convert normalized position [0,1] to grid coordinates
    int centerX = static_cast<int> (x * params.membraneWidth);
    int centerY = static_cast<int> (y * params.membraneHeight);

    // Apply impulse only within the membrane and impulse radius
    int idx =  getIndex(ix, iy, params.membraneWidth);
    if (mask[idx]) {
        // Calculate distance from the impulse center
        float dx =  ix - centerX;
        float dy =  iy - centerY;
        float distSquared =  dx * dx +  dy * dy;

        // Size of the impulse (radius squared in grid cells)
        const float impulseRadiusSquared = 2 5 .0 f// 5² = 25

        // Apply a Gaussian-shaped impulse within the radius
        if (distSquared < =  impulseRadiusSquared) {
            float dist =  sqrtf(distSquared);
            float falloff =  expf(-dist*dist / (impulseRadiusSquared4 .0));

            // Apply the impulse - negative for downward strike
            heights[idx] -=  strength * falloff;
        }
    }
}
```

[a19] **CudaBuffer** class for memory allocation *Return to report*

```
template<typename T>
class CudaBuffer {
private:
    T* devicePtr;
    size_t elementCount;
    size_t byteSize;

public:
    // Allocate memory on device
    void allocate(size_t count) {
        release();
        elementCount =  count;
        byteSize =  count *sizeof(T);
        CUDA_CHECK(cudaMalloc(&devicePtr, byteSize));
    }

    // Copy data from host to device
    void copyFromHost(const T* hostData) {
        CUDA_CHECK(cudaMemcpy(devicePtr, hostData, byteSize,
                              cudaMemcpyHostToDevice));
    }

    // Copy data from device to host
    void copyToHost(T* hostData) const {
        CUDA_CHECK(cudaMemcpy(hostData, devicePtr, byteSize,
                              cudaMemcpyDeviceToHost));
    }
};
```

[a20] **calculateOptimalBlockSize()** method to organize threads efficiently *Return to report*

```
dim3 calculateOptimalBlockSize(int dataWidth, int dataHeight) {
    // Default block size for 2D grid (8x8 = 64 threads per block)
    int blockSizeX = 8;
    int blockSizeY = 8;

    // For very small membranes, use smaller blocks
    if (dataWidth < 16 || dataHeight < 16) {
        blockSizeX = 4;
        blockSizeY = 4;
    }
    // For larger membranes, use larger blocks to increase parallelism
    else if (dataWidth >= 128 && dataHeight >= 128) {
        blockSizeX = 16;
        blockSizeY = 16;
    }

    return dim3(blockSizeX,  blockSizeY);
}
```

[a21] **initGLFW()** method to initialise OpenGL context through GLFW *Return to report*

```
bool VisualizationManager::initGLFW() {
    // Set error callback
    glfwSetErrorCallback(errorCallback);

    // Initialize GLFW
    if (!glfwInit()) {
        std::cerr < <" Failed to initialize GLFW"< <std::endl;
        return false;
    }

    // Set window hints for OpenGL version
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

    // Important for CUDA interop
    glfwWindowHint(GLFW_OPENGL_DEBUG_CONTEXT, GLFW_TRUE);
    glfwWindowHint(GLFW_VISIBLE, GLFW_TRUE);

    return true;
}
```

[a22] **initGLEW()** method to initialize GLEW *Return to report*

```
bool VisualizationManager::initGLEW() {
    // Initialize GLEW
    glewExperimental =  GL_TRUE;
    GLenum err =  glewInit();
    if (err !=  GLEW_OK) {
        std::cerr < <" GLEW Error: "< <  glewGetErrorString(err) <<std::endl;
        return false;
    }

    // Log OpenGL information
    std::cout < <" OpenGL version: "< <  glGetString(GL_VERSION) <<std::endl;
    std::cout < <" OpenGL renderer: "< <  glGetString(GL_RENDERER) <<std::endl;

    return true;
}
```

[a23] Custom GLSL shaders and membrane colouring algorithm Return to report

```
const char* vertexShaderSource = R" (
    # version 3 3 0  core
    layout (location =  0 ) in vec3  aPos;

    uniform mat4  model;
    uniform mat4  view;
    uniform mat4  projection;
    uniform vec3  color;

    out vec3  fragColor;

    void main() {
        gl_Position =  projection * view * model * vec4 (aPos, 1 .0 );

        // Color based on displacement (height)
        float displacement =  aPos.z;
        if (displacement >  0 .0 ) {
            // Red for positive displacement
            fragColor =  mix(color, vec3 (1 .0 , 0 .0 ),0displacement * 5 .0 );
        } else {
            // Blue for negative displacement
            fragColor =  mix(color, vec3 (0 .0 , 0 .0 ),1-displacement * 5 .0 );
        }
    }
)";

const char* fragmentShaderSource = R" (
    # version 3 3 0  core
    in vec3  fragColor;
    out vec4  FragColor;

    void main() {
        FragColor =  vec4 (fragColor, 1 .0 );
    }
)";
```

[a24] **initializeVisualization()** method of the **MembraneComponent** *Return to report*

```
void MembraneComponent::initializeVisualization(VisualizationManager& visManager) {
    // Initialize vertices for a simple grid
    std::vector<float>  vertices(membraneWidth * membraneHeight *3);
    for (int y = 0; y <  membraneHeight; y+ + ) {
        for (int x = 0; x <  membraneWidth; x+ + ) {
            int index =  (y * membraneWidth +  x) 3;

            // Calculate position in world space
            float xPos =  (x - membraneWidth /2 .0)f* cellSize;
            float yPos =  (y - membraneHeight /2 .0)f* cellSize;
            float zPos = 0 .0;f// Initially flat

            vertices[index] =  xPos;
            vertices[index + 1] =  yPos;
            vertices[index + 2] =  zPos;
        }
    }

    // Create a vertex buffer with initial data
    unsigned int vbo =  visManager.createVertexBuffer(
        vertices.size() * sizeof(float),
        vertices.data()
    );

    // Create Vertex Array Object
    vaoId =  visManager.createVertexArray();

    // Configure vertex attributes
    visManager.configureVertexAttributes(vaoId, vbo, 0, 3, 3 * sizeof(float), 0);
}
```

[a25] **prepareForVisualization()** method of the **MembraneComponent** *Return to report*

```
void MembraneComponent::prepareForVisualization() {
    // Using CUDA-OpenGL interop for direct GPU-to-GPU update
    if (g_enableCudaGLInterop && glInteropBuffer && glInteropBuffer-> isRegistered()) {
        // Map the buffer for CUDA access
        void* devicePtr =  glInteropBuffer-map();
        float3 * vertices =static_cast< float3 *> (devicePtr);

        // CUDA kernel updates vertices directly on the GPU
        updateVisualizationVertices(
            vertices,
            d_heights-> get(),
            d_circleMask-> get(),
            *kernelParams,
            visualizationScale
        );

        // Unmap the buffer when done
        glInteropBuffer-> unmap();
    }
    else {
        // Fallback to CPU-based update if interop is not available
        const auto& heights =  getHeights();
        std::vector<float>  vertices(membraneWidth * membraneHeight *3);

        for (int y = 0; y <  membraneHeight; y+ + ) {
            for (int x = 0; x <  membraneWidth; x+ + ) {
                int index =  (y * membraneWidth +  x) 3;
                int dataIndex =  y * membraneWidth +  x;

                // Position in world space
                float xPos =  (x - membraneWidth /2 .0)f* cellSize;
                float yPos =  (y - membraneHeight /2 .0)f* cellSize;
                float zPos = 0 .0;f

                // Apply height if inside the circular membrane
                if (isInsideCircle(x, y)) {
                    zPos =  heights[dataIndex] * visalizationScale;
                }

                vertices[index] =  xPos;
                vertices[index + 1] =  yPos;
                vertices[index + 2] =  zPos;
            }
        }

        // Update the OpenGL buffer with new vertex data
        glBindVertexArray(vaoId);
        glGetVertexAttribIuiv(0, GL_VERTEX_ATTRIB_ARRAY_BUFFER_BINDING, &vbo);
        glBindBuffer(GL_ARRAY_BUFFER, vbo);
        glBufferSubData(GL_ARRAY_BUFFER, 0, vertices.size() * sizeof(float), vertices.data());
        glBindBuffer(GL_ARRAY_BUFFER, 0);
        glBindVertexArray(0);
    }
}
```

[a26] **initializeVisualization()** method of the **BodyComponent** *Return to report*

```
void BodyComponent::initializeVisualization(VisualizationManager& visManager) {
    // Create vertices for a cylindrical shell
    std::vector<float>  vertices;
    std::vector<unsigned int>  indices;

    // Parameters for the cylinder
    const int numSegments = 3 6;  // Number of segments around the circumference
    const int numRings = 8;      // Number of rings along the height

    // Calculate vertices
    for (int ring = 0; ring < =  numRings; ring+ + ) {
        // Height position, with membrane at Z=0 and body extending down
        float z =  -(static_cast<float> (ring) / numRings) * height);

        for (int segment = 0; segment < =  numSegments; segment+ + ) {
            float angle =  static_cast<float> (segment) / numSegments) *2 .0 f M_PI;

            // X and Y for circular cross-section
            float x =  radius *cos(angle);
            float y =  radius *sin(angle);

            // Add vertex (x, y, z)
            vertices.push_back(x);
            vertices.push_back(y);
            vertices.push_back(z);
        }
    }

    // Calculate indices for wireframe rendering
    for (int ring = 0; ring <  numRings; ring+ + ) {
        for (int segment = 0; segment <  numSegments; segment+ + ) {
            // Get indices of the current quad's corners
            unsigned int topLeft =  ring * (numSegments +1) +  segment;
            unsigned int topRight =  topLeft +1;
            unsigned int bottomLeft =  (ring +1) * (numSegments + 1) +  segment;

            // Add lines for the wireframe
            indices.push_back(topLeft);
            indices.push_back(topRight);

            indices.push_back(topLeft);
            indices.push_back(bottomLeft);
        }
    }

    // Create buffers and set up rendering
    unsigned int vbo =  visManager.createVertexBuffer(
        vertices.size() * sizeof(float),
        vertices.data()
    );

    vaoId =  visManager.createVertexArray();
    visManager.configureVertexAttributes(vaoId, vbo, 0, 3, 3 * sizeof(float), 0);

    eboId =  visManager.createIndexBuffer(indices.data(), indices.size() * sizeof(unsigned int));
}
```

[a27] Code excerpt of the **AudioManager** class *Return to report*

```
class AudioManager {
private:
    // Singleton instance
    static std::unique_ptr< AudioManager>  instance;

    // Sample buffer for recording
    std::vector<float>  recordBuffer;

    // Audio settings
    int sampleRate;
    bool isRecording;
    double accumulatedTime;   // Accumulated simulation time
    double sampleInterval;    // Time between samples (1/sampleRate)

    // Sample interpolation state
    struct {
        float lastValue;       // Last sampled value
        float currentValue;    // Current sampled value
        double lastSampleTime; // Time of last actual sample
    } interpolationState;

    // Audio channels
    struct AudioChannel {
        std::string name;      // Channel name
        float gain;            // Channel-specific gain
        bool enabled;          // Whether this channel is active
        float currentValue;    // Last received value
    };
    std::vector< AudioChannel>  channels;
    bool useChannelMixing;     // Whether to mix all channels
    float masterGain;          // Global gain applied to all channels

public:
    void initialize(int sampleRate = 4 4 1)0 0
    void startRecording();
    void stopRecording();
    void processAudioStep(float simulationTimeStep, float currentSampleValue);
    bool writeToWavFile(const std::string& filename);
    // Channel management methods
    int addChannel(const std::string& name, float gain = 1 .0)f
    void setChannelGain(int channelIndex, float gain);
    void setChannelEnabled(int channelIndex, bool enabled);
    // ... other methods ...
};
```

[a28] Struct representing a virtual microphone Return to report

```
struct MembraneVirtualMicrophone {
    glm::vec2  position;  // Normalized position [0,1] on the membrane
    float gain;           // Microphone gain factor
    bool enabled;         // Whether the microphone is active
    std::string name;     // Name for the microphone (useful for UI)
};
```

[a29] Managing virtual microphones in the **MembraneComponent** Return to report

```
// In MembraneComponent class
std::vector< MembraneVirtualMicrophone>  microphones;
float masterGain;        // Master gain applied to the final mix
bool useMixedOutput;     // Whether to mix all microphones or use them separately

// Methods to manage microphones
int addMicrophone(float x, float y, float gain = 1 .0,fconst std::string& name = " ');
void removeMicrophone(int index);
void clearAllMicrophones();
void setMicrophonePosition(int index, float x, float y);
void setMicrophoneGain(int index, float gain);
void enableMicrophone(int index, bool enabled);

// Configuration presets
void setupSingleCenterMicrophone();
void setupStereoMicrophones();
void setupQuadMicrophones();
void setupCircularMicrophones(int count, float radius = 0 .4)f
```

[a30] **getInterpolatedSample()** method from the **AudioManager** Return to report

```
float AudioManager::getInterpolatedSample(double time) {
    // If we're at the very beginning, just return current value
    if (interpolationState.lastSampleTime <= 0.0) {
        return interpolationState.currentValue;
    }

    // Calculate simulation interval - the time between the last two simulation steps
    double simulationInterval =  accumulatedTime - interpolationState.lastSampleTime;

    if (simulationInterval <= 0.0) {
        return interpolationState.currentValue;
    }

    // Calculate interpolation factor (how far we are between previous and current time)
    double t =  (time - interpolationState.lastSampleTime) / simulationInterval;
    t = std::max(0.0, std::min(1.0, t)); // Clamp to [0,1]

    // Linear interpolation
    return interpolationState.lastValue * (1.0f - static_cast<float>(t))  +
           interpolationState.currentValue * static_cast<float>(t);
}
```

[a31] **AudioManager** combining audio signals from the membrane and body *Return to report*

```
// Get current membrane displacement at sample points
const auto& heights =  membrane-> getHeights();
float membraneOutput = 0 .0;f

// Calculate average of all microphone outputs
for (int i = 0; i <  membrane-> getMicrophoneCount(); i+ + ) {
    const auto& mic =  membrane-> getMicrophone(i);
    if (!mic.enabled) continue;

    // Convert mic position to grid coordinates
    int gridX = static_cast<int> (mic.position.x * membrane-> getMembraneWidth());
    int gridY = static_cast<int> (mic.position.y * membrane-> getMembraneHeight());

    // Get displacement if inside membrane
    if (membrane-> isInsideCircle(gridX, gridY)) {
        int idx =  gridY * membrane-> getMembraneWidth() +  gridX
        float displacement =  heights[idx] * mic.gain;
        membraneOutput + =  displacement;
    }
}

// Normalize and apply master gain
if (membrane-> getMicrophoneCount() >0) {
    membraneOutput /=  membrane-> getMicrophoneCount();
    membraneOutput *=  membrane-> getMasterGain();
}

// Process through body resonator
float combinedOutput =  membraneOutput;
if (bodyResonator) {
    float bodyOutput =  bodyResonator-> processInput(membraneOutput);
    combinedOutput =  membraneOutput *0 .7 f  bodyOutput *0 .3;f
}

// Add to audio buffer
audioManager.processAudioStep(timestep / 6 0 .0 f, combinedOutput);
```

[a32] **writeToWavFile()** method from the **AudioManager**

Return to report

```
bool AudioManager::writeToWavFile(const std::string& filename) {
    if (recordBuffer.empty()) {
        std::cerr << "Warning:  No samples to write to WAV file"<< std::endl;
        return false;
    }

    std::ofstream file(filename, std::ios::binary);
    if (!file) {
        std::cerr << "Failed  to open file for writing: "<<   filename <<std::endl;
        return false;
    }

    // WAV header parameters
    const int numChannels = 1; // Mono
    const int bitsPerSample = 16;
    const int bytesPerSample =  bitsPerSample /8;
    const int dataSize =  recordBuffer.size() * bytesPerSample;
    const int fileSize = 36 +  dataSize;
    const int byteRate =  sampleRate * numChannels * bytesPerSample;
    const int blockAlign =  numChannels * bytesPerSample;

    // Write WAV header
    // RIFF header
    file.write("RIFF", 4);
    file.write(reinterpret_cast<const char*>(&fileSize), 4);
    file.write("WAVE", 4);

    // Format chunk
    file.write("fmt ", 4);
    int fmtSize = 16;
    file.write(reinterpret_cast<const char*>(&fmtSize), 4);
    short audioFormat = 1; // PCM
    file.write(reinterpret_cast<const char*>(&audioFormat), 2);
    file.write(reinterpret_cast<const char*>(&numChannels), 2);
    file.write(reinterpret_cast<const char*>(&sampleRate), 4);
    file.write(reinterpret_cast<const char*>(&byteRate), 4);
    file.write(reinterpret_cast<const char*>(&blockAlign), 2);
    file.write(reinterpret_cast<const char*>(&bitsPerSample), 2);

    // Data chunk
    file.write("data", 4);
    file.write(reinterpret_cast<const char*>(&dataSize), 4);

    // Find maximum amplitude for normalization
    float maxAmplitude = 0.0f;
    for (const float& sample : recordBuffer) {
        maxAmplitude = std::max(maxAmplitude, std::abs(sample));
    }

    // Normalize only if necessary
    float normalizationFactor = 1.0f;
    if (maxAmplitude > 1.0f) {
        normalizationFactor = 1.0f / maxAmplitude;
    } else if (maxAmplitude < 0.1f) {
        // If very quiet, boost the signal
        normalizationFactor = 0.9f / maxAmplitude;
    }

    // Write audio data
    for (const float& sample : recordBuffer) {
        // Normalize, then convert float [-1,1] to 16-bit PCM
        float normalizedSample =  sample * normalizationFactor;
        // Clamp to [-1,1] range
        normalizedSample = std::max(-1.0f, std::min(normalizedSample, 1.0f));
        short pcmSample = static_cast<short>(normalizedSample  *32767.0f);
        file.write(reinterpret_cast<const char*>(&pcmSample), 2);
    }

    file.close();
    std::cout << "WAV  file saved: "<<   filename <<std::endl;
    std::cout << "Duration:  "<<   (recordBuffer.size() /static_cast<double>(sampleRate))
              << "  seconds"<< std::endl;
    return true;
}
```

[a33] Early discussion with Claude about the techstack for prototype development *Return to report*

Previous Context: Claude reviewing initial project plan, and a discussion on possibilities for a simple prototype

Prompt: okay, for the prototype let's scrap the blender importing

Claude:

That makes sense. Focusing on the simulation itself rather than model importing will streamline your prototype development. Here's a revised tech stack:

Core Technologies:

- **C++** for main codebase
- **CUDA** for GPU-accelerated membrane physics
- **OpenGL** for visualization of the membrane deformation

Streamlined Libraries:

- **GLFW** for window management
- **Dear ImGui** for a simple UI to control simulation parameters
- **miniaudio** for both real-time audio playback and file export
- **GLM** for math operations (vectors, matrices)

Build System:

- **CMake** with a simple configuration
- Git submodules for the header-only libraries

Instead of importing Blender models, your prototype would:

1. Generate a simple circular membrane mesh procedurally
2. Allow parameter adjustment through the ImGui interface (membrane tension, size, damping)
3. Visualize membrane deformation in real-time
4. Convert the physical simulation to audio
5. Play the audio and/or save to file

This approach gives you a functional prototype with user interaction while focusing your development efforts on the core GPU-accelerated physics simulation, which is the most critical aspect of your project. You can always add the Blender import pipeline later once your fundamental simulation is working correctly.

[a34] Asking claude to converse about a feature before writing any code Return to report

Previous Context: Talking about moving on to creating an initial graphics system for the CPU based prototype

Prompt: Okay, let's proceed to step 4, but before any sort of code implementation, let's converse about how we will go about it

Claude:

Happy to discuss step 4 before we implement anything. For visualizing the membrane grid in your OpenGL window, we'll need to cover several aspects:

Visualization Approach Options

1. **Wireframe Representation**:
   - Render the grid as lines connecting the points
   - Simple to implement but less visually informative
2. **Colored Height Map**:
   - Render the grid as quads/triangles with colors based on height
   - Good for visualizing the overall membrane shape
3. **3D Surface Rendering**:
   - Render as an actual 3D mesh with lighting
   - Most realistic but requires more OpenGL setup

Technical Requirements

1. **OpenGL Basics Needed**:
   - Vertex Buffer Objects (VBOs) to store membrane data

Reply continues…

[a35] Discussing architecture for starting the GPU accelerated version Return to report

Previous Context: Discussing moving on from the legacy CPU prototype and creating the architecture for a GPU accelerated version from scratch.

Prompt: Is this approach future proof? I want to flush out the simulation with the drum having a body and a lower membrane, an internal air cavity etc... would it be better to structure in a different way? maybe going with a more modular approach with a clear separation of concerns would be better.. what do you think

Claude:

That's an excellent consideration! When planning a more complex model with multiple components like a dual-membrane drum with an internal air cavity, we need to think about how the architecture will scale.

Let's reconsider the approach with future components in mind:

More Future-Proof Architecture

1. **Component-Based Design**: Rather than a single monolithic CudaMembrane class, we could design a more modular system with separate components that can interact:
   - CudaUpperMembrane
   - CudaLowerMembrane
   - CudaAirCavity
   - CudaDrumShell
2. **Coupling Interface**: Develop a standard coupling interface that allows these components to exchange boundary conditions and forces
3. **Simulation Manager**: Create a higher-level manager class that coordinates the simulation of all components and handles their interactions
4. **Unified Memory Management**: Implement a centralized memory management system for all GPU buffers
5. **Renderer Interface**: Design a visualization interface that any component can implement to be rendered

Benefits of This Approach

- **Modularity**: You can add new components without restructuring existing code
- **Isolation**: Each component can be simulated (and tested) independently
- **Flexibility**: Different simulation methods can be used for different components

Reply continues…

[a36] Providing Claude with a txt file export from a previous conversation Return to report

Prompt: The file is an export from our previous conversation, I am providing it for context. You also have access to the DrumForge Github, please review the codebase and architecture thoroughly. Please let me know your understanding before we continue

Claude:

> Thank you for providing the conversation context and access to the DrumForge codebase. After reviewing the files, I have a clear understanding of the project and the current issue:
>
> DrumForge is a physically-based drum synthesizer that uses CUDA for GPU-accelerated simulation of drum acoustics. The project employs:
>
> 1. A dual-approach to simulation:
>   - Membrane component: Uses Finite Difference Time Domain (FDTD) methods to simulate the drumhead
>   - Body component: Uses modal synthesis to simulate the drum shell resonance
> 2. The current issue that we've been facing is that the body component generates sounds with a consistent pitch regardless of its physical dimensions (radius, height, thickness), which is physically unrealistic.

Reply continues…