Tutorial Guide

Table of Contents

1 ABOUT THIS DOCUMENT:

This document describes the support provided by **OpenMPD** to integrate our core technology into Unity (core elements such as driver and solvers). More specifically, 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 the most common types of MPD content.

We also assume that you have a working setup (i.e. Acoustophoretic boards, firmware and calibration), as most of our examples require this to fully run. However, OpenMPD also provides a simulation mode where no device is required to be attached to the PC/Laptop and a Unity visualization would still be provided. However, having a device is strongly suggested to follow this tutorial.

To run all the examples provided in this documentation it would be necessary to install Unity 3D game engine version 2019.3 in your system. In theory, this framework is not constrained by the host version, however, the version 2019.3.4f1 is the version we used during the development of the framework so we suggest sticking to this version to avoid unnecessary issues. The solvers implemented in this framework use an independent GPU (we used Nvidia GPUs) to achieve high update rates of 10k ups necessary for some of the primitives to work properly (e.g., fast-moving particles - POV).

To run the examples in this document the necessary packages are already in the framework (Core.UnityPackage), then you are not required to install any additional modules. It is advisable to first understand this document and play with the provided demos before installing any additional examples. Anyway, to add any of the Unity packages you should open Unity first and go to the main menu (top left side, under the main unity logo in the UI) and select Assets>Import Package>Custom Package, then select the one to install and press Open and follow the instructions.

2 UNITY INTEGRATION OVERVIEW: FOLDERS AND GETTING STARTED

The latest version of OpenMPD framework and documentation can be found on "Github[1]". Once downloaded, the resulting project follows the usual structure of a Unity project, but some folders are particularly important for you to understand how to work with integrating OpenMPD into Unity (the current host):

- **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 illustrate how to use the integration.
- **GSPAT_Unity_Integration.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, otherwise, Unity will automatically open it when you edit any of the scripts in the scene. Probably the first time you open the framework this visual studio

[1] Link removed for anonymity

solution file will not be present, if that is the case just manually open any script from the Unity UI (double clicking on int), and then Unity will generate this file for you.

All these folders have already been set up for you with the latest version of our solver, 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 GS-PAT.

2.1 Scene Components: Getting started and initial remarks

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_Unity_Integration_Engine").
2. 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 1 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 1). This node contains two elements: *3D_Model* and *LevitatorOrigin*.

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 deserves a dedicated subsection describing its features.

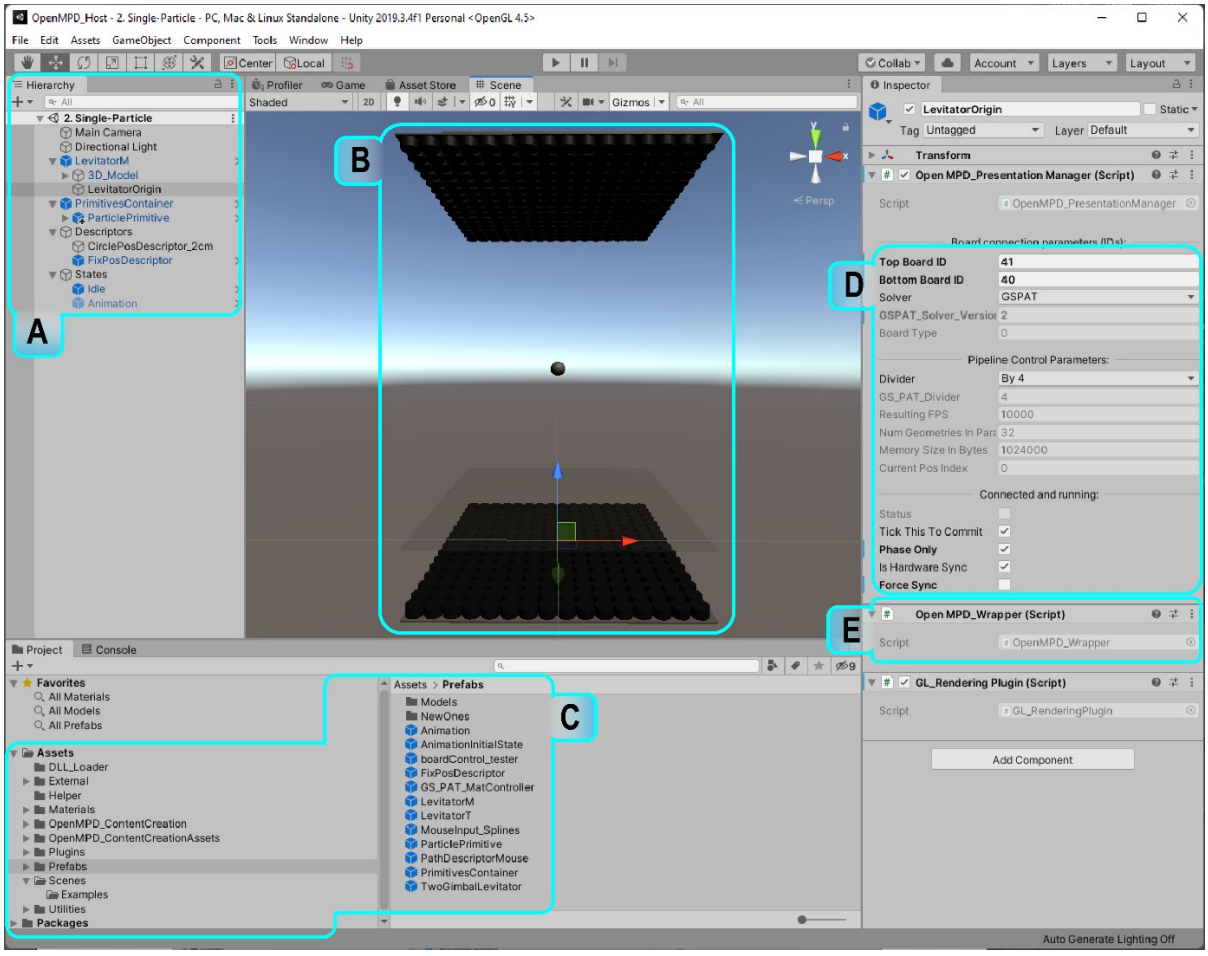

Figure 1: Overview of Unity after loading our empty scene, highlighting the structure of our Levitator prefab (A); configuration parameters to connect to the MPD device (B); basic component OpenMPD_Wrapper (C); and where these basic components and prefabs can be found (D).

2.1.1 The Levitator Origin

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

2.1.1.1 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 content 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 1.f) does not match the orientation of the axis used by Unity (you can see them in the top-right of the 3D scene view).

Note: it is important to note that the LevitatorOrigin/system of reference 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 (see the coloured arrows on the bottom board In Figure 1.B).

2.1.1.2 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 1 (*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 Open*MPD_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 26 and 24 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 t 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 2 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 2).

The *OpenMPD_PresentationManager* module will 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.

Note: The board IDs are associated/generated together with a configuration file where the data of the hardware used is stored (per board). For instance, USB serial number (to connect with your MPD boards), trasducers' mapping (to let the framework know, the way the transducers are 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 all the MPD boards must have all this data to be used together with OpenMPD.

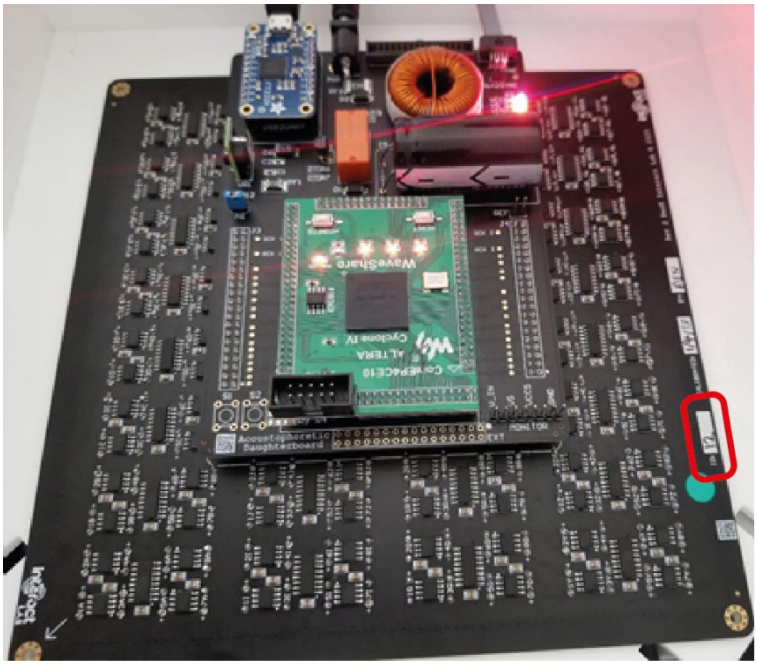

Figure 2: The board ID label is shown in the red box on the right-hand side.

2.1.2 Levitator and other prefabs

The *Assets* folder (see Highlight C on the left in Figure 1) contains all elements related to MPD development and, most relevant at this point, a *Prefabs* folder containing useful pre-assembled objects. The current repository of prefabs is shown in the bottom panel of Figure 1 (highlight C right).

The Unity toolkit is created to, whenever possible, allow the creation of MPD content simply by dragging and configuring the prefabs in this folder. We will illustrate the creation of a first simple application in the following subsection. It is worth noting that the framerate set on the *OpenMPD_PresentationManager* (Figure 1.D), is based on the hardware capabilities (a communication framerate between the driver & MPD hardware) then this value should match the device update rate. Other factors can be also customized, like numGeometriesInParallel (number of particles computed in parallel by the solver per update) and MemorySizeInBytes (memory allocated for the solver to use), however, we suggest keeping the default values for the initial testing(10000, 32 and 2000000 respectively).

Note: Inside the Scenes folder you can find a “Examples” folder with the completed version of the examples covered in this tutorial, however, we strongly suggest following the steps we provide in this document to build them yourself and use the ones in the “Full examples” folder just as reference.

3 INTEGRATION OF CORE ELEMENTS: *DRIVER* AND *SOLVER*

In this section, we will describe how the C++ DLLs that make the core of our system are integrated into Unity. Understanding this will be necessary in case you want to build custom DLLs for your applications, add new features or expose new functionalities. The elements related to loading DLLs can be found in the folder *“Assets/DLL_Loader”*. We will start by describing a simple C sharp component allowing us to dynamically link to our DLLs, and then explain how this is applied to both *Driver* and *Solver*.

3.1 Basic loading of DLLS: Kernell.cs and NativeWrapperBase.cs

Unity provides basic mechanisms to load methods from a native DLL. However, the basic approach links to such methods during Unity’s initialization, locking them while the environment is running (i.e. not only while you hit “Play”, but all the time that Unity is open). This is inconvenient if you are working on both the C++ side (e.g. fixing/adding something to the solver DLL) and the Unity side (e.g. to test if the fix worked), as you would need to restart Unity every time you changed the DLL.

To avoid this, we use an approach that allows our DLLs to be dynamically loaded during the experience execution and unloaded at the end (i.e. only active while we are in "Play" mode).

Kernel.cs provides static methods that allow us to load DLL libraries and retrieve functions exposed by that library. It uses the default (static) binding to the operating system DLL *kernel32.dll* and, as such, they do not change while Unity is running (i.e. this should not be a problem unless you were to change your operating system while keeping Unity running).

NativeWrapperBase.cs provides a base class to any DLL that we want to dynamically link to while in "*Play*" mode, making use of the methods provided by **Kernel.cs**. The wrappers to our core libraries will extend this base class, declaring and providing access to the internal methods that we need to expose.

3.1.1 NativeWrapperBase

This abstract class allows for dynamic loading/unloading of the DLL during "*Play*" mode. Derived classes (e.g. solvers like *GSPAT, naive*, *IBP* and the *driver*) can specify an arbitrary number of methods to expose, each with their own interface (i.e. return value and arguments) depending on their specific functionality. However, to support the dynamic loading/unloading behaviour implemented by *NativeWrapperBase* all derived classes **must implement a few methods with a fixed interface**. Derived classes can use their own name for these methods, but their interface must match the specification. The following table summarizes these methods, describing their interface, purpose and the field where derived classes can declare the name of their implementation (e.g. see *GSPAT_Solver.cs*, declaring `string initFuncName => @"GSPAT_CWrapper_Initialize"`).

Table 1: Fixed methods DLLs need to expose to be dynamically loaded/unloaded

Interface	Name	Description
`void initialize(void);`	`initFuncName`	Initializes the DLL for the first usage. Called when "*Play*" is pressed.
`Void release(void);`	`releaseFuncName`	Causes the DLL to deallocate all of its resources. Called when the application finishes (e.g. "*Stop*").
`bool isInitialized (void);`	`isInitializedFuncName`	[OPTIONAL] Returns true if the DLL is initialized already (e.g. resources, ports).
`void printFuncs(void *m(char*), void *w(char*), void *e(char*));`	`registerPrintFuncsName`	[OPTIONAL] Registers callback functions that DLLs can call to report notifications, warnings or errors.

NativeWrapperBase includes simple functionality, providing callbacks to react to Unity life cycle to load/unload the DLL when appropriate. While this is setup to react to the beginning and end of an application, other behaviours are simple to implement (e.g. load/unload when the component is enabled/disabled in the editor window) and these are not documented here.

However, *NativeWrapperBase* contains two main functions that support the main underlying functionality:

- *Activate()*: This method loads the DLL and its exposed methods. The method first loads the library (using *Kernel.cs*) and loads the fixed interface methods (e.g. initialize, release). It then checks if the DLL is initialized already and initializes it otherwise. Next, the method loads all methods exposed by the DLL, as declared in the derived class. Finally, it calls the *PostInit()* method, which allows derived classes to add any specific behaviours required for their operation (e.g. load a config file).
- *Deactivate()*: This method calls the release method in the DLL and unloads it.

4 THE OPENMPD DEVELOPMENT FRAMEWORK

The previous sections provided guidance on how to start using our Unity software package and a description of the integration and usage of our core elements from Unity. However, these elements still forced the MPD developer

to deal with many low-level aspects and creating MPD experiences with these resources alone would be hard (at best).

Our software package also includes support classes and Unity prefabs to support the creation of MPD experiences, some of which are introduced during our example in Section 2.1. This section provides an overview of the current elements and how these can be used to create MPD content.

4.1 Framework overview

The main elements in the framework are shown in Figure 3. Elements in grey show the basic infrastructure that becomes available when a *Levitator* node prefab is added to the scene. This basic infrastructure will allow the system to keep track of the MPD contents defined in the system, as well as the automatic computation of sound fields and delivery to the device. Elements in blue identify those included by the developer to create the MPD experience.

The framework is structured as three layers, separated by red dashed lines in the diagram. The top-most layer is related to the definition of MPD content and will be described first. The second layer is related to the automatic computation of sound fields and their delivery to the MPD device. This is the only layer interacting with our DLLs (bottom layer) and both will be explained together.

4.2 Defining MPD experiences: Content, primitives and descriptors

The definition of the elements in our MPD experiences is driven by our Content-Descriptor-State structure, decomposing them into high-level definitions (i.e. *Primitive*, updated at the low rates that Unity can afford) and low-level definitions (i.e. *OpenMPD_Descriptors*, potentially updated at very high rates ~10k ups).

4.2.1 High-level definitions: Primitive

This class represents a high-level definition of content, and they are the main entry point to describe interactive/dynamic behaviours during the experience. *Primitive* acts as a bridge between the Unity scene and our MPD content. Clients simply need to attach a *Primitive* to any Unity *GameObject*, to enable this object to act as a controller. That is, once a *Primitive* is attached to a node, the client will be able to control the overall location of the content, simply by updating the position/orientation of its associated node (i.e. actually, the *Particle* prefab provided simply contains an empty *GameObject* with a *Primitive* attached to it as the main component).

In the first example ("OpenMPD Tutorials_Examples.pdf", example 1) we show how a position update over the node can be done by manually selecting and translating it on the *Scene* view using the mouse and Unity UI tools (i.e., move tool). However, clients could update their position by any other means allowed by Unity, such as using OpenMPD animations, scripts reacting to users' inputs (e.g. hand movements detected using Leap motion, Kinect, etc.), and timeline animations, etc. Such updates in pose/orientation will automatically affect the *Primitive* (i.e. changes in the virtual particle will be reflected on the physical one). Please keep in mind that the available rendering volume for animations is 0.16m in *x*, 0.24m in *y* and 0.16m in *z* for the most common MPD setups (e.g., our hardware design we also release to be used together with the framework). It is worth noting that to date, there is no constraint implemented to warn you when exceeding the maximum rendering size, please be aware of it). However, the most advisable rendering volume size is 0.1cm x0.1cm x0.1cm (right in the centre of the levitator) as this ensures stronger trapping forces.

Primitive also contains configuration parameters (i.e. maxStepInMeters, maxRotInDegrees) that automatically limit the changes allowed to the transformations of the content over a single update (position and rotation), which is required to ensure continuity and smooth transitions on the motion of the content.

Note: To keep the basic levitator 3D model visible during runtime (to have it as a visual cue about the rendering volume), you can disable the "*Hide3DModelOnPlay*" scrip attached to the Levitator model.

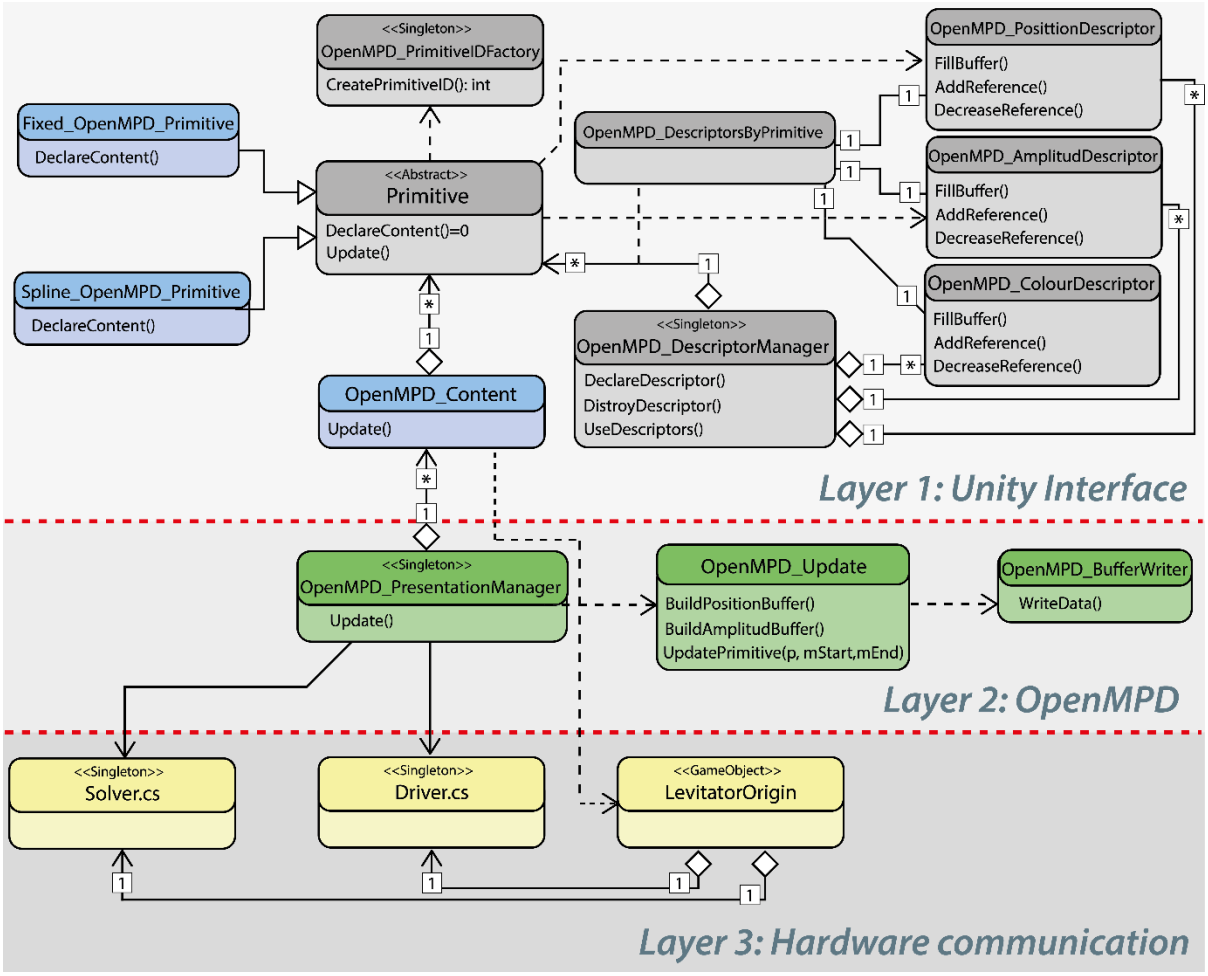

Figure 3: Classes in the OpenMPD development framework. Black elements describe the basic infrastructure available when a Levitator node is added to the scene. Blue elements identify user-created elements, used to describe the MPD experience.

4.2.2 Low-level definitions: OpenMPD_Descriptor, position and amplitude descriptors

This class represents a low-level definition of a single-point (particle, tactile or audio) content, updated at high update rates (e.g. 10KHz). *Primitive*s need to be attached to Unity game objects for them to be manipulated in the scene. However, they also need to be coupled to a high-level definition (i.e. an *OpenMPD_Content*). This is achieved by ensuring that each *Primitive* is attached to a direct child of an *OpenMPD_Content*. The *Primitive* will keep track of its current associated *OpenMPD_Content* (in its parent node), and it is even possible to migrate primitives across *OpenMPD_Contents* (i.e. making it join other "groups"/*OpenMPD_Contents*, or deactivating them by attaching them to disabled *OpenMPD_Contents*). However, the primitive should always remain as a direct child of an *OpenMPD_Cotent* node.

The main role of *Primitives* is to provide a pre-computed description of the positions and amplitudes to use for the point, specified at the target high-update rate. Position buffers should define a complete cycle (e.g. the last position in the buffer aligns with the first position in the buffer), and the positions will be in local coordinates to the associated *OpenMPD_Content* node.

4.2.2.1 Declaring different types of Primitives:

The base class *Primitive* provides the behaviour required to keep track of the OpenMPD_Content it is associated with (its high-level definition), as well as providing the required information to allow the presentation of the primitive. However, the base class contains no prior knowledge as to what type of low-level definition the primitive is using (i.e. POV content, point-based).

Subclasses of *Primitive* must be defined to provide such definitions. Each subclass of *Primitive* will extend a single method (i.e. *DeclareContent()*), in which it will fill in the positions and amplitudes buffers that the rendering engine must use over time. Each subclass of *Primitive* is free to use its own internal ways to define such buffers (e.g. user-defined parametric curves) and represents a way for users to define content.

At the time of writing, only point-based definitions are supported, encapsulated in our class *Fixed_OpenMPD_Primitive*. Such primitive simply declares a buffer with a fixed position (i.e. local (0,0,0,1) coordinates) and amplitude (i.e. amplitude 1). The catalogue of available primitives will grow as we develop our OpenMPD framework.

4.2.3 Dynamically managing primitives: descriptors and OpenMPD_DescriptorManager

As introduced earlier, each primitive will declare buffers with the positions and amplitudes to use. Each primitive must provide at least one position buffer and one amplitude buffer, which represent their current active configuration. *Primitives* are free to declare as many buffers as required (e.g. to enable changing states to present different shapes), but at each point in time, only one position and amplitude buffers will be used.

This behaviour is supported by the *MPD_DescriptorsManager* and the *descriptor* classes in the top-right of the diagram. The *OpenMPD_DescriptorManager* will keep track of the primitives declared in the system and, for each of them, it contains an *OpenMPD_DescriptorByPrimitive* describing the currently active position and amplitude buffers of each primitive. Subclasses of *Primitive* will be free to declare as many *OpenMPD_Positions_Descriptor* and *OpenMPD_Amplitudes_Descriptor* as required during their method *DeclareDescriptors*, but then they need to define their currently active descriptors. This can be done using the methods in *OpenMPD_Descriptors_Manager*:

- *UseDescriptors (primitiveID, posDesc_ID, ampDesc_ID)*: This method allows the primitive to redefine both the amplitude and position buffer descriptors used.
- *UseAmplitudesDescriptor(primitiveID, ampDesc_ID):* This method allows a primitive to redefine the amplitude buffer to use, such as when a sound source needs to start playing and audio stream.
- *UsePositionsDescriptor(primitiveID, posDesc_ID):* This method allows a primitive to redefine the positions buffer to use, such as when a POV content needs to reveal a different shape.
- *UseColourDescriptor(primitiveID, posDesc_ID):* This method allows a primitive to redefine the colour buffer to use in the colour rendering stage.

Note: The framework provides no automatic mechanism to guarantee “continuity” in how the transition between positions descriptors (descriptors with arbitrary starting positions). However, in the current version of the Framework, the old position buffer is only swapped once the cycle it defines is finished, and continuity can be guaranteed by ensuring all position descriptor buffers share the same starting and end positions.

4.3 Delivering MPD experiences: the MPD_PresentationManager

The *OpenMPD_PresentationManager* keeps track of all currently active *OpenMPD_Content* 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 GS-PAT solver.

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

- ***Standard***: it uses a semi-detached synchronization mode where the driver always keeps the last data package to keep the MPD device running using the last rendering state (previous data) and only updating when a new data package is received by the driver, and this becomes the new rendering state until the next update comes from Unity. This keeps the MPD hardware running with ~10k ups.
- ***ForceSync***: this mode forces the driver to execute the rendering based on the UI framerate, for instance, Unity’s ups, times the parallel geometries defined to be used in the GSPAT solver (up to 32).

This means that if Unity’s ups are ~200, then 200 x 32 will be the framerate that the driver will use to update the MPD hardware (this is ~6400 ups) which will be under the optimal framerate required by our MPD hardware. This mode requires an average framerate from Unity of ~320 ups.

OpenMPD framework supports the use of GPU-based custom solvers for sound field computation. MPD_PresentationManager is where the solver version can be selected using the field “Solver”.

Once we know a little bit more about our framework elements and the main structure, let's take a look at a set of examples in the file ”OpenMPD Tutorials_Examples.pdf”.