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

Supplementary Material

1. Introduction

Our Supplementary Material covers a range of materials that could not be included in the main paper, but were necessary for completeness. These include extended results, documents explaining specific elements (e.g. hardware design, tutorials) and the source code for OpenMPD and its example Unity client.

The current document provides a discussion of the most relevant elements covered, and acts as a central structuring element to parse all of this content, pointing to other documents were required.

More specifically, the current document is structured around 3 main topics:

1. *Hardware design summary*: OpenMPD requires specialized hardware to operate (i.e., phased array of transducers), and currently supports two type of devices: the open hardware designed by Morales et al [21] and our own hardware design (OpenMPD board), which includes some key extensions (namely, higher update rates and hardware synchronization). Section 2 introduces these platforms, providing details to allow replication and highlighting their differences. We also elaborate on specific aspects that device manufacturers should consider in order to implement OpenMPD drivers supporting their own devices.
2. *OpenMPD characterization*: The main paper characterizes OpenMPD’s performance, verifying how it is always capable of robustly maintaining the requirements of any of the primitives in Table 1 (>10KHz, with/without continuity/transitions). Section 3 provides a more extensive characterization, focusing on the behavior of each solver for different numbers of primitives and modalities (e.g., haptics, levitation and sound), as well as under different conditions (i.e. with or without hardware synchronization and using native C++ or our Unity client). The intention is not to compare the performance of each solver (these are not a core contribution of OpenMPD), but rather to characterize OpenMPD’s ability to support them all, as well as all required modalities.
3. *OpenMPD tutorials guide*: Section 4 provides a brief introduction to the source code and examples/tutorials provided in the other documents in this Supplementary Material. The explanation here describes how to set up the execution environment and how to run the different examples in the main paper. Please note this is only a brief summary for each tutorial, but pointers to dedicated external documents (one per tutorial) are provided.

Please also note that the contents of this Supplementary Material (i.e., this document, external files and source code) is only a subset of what is available in our GitHub page and they should be taken only as introductory material. An interested reader should refer to GitHub or more extensive instruction, as well as up-to-date examples.

1. Hardware design summary

We here provide a description of the two hardware platforms currently supported by OpenMPD: the open hardware designed by Morales et al. [21] and our own hardware design, built as an extension of this prior hardware but with key extensions for optimum performance (i.e. increased update rates and hardware synchronization). We then describe our communication protocol and board definition file format. These are key aspects for device manufacturers creating OpenMPD drivers, but they also illustrate the ability of OpenMPD (and underlying acoustic solvers) to deal with a variety of devices (e.g., variable number of transducers, layouts, intensities, etc.). By doing this we aim to improve the acquisition/generation of MPD devices facilitating the dissemination of the technology and the development of new devices and techniques around it.

* 1. SonicSurface

The first device supported is the open hardware design by Morales et al.[21], chosen due to its public availability and also to try and reinforce the community of developers and early adopters developed around its Instructables page[[1]](#footnote-1). We refer the interested reader to these pages for a comprehensive description of its design, assembly instructions and components required for fabrication (i.e. bill of materials).

* 1. OpenMPD board

Our custom board is an extension of SonicSurface, with two key modifications: i) the devise uses a D2XX USB communication protocol; and ii) it includes a hardware synchronization mechanism across all boards connected (typically 2 in our top-bottom setup). We discuss these differences first. Then we describe the main components in the board (making reference to required design files) and finally provide a subsection dealing with mechanisms to allow other manufacturers to create OpenMPD drivers for their own devices.

* + 1. Key differences:

The D2XX communication allows for a higher update rate when compared to SonicSurface. SonicSurface allows for a baud-rate of 203400bps. Given the packet size used (257B/packet), this translates into a maximum of 100 updates per second, which is not enough for some of the contents supported by OpenMPD. Our communication protocol allows for up to 13000 updates per second, which is above the recommended optimum value of 10000 updates per second [12].

Our hardware synchronization mechanism ensures that all boards connected apply their updates (i.e., transducers’ phases and amplitudes) synchronously. The *sync out/sync in* lines in SonicSurface ensure that all boards share the 40KHz of the master board, which is key to ensure that all transducers’ phase delays are relative to a common clock (i.e., the rising edge of the 40KHz signal represents a phase delay of zero). However, this mechanism alone does not ensure that all boards will apply their updates at the same time (i.e., on the same rising edge of the 40KHz clock).

Our board extends the communication line between boards (i.e. *sync out/sync in* lines) with a *trigger* line, which indicates (i.e., on rising edge) when boards can apply the updates/packages received from the client PC. The *trigger line* can be configured to use variable rates (e.g., 40KHz, 20KHz, 13.3KHz, 10KHz, 8KHz), but these are always a divider of the base frequency of 40Khz (i.e. the client PC can define the *divider* to use), ensuring that the rising edge of *trigger* line always matches the rising edge of the *sync out* line.

Please note a software synchronization mechanism is also supported (e.g. for SonicSurface), using the CPU clock to send each message at the time required. Such approach can introduce timing errors, due to the OS thread scheduler, or to variable delays in USB transfer. The benefits of using hardware synchronization are later explored in Section 3.1.

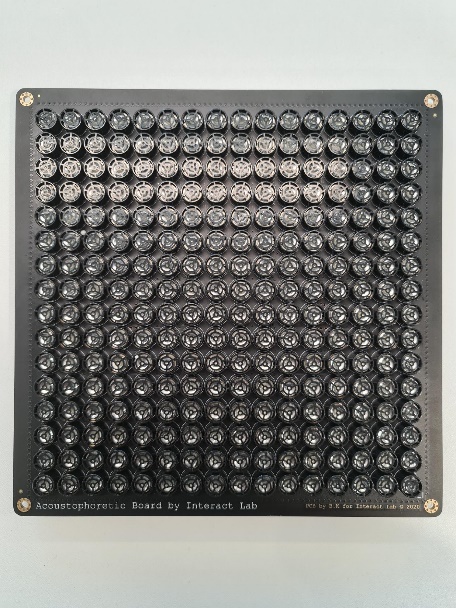
* + 1. Hardware design:

Each board in our PBD setup is composed of two main components, the *Daughterboard* (control and power board) and the *Motherboard* (transducers, shift registers and MOSFETs), shown in Figure 1. The files of this boards are also part of this submission (MotherBoard\_PCB and DaughterBoard\_PCB).

* + - 1. Daughterboard

The daughterboard is the one in charge of stablishing the direct communication with the client PC and update the phases and amplitudes of each transducer. This board (see Figure S1, left), encapsulates power-conditioning from a 20V power supply, the USB communication module to the client PC, and an EP4CE10 Altera Cyclone IV FPGA. A detailed description of all board components, IOs, physical configurations (e.g., master/slave jumpers) and assembly can be found in the document “Hardware Design Guide.pdf”. The design files and a complete bill of materials can be found in files “Daughterboard\_Altium\_files” and “Daughterboard\_Power.pdf” respectively.

A close-up of a circuit board

Description automatically generated with medium confidence  A picture containing text, electronics, plaque

Description automatically generated

Figure S1 Main components in our OpenMPD boards: *Daugtherboard* (left) and *Motherboard* (centre and right).

* + - 1. Motherboard

The array of speakers also called motherboard is less complex than the daughter board. This PCB houses all 256 ultrasound transducers (see Figure S1, centre and right). Its entire IO and power inputs are through the two coupling sockets that mate with the *Daughterboard*. This board replicates the architecture of SonicSurface, with 32 shift registers (controlled by 32 individual lines from the FPGA), shift registers each controlling 8 transducers, as well as the amplifier ICs and associated bypass capacitors. Again schematics and assembly details can be found in “Hardware Design Guide.pdf”, while files “Motherboard\_Altium\_files” and “Motherboard\_Top.pdf” include required design files and bill of materials.

* + - 1. Communication protocol

The 256 transducers in each board are updated by sending a packet (512 bytes), which includes 256 phases and then 256 amplitudes, each encoded into a single byte *mi* (Left in Fig. S2). The highest bit (b7) in the first byte of an update (*m0*) is typically set to one, to indicate the beginning of the packet. Otherwise, the highest bit (b7) in all other bytes (*mi*) is set to zero.

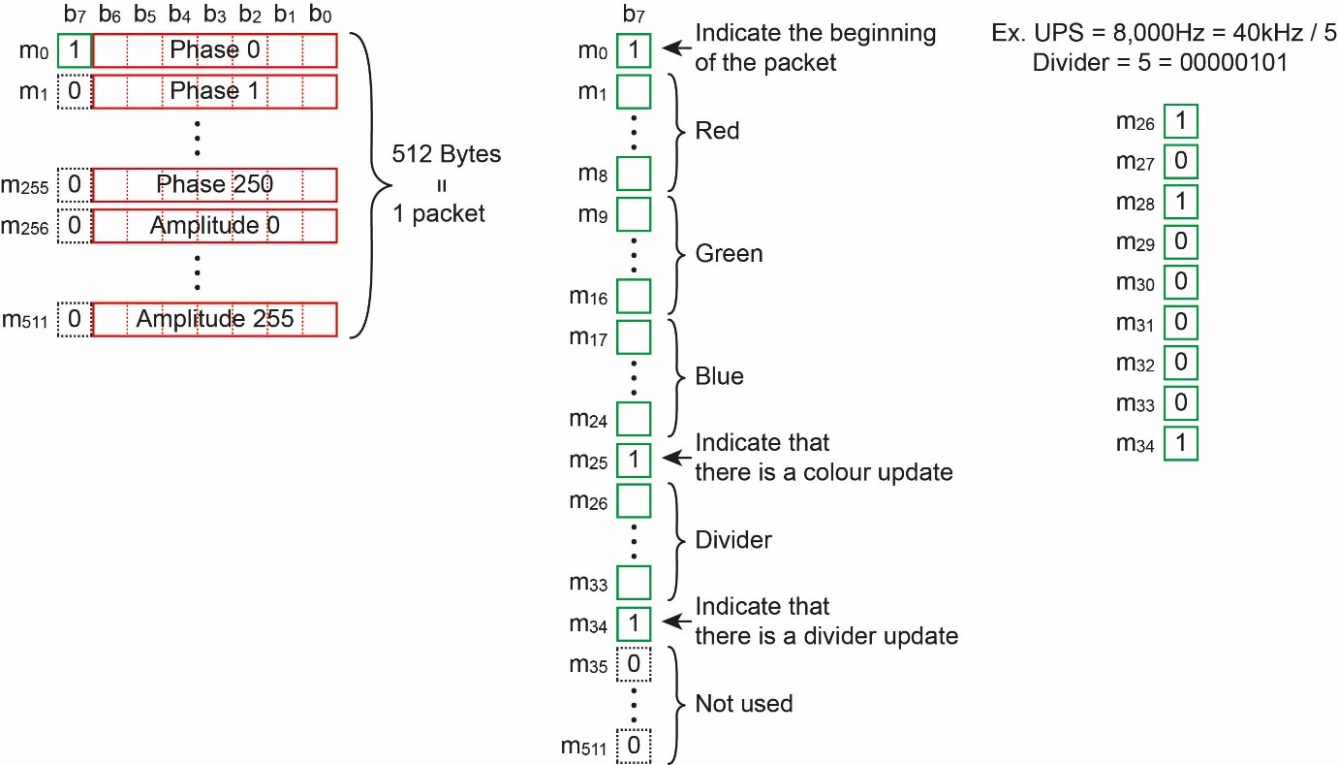
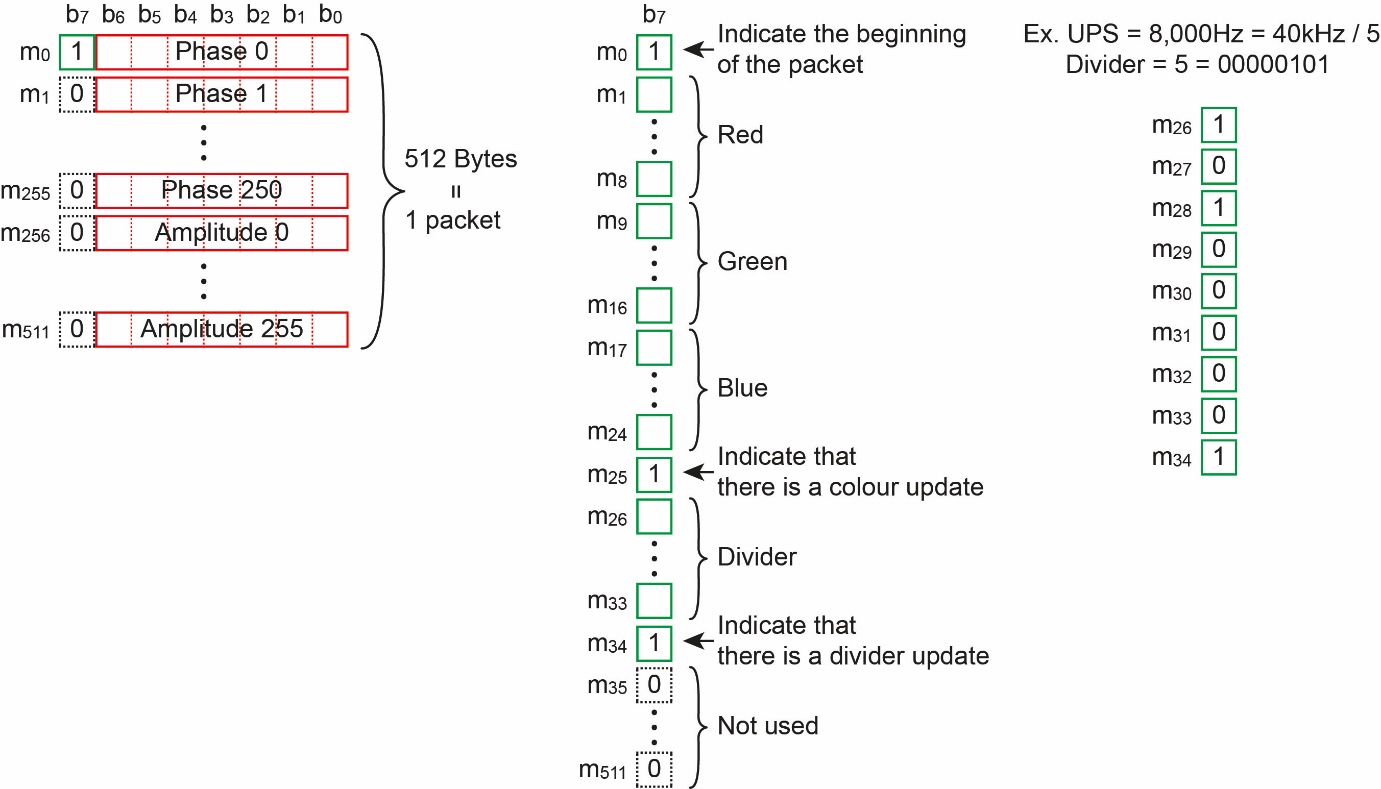


Figure S2: (Left) Packet including 256 phases and 256 amplitudes. (Right) Example setting the update rate of the device to 8,000 UPS, by using a divider = 5, encoded in the highest order bit of the bytes (m26 to m34).

**Phase discretization:** While the phases of the transducers can be any real number, the board only uses 128 discrete values [0..127]. Thus the phases need to be discretized as in Figure S3, requiring only the 7 lower bits in the byte to encode such values (leaving the highest bits available to indicate the beginning of a message).

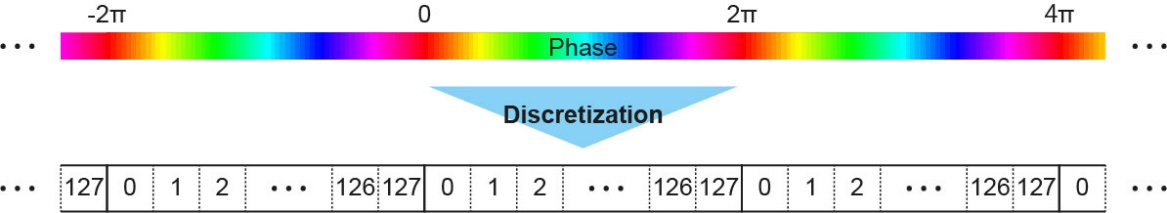


Fig. S3: Discretization of phases of the transducers.

**Amplitude discretization:** Our acoustophoretic board uses 65 discrete values of amplitude [0..64]. The amplitude of the transducers do not vary linearly with duty cycle of the square wave signal (that is, a control signal with 25% duty cycle does not result in half the amplitude of a control signal using a 50% [12]). Thus, input of amplitude in the range [0..1] (1 indicates the maximum amplitude of the transducers) needs to be converted to duty cycle before being discretized (Figure S4).

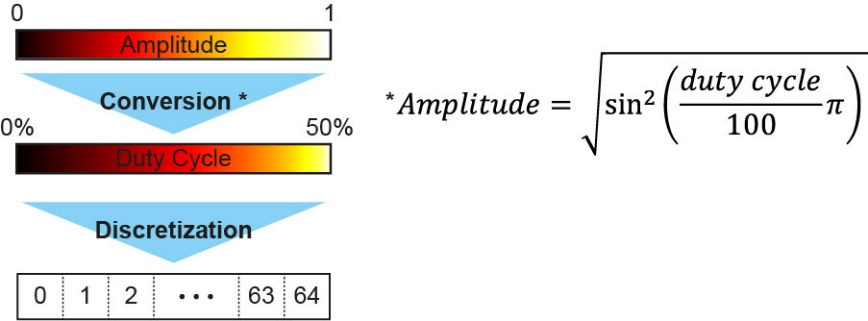


Fig. S4: Discretization of amplitudes of the transducers.

**Update rate divider:** The update rate of the board can be configured from the client PC by setting an adequate divider of the base frequency (40KHz). For example, update rates of 8,000 UPS can be achieved by setting a divider value of 5 (= 40,000 / 8,000). The highest bits (b7) of the bytes (m26 to m33) are used to represent the value of the divider, and the highest bit of the byte (m34) is used to indicate that the current packet contains a divider update. The divider needs to be set only once before starting the application and the device will retain this state until a new divider is set. Dividers can also be updated at any point by a client.

* + - 1. Device configuration file:

We use a file format to describe the intrinsic properties of the device used, which is used by OpenMPD during application set-up to connect to the boards/devices. This file format is designed as to allow flexible support for other types of devices, such as those featuring a different number of transducers, different arrangements/positions and transducers’ features (e.g. phase delays and output pressure). We here describe the structure of such files, as a reference for manufacturers wishing to provide support for their own devices.

The files are structured in several sections, each with a specific purpose:

1. **Hardware ID:** Our boards use a USB connection (FTD2) to achieve high bandwidth which have an associated hardware ID (e.g. our first board with ID 1 had hardware ID “FT4TKZL8”). This field contains such name, making this mapping transparent to the user. Other device manufacturers should use this field to store identifiers allowing their drivers to connect to the correct device, should several of them be connected to the same host PC.
2. **Number of transducers:** Our current boards use 256 transducers per device, but the underlying solvers are agnostic to this (i.e. can use a different number of transducers) allowing support for devices with other number of transducers. Hence, this field defines the number of transducers in the board used.
3. **Transducer positions:** Theplacement of the transducerscould also change from device to device (e.g. non-square boards), so this field describes the position of each of the transducers in coordinates **local to the board**. We use the system of reference in Figure S5 (XYZ axis encoded as R-G-B arrows) and meters as our unit.
4. **Transducer IDs to PIN IDs:** Transducers are logically arranged as a linear array. For instance, in our current boards transducer 0 is placed in the top left corner of the board, while transducer 255 would be located in the bottom right. However, the hardware can (and ours, for instance, does) use a different arrangement, reflecting how transducers are connected to the pins in the circuitry (see documentation on *Board Manufacturing in* “*Hardware Design Guide.pdf”*). Similarly, other manufacturers could use alternative pin mappings. This field contains a mapping describing, for each transducer ***t*** (e.g. ordered from top left to bottom right in our current boards) its associated hardware PIN **p** (e.g. ordered to match hardware requirements). This is required by the board driver to send update packages to the boards according to their actual PIN layout.
5. **Phase correction (per PIN ID):** Each transducer has a different response which must be considered for optimum use of the board. This field contains the static phase offset correction required. Please note that this is stored in degrees for each transducer, but the order is that of **the PIN ID of the board** (e.g. not the nice top-left to bottom right transducer order).
6. **Amplitude correction (per PIN ID):** The maximum amplitude of each transducer also shows some variance, which we store here. Particularly, this field contains the pressure in Pascals delivered by each transducer at 1m distance. Again, this is stored according to **PIN ID.**



Fig. S5: (Left) Axis convention used by OpenMPD devices; (Centre) Position of the first transducer as per the devices configuration, which does not match a linear arrangement; (Right) mapping used by our OpenMPD boards (please note this is different from SonicSurface, and such devices need to reflect this in their configuration files ).

This file format can be formalized as follows:

**Lexical categories:** Our grammar makes use of the following low level lexical categories, which can all be easily parsed with conventional functions (e.g. *fscanf()*):

* **float**: Single precision floating point number
* **integer**:
* **string**: Null terminated character string

The description also makes use of other conventional separators, which we identify using conventional C notation (e.g. **‘\n’**, **‘,’**), in boldface, to indicate they are lexical categories.

**Syntactic level:** We use a regular grammar which can be parsed with an LL(0) interpreter. Clauses are identified in *italics* and we also use conventional notation for regular expressions. Semantic actions are not included. *n* represents the number of transducers.

*File* 🡪*HardwareID NumTransducers NumLevels TransducerPositions PINMapping*

*PhaseCorrection* [*AmplitudeCorrection]* ***eof***

*HardwareID* 🡪 **string ‘\n’**

*NumTransducers* 🡪**integer ‘\n’**

*NumLevels* 🡪 **integer ‘\n’**

*TransducerPositions* 🡪[**‘(’ float ‘,’ float ‘,’ float ‘)’ ‘,’**]*n* **‘\n’**

*PINMapping* 🡪[**integer ‘,’**]*n* **‘\n’**

*PhaseCorrection* 🡪[**integer ‘,’**]*n* **‘\n’**

*AmplitudeCorrection* 🡪[**float ‘,’**]*n* **‘\n’**

1. performance characterization and multimodal support

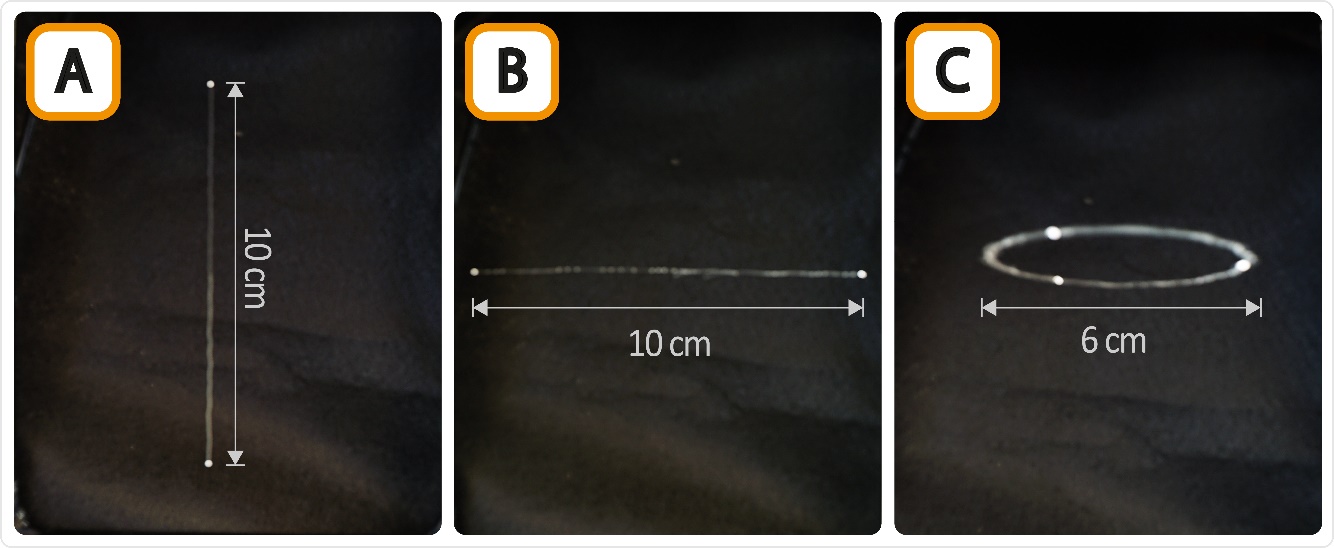
The main paper characterizes OpenMPD’s performance, verifying how it is always capable of maintaining stable update rates of 10KHz, with/without trap continuity and for any of the acoustic solvers supported (i.e., Naïve, IBP and GS-PAT). This ensures OpenMPD’s ability to meet the requirements of all the multimodal primitives, summarized in Table 1.

The current section provides a more in depth characterization of the performance of each of these solvers when used with OpenMPD, for different number and types of modalities (i.e., levitation, haptics and parametric audio), as well as under different conditions (i.e. with or without hardware synchronization; using native C++ or our Unity client). The intention is not to compare the performance of each solver (these are not a core contribution of OpenMPD), but rather to characterize OpenMPD’s ability to support them all, as well as all required modalities.

We structure this study in two subsections, a first one using levitation traps and speed tests (i.e., required for visual content) and a second one using amplitude descriptors of varying frequencies (i.e., within the range of tactile receptors and audible sound) to characterize support of each solver for both audio and haptics.

* 1. Speed Tests

We conducted a series of speed tests inspired on those described in [12], but extended to consider several particles and acoustic solvers. We also characterize the impact of some of the key mechanisms within OpenMPD in performance (i.e., synchronization and integration in higher-level clients, such as Unity).

Figure S6: Speed tests conducted: Single particle vertical test (A); Single particle horizontal test (B); and Multi-particle horizontal test (C).

* + 1. Single particle tests:

We replicated the speed tests conducted in [12], characterizing the maximum speed achievable for a single particle traveling along the vertical and horizontal directions, as these are the two main directions defining the strength of the acoustic trap [26]. Accelerations, rather than maximum speeds achieved are reported, as these are a more representative metric for the strength of the trap along the direction tested.

Particularly, we made use of ~2mm particles and performed tests characterizing maximum displacement speeds for each of our 3 solvers (Naïve, IBP and GS-PAT, all of them operating in phase-only mode). Linear paths of 10 cm were used for these tests, with the particles starting at 5cm to the left and stopping at 5 cm to the right of the centre of the device (i.e. or 5 cm above/below the centre, for the vertical tests). Particles started at rest and were constantly accelerated to reach maximum speed at the centre of the array. They were then constantly decelerated until brought back to rest at a position 10 cm away from the starting position (e.g. see Figure S6.A and B).

While exploring potential maximum accelerations (a*max*), we started with an initial acceleration of 10m/s2, increasing first in coarse steps of 100m/s2 and, after the first failure, in finer steps of 10 m/s2 from the last successful value. We performed 10 tests at each acceleration, and only considered the test successful (i.e. and tested the next higher value) if 10/10 repetitions were successful.

Beyond testing 2 different directions (vertical and horizontal) and our 3 solvers, we also tested the following conditions:

* *Sync mode:* OpenMPD supports devices with and without hardware synchronization (see Section 2.2.1). In order to avoid effects related to the hardware used (SomicSurface or OpenMPD board), we used an OpenMPD board in both cases, enabling or disabling our synchronization mechanism. The benefits of using higher update rates (like those supported by the OpenMPD board) on speed tests were already explored in [12], and not explored here.
* *Implementation level*: The core of OpenMPD is implemented as C++ libraries, while higher level clients (Unity) use wrappers for C sharp. A client interacting with the native C++ OpenMPD libraries represents the best case, where framerate and particle speed could have the highest performance, while clients using Unity could experience performance loses. We hence compared performance when using both native (C++) and high-level (Unity) clients for our speed tests.

Our results show the relevance of our hardware synchronization mechanism, with very significant gains for particles traveling along the vertical direction (i.e., ~3x higher accelerations), while gains are much smaller along the horizontal direction. This result is to be expected, as a particle travelling perpendicularly to the array (vertically) requires higher phase differences at each solver update than a particle travelling parallel to the array. Thus, the effect of one board applying updates a few frames ahead or behind the second board become much more significant along vertical paths. These results also show similar levels of performance (i.e. accelerations) and framerate both for the Unity and C++ clients.

Finally, the different solvers show a very similar performance in all cases and under all conditions, with differences between them being attributable to the particularities of each algorithm (refer to [28] for a more in-depth exploration and discussion of these differences), as well as the strength of acoustic traps along each direction (i.e. higher in the vertical direction [26]). In any case, these results illustrate how OpenMPD can support consistent maximum accelerations and update rates for every solver, independently of the direction, type of client and synchronization used.

Supplementary Table 1: Maximum particle accelerations achieved per direction (vertical/horizontal), solver (Naïve, IBP, GS-PAT), synchronization mechanism (Software/Hardware) and client type (C++/Unity).

Table

Description automatically generated

* + 1. Multiple particle tests:

We extended the circular speed tests conducted in [12], extending them to several particles (i.e., 2,3,4,5, and 6 particles). Again, angular accelerations achieved (i.e., rather than linear or angular speeds) are reported, and the tests compare the performance of the various solvers (Naïve, IBP and GS-PAT, all of them operating in phase-only mode), synchronization mode (hardware or software) and type of client (C++ or Unity).

Supplementary Table 2: framework speed performance tests



The tests made use of ~2mm particles, homogeneously placed along a circle of 5cm radius (e.g. see Figure X.A and B). The particles started from rest and were slowly accelerated until the target angular speed was achieved. Particles needed to retain such speed for at least 3s, before a constant deceleration was applied to bring them back to rest.

Similarly to previous tests, we started with an initial target acceleration of 10m/s2, increasing first in coarse steps of 100m/s2 and, after the first failure, in finer steps of 10 m/s2 from the last successful value. We performed 5 repetitions for each acceleration tested, and only considered it successful (i.e. and tested the next higher value acceleration) if all particles came back to rest and all tests were successful.

Our results are summarized in Supplementary Table 2. These results again show the relevance of our hardware synchronization mechanism, even if in this case the circle was rendered horizontally (which showed smaller effects in the single particle tests). This effect can again be explained due to the phase changes per update required from each transducers, as these become higher when solving for multiple particles. Thus, the effect of one board applying updates a few frames ahead or behind the second board can result in very different phases being delivered.

Again, the results also show similar levels of performance (i.e. accelerations) and framerate both for the Unity and C++ clients, as well as consistent angular accelerations and update rates for every solver, independently of the number of particles, type of client and synchronization mode used.

* 1. Variable amplitudes: haptics and audio generation and audio quality.

We here illustrate the capabilities of each of the solvers supported by OpenMPD to create primitives with time-varying amplitudes between 100Hz and 5KHz. This range is selected as it covers the range of frequencies detected by both Pacinian mechano-receptors (i.e., 200-300Hz for haptics) and the human primary auditory range (i.e. 2KHz-5KHz).

More specifically, we trapped three levitated particles arranged in a line (see FigureS7) and used a chirp audio signal from 100Hz-5KHz (spectrogram shown in Figure S8.a) to modulate the amplitude of the middle trap. We used the same amplitude modulation method (i.e., upper-sideband modulation) used in [12] to encode the signal and represented the modulated data as an OpenMPD amplitude descriptor. We recorded the sound generated with an Audio-Technica PRO35 microphone placed 5cm in front of the central particle.

A planet in space

Description automatically generated with low confidence

Figure S7: Audio test.Tree acoustically captured particles and the microphone setup. The central particle is

We recorded the audio generated when using each of the different solvers: Naïve, IBP and GS-PAT, and the resulting spectrograms are shown in Figure S8.b, S8.c and S8.d, respectively. This analysis illustrates the ability of any of these solvers to generate audible sound within the target frequency range (i.e., 100-5KHz), confirming their ability to produce primitives encoding amplitude modulations that can address both to tactile and audio primitives.

The results also show the usual artefacts related to the update frequency used of 10KHz, such as aliasing effects (i.e. shown in the spectrogram as duplicates of the input chirp signal, every 5KHz) or harmonics (e.g., signals at 2x the frequency of the input signal).

A second interesting observation is the ability of IBP to still produce audible sound, even if this algorithm does not allow for variable amplitudes of the transducers. This behavior can be obtained only when several primitives are created, as the phases of each transducer interfere constructively/destructively to achieve the required amplitude at the target point. This behaviour cannot be obtained if only one primitive is created (a constant maximum amplitude would be created instead).

A picture containing text, display

Description automatically generated

Figure S8: Spectral analysis of the audio response of our OpenMPD framework. (a) Signal used for input: chirp (from 100 Hz to 5,000 Hz). Output of the system when we use different solvers: (b) Naïve, (c) IBP and (d) GS-PAT.

1. OpenMPD Tutorials

This document provides an overview of how **OpenMPD** integrates our core libraries into Unity (core elements such as engine, drivers and solvers) and how developers can use such integration. It also provides an overview of the examples included in the framework (together with a brief description of each one). The full examples are not part of this document, but they are included in individual files in the Supplementary Material accompanying this submission and timely referenced along this section.

The software provided (and our explanation of it) can be broadly structured in two categories:

* *Integration of core elements into Unity*: We will describe the basic elements’ structure in C#/Unity.
* *OpenMPD framework supporting MPD (Multi-modal Particle-based Display) content creation (classes, prefabs):* A set of classes developed in C# that facilitate the usage of the solvers, by supporting most common types of MPD content.

The latest version of OpenMPD framework and documentation can be found attached to this submission and it is also available online through our GitHub repository. Once downloaded, please follow the setup instructions from the document “OpenMPD\_FrameworkSetup.pdf”. After the setup stage, our Unity project follows the usual structure of any project using this platform, but some folders are particularly important for you to understand how the integration of OpenMPD into Unity works:

* **Base folder (OpenMPD)**: This is the “working directory” that Unity will use. That is, when you run your Unity game, it will be the base directory of your .exe file. This folder needs to contain the resources required to support execution, such as the external DLLs used by *the Drivers* and *the supported Solvers* and the OpenCL shaders (i.e. *hologramSolver2.cl*). If you make changes to the solver that involve external resources (i.e. new libraries, new shaders, textures, etc.), make sure these are copied into this folder.
* **External:** This folder contains our (external) DLLs encapsulating *the Drivers* and *Solvers* (each in a separate folder)*.* These are the output files produced by the Visual Studio (VS) C++ Solution that we used to build the low-level version of OpenMPD framework. If you want to make a custom version of the solvers or drivers, you will need to manually copy your resulting DLL files (from the “x64” folder in your VS C++ Solution).
* **Assets:** This folder contains the Unity content supporting your development. This includes C# scripts (i.e. classes), prefabs (i.e. the “Levitator” node) and the example scenes that we will use as tutorials here to illustrate how to work with OpenMPD from Unity.
* **OpenMPD\_Unity\_Example.sln**: This is the VS C# Solution containing all the scripts and examples that we will use in this document. You can open it by double clicking on it or, otherwise, Unity will automatically open it when you edit any of the scripts in the scene.

All these folders have already been set up for you with the latest version of our framework, so you should be able to jump straight into Unity. Please note there are many other sub-folders and C# projects. These are default Unity elements and do not hold any specific information related to OpenMPD.

* 1. Basic infrastructure and key concepts

Getting started is then as simple as opening Unity (we used version 2019.3.4f1, but OpenMPD framework should be compatible with any version):

* + - 1. Click on File🡪Open Project and select our project folder (“<Installation folder> \OpenMPD\_Host”).

1. Click on File🡪Open Scene and select “1. Single-Particle”.

You will find an almost empty scene, with our *Levitator* at the centre, as shown in Figure S9 highlight B. Even if almost empty, there are a few relevant things that you need to know. The most immediate one is the *Levitator* node itself (see highlight A in Figure S9). This node contains two elements: *3D\_Model* and *LevitatorOrigin*.

* + 1. 3D\_Model:

The *3D\_Model* simply contains a minimalistic reconstruction of the levitator, and it only serves as a visual reference while creating your experiences. By default, the *3D\_Model* gets hidden when you press “*Play*” (you can disable the script *Hide3DModelOnPlay* attached to 3*D\_Model*, to avoid this). *Levitator-Origin* is much more relevant, and it will be described in the next subsection.

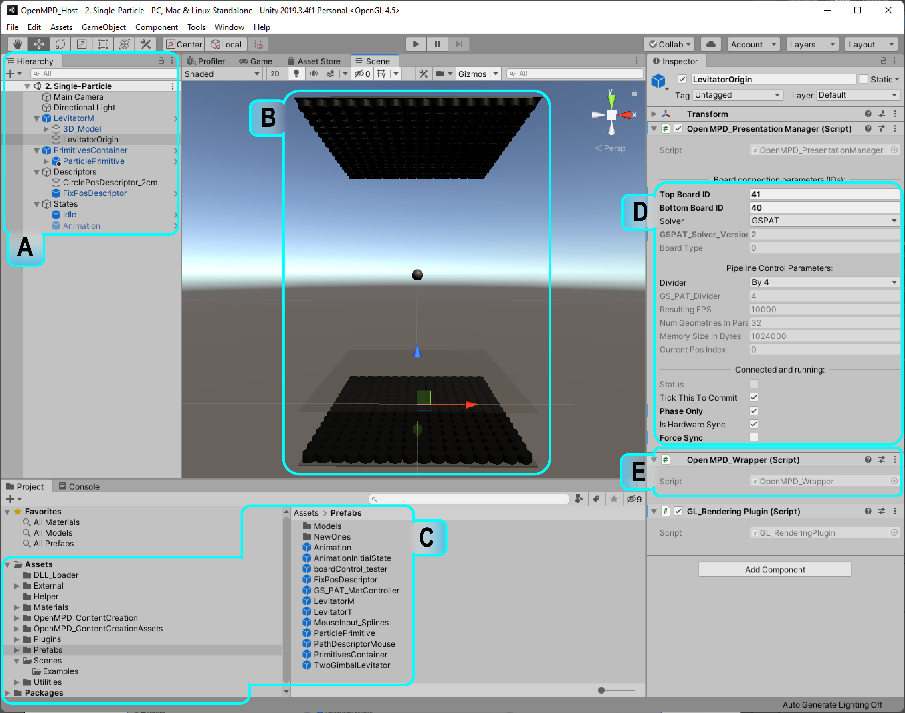


Figure S9: Overview of Unity after loading our empty scene, highlighting the structure of our Levitator prefab (A); configuration parameters to connect to the MPD device (B); basic component OpenMPD\_Engine (C); and where these basic components and prefabs can be found (D).

* + 1. The Levitator Origin

The *LevitatorOrigin* provides the basic infrastructure you need to create MPD experiences, serving several purposes.

* + - 1. LevitatorOrigin as a system of reference:

The *LevitatorOrigin* identifies the system of reference of our levitation setup (i.e. its (0,0,0) point). That is, you can place the *Levitator* node (i.e. the parent of *LevitatorOrigin*) however you like into your Unity scene. You can place your MPD contents however you like also. The *LevitatorOrigin* will be there to take those coordinates from the primitives in Unity space and make them relative to your device.

This node also deals with mismatches in the way coordinate axis are used by Unity and *OpenMPD*. As you can see, the axis in the levitator (the three arrows overlaid on the bottom board in Figure S9.B) does not match the orientation of the axis used by Unity (you can see them in the top-right of the 3D scene view). It is important to note that the *LevitatorOrigin* is located in the centre of the Unity scene (0,0,0), and this is visually located in the centre of the bottom board of the levitator model in the scene. You are free to move the parent *Levitator* node to fit the arrangement of the environment/application you are creating, but you should not change the location of the *LevitatorOrigin* or the *3D\_Model*, as this would create a mismatch between where you see your levitator (*3D\_Model*) and where OpenMPD thinks it is (*Levitator\_Origin)*.

* + - 1. LevitatorOrigin as a basic infrastructure for MPDs:

This node also holds two basic elements that allow us to create MPD content called Open*MPD\_PresentationManager* and *OpenMPD\_Wrapper* (which internally handles the calls to both the driver and the solver to be used*)*, as shown in the right-hand side of Figure S9.D (*Inspector* panel). Their specific role and functionality will be explained later in “Sections 3 and 4.3”, but by now you simply need to rest assured that the underlying elements needed for your MPD experience to work are in place (and be aware of where they are).

The *OpenMPD\_PresentationManager* module holds the parameters that allow us to connect to our specific PAT device (see highlight B). At the time of writing, our device used IDs 41 and 40 for the bottom and top boards respectively (See the note below about what *board ID* a is). Make sure to change these to the specific IDs of your device, to be able to run our examples or just set both top and bottom Ids to 0 to enter the simulation mode where no device is required to be attached (in this mode only Unity visualization is used, however, while the solver would be fully working, the driver would not initialize the connection with the hardware). If you use the hardware PCB schematic we provided to build your MPD boards, you will see a label with its correspondent ID number (see Figure S10 red box and read the note about the board Ids at the end of this section). This ID number is associated with a serial number of the communication board comport, that our software uses to recognize and initialize the connected devices (see Figure S10).

A picture containing text, electronics

Description automatically generatedThe *OpenMPD\_PresentationManager* modulewill also ensure that these DLLs are loaded when you hit “*Play*” (and released when you “*Stop*” it). Both the Solver and Diver DLLs are internally configured to print out their notification, error and warning messages on the Unity console during execution (see the tab in the bottom panel, to the right of the “*Project*” tab).   
**Important Note:** The board IDs are associated/generated together with a configuration file where the data of the hardware used is stored (see Section S1). For instance, USB serial number (to connect with your MPD boards), transducer mapping (to let the framework know, how are the transducers located or sorted out into the hardware layout), amplitude levels per transducers (for amplitude regularization/ compensation) and transducers’ phase adjustment or calibration (to adjust transducers’ phase shifts before solvers computation). Note that note all the MPD boards must have all this data to be used together with OpenMPD.

Figure S10: Board ID label is shown in the red box on the right-hand side.

* 1. Framework overview

The *MPD\_PresentationManager* keeps track of all currently active *MPD\_Content*s in the scene (i.e. those that are not disabled). The manager will produce one *MPD\_Update* per execution cycle (at Unity rates), containing all the data (i.e. matrices from each *MPD\_Content*, buffers from each *MPD\_Primitive*) required to issue commands to the underlying solver.

The rendering manager is also responsible for reading the results from the solver and delivering them to the underlying MPD device. MPD\_RenderingManager has two main operation modes:

* ***Standard***: it uses a semi-detached synchronization mode where the OpenMPD engine always keeps the last primitive update produced by Unity (primitives’ matrices and descriptors used) to keep the MPD device running. This keeps the MPD hardware running with ~10k ups, but the movement of the primitives might be jittery if the Unity client does not keep up with the update rate required. That is, in a conventional case, with 10KHz (divider 4) and 32 parallel solutions per computation, Unity should aim for 312 fps (i.e., 10000/32). Running at lower/higher rates will cause jitter, as the same matrix is applied to the primitive several times (<312fps) or some matrices are skipped (>312fps).
* ***ForceSync***: This mode forces the OpenMPD engine to couple the acoustic solver to Unity’s framerate, delivering 32 acoustic updates per Unity frame. This ensures each matrix update is only applied once and can maximize the smoothness of the animations. However, if Unity falls below the target rate of 312fps, the device will operate at a suboptimum update rate. That is if Unity’s ups are ~200, then 200 x 32 (this is ~6400 ups) will be the framerate that OpenMPD engine will use to update the MPD hardware, which will be under the optimal framerate required by our MPD hardware. This mode is highly recommended, as long as Unity performance can meet the minimum of 312 fps.

Finally, OpenMPD framework supports the use of GPU-based custom solvers for the sound field computation (Naïve, IBP and GSPAT), which you can select from the. *MPD\_PresentationManager* by using the field **“**Solver**”** on the inspector tab.

Once we know a little bit more about our framework elements and their main we will take a look at the layout of the examples you will find in OpenMPD. In the following subsections, we will briefly describe each example and reference it to where you can find a full description for it.

* + - 1. Independent particles: High-level transformations

This is the simplest example in this document, and it shows how to direct manipulations can be achieved in OpenMPD. It shows how to control each independent particle in the scene by direct transformations through the Unity UI by simply selecting and moving or rotating the particle or group of particles in the scene.

This can be done by using the Unity UI tools usually on the top left (translate and rotate) or by manually changing the translation and rotation values per axis on the transform section inside the inspector tab (at least one GameObject/Particle must be selected).

This particular example takes advantage of the Unity update rate by sending the transformation matrices from each active primitive in the scene to OpenMPD once per Unity update call (~312ups). We also call this update type high-level updates, and they are the simplest way to control primitives. However, these updates are limited by Unity’s framerate (312 ups, although OpenMPD will apply linear smoothing between updates to reach the target 10KHz). More advanced behaviours (e.g. PoV content, spatio-temporal haptics) require the use of *Descriptors*, which we address in the next tutorial.

Please find the full description of this example in the file “OpenMPD Tutorials\_Examples.pdf” under section 1.

* + - 1. Single particle: fast-motion (PoV effect)

In this example, we will use a single particle to exemplify how can we generate a circle animation by accelerating the particle in a circular closed path. Opposite to example 1 (high-level transformations). Where the particle displacement is set by Unity and constrained by its update rate (high-level updates ~312ups), in this example we will use low-level descriptors to define the particle trajectory and velocity we need, allowing update rates up to 10k ups. Please find the full description of this example in the file “OpenMPD Tutorials\_Examples.pdf” under section 2.

* + - 1. Multiple Particles: Independent Animations

In this example, we also follow the content-descriptor-state structure shown in example 3 (same scene elements). We are going to start from the circle animation, however, we will use two particles to draw the circle in the centre of the rendering space, using two methods; a) independent descriptors to describe the circular motion per primitive and b) sharing the same descriptor but adjusting starting indices on the animation states. This example will use low-level updates during the rendering state, however, a hybrid mode (low-level + high-level updates) is perfectly supported. Please find the full description of this example in the file “OpenMPD Tutorials\_Examples.pdf” under section 3.

* + - 1. Levitated Props: Multiparticle Structures

In this example, we manipulate a levitated prop as a spatially constrained particle structure. This means that the spatial relationship between the particles in the arrangement is fixed (e.g., four particles in a square shape will maintain the shape when translations and rotations are applied). This is similar to **Exercise B** in example 1 (see “OpenMPD Tutorials\_Examples.pdf” under section 1). In other words, any matrix transformations applied to the parent will affect all the children as a group. In this example, we use the combination of low-level and high-level updates during the rendering state. Please find the full description of this example in the file “OpenMPD Tutorials\_Examples.pdf” under section 4.

* + - 1. Multiparticle Transitions

As shown in previous examples, low-level descriptors are an excellent way to improve particle displacements and reach higher particle speeds that would not be possible using high-level updates alone. However, OpenMPD support queued animations, where a correct transition between high-speed animations become crucial to not only maintain animation continuity but to avoid rendering failures (particle flying away from the trap). Thus, animation transitions are an important element when working with queued animations in MPD systems, as they are the link between stages in the overall experience (considering stages as different animation/shapes/paths to go through).

We can render multiple animations using a single particle by using the correct transitions across animations. In this example, we will use a set of three circles that will combine into one, but rotating and translating independently to each other. We will also be able to move each of the circles independently, making them join/leave the shape without colliding with other particles. This example takes advantage of our hybrid updates (low-level circle descriptors and interactive high-level updates to join/leave). Please find the full description of this example in the file “OpenMPD Tutorials\_Examples.pdf” under section 5.

* + - 1. Amplitude Modulation: Haptics

In this example, we show how haptics can be generated using OpenMP. As per the main paper, haptics can be done mainly by using one of two methods, i) spatial modulation (modulating the particles positioning) or ii) frequency modulation (modulating the acoustic trap amplitude). Spatial modulation can be done by using our example 2 (simply set the circle descriptors to spin at a rate of 20Hz and try it on your hand).

Instead, this example shows how to modulate the amplitude to achieve haptics via frequency modulation. To do it, we will use an amplitude descriptor to generate a haptic/tactile point where we will be able to control the frequency to generate different stimulation types on the hand’s mechanoreceptors. Please find the full description of this example in the file “OpenMPD Tutorials\_Examples.pdf” under section 6.

* + - 1. Parametric Audio Generation

This example will highlight the OpenMPD capability of supporting parametric audio applications. While the standard audio sampling rate is ~44kHz, OpenMPD works at 10kHzas it is the update rate set as standard (this depends on the hardware). Then the audio file we will play in this example will be resampled to 10kHz (if your MPD device operates at a lower update rate the audio file should be resampled to match it). Similar to example 6, OpenMPD uses the amplitude descriptor to play the audio file. This means that if you have the amplitude array of your audio file you can follow the Amplitude modulation example and instead of using AmplitudeDescritporFreq.cs to generate the array of amplitudes, use ReadAmpDescriptorCSV.cs to read your file and use it as any amplitude descriptor.

As the procedure to play parametric audio files is quite similar to the one described in example 6, we will explain our method to process the audio file to generate the AM array out of it. However, the tool we need is not contained in the Unity implementation, but in the low-level version of it. Then we will need to open the “OpenMPD\_Native” solution which is the C++ version for the framework. We used Visual Studio 2019 for this implementation, while other VS versions can be compatible, using this the vS2019 version is suggested. Please find the full description of this example in the file “OpenMPD Tutorials\_Examples.pdf” under section 7.

* + - 1. Reading Descriptors from CSV file

As we have observed, OpenMPD takes arrays of data to describe the primitives' behaviour in form of descriptors. So far the descriptors were generated inside the framework, however, OpenMPD supports a descriptor definition from external files. In this example we will show the implementation to generate position descriptors from a CSV file data, however, the same method is available for amplitude and colour descriptors. Please find the full description of this example in the file “OpenMPD Tutorials\_Examples.pdf” under section 8.

1. https://www.instructables.com/SonicSurface-Phased-array-for-Levitation-Mid-air-T/ [↑](#footnote-ref-1)