OpenMPD: A low-level presentation engine for Multimodal Particle-based Displays

ROBERTO MONTANO-MURILLO, University College London, Ultraleap, UK
RYUJI HIRAYAMA, University College London, UK
DIEGO MARTINEZ PLASENCIA, University College London, UK

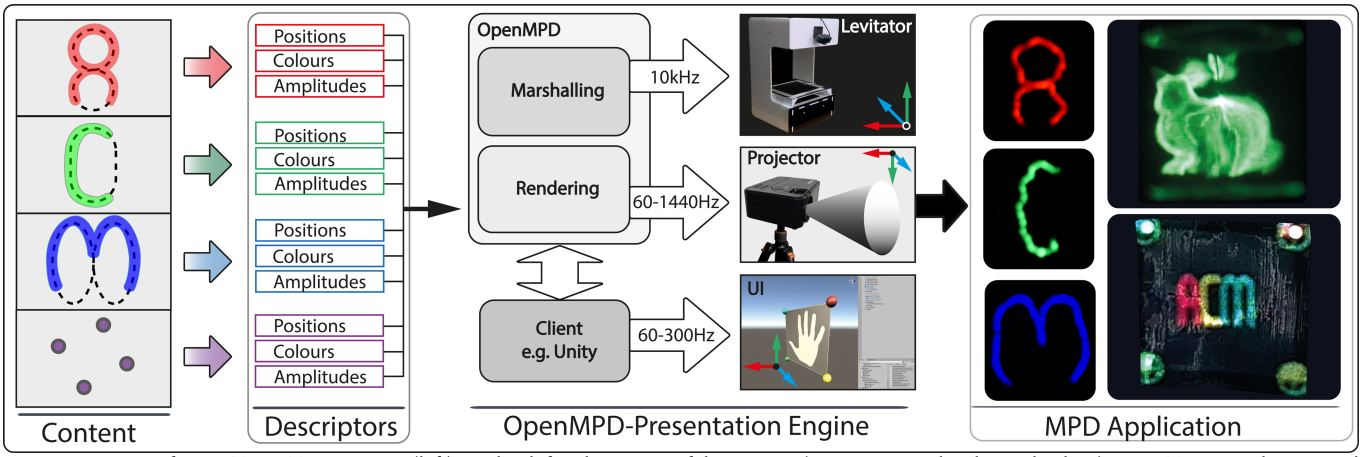


Fig. 1. Overview of OpenMPD: MPD content (left) can be defined in terms of descriptors (positions, amplitudes and colour). OpenMPD coordinates and synchronizes all the processes involved, even if each operates at a different rate (e.g., 10KHz for the acoustic solver, 60Hz for render). OpenMPD can be integrated with higher-level client engines (Unity) and used to present novel MPD content (right).

Phased arrays of transducers have been quickly evolving in terms of software and hardware with applications in haptics (acoustic vibrations), display (levitation) and audio. Most recently, Multimodal Particle-based Displays (MPDs) have even demonstrated volumetric content that can be seen, heard, and felt simultaneously, without additional instrumentation. However, current software tools only support individual modalities and they do not address the integration and exploitation of the multimodal potential of MPDs. This is because there is no standardized presentation pipeline tackling the challenges related to presenting such kind of multi-modal content (e.g., multimodal support, multi-rate synchronization at 10 KHz, visual rendering or synchronization and continuity). This paper presents OpenMPD, a low-level presentation engine that deals with these challenges and allows structured exploitation of any type of MPD content (i.e., visual, tactile, audio). We characterize OpenMPD's performance and illustrate how it can be integrated into higher-level development tools (i.e., Unity game engine). We then illustrate its ability to enable novel presentation capabilities, such as support of multiple MPD contents, dexterous manipulations of fast-moving particles or novel swept-volume MPD content.

Authors' addresses: Roberto Montano-Murillo, University College London, Ultraleap, UK, roberto.montano@ultraleap.com; Ryuji Hirayama, University College London, UK, r.hirayama@ucl.ac.uk; Diego Martinez Plasencia, University College London, UK, d.plasencia@ucl.ac.uk.

0730-0301/2023/4-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

CCS Concepts: • **Computing methodologies** → **Physical simulation**.

Additional Key Words and Phrases: Particle-based Display, Acoustic Levitation, Presentation Engine.

ACM Reference Format:
Roberto Montano-Murillo, Ryuji Hirayama, and Diego Martinez Plasencia. 2023. OpenMPD: A low-level presentation engine for Multimodal Particle-based Displays. *ACM Trans. Graph.* 1, 1 (April 2023), 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Phased arrays of transducers support interactive applications in several domains, such as haptics, levitation or parametric audio. Recent advances in ultrasound control have demonstrated the ability to simultaneously combine all these capabilities, resulting in a new type of technology that we here refer to as Multimodal Particle-based Displays (MPDs). MPDs use physically levitated particles as display elements, either as sparse 3D voxels [Ochiai et al. 2014; Omirou et al. 2015; Sahoo et al. 2016]; as anchors to control levitated props (e.g. threads or 2D shapes) [Morales et al. 2019; Omirou et al. 2016]; or as quickly moving particles revealing 3D content, by exploiting the Persistence of Vision (PoV) effect [Fushimi et al. 2019].

Crucially, such visual MPD content can be combined with other modalities such as haptics or parametric audio [Hirayama et al. 2019; Plasencia et al. 2020], creating multimodal experiences that users can see, hear and feel with their bare eyes, ears and hands, and opening novel opportunities for entertainment or digital signage.

While hardware and algorithms for MPD presentation are becoming available, there is a lack of a standardized underlying presentation engine, as to allow controlled and simple exploitation of the novel range of capabilities of MPDs. This situation is akin to the appearance of OpenGL. While acceleration hardware and rendering algorithms existed before, OpenGL provided manufacturers and programmers with a standard rendering interface, enabling the exploitation of the full capabilities of the hardware and simplifying the creation and portability of 3D applications. We here aim to provide an advanced equivalent to OpenGL, to the field of MPDs.

This paper introduces ***OpenMPD*** (see Figure 1), a low-level presentation engine allowing structured exploitation of any combination of MPD content (i.e., visual, tactile, audio) while dealing with the challenges specific to MPD content presentation (formalised as ***C1-C4*** in Section 3).

For instance, our multi-rate runtime cycle (***C2***) allows us to combine very high computation rates for the sound-field (i.e., 10K soundfields per second, for optimum acoustic control [Hirayama et al. 2019]), with lower rates for control and rendering processes (i.e., hundreds of Hz, as typically used by rendering engines or tracking devices supporting interaction). Our low-level synchronization (***C4***) allows us to retain accurate interoperation among these processes. All of these enable novel capabilities, such as enabling colour projection onto high-speed particles and swept displays, as well as or *dexterous manipulations* of PoV content changing shape or precisely merging with other PoV paths.

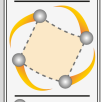
We first describe the abstractions allowing the definition of multimodal content for ***OpenMPD***. Next, we describe the stages within the ***OpenMPD*** presentation engine, detailing key algorithms for data marshalling, acoustic and visual rendering, and synchronization. We then characterize its performance and illustrate how ***OpenMPD*** can be integrated into higher-level tools (i.e., Unity game engine), as a way to facilitate content creation and stimulate MPD adoption. Finally, we use this platform to showcase the presentation capabilities of ***OpenMPD***, such as in allowing combinations of multiple MPD primitives, *dexterous manipulation* of PoV paths (e.g., high-speed particles morphing into different shapes or merging with other particles, to create PoV contents jointly), as well as novel swept-volume MPD content.

2 RELATED WORK AND BACKGROUND

Particle trapping and control has been an extensive field of research within physics, following a range of approaches [Brandt 1989]. Almost all of these approaches have been explored for display purposes, either by using air flows [Heo and Bang 2014], magnets [Lee et al. 2011], optical [Smalley et al. 2018] or electro-magnetic traps [Berthelot and Bonod 2019]. However, only acoustics-based particle displays have demonstrated the multi-modal capabilities of MPDs.

Hence, we focus our review on the range of existing acoustics-based (i.e., ultrasound) displays, particularly, on the requirements of the different types of content that they enable (i.e., visual, tactile, auditive). These requirements will inform the features ***OpenMPD*** must support. Finally, we analyse existing tools supporting the definition of such MPD content (i.e., visual, tactile and sound, either individually or combined) and identify existing gaps in this space, justifying the need for our presentation engine.

Table 1. Summary of existing MPD contents and associated requirements.

	Primitives	Position Update Rates (Hz)	Amplitude Update Rates (Hz)	Continuity Required	References
Independent Particles		< 100 Hz	0 Hz	yes	Omirou et al, 2015 Sahoo et al, 2016 Omirou et al, 2016
Spatially Structured Particles		< 100 Hz	0 Hz	yes	Ochiai et al, 2014 Marzo et al, 2015 Marzo et al, 2018 Morales 2019
PoV Content		10k Hz	0 Hz	yes	Hirayama et al, 2019 Fushimi et al, 2019 Martinez et al,2020
AM Tactile Content		3k Hz	10k Hz	no	Carter et al, 2013 Hoshi et al. 2019
PM Tactile Content		10k Hz	0 Hz	yes	Frier et al. 2018 Takahashi et al. 2018
Audio Source		< 100 Hz	10k - 40k Hz	no	Hirayama et al, 2019 Martinez et al,2020

2.1 Levitation-based displays

Single frequency ultrasound, the most conventional approach towards acoustic levitation, was first observed to trap dust particles in the lobes of a standing wave more than 150 years ago [Stevens 1899] and has been used to support different types of content, which we analyse next and summarize in Table 1.

The earliest type of content was the levitation of *independent particles*, used as 3D display voxels. Early approaches only allowed control of particles at constrained locations [Omirou et al. 2015, 2016; Sahoo et al. 2016], or as a group [Ochiai et al. 2014]. Later advances enabled free 3D positioning of one [Marzo et al. 2015] or several particles [Marzo and Drinkwater 2019] at interactive rates.

Using *spatially structured particles* (i.e. particles connected to physical props, such as threads [Marzo and Drinkwater 2019] or cloth [Fender et al. 2021; Freeman et al. 2019; Morales et al. 2019]), allowed for content featuring continuous and expressive projection surfaces. The underlying requirements regarding ultrasound control algorithms and framerate remained unchanged.

Volumetric *PoV Content* was demonstrated by leveraging single [Hirayama et al. 2019] or multiple fast-moving particles [Plasencia et al. 2020], even in the presence of physical objects [Hirayama et al. 2022] but this required algorithms supporting high update rates of the particle positions and trapping intensities (i.e. ideally, above 10 KHz). While colouring of *PoV Content* was solved for the single particle case [Hirayama et al. 2019], a rendering/illumination solution is still yet to be proposed for the multi-particle case [Plasencia et al. 2020], and one we address in Section 5.4.

Finally, as a common feature of these types of content (*independent particles*, *spatially structured* and *PoV*), they all rely on external

elements to reveal content (i.e., the particles and props), which should not be dropped. Once these elements are loaded into the traps, they must retain *trap continuity* throughout the experience, avoiding sudden jumps as to keep particles/props within the traps.

The same algorithms supporting *PoV Content* also support *Tactile* and *Audio source* content [Plasencia et al. 2020], but their requirements vary depending on the approach. Approaches such as *Amplitude Modulation* (AM) require low position update rates, but their amplitudes need to be rapidly modulated to create different tactile sensations [Carter et al. 2013; Hoshi et al. 2010]. In contrast, approaches based on *Position Modulation* (PM), such as lateral [Takahashi et al. 2018] or spatio-temporal modulation [Frier et al. 2018], rely on very high position update rates. Parametric audio quality (i.e., reproducible frequencies) is directly related to the algorithm's ability to quickly modulate amplitudes (i.e., instead of positions).

Even if all these contents can be created by the same algorithms (GS-PAT), their exploitation requires dealing with very different and demanding requirements, which we summarized in Table 1.

2.2 Content creation support for MPDs

At the moment of writing, there are no platforms explicitly designed to support the creation of the complete range of multi-modal content supported by MPDs. However, some limited support can be found for each independent modality by the platforms below.

Ultraleap SDK [Ultraleap 2022] provides a robust API supporting a range of tactile approaches (i.e., AM, PM), sensing devices and integration within Unity. Their solution, however, is only limited to haptics and their proprietary algorithm [Long et al. 2014]. Ultraino [Marzo et al. 2017] provides a java API to generate levitation traps and even animations. However, the toolkit is focused on acoustics, providing support for low-level simulations rather than for the creation of interactive experiences (e.g., no support for high update rates or sensor devices). Ultraino includes the algorithm described by [Morales et al. 2019], providing a way to optimize the placement of particles on cloth props (i.e., LeviProps) for optimum trapping. However, no further support is provided for other key features such as projection, real-time synchronization or *trap continuity*. OptiTrap [Paneva et al. 2022] is another example of a content creation tool. In this case, the algorithm accepts the shape/PoV path to render, and computes the position and timing of the traps that will lead to optimum rendering. However, this algorithm itself does not provide the low-level mechanisms allowing presentation and relies on an early release of ***OpenMPD*** for presentation.

ArticuLev [Fender et al. 2021] stands as the closest candidate to support MPD presentation. Its main focus is to support initialization for levitation-based displays, detecting props (beads, threads or cloth) and levitating them to their initial location to allow the application to start. However, the platform also provides high-level tool to assemble, articulate and animate them, as well as integration into Unity to facilitate development. However, ArticuLev fails to support high position update rates (i.e., maximum 3 KHz, instead of the 10 KHz required for optimum MPD control), relies on a sub-optimum algorithm (i.e., *Naïve/BP* [Marzo and Drinkwater 2019]), provides no support for amplitude modulation (used by audio and tactile content); and lacks low-level synchronization mechanisms required for projection on multi-particle PoV Content and *dexterous manipulations* (e.g., morphing PoV shapes or combining them).

3 OPENMPD: CHALLENGES AND SCOPE

The MPD contents in Table 1 frame the design space of applications that ***OpenMPD*** must support. Its aim is to provide a standardized presentation engine for their systematic exploitation, which entails addressing the following 4 main challenges:

- *Structured multi-modal content definition (**C1**)*: ***OpenMPD*** allows structured declarative definition of all types of content in Table 1. These definitions allow real-time control and mofidication of content (e.g., changing visual shapes, audio or tactile textures) and are presented in Section 4.
- *Variable update rates (**C2**)*: ***OpenMPD*** provides a multi-rate engine, with synchronized interoperation of 3 processes: sound-field computation (i.e., 10 KHz), rendering and client logic (at hundreds of hertz), as detailed in Sections 5.1-5.3.
- *Multi-particle PoV rendering (**C3**)*: ***OpenMPD*** provides a projector-based rendering algorithm to colour multi-particle PoV content, synchronized to the sound-field computation, described in Section 5.4.
- *Continuity and synchronization requirements (**C4**)*: ***OpenMPD*** provides mechanisms to retain *trap continuity* of the content (i.e., for *independent particles*, *spatially structured* or *PoV*). That is, the levitated components (particles, props) must remain within their traps, even if the shapes change to reveal a new content (i.e., change in PoV shape), or to support *dexterous manipulations* (i.e., controlled interactions among fast-moving particles). Exploiting these requires an interplay of content definition (see Section 4.3), software (see Section 5.2) and hardware mechanisms (detailed next).

3.1 Hardware support and scope of application

The current ***OpenMPD*** implementation supports state-of-the-art algorithms for acoustic manipulation (i.e., *Naïve/BP*, *IBP* [Marzo and Drinkwater 2019]; and *GSPAT* [Plasencia et al. 2020]), all of them operating at optimum rates of 10 KHz. We also use top-bottom setups with 256 transducers per board (see Figure 2a), as the most common setup used in current MPD systems [Freeman et al. 2018; Hirayama et al. 2019; Omirou et al. 2016; Plasencia et al. 2020].

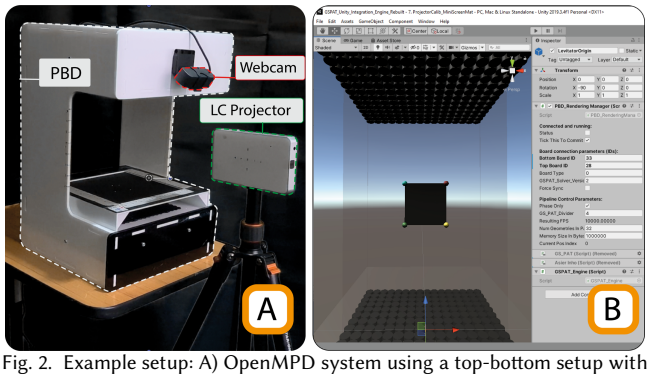


Fig. 2. Example setup: A) OpenMPD system using a top-bottom setup with 512 transducers and an LC projector; B) OpenMPD integration into Unity.

We currently support 2 hardware platforms. The first one is *SonicSurface* [Morales et al. 2021], an open-hardware design that already benefits from a community of early adopters, but is limited to lower update rates (i.e., 100 Hz). The second one is our own hardware, created as an extension of *SonicSurface* but modified for higher update rates (10 KHz), 128 levels of phase resolution and 64 levels of amplitude resolution. Beyond these, the key difference between both hardware platforms lies in their synchronization mechanisms. The boards in *SonicSurface* apply phase updates as soon as a message arrives. This introduces variability in the time at which each board is updated, due to communication delays or OS interruptions, significantly hindering performance if used at high rates of 10 KHz. Our custom boards include a hardware synchronization mechanism ensuring both boards are updated at the same time and at a rate of 10 KHz. Our Supplementary material provides a detailed description of both devices (design, fabrication), as well as an evaluation of the relevance of such hardware synchronization.

It is also worth noting that ***OpenMPD*** can be adapted to other layouts, update rates (e.g., 8, 13, 20 or 40 KHz) or hardware designs [Foisy 2021; Inoue et al. 2018; Marzo et al. 2017; Zehnter and Ament 2019], with further details in Supplementary Material.

OpenMPD is designed to deal with the low-level presentation challenges above, but also to be integrated with higher-level tools (i.e. Unity in Figure 2b) which can deal with other aspects, such as defining the client logic, interfacing with input devices or content detection and initialization [Fender et al. 2021]. Sections 4 and 5 focus on ***OpenMPD***, detailing content definition, its internal stages and mechanisms, while Section 6 showcases our integration with higher level tools (Unity) and examples created on top of them.

4 HIERARCHICAL CONTENT DEFINITION

Solvers such as *GS-PAT* support the (acoustic) delivery of all the types of content summarized in Table 1 (i.e., illumination will require the rendering technique described in Section 5.4). In all cases, content can be defined as a combination of points, and each point is defined as a series of 3D positions and amplitudes over time.

For instance, Figure 3 shows an example of a multimodal content definition. The positions define the location of the trap over time (i.e., a *PoV content*), while the amplitudes encode the audible sound and, in both cases, they are sampled at 10 KHz. The visual appearance (i.e., parts of the shape to be coloured in blue, in Figure 3) can be trivially included as a colour buffer (i.e., colour at each point).

However, even if this definition only contains one cyclic repetition of the PoV content (i.e., 1,000 samples, revealed within 0.1s), this would still require up to 9,000 floating point values to encode it (i.e., assuming homogeneous coordinates and RGBA colour encoding). Any animation (e.g., a butterfly flapping its wings over 2 seconds) would require even larger definitions (e.g., 20K samples, for a total of 180K floating points).

Such naïve definitions would be hardly compatible with real-time updates of the content (e.g., to move it or change the audio), if all this data needed to be updated at the intended rate of 10 KHz. Instead, ***OpenMPD*** uses a hierarchical definition for content, somewhat similar to mesh rendering in OpenGL, as described next.

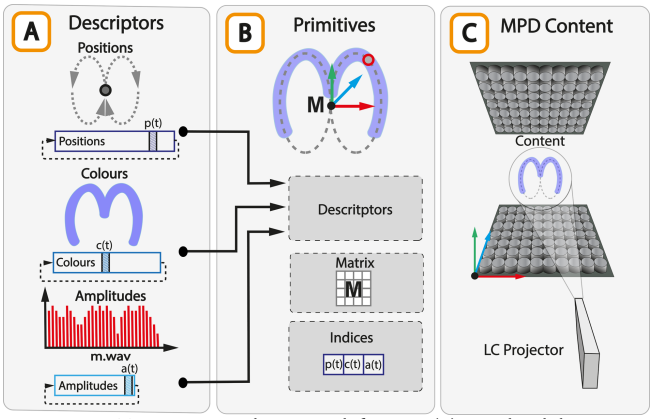


Fig. 3. OpenMPD structured content definition. (A) Low-level descriptors are used to define the content's positions, amplitudes and colours in local coordinates and are sampled at high rates (10KHz). (B) Primitives group descriptors to define MPD contents, which can be freely placed in space using a 4x4 matrix and presented in the device (C).

4.1 Low-level Descriptors: Position, Amplitude and Colour

Descriptors contain low-level definitions of MPD content as buffers of colours, amplitudes or positions in coordinates local to the content (Figure 3a). This allows for a declarative definition of MPD content, as a combination of position, amplitude and colour descriptors, with each element being freely interchangeable.

As such, descriptors are equivalent to OpenGL resources like vertex arrays or textures, which can then be reused across several ***OpenMPD*** contents. *Descriptors* are stored in GPU memory and are internally managed by the engine to present the content at 10 KHz.

OpenMPD assumes all *Descriptors* (independently of their nature) contain a minimum, cyclic definition, which will be repeated until a new *Descriptor* is used (i.e., to morph into a new shape, or to play a new sound or tactile pattern).

4.2 High-level Primitives

Primitives contain high-level definitions for MPD content and are somewhat equivalent to an OpenGL rendering call. The *Primitive* groups the resources (i.e., position, amplitude and colour *Descriptors*) defining the MPD content and a 4x4 matrix to place the *Primitive* freely within the levitator (scene) with minimum CPU/GPU traffic.

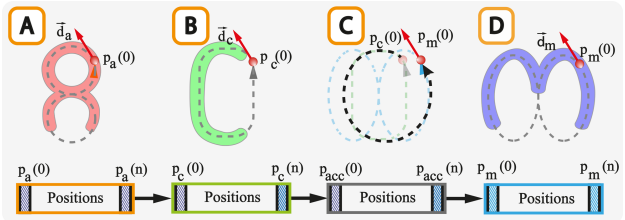


Fig. 4. Cyclic definitions can be ensured by matching the initial and final positions of each path ($p_a(0) = p_a(n)$); Transition continuity is ensured by sharing the same starting positions ($p_a(0) = p_b(0)$), and velocity vectors ($d_a(0) = d_b(0)$), or by using *accommodation paths* (C).

While OpenGL rendering calls are stateless (i.e., the state is represented by the parameters used in the call and thread context), ***OpenMPD*** *Primitives* do retain an internal state. This state describes the matrix used (i.e., update the matrix, instead of several KB of data), the *Descriptors* currently bound to the *Primitive*, and their indices (i.e., current *Descriptor* sample, represented as *p(t)*, *a(t)* and *c(t)* in Figure 3). This state is required in order to retain synchronization and continuity, as explained in Section 4.3.

Please note that the state is associated with the *Primitive* and not to its *Descriptors* (see Figure 3b). Thus, several primitives can reuse the same *Descriptor* to render the same shape or audio, but at different points in space (i.e., *Descriptors* are in local coordinates; *Primitives*' matrices can adjust them to different device's coordinates) and progress in time (i.e., different values for *p(t)*, *a(t)* and *c(t)*).

Even if *Primitives* share one *Descriptor* (e.g., positions), each *Primitive* can combine it differently with other descriptors. That is, even if sharing the same position descriptor, each *Primitive* can use a different colour buffer (i.e., to tailor its visual aspect from 'A' to 'C', as in the example in Figure 1, left), or use a different amplitude descriptor with varying amplitudes, to add sound.

For simple contents, *Primitives* can use descriptors containing a single sample (i.e., descriptors with a fixed position and a fixed amplitude for an *independent particle*). More complex contents, such as *PoV Content* or *AM Tactile* will require descriptors defining the shape (e.g., 0.1s of a 3D path for *PoV Content*) or tactile texture (e.g., 5ms of time-varying amplitudes, for a 200 Hz tactile pattern).

It must be noted that the length of each of the *Descriptors* bound to a *Primitive* can differ. As such, they can retain the minimum length required to encode a given path, sound effect or tactile sensation, independently of the other *Descriptors* they will be used with.

4.3 Dexterous manipulations: Continuous and synchronized transitions

The structured definitions above can describe any type of MPD content as well as dynamic changes by swapping descriptors. However, the most demanding transitions (e.g., *PoV Content*) must retain *trap continuity* to be feasible. ***OpenMPD*** facilitates this by assuming cyclic descriptors and ensuring that descriptor changes are enqueued and only applied when the current descriptor is finished (e.g., when $p(t)$ reaches the end of the buffer, in Figure 3).

This way, the content creator can define compatible *continuous transitions*, by ensuring that the descriptors involved a match their initial positions and velocities, as seen in Figures 4a and 4b. Such *continuous transitions* might not be achievable for certain shapes due to different initial positions, such as the 'M' shape in Figure 4d. In this case, an *accommodation path* can be used to join the initial positions in both descriptors (e.g., see $p_c(0)$ and $p_m(0)$ in Figure 4c)), and will become necessary to morph shapes or slowly accelerate a particle to reach the initial state to render a PoV shape.

It must be noted that *accommodation paths* are by nature non-cyclic descriptors (i.e., they connect different starting positions/states of two paths) and would lead to continuity failure if run cyclically (i.e., in Figure 4c, a particle reaching the end of the *accommodation path* at $p_m(0)$ would be dropped, as the next trap would be created in the non-adjacent initial position $p_c(0)$). As such, the *accommodation path* must be immediately enqueued together with the new cyclic path, so that continuity is retained. That is, a transition from shapes 'A' to 'M' in Figure 4 must be achieved by enqueueing paths P_a and P_m to the primitive as a single transaction (*commit*), but no changes are required to paths P_a or P_m.

5 OPENMPD: PRESENTATION ENGINE

The previous section defined the steps a developer must take to create content. Here we focus on the structure and mechanisms within ***OpenMPD*** that allow such content to be presented.

The ***OpenMPD*** presentation engine is summarized in Figure 5. Boxes represent main data structures while computing resources are represented as rounded rectangles distributed across the CPU (i.e., threads) and GPU (i.e., high-performance acoustic solvers, as OpenCL kernels; and PoV rendering, as OpenGL shaders).

The threads involved include both core ***OpenMPD*** threads, such as sound-field computation or visual rendering, but also services called from the client thread(s) to control the MPD experience.

We here describe the main data structures and functionalities in ***OpenMPD***, according to each of its main threads (i.e., ***OpenMPD***

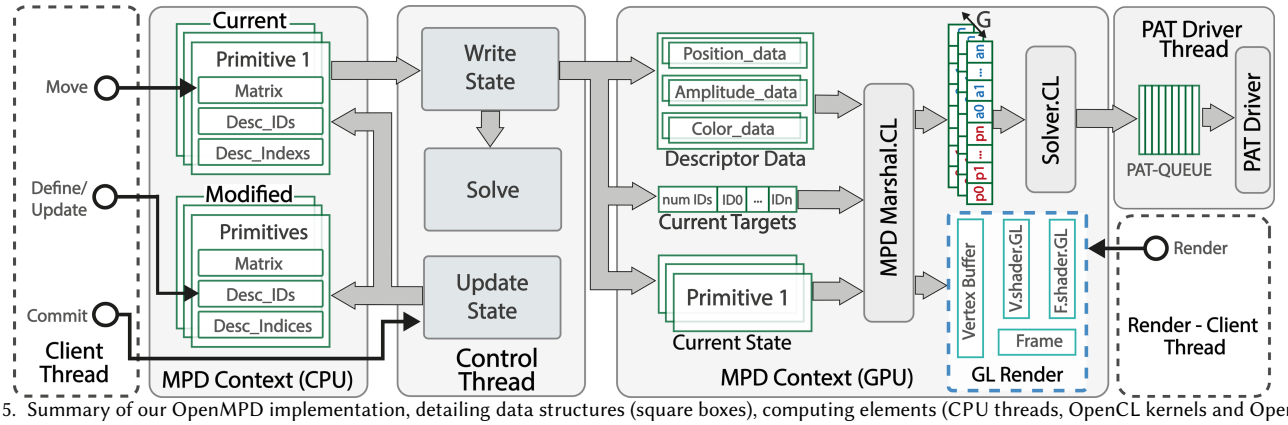


Fig. 5. Summary of our OpenMPD implementation, detailing data structures (square boxes), computing elements (CPU threads, OpenCL kernels and OpenGL shaders, all as rounded boxes) and main data flows among processes.

or client). A comprehensive description (all classes and methods) can be found in Supplementary Material and our GitHub page [1].

5.1 MPD_Context and the client thread:

Clients interact with ***OpenMPD*** through an *MPD_Context*. Such context encapsulates all details related to an MPD experience on a given device, exposing the interfaces required to declare and control all MPD content (i.e., *lollipops*, in Figure 5). An *MPD_Context* currently supports up to 32 simultaneous primitives, considered enough as only up to 12 independent particles have been levitated to date with these setups [Marzo and Drinkwater 2019] and prior MPD experiences involve even fewer primitives [Plasencia et al. 2020].

An *MPD_Context* operates in a dual scene premise. Client calls to *Define/Update* MPD content (e.g., primitives and/or descriptors) are temporarily applied to the *Modified* scene. However, such changes will not come into effect until a call to *commit* is invoked, at which point the *Current* scene is updated. This is crucial to ensure updates on the scene happen in a synchronized manner, which is required for continuity control (i.e., *accommodation* followed by *cyclic* paths, as in Section 4.3) and *dexterous manipulations* (Section 6.2). Also, descriptor updates can be computed offline on the client thread and applied when ready, not interfering with the 10 KHz update rate.

It is worth noting that changes to the transformation matrix in each primitive are an exception to this general rule and are applied immediately to both scenes (i.e., smoothed for continuity, as explained in Section 5.2). This allows the client to control primitive position/orientation in real-time (i.e., see *move* interface in Figure 5), with *commits* being reserved for changes to the structure of the scene/content, where fine synchronization might be needed. Finally, the internal state of each primitive (shown as the current *Desc_Indices* in Figure 5) is not exposed to the client. These are solely managed by ***OpenMPD*** to ensure synchronization at the sound-field rate (i.e., 10 KHz) as well as *trap continuity*.

5.2 OpenMPD control thread

This is arguably the most relevant thread within ***OpenMPD***, dealing with the hybrid CPU/GPU architecture, performing data marshalling required to feed related services (e.g., sound-field computation and visual rendering) and retaining synchronization between their differing rates (i.e., 10 KHz for sound-field, 60-1440 Hz for rendering, 200-300 Hz for client logic). The functionality in this thread is divided into 3 steps, summarized next.

5.2.1 Write State: This stage copies the *Current* scene into the GPU in a single CPU/GPU transaction, copying all potential 32 primitives, even if only a few are active. The specific primitives to be presented are encoded in a separate buffer (*CurrentTargets*), defining the number of primitives currently being used and their IDs.

5.2.2 Solve/Data Marshalling. This stage encapsulates the core functionality in this thread, executed in a single OpenCL kernel (*MPD_Marshal.cl*). The kernel accesses the data on the different *Primitives*' descriptors, adapting them into the formats required by the sound field and rendering threads, as shown in Figure 6.

First, the position and amplitude descriptors in each primitive are interleaved (Figure 6), producing arrays of target positions and amplitudes to be fed to the acoustic solver at the target 10 KHz rate. Second, position and colour information are transformed into vertices describing a coloured strip per primitive (i.e., the particle trail to be rendered). While the sound-field computation is assumed to be fixed at 10 KHz, visual rendering will operate at much lower rates and the marshal accounts for these variable rates. That is, when rendering at 60 Hz (Figure 6), the coloured strips typically span a window of 33 ms (2 frames) worth of positions and colours from the descriptors' current indices (see Section 5.4). This accounts for variability in rendering time and delays related to the projector's processing, but the offset and size of this time window are adjustable by the client. Finally, the positions generated from the *Descriptors* (for the acoustic solver or visual rendering) must be multiplied by the *Primitives*' matrices. However, the client thread is only expected to update such matrices at conventional interactive rates (e.g., 200 Hz, in the example in Figure 6), while the marshal must deal with position samples at a 10 KHz rate, which could cause discontinuities (i.e., 10K/200=50 samples using one matrix, 50 using the next one).

To avoid this, the marshal computes a matrix for each sample as a simple linear interpolation between the two latest matrix updates provided by the client. Such matrix interpolations introduce errors which will only be negligible if orientation updates between matrices are small. By default, our example client (Section 6) limits the update of *Primitive* nodes to 400 mm/s and 400 degrees/s (i.e., a 2-degree difference between updates, for our example client at 200 Hz), to ensure interpolation does not introduce issues. These limits are adjustable by the developer, but otherwise transparent.

5.2.3 Update State: As the last step, the marshal updates the state of each *Primitive*. This includes updating the indices in each of its descriptors, as to indicate the current position/colour/amplitude to be used. This also includes identifying when an iteration has been completed in any given descriptor, transitioning to the next descriptor in the queue (e.g., at the end of an *accommodation path*) or resetting the index to the beginning of the descriptor (i.e., for a *cyclic path*). This step is critical to support the controlled transitions described in Section 4.3 and illustrated in Section 6.2.

5.3 Sound-field computation thread:

This thread is in charge of delivering the data generated by the control thread to the acoustic solver, retrieving the resulting sound-field and transmitting it to the MPD device. ***OpenMPD*** currently supports 2 devices: i) the open-hardware device by (Morales et al., 2021) which updates transducers as soon as a message is received; and ii) our custom extension, with higher update rates and applying updates in synchronization with a real-time engine. Other devices can be included as detailed in Supplementary Material, but these 2 illustrate devices' constraints that ***OpenMPD*** needs to deal with.

In order to support these, ***OpenMPD*** includes 2 modes of operation: one for devices with hardware synchronization (see Section 3.1) and another using software synchronization. In the first case,

[1] https://github.com/RMResearch/OpenMPD.git

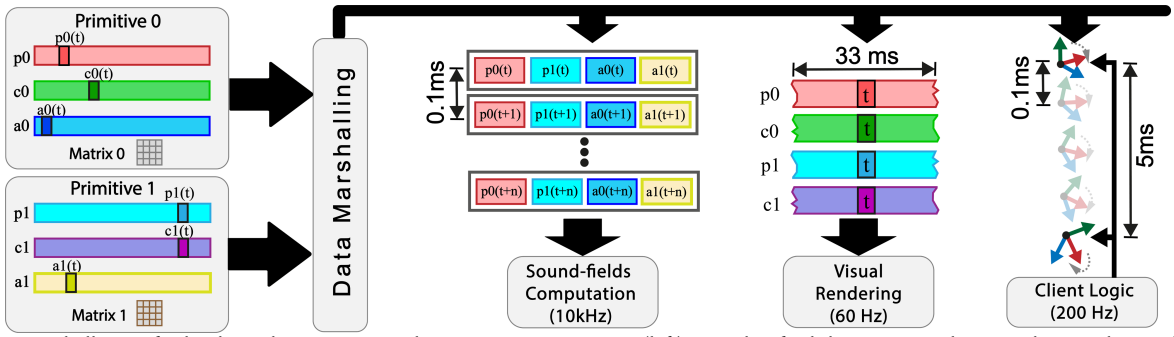


Fig. 6. Data marshalling to feed independent processes. The current Primitive state (left) is used to feed the acoustic solver, interleaving data and feeding it at 10KHz. Buffers for visual rendering are updated, selecting the positions and colours to be rendered in the next frame. Position updates (provided at 200 matrices/s) are interpolated to produce 10K matrices/s and fed to the solver.

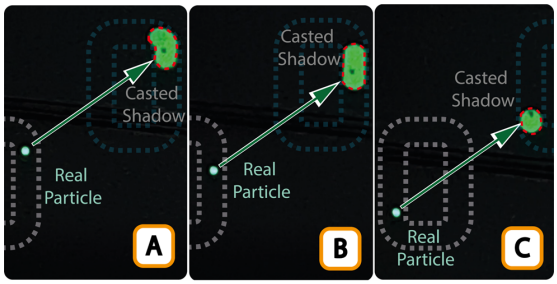


Fig. 7. Color projection example: particle moving along a closed path and color projection onto each section of the path.

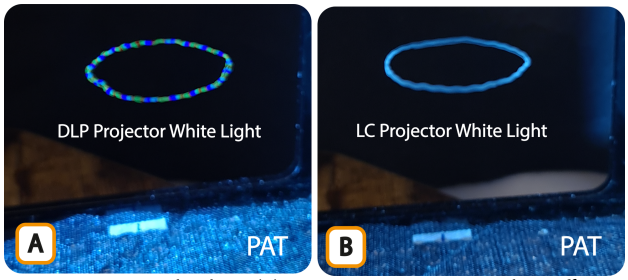


Fig. 8. Projection technology. (A) DLP projectors cause rainbow effects on the content; (B) LC projectors allow for correct colour reproduction.

the thread sends update packages at the intended rate (10 KHz), but it is the devices' responsibility to apply them at the correct time, using their internal real-time clock. In the second case, the thread uses the CPU clock to determine when each package should be sent, and the device is expected to apply them as soon as they arrive. Critical deadlines cannot be ensured in this way, as communication delays or the OS thread scheduler might introduce variable delays.

This thread also encapsulates 3 different acoustic solvers: *Naïve* [Marzo and Drinkwater 2019], *IBP* [Marzo and Drinkwater 2019] ; and *GS-PAT* [Plasencia et al. 2020], all implemented on top of OpenCL for multi-platform support. The *Naïve* solver was selected because of its relative simplicity, while *GS-PAT* provides more robust solutions and is compatible with devices allowing phase and amplitude modulation. *IBP* provides similar robustness to *GS-PAT* but is limited to phase-only devices (i.e., suboptimum for audio and haptics) and provides limited support for high computing rates (i.e., limited to less than 8 simultaneous primitives). A characterization of their performance, stability, the effect of synchronization and support for different primitives is later discussed in Section 5.5.

5.4 Visual rendering

OpenMPD includes a projector-based solution for visual rendering of any MPD content, particularly necessary for multiple, high-speed particles (i.e., a challenge not covered to date). Support is provided via extension DLLs, remaining independent from specific technologies (e.g., OpenGL, DirectX). Our example OpenGL implementation (*GL_Renderer*, to the bottom left of Figure 5) illustrates this behaviour and is used in our client integration in Section 6.

In order to remain implementation agnostic, the control thread shares the (position and colour) descriptor definition buffers with the rendering DLL, which is responsible for translating them into the adequate format (e.g., OpenGL vertex buffers). During runtime, the control thread specifies the indices within these buffers (particle positions), as well as the transformation matrices that should be used in the current frame, as seen in Section 5.2. The number of particle positions (indices) to reveal in each frame can be tailored by the client as to adapt to the projector's framerate used (e.g., 33ms of positions in Figure 6). This allows projection only on the parts of the path that the particle will reveal during that frame (see Figure 7), minimizing interference with other contents projected (e.g., paths overlaid on the image projected on a LeviProp).

Please note that projector synchronization plays a key role to effective rendering with ***OpenMPD***. While ***OpenMPD*** will retain synchronization across processes (control, sound-field and render) and acoustic boards, mechanisms derived from VESA Adaptive Sync must be used to retain projector frame synchronization.

The client thread is in turn responsible for launching the required (OpenGL) rendering threads, as well as invoking the rendering DLL once per frame, providing the P (Projection) and V (View) matrices describing the position of the virtual cameras. The client is also responsible for projector calibration (i.e., P and V matrix), as well as the rendering of any content other than particles. This retains ***OpenMPD*** independence from the details of the underlying 3D scene, which is solely managed by the client. Our solution is also compatible with multi-projector setups, at the expense of clients

Table 2. OpenMPD performance per solvers and number of Primitives.

Solvers	Particles	Maximum Rate	Cyclic	Transitions
Naive	2	40001 ± 91.97	10000 ± 2.05	10000 ± 2.41
	4	40000 ± 18.02	10000 ± 2.13	10000 ± 2.08
	6	39991 ± 75.56	10000 ± 10.19	10000 ± 2.06
	8	40000 ± 10.94	10000 ± 7.32	10000 ± 2.50
	16	—	—	—
IBP	2	40000 ± 46.89	10000 ± 2.60	10000 ± 2.56
	4	40000 ± 11.96	10000 ± 2.48	10000 ± 2.04
	6	40000 ± 15.48	10000 ± 2.18	10000 ± 2.27
	8	40000 ± 75.90	10000 ± 3.13	10000 ± 2.41
	16	—	—	—
GSPAT	2	15668 ± 178.2	10000 ± 3.30	10000 ± 5.83
	4	15683 ± 203.2	10000 ± 7.55	10000 ± 6.58
	6	15374 ± 113.9	10000 ± 9.44	10000 ± 8.92
	8	13906 ± 239.5	10000 ± 3.01	10000 ± 4.66
	16	12889 ± 171.8	10000 ± 2.33	10000 ± 5.71

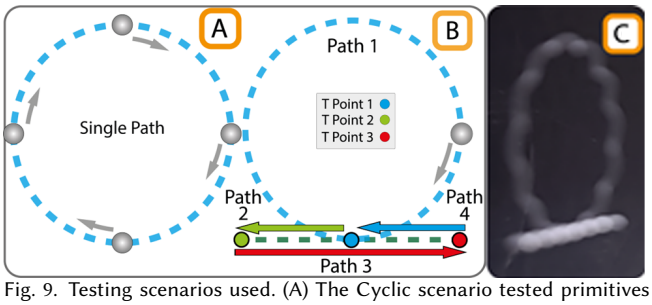


Fig. 9. Testing scenarios used. (A) The Cyclic scenario tested primitives cyclically traversing a circular path. (B) *Transitions* combines circular and linear paths. (C) Example of the *Transitions* case, shown as a POV Primitive.

calling the rendering DLL once per frame and camera/projector, as illustrated in Section 6.3.

Rendering on high-speed particles requires projectors that do not make use of time-multiplexing techniques. Figure 8a shows the effects of using such a projector (DLP technology) to render a white path on a PoV particle moving along a circle. Time multiplexing causes a rainbow effect on the path (i.e., alternating red, green and blue patches, due to the colour wheel), not reproducing the intended colours. Figure 8b shows correct colouring can be achieved by using an LC projector, avoiding such rainbow effects.

5.5 Performance characterization and validation

Our performance characterization and validation is divided into 2 sections, depending on whether they characterize the performance of key ***OpenMPD*** mechanisms (see 5.5.1.), or if they validate the performance of previously existing elements (e.g., solvers) integrated within ***OpenMPD*** (summary in 5.5.2, details in SM3).

5.5.1 OpenMPD Characterization: Table 2 characterizes the computing rates of the 3 solvers supported, for a variable number of primitives (2, 4, 8, 12 and 16). The column *Max Rate* describes the maximum rate that each solver can achieve (capped at transducers rate of 40 KHz), as the number of particles increases.

It is worth clarifying why *IBP* and *Naïve* achieve higher maximum rates than *GS-PAT* (i.e., given that *GS-PAT* was designed explicitly for high rates). In order to push *IBP* and *Naïve* above 10 KHz, we make aggressive use of the local memory cache in every OpenCL computing group. However, the caches in the GPUs tested (i.e., GTX1050Ti, GTX 1660) can only hold data (i.e., propagation matrices) for up to 8 primitives on a 512-transducer device.

In contrast, our *GS-PAT* implementation uses the general GPU VRAM, with worse performance but allowing higher scalability. Similar optimizations could be included for *GS-PAT*, at the expense of a more complex control (e.g., use the cache but dynamically swap to the VRAM implementation, for more primitives).

In any case, these tests show how all solvers provide rates above the target rate of 10 KHz (with *IBP* and *Naïve* limited to 8 points). However, while maximum rates are important, maintaining a stable framerate is much more critical. Any descriptor contains shapes, audio and tactile textures, all sampled at a specific target rate (e.g.,10 KHz), and accurate delivery relies on them being presented at that exact rate. Other features, such as *trap continuity* and *dexterous manipulations* also heavily rely on a stable framerate over time.

As a next step, we tested the ability of each solver to retain stable framerates at a fixed rate (10 KHz), as the number of *Primitives* changed. We tested 2 different scenarios: *Cyclic* and *Transitions*. The *Cyclic* scenario used several *Primitives* cyclically travelling along a circular path, at even distances. The *Transitions* scenario presents *Primitives* repeatedly transitioning between a circular and a linear path descriptor (see Figure 9), to see if these transitions affected framerate stability. We conducted 20 profiling sessions of 5 seconds per case (solver and number of primitives) and report the average framerate and standard deviations measured in Table 2.

The results show very stable framerates with minimum standard deviations, even where *trap continuity* is required, illustrating ***OpenMPD***'s ability to support all primitives in Table 1. Solvers without amplitude control are not suitable for some haptics and audio primitives, and developers must select a suitable solver.

5.5.2 OpenMPD Validation. The characterization above focuses on ***OpenMPD***'s software capabilities to present content, independently of its actual feasibility. We also conducted speed tests with single and multiple particles (i.e., for display/levitation), as well as characterization (spectrograms) of ***OpenMPD***'s ability to deal with variable amplitudes (i.e., for audio and haptics). These tests compare the performance of our 3 solvers, either using native C++ clients or our Unity integration, and with and without the hardware synchronization mechanism introduced by our hardware.

The details of these tests are relegated to Supplementary Material SM3, as they are more closely related to characterizing the performance of previously existing acoustic solvers, rather than key mechanisms within ***OpenMPD***. However, the results validate the ability of ***OpenMPD*** to represent all required modalities, providing consistent results for all solvers (albeit according to each solver's capabilities). The results also show the relevance of our hardware synchronization mechanism, sometimes tripling the maximum accelerations achievable when compared to a device without such mechanism. Finally, the results show negligible effects of the type of client used (native C++ or Unity), indicating minimum overhead is added by integrating ***OpenMPD*** into a higher-level tool.

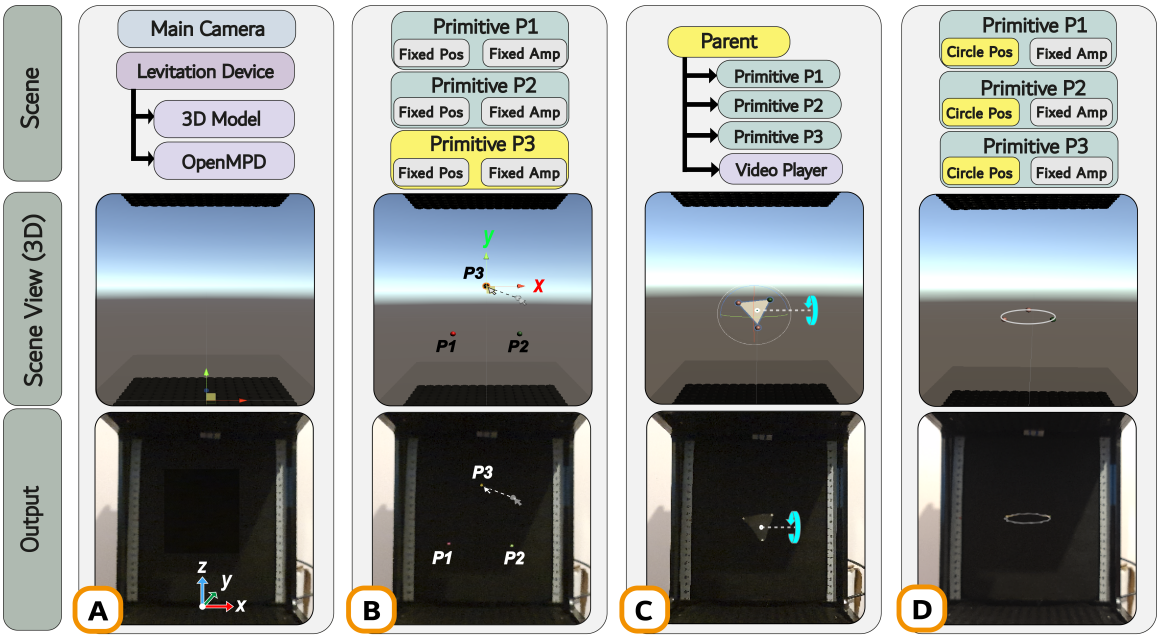


Fig. 10. Examples of simple content creation with our client: (A) Initial empty scene (top), with its virtual representation (middle) matching the real device 1:1 (bottom); (B) Example of using *Primitives* as Unity nodes, to support *independent particles*; (C) extended with extra Unity nodes to support LeviProps; (D) Updating the *Primitives* position descriptors to produce PoV content.

6 EXAMPLE CLIENT AND APPLICATIONS

We created an example ***OpenMPD*** client, using Unity 3D as the host game engine (see Figure 10). Unity was chosen as an illustrative example, but also due to its wide adoption and community support. In any case, ***OpenMPD*** is not limited to this and could be integrated into other game engines (e.g., Unreal, Ogre3D).

We used this Unity integration to implement a set of examples, showcasing key functionalities within ***OpenMPD***. Details on how to implement these, as well as additional examples (i.e., multimodal primitives and OptiTrap shapes) with ***OpenMPD*** and this Unity integration, can be found in Supplementary Material SM4. For further examples, tutorials and up-to-date documentation on ***OpenMPD*** and our Unity integration, please refer to our GitHub website[1].

6.1 Basic OpenMPD content creation

Figure 10a shows an empty ***OpenMPD*** scene, used as the starting point to create applications. The scene contains a node (*Levitation Device*) with a 3D model of the device used, as well as a script providing access to ***OpenMPD*** functions (i.e., a C# sharp wrapper for our C++ DLLs). The Unity integration operates under the premise of maintaining a 1:1 match between the virtual scene and the real world. The 3D model of the device allows content creators to define primitives within the device's working volume, and the ***OpenMPD*** script deals with differing axis conventions between the client (Unity) and the device (see different axis orientations in middle and bottom images in Figure 10a).

Developers can create MPD content simply by adding *Primitive* prefab nodes to the 3D scene (Figure 10b). Physical traps are automatically created in the real device to match the position of the virtual *Primitive* nodes, which the developer can control simply by moving the nodes in the Unity scene. By default, *Primitives* use fixed position (i.e., one sample at (0,0,0,1)) and amplitude descriptors (i.e., one sample with amplitude 1), easily supporting applications using *independent particles* (see Figure 10b, bottom).

More complex content can be created by adding additional Unity components, like a 3D textured triangle to create a LeviProp (i.e., *spatially structured particles*, in Table 1). In Figure 10c, we use a *Parent* Unity scene node, to group and move children *Primitives* as a whole, and other conventional Unity nodes to add a 3D object (e.g., triangle polygon) with a dynamic texture (*VideoPlayer*).

Other types of content in Table 1 require replacing the default descriptors in each *Primitive*. For instance, in Figure 10d, the default (fixed) position descriptors have been replaced by descriptors defining a circular PoV path (e.g., *Circle_Pos*). The Unity scene (middle) shows the positions for each *Primitive*, as well as the coloured visual paths (see Section 5.4), which are synchronized with the acoustic solver and physical particles (Figure 10d, bottom).

Replacing the amplitude descriptor in any of these primitives by one encoding a time-varying can transform any of these primitives into a *AM Tactile Content* or an *Audio Source* (Table 1). This is illustrated in our video figure and Supplementary material includes tutorials to demonstrate both the creation of haptics and audio.

Please note, the developer must make sure to follow the considerations discussed in Section 4 when creating descriptors (i.e., cyclic definitions, *accommodation paths*, grouping synchronized transitions into a single *commit*) in order to ensure correct operation. Similarly, the developer will need to calibrate the LC projector (i.e., intrinsics/extrinsics) and apply such settings to the camera (i.e. P,V

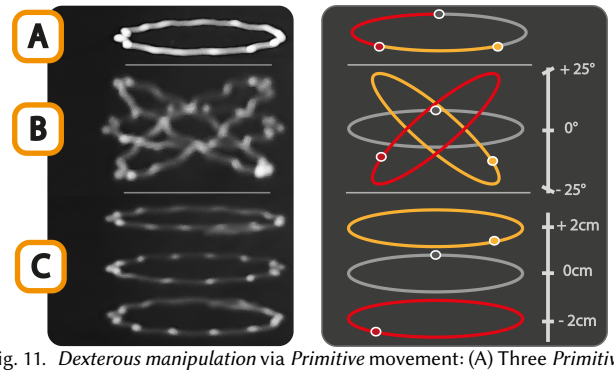


Fig. 11. *Dexterous manipulation* via *Primitive* movement: (A) Three *Primitives* overlap, to render a single PoV circle; *Primitives* change orientation (B) or position (C), allowing synchronized transition to other effects.

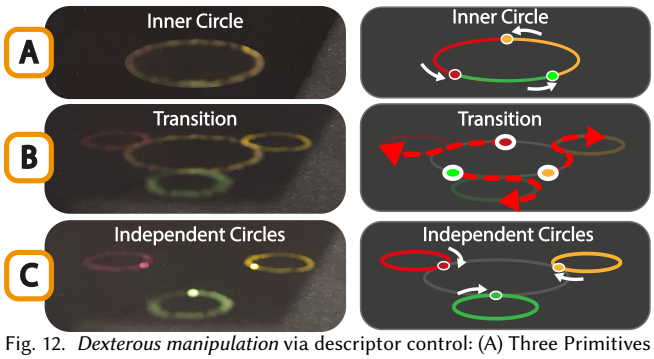


Fig. 12. *Dexterous manipulation* via descriptor control: (A) Three Primitives rendering a PoV circle. (B) Using *accommodation paths* to transition to a new initial position (and speed); (C) Primitives in their new cyclic state.

matrices), to ensure correct projection of the virtual content onto the physical particles and props.

Our GitHub implementation[1] contains some utility Unity scripts to help support some of these steps (e.g., define descriptors, projector calibration) and other types of content (*AM/PD Tactile* and *Audio* contents, as shown in the video figure). This is far from a complete content creation suite and one that we will continue to extend, (hopefully) with the support of the ***OpenMPD*** developer community.

6.2 Dexterous Manipulations with OpenMPD

The examples in the previous section and the accompanying video figure demonstrate the use of ***OpenMPD*** to create the range of content described in Table 1, with simple integration into a higher-level client host (Unity). However, ***OpenMPD***'s synchronization mechanisms allow for new types of content arising from the accurate control of high-speed moving particles, which we refer to as *dexterous manipulations*.

Figure 11 extends the example in Figure 10d, by modifying the location (i.e., matrix) of each of the 3 *Primitives* in the circle. Independent rotations applied to each *Primitive* can create an atom-like structure, as seen in Figure 11b. Alternatively, each circle can be translated up and down (Figure 11c). In either case, it is ***OpenMPD*** synchronization that allows particles to split or to merge back into a circle, avoiding particle collisions. Please note that all these manipulations simply require updates to each *Primitives*' matrix (i.e., updated at the client's rate of 200-300 Hz), with ***OpenMPD*** transparently managing synchronization with the solver at 10 KHz.

Figure 12 illustrates another example of *dexterous manipulation*, this time as the result of synchronized updates to their position descriptors. Here, the 3 previous *Primitives* tracing the circle are each provided an *accommodation path* and a new circle descriptor (all committed into a single transaction). This allows controlled transitions from a circle (Figure 12a) to 3 independent PoV circles (Figure 12b), or back to the initial circle (Figure 12c), when *Primitive*'s paths are updated again.

Please note that it is the developer's responsibility to design and use compatible paths (i.e., *accommodation* or *cyclic paths*, with position/speed continuity; see Section 4.3). Our examples illustrate how correct usage of ***OpenMPD*** mechanisms enables such control.

6.3 Swept volume MPD content

This section describes an example of MPD content inspired by swept-volume volumetric displays (see Figure 13), such as Voxon Vx1 [Vox 2022] or Perspecta [Cossairt et al. 2007]. Such content is implemented as a *spatially structured particles* primitive (i.e., a LeviProp [Morales et al. 2019]), built with a square of 3x3cm SuperOrganza fabric attached to 4 particles. The prop quickly spins around the vertical axis at 5 revolutions per second, so that each 3D voxel is revealed at 10 Hz (i.e., PoV rate). We use a custom rendering engine (implemented using OpenGL and glfw), allowing access to the low-level synchronization and timing options, required to maintain the consistency of the projection on the prop. This is combined with a LightCrafter projection engine. This projector allows 24 binary images (1bpp) to be encoded into a single RGB colour image (24bpp), allowing a final rendering rate of 1440 fps. As it only uses one colour channel at 1bpp, no issues related to DLP time multiplexing occur.

To reveal our model 10 times per second at 1440 fps, we precomputed 144 slices of our example model (i.e., Stanford bunny) and stored them in a single texture array. The projection was managed as a conventional projection mapping problem, with a scene reproducing the 4 *Primitives* (with required descriptors), the physical prop (i.e., textured quad matching the shape and location of the LeviProp) and the virtual cameras matching the intrinsics and extrinsics of our projector. Even though we use V-Sync for rendering, small variabilities in the output timing would result in instability and jitter on the content projected. We avoided this by making use of the processors' clock, and adjusting the textures to render at each point according to the time measurements.

Our setup uses a mirror, physically splitting our projector output into two separate outputs (i.e., virtual cameras), each from a different perspective. This ensures that at least one camera/projection is able to project onto the prop, even if the prop is parallel to the other view (i.e., the prop's normal is perpendicular to the camera's Z vector). This example illustrates the relevance of the low-level synchronization between the rendering process and the acoustic solver described in Section 5.4, but also the ability to combine particle rendering with other outputs produced by the client (i.e., the texture slices, revealing the bunny) and multi-projector support.

Our solution is still limited by the remaining hardware used. A graphics card providing accurate and steady frame delivery (e.g.,

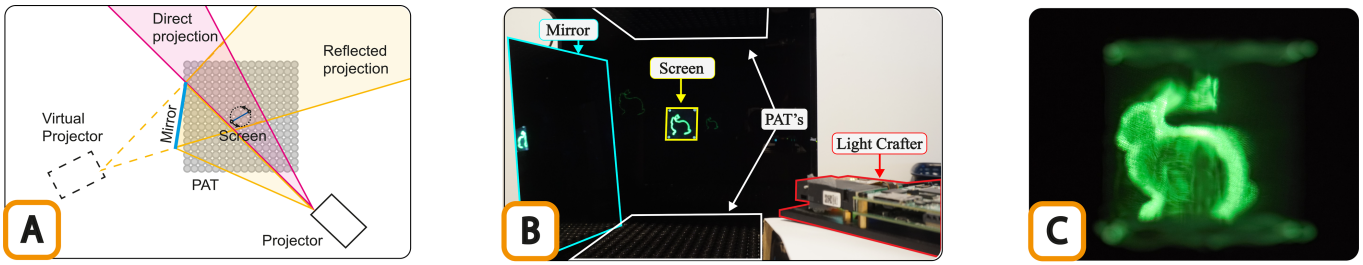


Fig. 13. Example swept volume MPD Content. (A) Diagram of our setup, including PATs, a projector and a LeviProp, spinning 5Hz (each voxel revealed at 10Hz); (B) Image of the real setup and example slice across a section of the model; (C) Final result.

Quadro Sync) would remove the need for dynamically adjusting projection or would allow for synchronized multi-projector solutions (i.e., instead of physically splitting the output of a single projector). Manual tuning of the GPU to projector delay is required (i.e., see '*Test*' pattern in our video figure), which could be avoided by using techniques such as those in [Knibbe et al. 2015].

7 DISCUSION

OpenMPD provides an open platform to support the delivery of novel and underexplored types of content enabled by MPD technologies. The current implementation covers the range of existing MPD primitives, the most common setups and acoustic solvers, and provides dedicated mechanisms to retain low-level synchronization, key for visual rendering of multi-particle PoV content, or to enable *dexterous manipulations*.

However, beyond this set of individual features, ***OpenMPD*** stands as an effort to standardize the definition and exploitation of MPD content. We strongly believe this is an important and timely step towards increased adoption of MPD technologies. First, it provides an open standard for the diverse range of PAT devices available. Practitioners [Foisy 2021], researchers [Inoue et al. 2018; Marzo et al. 2017; Zehnter and Ament 2019] or companies [Ultraleap 2022]) can contribute ***OpenMPD*** drivers for their devices, advancing towards reusability and portability of MPD applications. This can also facilitate reproducibility and baseline comparisons against prior art for new advances (e.g., solvers, devices) in the field.

Second, our advocacy for open-software, combined with open-hardware and integration with a popular engine (Unity) attempts to facilitate access to this technology to researchers, 3D developers and media artists, as a way to stimulate the development of a community around this technology. Such contributions could include simply novel applications (i.e., by using the current implementation), novel tools to support content creation (i.e., by extending our Unity scripts) or even contributions to the core of ***OpenMPD*** (i.e., improvements to performance, extend rendering to DirectX, CUDA or Vulcan, support for novel acoustic solvers).

Interactions with our early adopters (i.e., researchers and artists across 9 countries and 4 continents) showed us the very different approaches and workflows that each of them uses. The mechanisms, guidelines and tools included in ***OpenMPD***, while necessary for the exploitation of MPD content, are far from sufficient to address their content creation needs. Specific higher-level content creation tools are needed (e.g., akin to Gimp or Blender for 3D industries), as the content creator is still responsible for providing feasible content (e.g., path descriptors). While this philosophy is no different to, say, OpenGL (e.g., the client must provide correct winding order, normals, tangents, etc.), the aspects that an ***OpenMPD*** developer needs to consider to create feasible content involve specific challenges related to dealing with a physical system.

For instance, as illustrated by [Paneva et al. 2022], the content defined by the developer must comply with the maximum speeds and accelerations that the device can provide. Their algorithm allows a single particle to reveal the desired shape while complying with these constraints. However, content creation algorithms should also be considered to deal with multiple particles (i.e., to reveal the same shape or to deal with several independent MPD primitives), to allow dynamic transformations (e.g., rotations, translations), or even to deal with the variability in the props used (i.e., particles, cloth).

Other required additions would include the integration of tools to deal with the detection and initialization stages of MPDs [Fender et al. 2021] and dedicated support for sensor technologies. Further advances could include intuitive tools to define PoV paths, but also their colour and amplitude at each point, better rendering methods (e.g., beyond our projector-based solution) or even support for larger transducer arrays or other transducer layouts.

8 CONCLUSION

We presented ***OpenMPD***, a low-level presentation engine supporting all types of MPD content proposed to date, integrating state of the art acoustic algorithms and dealing with the pitfalls and synchronization requirements required to support real time control of the MPD content (e.g., interactive control), visual rendering (e.g., multi-particle and/or swept-volume PoV content) and device control at optimum rates (i.e., 10 KHz).

We provided the abstractions required to allow structured definition of all types of MPD content (i.e., in terms of *Descriptors* and *Primitives*), and described the ***OpenMPD*** implementation, focussing on the mechanisms that allow it to cope with multi-rate processes (i.e., acoustic solvers at 10 KHz, visual render and logic at hundreds of Hz), multi-particle PoV rendering and low-level synchronization mechanisms to ensure *trap continuity*. We also characterize the performance of our engine, demonstrating stable framerates around 10 KHz, independently of the number of particles or solver used (i.e., *Naïve*, *IBP* or *GS-PAT*). We then demonstrated integration of ***OpenMPD*** into a high-level engine (Unity), using it to illustrate a range of contents raging from independent particles to LeviProps, PoV content, as well as novel possibilities, such as *dexterous manipulations* and swept-volume MPD content. Most

importantly, ***OpenMPD*** offers a completely open platform aiming to standardize and stimulate access and adoption of MPD technology.

ACKNOWLEDGMENTS

This work was supported by the AHRC UK-China Research-Industry Creative Partnerships (AH/T01136X/2), by the EU Horizon 2020 research and innovation programme under grant agreement No 101017746 and by EPSRC through their prosperity partnership program (EP/V037846/1). The authors would also like to thank Mr. Eimontas Jankauskis for his support in creating the videos and figures.

REFERENCES

2022. *Photonics Voxon.* voxon.co/voxon-vx1-available-for-purchase/

Johann Berthelot and Nicolas Bonod. 2019. Free-space micro-graphics with electrically driven levitated light scatterers. *Opt. Lett.* 44, 6 (Mar 2019), 1476–1479. https://doi.org/10.1364/OL.44.001476

E. H. Brandt. 1989. Levitation in Physics. *Science* 243, 4889 (1989), 349–355. https://doi.org/10.1126/science.243.4889.349 arXiv:www.science.org/doi/pdf/10.1126/science.243.4889.349

Tom Carter, Sue Ann Seah, Benjamin Long, Bruce Drinkwater, and Sriram Subramanian. 2013. UltraHaptics: Multi-Point Mid-Air Haptic Feedback for Touch Surfaces. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology* (St. Andrews, Scotland, United Kingdom) *(UIST '13)*. Association for Computing Machinery, New York, NY, USA, 505–514. https://doi.org/10.1145/2501988.2502018

Oliver S. Cossairt, Joshua Napoli, Samuel L. Hill, Rick K. Dorval, and Gregg E. Favalora. 2007. Occlusion-capable multiview volumetric three-dimensional display. *Appl. Opt.* 46, 8 (Mar 2007), 1244–1250. https://doi.org/10.1364/AO.46.001244

Andreas Rene Fender, Diego Martinez Plasencia, and Sriram Subramanian. 2021. ArticuLev: An Integrated Self-Assembly Pipeline for Articulated Multi-Bead Levitation Primitives. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 422, 12 pages. https://doi.org/10.1145/3411764.3445342

Dan Foisy. 2021. How a Basement Inventor Builds Volumetric Displays. In *ACM SIGGRAPH 2021 Labs* (Virtual Event, USA) *(SIGGRAPH '21)*. Association for Computing Machinery, New York, NY, USA, Article 2, 2 pages. https://doi.org/10.1145/3450616.3470531

Euan Freeman, Asier Marzo, Praxitelis B. Kourtelos, Julie R. Williamson, and Stephen Brewster. 2019. Enhancing Physical Objects with Actuated Levitating Particles. In *Proceedings of the 8th ACM International Symposium on Pervasive Displays* (Palermo, Italy) *(PerDis '19)*. Association for Computing Machinery, New York, NY, USA, Article 2, 7 pages. https://doi.org/10.1145/3321335.3324939

Euan Freeman, Julie Williamson, Sriram Subramanian, and Stephen Brewster. 2018. Point-and-Shake: Selecting from Levitating Object Displays. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3173574.3173592

William Frier, Damien Ablart, Jamie Chilles, Benjamin Long, Marcello Giordano, Marianna Obrist, and Sriram Subramanian. 2018. Using spatiotemporal modulation to draw tactile patterns in mid-air. In *International Conference on Human Haptic Sensing and Touch Enabled Computer Applications*. Springer, Springer, 270–281. https://doi.org/doi.org/10.1145/3173574.3173592

Tatsuki Fushimi, Asier Marzo, Bruce W Drinkwater, and Thomas L Hill. 2019. Acoustophoretic volumetric displays using a fast-moving levitated particle. *Applied Physics Letters* 115, 6 (2019), 064101. https://doi.org/doi.org/10.1063/1.5113467

Yunsil Heo and Hyunwoo Bang. 2014. Levitate. *Leonardo* 47, 4 (08 2014), 402–403. https://doi.org/10.1162/LEON_a_00849 arXiv:direct.mit.edu/leon/article-pdf/47/4/402/1575878/leon_a_00849.pdf

Ryuji Hirayama, Giorgos Christopoulos, Diego Martinez Plasencia, and Sriram Subramanian. 2022. High-speed acoustic holography with arbitrary scattering objects. *Science Advances* 8, 24 (2022), eabn7614. https://doi.org/10.1126/sciadv.abn7614 arXiv:https://www.science.org/doi/pdf/10.1126/sciadv.abn7614

Ryuji Hirayama, Diego Martinez Plasencia, Nobuyuki Masuda, and Sriram Subramanian. 2019. A volumetric display for visual, tactile and audio presentation using acoustic trapping. *Nature* 575, 7782 (2019), 320–323.

Takayuki Hoshi, Masafumi Takahashi, Takayuki Iwamoto, and Hiroyuki Shinoda. 2010. Noncontact Tactile Display Based on Radiation Pressure of Airborne Ultrasound. *IEEE Transactions on Haptics* 3, 3 (2010), 155–165. https://doi.org/10.1109/TOH.2010.4

Seki Inoue, Yasutoshi Makino, and Hiroyuki Shinoda. 2018. Scalable Architecture for Airborne Ultrasound Tactile Display. In *Haptic Interaction*, Shoichi Hasegawa, Masashi Konyo, Ki-Uk Kyung, Takuya Nojima, and Hiroyuki Kajimoto (Eds.). Springer Singapore, Singapore, 99–103. https://doi.org/doi.org/10.1007/978-981-10-4157-0_17

Jarrod Knibbe, Hrvoje Benko, and Andrew D. Wilson. 2015. Juggling the Effects of Latency: Software Approaches to Minimizing Latency in Dynamic Projector-Camera Systems. In *Adjunct Proceedings of the 28th Annual ACM Symposium on User Interface Software and Technology* (Daegu, Kyungpook, Republic of Korea) *(UIST '15 Adjunct)*. Association for Computing Machinery, New York, NY, USA, 93–94. https://doi.org/10.1145/2815585.2815735

Jinha Lee, Rehmi Post, and Hiroshi Ishii. 2011. ZeroN: Mid-Air Tangible Interaction Enabled by Computer Controlled Magnetic Levitation. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology* (Santa Barbara, California, USA) *(UIST '11)*. Association for Computing Machinery, New York, NY, USA, 327–336. https://doi.org/10.1145/2047196.2047239

Benjamin Long, Sue Ann Seah, Tom Carter, and Sriram Subramanian. 2014. Rendering volumetric haptic shapes in mid-air using ultrasound. *ACM Transactions on Graphics (TOG)* 33, 6 (2014), 1–10.

Asier Marzo, Tom Corkett, and Bruce W Drinkwater. 2017. Ultraino: An open phased-array system for narrowband airborne ultrasound transmission. *IEEE transactions on ultrasonics, ferroelectrics, and frequency control* 65, 1 (2017), 102–111.

Asier Marzo and Bruce W Drinkwater. 2019. Holographic acoustic tweezers. *Proceedings of the National Academy of Sciences* 116, 1 (2019), 84–89.

Asier Marzo, Sue Ann Seah, Bruce W Drinkwater, Deepak Ranjan Sahoo, Benjamin Long, and Sriram Subramanian. 2015. Holographic acoustic elements for manipulation of levitated objects. *Nature communications* 6 (2015), 8661.

Rafael Morales, Iñigo Ezcurdia, Josu Irisarri, Marco A. B. Andrade, and Asier Marzo. 2021. Generating Airborne Ultrasonic Amplitude Patterns Using an Open Hardware Phased Array. *Applied Sciences* 11, 7 (2021). https://doi.org/10.3390/app11072981

Rafael Morales, Asier Marzo, Sriram Subramanian, and Diego Martinez. 2019. LeviProps: animating levitated optimized fabric structures using holographic acoustic tweezers. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. Association for Computing Machinery, New York, NY, USA, 651–661.

Yoichi Ochiai, Takayuki Hoshi, and Jun Rekimoto. 2014. Pixie dust: graphics generated by levitated and animated objects in computational acoustic-potential field. *ACM Transactions on Graphics (TOG)* 33, 4 (2014), 1–13.

Themis Omirou, Asier Marzo, Sue Ann Seah, and Sriram Subramanian. 2015. LeviPath: Modular acoustic levitation for 3D path visualisations. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. Association for Computing Machinery, New York, NY, USA, 309–312.

Themis Omirou, Asier Marzo Perez, Sriram Subramanian, and Anne Roudaut. 2016. Floating charts: Data plotting using free-floating acoustically levitated representations. In *2016 IEEE Symposium on 3D User Interfaces (3DUI)*. IEEE, IEEE, New York, NY, USA, 187–190.

Viktorija Paneva, Arthur Fleig, Diego Martínez Plasencia, Timm Faulwasser, and Jörg Müller. 2022. OptiTrap: Optimal Trap Trajectories for Acoustic Levitation Displays. *ACM Trans. Graph.* (feb 2022). https://doi.org/10.1145/3517746 Just Accepted.

Diego Martinez Plasencia, Ryuji Hirayama, Roberto Montano-Murillo, and Sriram Subramanian. 2020. GS-PAT: High-Speed Multi-Point Sound-Fields for Phased Arrays of Transducers. *ACM Trans. Graph.* 39, 4, Article 138 (jul 2020), 12 pages. https://doi.org/10.1145/3386569.3392492

Deepak Ranjan Sahoo, Takuto Nakamura, Asier Marzo, Themis Omirou, Michihiro Asakawa, and Sriram Subramanian. 2016. Joled: A mid-air display based on electrostatic rotation of levitated janus objects. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. Association for Computing Machinery, New York, NY, USA, 437–448. https://doi.org/doi.org/10.1145/2984511.2984549

DE Smalley, E Nygaard, K Squire, J Van Wagoner, J Rasmussen, S Gneiting, K Qaderi, J Goodsell, W Rogers, M Lindsey, et al. 2018. A photophoretic-trap volumetric display. *Nature* 553, 7689 (2018), 486. https://doi.org/doi.org/10.1145/3386569.3392492

W. Le Conte Stevens. 1899. Review of A Text-Book of Physics–Sound. *Science* 9, 234 (1899), 872–874. http://www.jstor.org/stable/1626789

Ryoko Takahashi, Keisuke Hasegawa, and Hiroyuki Shinoda. 2018. Lateral Modulation of Midair Ultrasound Focus for Intensified Vibrotactile Stimuli. In *Haptics: Science, Technology, and Applications*, Domenico Prattichizzo, Hiroyuki Shinoda, Hong Z. Tan, Emanuele Ruffaldi, and Antonio Frisoli (Eds.). Springer International Publishing, Cham, 276–288.

Ultraleap. 2022. *Developer SDK.* www.ultraleap.com/developers/

Sebastian Zehnter and Christoph Ament. 2019. A Modular FPGA-based Phased Array System for Ultrasonic Levitation with MATLAB. In *2019 IEEE International Ultrasonics Symposium (IUS)*. 654–658. https://doi.org/10.1109/ULTSYM.2019.8926194