G-SpAR: GPU-Based Voxel Graph Pathfinding for Spatial Audio Rendering in Games and VR

Mirza Beig[1], Bill Kapralos[1], Karen Collins[2], and Pejman Mirza-Babaei[1]
[1]University of Ontario Institute of Technology, Oshawa, Canada.
[2]Unviversity of Waterloo, Waterloo, Canada.

Abstract— **The influx of investment in virtual and augmented reality in recent years has brought with it a renewed interest in spatial audio. Spatial audio in virtual environments and games has been neglected previously due in part due to computationally expensive nature of the processing involved. Here we introduce G-SpAR, a GPU-based spatial audio renderer for the Unity3D game engine. The system develops a pathfinding method that shortens occlusion and obstruction, increasing performance in-game.**

Keywords—Spatial audio, acoustical modeling, graphics processing unit (GPU), real-time, pathfinding

I. Introduction

Interest in spatial has been increasing in recent years with the rapid rise of virtual and augmented reality (VR and AR respectively). This explosion of enthusiasm, comes with a need for, and greater interest in, further developing spatial audio tools. Realistic and effective spatial audio technology has existed for many years in high-end simulations. However, the challenge with respect to consumer systems is the deployment of realistic spatial audio in a computationally inexpensive manner that is also platform agnostic, and easy to use given the difficulties associated with modeling the human listener (e.g., head-related transfer functions (HRTFs)), and the room acoustics. Here, we ignore the listener and focus solely on recreating the room acoustics (see for example [1] for a detailed discussion on simulating listener-specific characteristics). Many spatial audio approaches employ geometric acoustics, whereby it is assumed that sound and rays behave in a similar manner. This is in contrast to wave-based methods whereby the aim is to recreate a particular sound field by approximating the wave equation using numerical approximations (e.g., finite element methods, boundary element methods, and finite difference time domain methods instead ([2][3][4]).

Geometric acoustics models sound propagation as straight lines ("rays") that interact with the surface geometry and materials of the virtual environment. The acoustics of an environment is then modeled by tracing these "sound rays" as they propagate through the environment while accounting for any interactions between the sound rays and any objects/surfaces they encounter before reacing the listener. Mathematical models are used to account for sound source emission patterns, atmospheric scattering, and the medium's absorption of sound energy as a function of humidity, temperature, frequency, and distance. At the listenera room an echogram, which describes the distribution of incident sound energy (rays) at the receiver over time, is obtained. The echogram is then post-processed to provide a room impulse response which is used to filter a sound and recreate the specific listening environment. Geometric acoustic models are only valid approximations for high frequency sound propagation; low frequency wave effects such as diffraction are ignored (see [5]). Current graphical-based application (e.g., video games and virtual environments), employ advanced graphical rendering techniques (such as real-time ray-tracing and radiosity), and other advanced lighting and shading techniques that are implemented using the graphics processing unit (GPU). As will be described in the following section, there is an increased demand for the application of such advanced technqiues to sound rendering, thus taking advantage of the power inherent in GPUs. There have been various approaches to using the GPU for spatial sound generation, leading to various interactive rate spatial sound methods and techniques. Here, we present the GPU-Spatial Audio Renderer ("G-SpAR"), a spatial audio rendering component that employs pathfinding and runs on the GPU to the acoustics of an environment in a computationally efficient manner.

II. Background

A. Spatial Audio Rendering

Audio-based ray tracing using the GPU has been implemented by Jedrzejewski [6] to compute the propagation of acoustic reflections in highly occluded environments. The method also allows for the sound source and the listener to move throughout a simulation without the need for a long pre-computation. Jedrzejewski takes advantage of the fact that in acoustics, as opposed to graphics, objects other than walls do not contribute significantly to the sound wave modifications and therefore can be ignored during the computation. As a result, only polygons that represent walls are taken into account. To make the system more efficient, each ray is intersected with a plane rather than a polygon.

Röber et al. [7] describe a ray-based acoustical modeling method that employs the GPU. Their framework was designed along existing (computer graphics) GPU-based ray tracing systems suitably modified to handle sound wave propagation. The system accepts a 3D polygonal mesh of up to 15,000 polygons and pre-processes it into an accessible structure. All signal processing, including HRTF filtering and delay filtering, is programmed as fragment shaders and for each task, a single shader is developed. Cheng [8] has also developed a GPU-based method for simulating room acoustics in real-time. The method computes the reflected and transmitted acoustic response from a number of sound sources to a stereo listener in arbitrary triangle-based geometry.

978-1-7281-1884-0/19/$31.00 ©2019 IEEE

Tsingos and Gascuel developed a method that employs the GPU to perform fast sound visibility calculations that can account for specular reflections (diffuse reflections were not considered), absorption, and diffraction caused by partial occluders [9]. Specular reflections are handled using an image source approach, while diffraction is approximated by computing the fraction of sound that is blocked by obstacles on the path from the sound source to the receiver by considering the amount of volume of the first Fresnel ellipsoid that is blocked by the occluders. Although their approach is not completely real-time, it is "capable of achieving interactive computation rates for fully dynamic complex environments" [9]. Tsingos and Gascuel later introduced another occlusion and diffraction method based on the Fresnel-Kirchoff optics-based approximation to diffraction [10]. Tsingos et al. [11] describe a high quality, GPU-based first order sound scattering modeling method that is based on a surface integral formulation and Kirchhoff's approximation. Their method is capable of modeling both diffraction and reflection in an arbitrarily complex environment. Experiments indicate their method fares well with boundary element methods (BEMs), although greater work remains to allow for higher order sound scattering and to overcome the fact that the method is prone to aliasing. Cowan and Kapralos [12] introduced a GPU-based occlusion method capable of real-time operation even for complex virtual environments and games. Occlusion/diffraction effects are computed by rendering the scene (using the GPU) from the perspective of the sound source. The method is capable of approximating acoustical occlusion and diffraction effects in real-time for detailed scenes containing many objects of complex shape. It works best in larger, open environments with several occluding objects. However, environments with interconnecting rooms can be problematic. This issue was addressed in later work by the same authors [13]. The method is computationally efficient, allowing it to be incorporated into real-time virtual environments and games where the scene is arbitrarily complex.

B. *Pathfinding*

Several algorithms exist for pathfinding in games, along with numerous variations suited for special cases and tasks, including breadth-first search (BFS), Dijkstra's algorithm, and its heuristic-guided variant, A-Star (A*)[6]. BFS expands one step towards connected vertices every iteration from a given source vertex and assumes a virtually non-existent cost of traversal each time. It can be used to find the least number of steps required to obtain between two points, which may also be the shortest path if costs are irrelevant to the results of the search. Dijkstra's algorithm is similar, except that it allows for weighted costs for traversal between vertices, allowing for a better definition of the shortest path. A* is a variant of Djikstra's algorithm that applies a heuristic to guide the search so that the search tends to gravitate towards the most likely shortest path. As a result, it is more performant than Djikstra's algorithm when requiring a single path to a target for any given source. The use of pathfinding provides an opportunity for optimization by reducing the requirement to a single source, shortest path problem. All sound sources in the virtual environment are perceived by a single listener. For this reason, A* is unlikely to be useful when there are more than a few sound sources in the scene as each source requires a unique, or partially unique traversal for the heuristics-guided algorithm. By using an expanding BFS-type search, the shortest path to every vertex in the graph can be resolved from a single source. With respect to performance, the addition of sound sources is then insignificant since a path to the vertex containing the sound source already exists. However, real-time pathfinding on a large enough dataset remains a complex and intensive task that can impact performance. As such, the operations involved are often offset to worker threads or solved in sequential steps over several frames with a limit to the maximum number of vertices or paths processed in a single update cycle to prevent impacting the performance of a real-time application too severely (Unity does this automatically as part of their built-in pathfinding solution).

C. *GPU-Based Pathfinding*

Various libraries for GPU-graph processing do exist, including NVIDIA's nvGraph, and Gunrock – both of which build upon NVIDIA's GPU parallel computing platform. Despite the availability of these libraries, we have not found any complete implementations of a fully 3D GPU-accelerated pathfinding algorithm for real-time use in games, and existing research that has been done in this area has yet to be tested on modern, consumer-level GPUs such as those found in today's gaming PCs. Prior work regarding parallel processing for pathfinding algorithms has employed parameters that do not apply to most game scenarios. For example, Bleiweiss [14] uses the A* algorithm on the GPU with a maximum testing block of 20×20 nodes, which is unlikely to provide a sufficient resolution for realistic pathfinding applications in most games, since the detail of the level geometry and obstacles would be overly simplified.

III. G-SPAR: VOXEL-BASED GPU SPATIAL AUDIO

By considering existing sound propagation pathfinding methods that have been used in popular games for play-by-sound mechanics, such as Overwatch (2016), we accelerated the process for dynamic 3D environments through the use of general purpose GPU (GPGPU) technology that is hardware agnostic. As a fall back, the system can still use the CPU, for easy cross-platform deployment to mobile and other lower-end hardware systems. The decision of whether to use the GPU is left to the discretion of the developer. Dynamically switching between CPU-only and GPU-assisted processing during runtime is also fully supported, allowing developers to profile the execution time of G-SpAR components themselves from within their application to determine which one should be used. We built G-SpAR to work with Unity, one of the most common game development engines. One important feature that makes Unity an ideal choice for virtual environments is the built-in 3D audio spatialization integration that includes HRTF transforms. G-SpAR employs component-based approximations that enhance existing solutions for environmental occlusion and obstruction modelling using four distinct methods that are each suited for scaling on their target platforms (e.g., lower-end mobile devices, and modern high performance gaming PCs) and the nature of the sound source environment (dynamic/static).

A. *Raycast and Lightmap Methods*

The first method is a simple on/off state raycast system that checks the line of sight between the sound source and the listener. If the line of sight is obstructed, the sound signal is occluded either completely or partially using a low-pass filter. This is a commonly used techniques in games of recent years.

The second method makes use of precomputed lighting information positioned at the sound source that, instead of being applied over the surface geometry in the game as a lightmap, is sampled directly as a texture to retrieve information about the sound throughout the scene. Light and sound sometimes behave in a similar manner when encountering physical surfaces. With this in mind, the high-quality information within the lightmap can be used as a "soundmap" to model the room acoustics. However, this approach is limited as it only works in 2.5D space (flat plane and height), and the sound source must be in a fixed location, as the texture is projected to the surface (not volumetric), and cannot be calculated in real-time without a significant performance impact. Depending on the size and configuration of compression for each texture, it may also introduce long load times and significantly increased storage requirements. Pre-computed lighting may require significant time to calculate and render into a texture. However, this approach provides the opportunity for creative use, since different light types can be used for a sound source and multiple lights representing the propagation of that sound source can be baked into a soundmap

B. *Pathfinding Method*

The third method uses 2.5D pathfinding and has the added benefit that a portion of the geometry in the scene can be dynamic. The pathfinding data is used to calculate the shortest distance that the sound would have to travel around the level geometry to reach the listener. This data is then compared to the direct point-to-point distance and the difference is used to filter the frequency and intensity of the source. Since calculating the path using a navigation mesh in 2.5D is quicker, this data is all processed using the CPU. Using this information, simple propagation can be modelled by taking the last visible corner (the last point determined by a raycast) of the path from the listener to the source and projecting a virtual sound source some distance equal to the remaining distance of the path from this corner to the source in the direction of the corner from the listener. When the GPU component of G-SpAR is completed, a path from any point on the node graph is possible, but we trace only a path for the two points we need between the listener and the source. The number of paths traced depends on the number of sound sources. Pathfinding-based sound propagation and/or occlusion has been consistently used in AAA video game titles such as Tom Clancy's Rainbow Six Siege (2015) for "play by sound" mechanics. Some of the limitations of the pathfinding-based propagation systems in these games involve limited spatial dimensions and limited or simulated environment dynamics [15]. Our implementation of pathfinding-based sound propagation and occlusion is also scalable to mobile devices and can handle fully destructible environments.

C. *GPU-Based Voxel Graph Pathfinding*

The final method—and our particular innovation—builds upon the pathfinding-based solution described above for approximating audio propagation while also circumventing the 2.5D limitation entirely by sampling the current scene as a voxel graph and making use of the highly parallel nature of modern GPUs to process this data. Combined with additional optimizations on the CPU, the extreme throughput of the GPU here allows for multiple (potentially several thousands of) real-time pathfinding calculations to be completed within a single frame to use as raw data for DSP once retrieved back to the CPU. As a result, the entire implementation down to the end of the DSP chain, even with multiple active sound sources and dynamic level geometry can be run in a complex-geometry at over 60 frames per second. To our knowledge, this technique is unique and serves as both a proof of concept and an actual implementation of this kind of GPGPU computing for approximating the acoustics of sound.

IV. Implementation and System Details

To implement G-SpAR, we created a complete framework from the CPU to the GPU using C# and Unity's device-independent compute shader language. This provided maximum control over the pipeline allowing us to test multiple algorithms and still retain the potential for multi-platform deployment later on. A complete source code listing is available (see [16]).

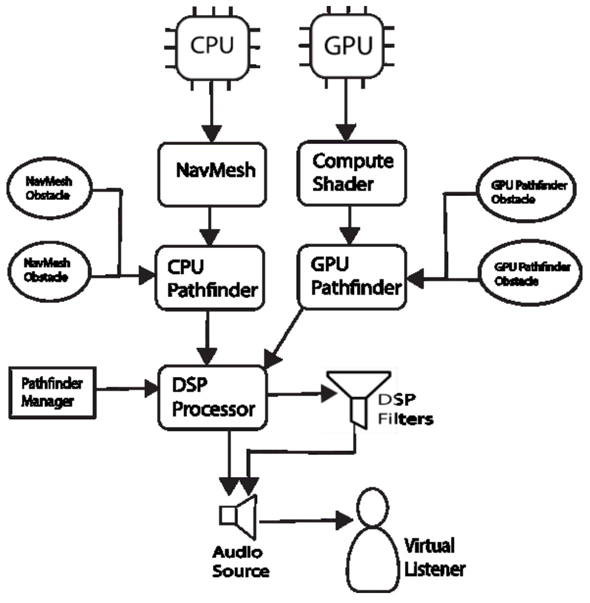

Fig. 1. High-level flow diagram of the overall structure of G-SpAR. NavMesh obstacles are handled directly by Unity's system, whereas GPU pathfinder obstacles update node states on the CPU which are then re-sent to the GPU.

An overview of the system can be seen in Fig. 1. Pathfinding primarily functions through three custom components and a single compute shader. There are two component scripts that handle all the pathfinding-related operations on either only the CPU using Unity's NavMesh system, or both the CPU and GPU using our pathfinding implementation (including the data as it is transferred to and from the compute shader). Both return similar results and have almost identical function names – the difference is in how they retrieve those results for their roles as the "pathfinder". Of special interest here is the latter, which involves the GPU. The third component (labelled as the DSP processor) selects which pathfinder to use, and processes the results to control low-pass, reverb, and intensity filters applied to a specific audio stream. There is only a single active pathfinder required for each type in the scene (so at most there will be the CPU pathfinder and GPU pathfinder), but there may be multiple DSP processors attached to GameObjects (that is, Unity's base class objects) that have Unity's AudioSource component (that is, the audio clip playback component), for manipulating the audio through DSP filters.

The process begins on the CPU through a pre-processing stage that first takes into account all of the static geometry in the scene for the duration of the session when the pathfinder component first “awakes” on scene load. This geometry must be specifically designated and assigned to a layer that is configured on the pathfinder component to use as a search mask. The pathfinder is given a three-dimensional vector representing the discrete scale of the area in which the algorithm will operate as an axis-aligned “room” or grid. The resolution of this grid is then defined by another value that represents the uniform cell size. The representation of a single cell in the pathfinder is through a custom C# struct called a node. A node is defined by an accumulative array index, its vector [x, y, z] index, and world position. It is essentially a vertex in the pathfinding graph. A global array of nodes containing all of the cell data is then created, along with two additional global integer and Boolean arrays that will record the traversable state of that node and whether the state may have been changed by dynamic geometry. Both the integer and Boolean arrays are of the same size as the node array and are treated and used as Boolean variables specifying whether the node is traversable and whether its traversable state may have been changed as a result of dynamic geometry moving into the node respectively. However, since the integer array is passed to the compute shader, it must be stored as an array of integers due to the minimum stride of compute buffers being four bytes. The pathfinder then scans each cell area and performs an axis-aligned box test to check for colliders, marking the integer value in the traversable array as either 0 for not traversable, and 1 for traversable.

The next stage of the pathfinder takes place in the update loop (every frame) and contains the code for the compute shader kernel dispatches. Here, Djikstra’s algorithm was found to be more performant due to its more reserved use of compute shader kernel dispatches. Optimizations are set in place to determine if the pathfinder must engage the GPU. In the main update loop, dynamic obstacles are handled at a variable frequency that can be set by the user. In this case, we used a default minimum update delay value of 0.125 s. If an obstacle is detected, the axis-aligned bounding box defining its collider is used to mark all nodes within that area as “dirty” and checked again using an isolated box test for traversable state updates. The array containing these states is then sent to the GPU in one call. It is important to minimize large data transfers between the CPU and GPU, as this can create a performance bottleneck. Therefore, the node data is passed in only once at the start of the scene while the traversable state is stored in a separate array consisting only of integers rather than being a part of the full struct. This allows dynamic scene geometry to update the traversable state of nodes without having to communicate unchanged data between the host and device. The amount of data transferred depends on the size of the buffer (number of nodes) multiplied by the length of the individual elements in bytes. Although not previously viable, advances in technology have allowed for much faster transfer speeds, making the possibility of large-scale data handovers and solutions between the host and device possible.

Next, we check whether the listener GameObject has moved to a position that is different from a previous update cycle, otherwise there is no need to update the internal grid paths at all. If the dynamic obstacle check is executed and obstacles are found, or if the listener has moved to a new cell, the pathfinder marks the grid for updating and dispatches the required shader kernels for execution. This is an inherently useful advantage of using a discrete grid for pathfinding, as the listener is effectively treated as the player’s ear. If the player is not moving and the scene remains static, then there is no need to recalculate new paths since the results are guaranteed to be the same when using a BFS search, which calculates a path to every vertex in the graph. The faster the player moves and the higher the resolution of the grid, the more likely it is that an update will be required and the pathfinder will engage the GPU. For GPU benchmarking, we moved the player at a constant maximum speed of 10 m/s (this is close to the maximum recorded sprint speed for humans), or 10 cells/s. This results in an actual maximum potential update frequency of 10/s and conforms to a game-like scenario with a grounded player character in VR. Unlike multi-frame CPU solutions, updated feedback is guaranteed on movement within the same frame. If an update to the grid is required, the pathfinder sends the position of the listener to the compute shader and begins a loop that dispatches an expand frontier kernel for Djikstra’s algorithm indirectly, as well as another kernel directly with a single active thread to monitor completion. In DirectCompute, when dispatching a kernel directly, the number of threads to execute on [x, y, z] are pre-determined in the call, whereas an indirect dispatch takes in an argument buffer that can be updated on the GPU-side as a means for dynamic parallelism, where the number of threads launched per dispatch can change through code executed on the GPU. Once the second kernel determines that expansion has completed and there are no more nodes left to explore on the frontier, the first kernel becomes inactive until all dispatch calls from the pathfinder’s loop have been exhausted. At this point, the grid is up-to-date and all nodes stored on the GPU have the correct neighbor indices that can be traced to the starting vertex node. The pathfinder can then dispatch a final calculation kernel once that can be used to update and retrieve an array of indices containing all the path data in sequence (calculated in parallel, per path), and another array containing the length of all the paths in meters. The length of the path is a sum of the distances between the corners. The DSP processor component works individually on every AudioSource GameObject in the scene and must be attached as an additional component to each one to have it work in conjunction with the pathfinder. This replicates existing solutions that use pathfinding as the base for approximating propagation, except that it can query paths using the custom GPU pathfinder for results (it can switch between choosing the appropriate pathfinder at runtime).

A raycast from every DSP processor’s position (based on its GameObject Transform component which provides position, rotation and scale information), is executed towards the listener. A hit triggers an indication of some obstruction, which in turn creates a request to the target pathfinder for a path to the current DSP processor. The length of this path is compared to the direct, straight-line distance between the sound source and the listener, and the difference ratio to the maximum range of the audio source is then used as an input to various functions with exposed properties in Unity. Additional propagation is modelled in the DSP processor by sampling the last visible corner from the perspective of the listener, and using that to instantiate a duplicate sound source positioned the remaining distance away from the angle the listener to the corner. This is an approximation of the last reflected sound wave reaching the listener and also simulates sound that "curves" or diffracts around corners so that its position is not heard as strongly directly from its initial position, but can be used to trace back to the original source if the player carefully aligns themselves and follows the duplicate sound. While the GPU-accelerated 3D pathfinding component of the system is limited to high-performance platforms that support compute shaders, the 2.5D CPU variant is fully compatible with all platforms that Unity supports. The pathfinder component is not tied to the sound part of the system and can be used for other tasks in a game by simply requesting a path from any other part of the game loop. It is an approximation that can be used as-is, or in conjunction with other systems such as Steam Audio or Google Resonance to fill in the gap for real-time, dynamic occlusion modelling.

V. Benchamarking Results

TABLE I. All systems used for benchmarking

Platform	CPU	GPU	RAM
Windows 10 desktop	Intel Core i7-6700K @ 4.00 GHz (8 CPUs)	NVIDIA GeForce GTX1070 @ 1506 MHz (8096 MB)	32,768 MB
Android 7.0 Smartphone	Qualcomm Snapdragon820 @ 2.15 GHz (4 CPUs)	Qualcomm Adreno 530 @ 624MHz	4,096 MB

G-SpAR was deployed on Windows 10, and Android 7 devices (see Table I). For optimization during development, performance was monitored using Unity's built-in profiler. We developed a custom profiler integrated into the global pathfinder manager that recorded execution times for the entire, framerate-unlocked application (disabled vertical synchronization), and isolated execution times for the spatial audio system over a specified duration. The parameters recorded were the average execution time (calculated as the sum of values over the recording duration divided by the number of frames executed), and the minimum and maximum execution times. From these we derived the rounded integer frame rates. Performance tests were recorded with a duration of 60.0 s (even with the frame rate of the application unlocked, the maximum frames per second that can be executed by mobile devices is fixed at 60, 30 or lower due to mandatory vertical synchronization).

A. *Test Scene*

The test scene consisted of 10 sound sources spread out in a volume of 100 m^3 with a player size to scale of about 2.0 m. The player moved at the maximum speed of 10 m/s at all times for the GPU tests with dynamic obstacle delay values (when enabled) set to either 0.125 s or 0.0 s (see benchmark tests). There was one physics-enabled dynamic object in the scene as a large 4 m^3 cube to ensure the dynamic obstacle part of the pathfinder would execute. The actual playable area was 100 m × 10 m × 100 m, but the full 100 m^3 node block was processed (see Fig. 2). The uniform node size was fixed to 1.0 m.

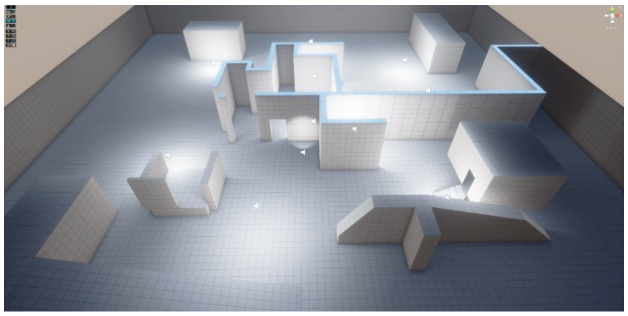

Fig. 2. Benchmarking level in Unity's editor with 10 sound sources.

Windows 10 significantly outperformed Android 7 and easily executed and updated the GPU pathfinder every frame while maintaining an average of just over 60 fps. At its worst, the GPU-based spatial audio components took 18.55 ms, and at its best, it took 11.25 ms while engaging and exchanging buffer data with the GPU. If dynamic obstacle updates were throttled to execute at a fixed frequency of eight times per second (every 0.125 s), the average performance jumped significantly to over 700 fps for the runtime total, and over 900 fps for just the spatial audio: the timings are separated into total runtime average and spatial audio runtime averages to account first for all parameters, and then only on G-SpAR's calculations. With limited dynamic updates, the Spatial Audio Max was 19.94 ms, which is comparable to the spatial audio Max with full dynamic updates every frame timing in at 18.55 ms. This is expected, as the amount of data to process will be nearly identical, except that updates will execute at a lower frequency. This is also why the Spatial Audio Min was much lower for GPU (Dynamic 0.125) at 0.39 ms than for GPU (Dynamic 0.0), since during the frames when dynamic updates are not being processed, the GPU pathfinder behaves the same as if all geometry is static. The Spatial Audio Min for GPU (Static) confirms this with a similar Spatial Audio Min to GPU (Dynamic 0.125). A summary of the Windows 10 results for the runtime and spatial audio tests are available in Tables II and III respectively.

The Android 7 was not performant to the point of being useful in an actual game that would require sustaining 30-60 fps. As mentioned earlier, when the dynamic geometry part of the GPU pathfinder is inactive, it behaves as if having only static geometry, and therefore, the spatial audio Min between GPU (Static) and GPU Dynamic (0.125) should be similar. The large gap between the runtime values is most likely due to the Android's overall low performance with the GPU pathfinder. Of the systems capable of GPU-accelerated spatial audio processing, the Android resulted in the worst performance by a large margin. Instead of the GPU (Dynamic 0.125) measurement being similar to GPU (Static), it is much closer to GPU (Dynamic 0.0), indicating that it was never able to "rest" between frames, since by the time the next frame could be processed, over 1/8th of a second had passed. The significantly lower GPU (Static) Spatial Audio Min can be explained by considering the value being in range of the CPU Spatial Audio Max. While there is no movement for GPU (Static), no recalculation is performed on the GPU, and any overhead is from having the GPU pathfinder pipeline running. This is reported in Unity's profiler, and can be inferred visually on the Android, which does not produce frame-lag if simply idling without character movement across graph cells (during which the benchmark total is at least 60 fps), but then suddenly produces lag when moving. During the benchmark, the character is supposed to be moving at an average of 10 cells/s, but in one frame, due to the time delta between frames, the character may not have moved enough to engage the GPU via triggering a recalculation and graph update. A summary of the Android 7 results for the runtime and spatial audio tests are available in Tables IV and V respectively.

TABLE II. SUMMARY OF RUNTIME TESTS ON WINDOWS 10

Pathfinder	Runtime Total Min	Runtime Total Max	Runtime Total Avg
CPU	0.56 ms (1,774 fps)	3.88 ms (258 fps)	0.90 ms (1,107 fps)
GPU (Static)	0.78 ms (1,275 fps)	16.67 ms (60 fps)	1.32 ms (758 fps)
GPU (Dynamic 0.125)	0.80 ms (1,257 fps)	21.93 ms (46 fps)	1.57 ms (638 fps)
GPU (Dynamic 0.0)	12.06 ms (83 fps)	20.20 ms (49 fps)	16.07 ms (62 fps)

TABLE III. SUMMARY OF SPATIAL AUDIO TESTS ON WINDOWS 10

Pathfinder	Spatial Audio Min	Spatial Audio Max	Spatial Audio Avg
CPU	0.23 ms (4,264 fps)	2.51 ms (399 fps)	0.26 ms (3,884 fps)
GPU (Static)	0.37 ms (2,711 fps)	16.02 ms (62 fps)	0.85 ms (1,177 fps) (758 fps)
GPU (Dynamic 0.125)	0.39 ms (2,552 fps)	19.94 ms (50 fps)	1.09 ms (916 fps)
GPU (Dynamic 0.0)	11.25 ms (89 fps)	18.55 ms (54 fps)	14.99 ms (67 fps)

B. *In-Game Test Methods*

To test and confirm the applicable performance of the system in a VR environment, we developed a fully functional 3D game optimized for deployment using the system on mobile and desktop/VR platforms. The game places an emphasis on the purposeful manipulation of sound and light in its gameplay mechanics across multiple levels, and demonstrates how G-SpAR can be used in an actual development scenario. The game emphasizes play-by-sound mechanics to the point of even allowing players to catch the attention of a lingering enemy AI using their voice through a microphone, which is then projected into the game world and propagated to the AI as if it were originally sourced from inside the virtual environment. The bulk of the gameplay takes place in a darkened underground labyrinth faintly lit by incandescent light bulbs and torches. The player must safely navigate the labyrinth using the faint cues of light and propagated sound to solve puzzles, find/collect keys, and avoid a dangerous lingering enemy AI that can detect the player's movements (including in-game interactions) using simulated hearing and sight. To escape to the next area, they must use the keys they find on chests scattered throughout the level without alerting the enemy and then activate a "teleportation monolith". The dimensions of all of the rooms in the game are smaller than the dimensions of the rooms in the benchmarking environment and despite physics, AI, lighting, and other active game-relevant calculations, runtime performance was smooth. All sounds emitted in 3D were spatialized with HRTFs using Unity's native Oculus integration. This did not include environmental propagation. The keys, the enemy AI, and other select sources in the game use G-SpAR to convey additional important information about the game state using sound, such as obstruction and occlusion, allowing players to "hear" what they're looking to find (or avoid) by tracking sounds around corridors. However, the enemy similarly "listens" to the player, and taking into account how the sound should propagate path-wise, may attempt to navigate to any disturbances it detects. The listening behaviour reads in all disturbances logged to a global sound source "monitor", which holds the position and initial volume of the source. Player footsteps and interactions are added to the monitor. The player can take advantage of this behavior to deceive the enemy by purposefully creating noise in one area and then sneaking away as it approaches to investigate.

The majority of interactions in the game produce 3D spatialized sounds based on physics such as the player running into and against walls or other objects, and object-to-object collisions such as impacts and scrapes (accounting for relative velocity and pressure). In addition to providing a constant stream of sonic information to the player, this has additional gameplay consequences in the underground labyrinth where this information can also be decoded by the enemy. Interactions with objects in the environment, such as doors and chests (both featuring squeaky hinges), can attract enemy attention. The enemy has spatialized and propagated footsteps, growls, barks, and leaves a trail of emissive particles. Additionally, its eyes glow and a spotlight attached to the head brightens anything the enemy is looking at, such as walls. The player uses these cues to remain aware of the enemy's position at all times. If the enemy catches the player, the player is killed and the level is restarted. In other words, spatial sound was an important component not just as background audio, but as an interactive element of the game that demands attention from the player. During runtime, our propagation system can be switched between being entirely inactive (using Unity's default implementation), using only the CPU for 2.5D processing, or additionally taking advantage of the GPU to simulate fully 3D propagation acoustics so that the differences can be observed. Runtime measurements were performed using the same custom profiler as described above. The second level of the game was tested as it contained the largest and most complex design with the most interactions and longest likely gameplay length. All measurements were recorded over a duration of 60.0 s. The same systems were used and the GPU pathfinder was benchmarked when possible, with the dynamic frequency set to run every 0.125 s (the gameplay did not require dynamic geometry updates every frame). The playable area's bounding volume for the level was approximately 50 m × 15 m × 60 m. The node size was set to 0.48 m to accommodate finer details in the level geometry than that of the benchmark level. Finally, the player moved at a variable rate between no movement, crouching slow/fast, walking, and running at full speed. This results in a move speed range between 0.0 m/s to 4 m/s while performing any of the interactions that were part of the level.

TABLE IV. SUMMARY OF RUNTIME TESTS ON ANDROID 7

Pathfinder	Runtime Total Min	Runtime Total Max	Runtime Total Avg
CPU	6.83 ms (146 fps)	49.33 ms (20 fps)	16.78 ms (60 fps)
GPU (Static)	13.15 ms (76 fps)	980.62 ms (1 fps)	534.07 ms (2 fps)
GPU (Dynamic 0.125)	529.80 ms (2 fps)	1195.18 ms (1 fps)	817.41 ms (1 fps)
GPU (Dynamic 0.0)	493.09 ms (2 fps)	1272.30 ms (1 fps)	863.54 ms (1 fps)

TABLE V. SUMMARY OF SPATIAL AUDIO TESTS ON ANDROID 7

Pathfinder	Spatial Audio Min	Spatial Audio Max	Spatial Audio Avg
CPU	0.65 ms (1532 fps)	23.05 ms (43 fps)	1.27 ms (790 fps)
GPU (Static)	5.42 ms (184 fps)	961.99 ms (1 fps)	519.63 ms (2 fps)
GPU (Dynamic 0.125)	515.32 ms (2 fps)	1181.29 ms (1 fps)	799.50 ms (1 fps)
GPU (Dynamic 0.0)	478.23 ms (2 fps)	1253.31 ms (1 fps)	845.64 ms (1 fps)

TABLE VI. SUMMARY OF GAME TESTS ON WINDOWS 10

Pathfinder	Runtime Total Min	Runtime Total Max	Runtime Total Avg
CPU	1.66 ms (603 fps)	8.87 ms (113 fps)	2.09 ms (479 fps)
GPU (Dynamic 0.125)	2.47 ms (404 fps)	15.06 ms (66 fps)	3.50 ms (286 fps)

TABLE VII. SUMMARY OF SPATIAL AUDIO GAME TESTS ON WINDOWS 10

Pathfinder	Spatial Audio Min	Spatial Audio Max	Spatial Audio Avg
CPU	0.60 ms (1671 fps)	4.92 ms (203 fps)	0.71 ms (1416 fps)
GPU (Dynamic 0.125)	0.77 ms (1294 fps)	13.41 ms (75 fps)	2.01 ms (497 fps)

TABLE VIII. SUMMARY OF GAME TESTS ON ANDROID 7

Pathfinder	Runtime Total Min	Runtime Total Max	Runtime Total Avg
CPU	7.48 ms (134 fps)	74.66 ms (13 fps)	17.05 ms (59 fps)
GPU (Dynamic 0.125)	426.60 ms (2 fps)	623.78 ms (2 fps)	531.89 ms (2 fps)

TABLE IX. SUMMARY OF SPATIAL AUDIO TESTS ON ANDROID 7

Pathfinder	Spatial Audio Min	Spatial Audio Max	Spatial Audio Avg
CPU	2.12 ms (472 fps)	28.21 ms (35 fps)	3.46 ms (289 fps)
GPU (Dynamic 0.125)	396.82 ms (3 fps)	570.02 ms (2 fps)	476.69 ms (2 fps)

A summary of the Windows 10, and Android 7 results is provided in Tables VI, VII and Tables VIII, IX respectively. Results are similar to those of our benchmark levels, with a noticeable increase in the Runtime Average as a result of increased overall environment complexity (graphics quality, particle systems, physics, player calculations, interactions, etc.). This is not necessarily evidenced in Table 4, although Unity's profiler reported increased times used by several components unrelated to the spatial audio system. Regardless, the spatial audio times are also worse for both the CPU and GPU spatial audio systems than the benchmark level. The explanation for the

CPU system can be attributed to the greatly increased NavMesh complexity.

The GPU system was also less performant than the benchmark level due to a significantly higher node density, despite the smaller bounding volume area. The benchmark level node size was 1.0 m, but for the second game level it is less than half of that at 0.48 m. If the node size was not changed, the total number of nodes would be the same as the bounding volume in m^3, which is 45,000. However, because of the reduced size and the square-cube law, the actual total node count is 403,000 and the density is about eight times as high. This means that as the player moves at full speed (4 m/s), they may trigger up to approximately 16 (4×4) GPU graph recalculations / updates per second with forward movement even with static geometry, as opposed to 10 for the benchmark level. Updates may be faster for ungrounded movement where the player may accelerate downwards from the effects of gravity (e.g., after jumping off the ground, stairs, stacked boxes, etc.)

VI. Summary

Here we have presented G-SpAR, a cross-platform, generic solution for dynamic occlusion and obstruction that is more advanced than the simple raycast and binary techniques most often used in games. We have developed a solution that circumvents many of the current limitations in existing spatial audio technologies to provide a perceptual approximation of spatial cues from the environment, including occlusion, reflection, and diffraction. Leveraging the parallel nature of modern GPUs for processing large amounts of data with a built-in CPU fallback, we were able to integrate a hardware-independent sound propagation system into a cross-platform game as a demonstration of the viability of this implementation. Moreover, our system requires minimal setup, effectively to the point of simply "tagging" sound sources with a custom component that makes the pathfinding requests, while a global manager defines the overall pathfinding variables to use at runtime. Our system is not as accurate in modelling sound propagation as that of more fine-tuned geometrical acoustics and computationally expensive wave-based acoustical modelling. However, sacrificing some accuracy implies that the system is noticeably faster at runtime, requires no time-consuming offline precomputation stage, and is able to perform the calculations within a single frame without the need for asynchronous delays. Future tests can determine if players notice the discrepancy in accuracy, or have a preference for a particular method of spatial rendering. From desktops to mobile smartphones, runtime measurements in an isolated benchmark level and complete game environment demonstrate our low impact on high-end GPUs and scalability to suboptimal hardware when using only the CPU. One current-gen smartphone was able to run the GPU simulation at interactive rates. Although the GPU part of the system is still not suitable for mobile devices when considering large simulation bounds (such as those tested), it is likely that in the future more powerful mobile GPUs will give way to better GPGPU capability, thus improving performance and the viability of GPU-accelerated spatialized sound for large scenes on smartphones. Moreover, further improvements and research can still be undertaken into optimizing the GPU algorithm through exploring the viability of storing level data in different, more easily traversable and efficient graph-like structures, such as octrees, or simply storing the data offline from a precomputation stage, separate from the runtime part. Finally, future work will also examine the effectiveness of G-SpAR via human-based listening tests in order to quantify its accuracy.

Acknowledgment

The authors acknowledge the funding of this work as part of a larger project on sound in games by the *Social Sciences and Humanities Research Council of Canada* (SSHRC). The financial support of *Natural Sciences and Engineering Research Council of Canada* (NSERC) is also acknowledged..

References

[1] B. Kapralos, M. Jenkin, and E. Milios. Virtual audio systems. *Presence: Teleoperators and Virtual Environments*, 17(6): 527–549, 2008.

[2] N. Tsingos, I. Carlbom, G. Elko, T. Funkhouser, and B. Kubli. Validation of acoustical simulations in the "Bell Labs Box." *IEEE Computer Graphics and Applications*, 22(4), 28–37, 2002.

[3] L. Savioja. Modeling techniques for virtual acoustics. PhD thesis, Telecommunications Software and Multimedia Laboratory, Helsinki University of Technology. Helsinki, Finland, 1999.

[4] R. Mehra, N. Raghuvanshi, L. Savioja, M.C. Lin, and D. Manocha. An efficient GPU-based time domain solver for the acoustic wave equation. Applied Acoustics 73:83–94, 2012.

[5] L. Savioja and U.P. Svensson. "Overview of geometrical room acoustic modeling techniques," *Journal of the Acoustical Society of America* 138(2), 708–730 (2015).

[6] M. Jedrzejewski, Computation of Room Acoustics on Programmable Video Hardware. Master's thesis, Polish-Japanese Institute of Information Technology, Warsaw, Poland, 2004.

[7] N. Röber, U. Kaminski, and M. Masuch. Ray acoustics using computer graphics technology. In Proc. of the *International Conference on Digital Audio Effects*, Bordeaux, France, Sep. 10-15, 2007.

[8] Z. Cheng. 2014. Design of a real-time GPU accelerated acoustic simulation engine for interactive applications. Ph.D. Dissertation. University of Illinois at Urbana Champaign. http://hdl.handle.net/2142/50364

[9] N. Tsingos, and N. Gascuel. Soundtracks for computer animation: Sound rendering in dynamic environments with occlusion. Graphics Interface '97, pp. 9-16 (1997).

[10] N. Tsingos, and N. Gascuel. Fast rendering of sound occlusion and diffraction effects for virtual acoustic environments. In *Proc. 104th Convention of the Audio Engineering* pp. 1–14 (1998).

[11] N. Tsingos, T. Funkhouser, A. Ngan, and I. Carlbom. Modeling acoustics in virtual environments using the uniform theory of diffraction. 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2001), pp. 545–552, (2001).

[12] B. Cowan and B. Kapralos. "GPU-based real-time acoustical occlusion modeling," Virtual Reality, 14: 3. pp. 183–196 (2010).

[13] B. Cowan and B. Kapralos. "Interactive rate acoustical occlusion/diffraction modeling for 2D virtual environments & games," In *Proc. 2015 6th International Conference on Information, Intelligence, Systems and Applications* (IISA), Corfu, (2015).

[14] A. Bleiweiss. "GPU Accelerated Pathfinding," In *Proc. 23rd ACM SIGGRAPH/ EUROGRAPHICS Symposium on Graphics Hardware*, pp. 65 – 74 (2008).

[15] S. Lawlor and T. Neumann, "Overwatch - The Elusive Goal: Play by Sound," In *Proc. Game Developer's Conference*, 14 - 18 March, San Francisco, https://www.gdcvault.com/play/1023317/Overwatch-The-Elusive-Goal-Play (2016).

[16] M. Beig. Scalable immersive audio for virtual environments. Master's Thesis., Computer Science, University of Ontario Institute of Technology, may, 2018.