|  |
| --- |
| University College of London (UCL) |
| GS-PAT Overview: Core C++ algorithms and examples. |
| Multi-Sensory Devices Group |

|  |
| --- |
| Diego Martinez Plasencia  3-30-2020 |

Change Log

|  |  |
| --- | --- |
| Date | Change |
| 30/03/2020 | Initial version (Diego Martinez Plasencia) |
| 06/01/2021 | AsierInho\_V2 update (Roberto A. Montano Murillo) |
| 05/02/2021 | Between Docs Referencing - Public (Roberto A. Montano Murillo) |
|  |  |
|  |  |
|  |  |

Table of Contents

[1 About this document: 3](#_Toc63444782)

[2 Project Overview: 3](#_Toc63444783)

[3 AsierInho\_v2 4](#_Toc63444784)

[3.1 Overall module structure 4](#_Toc63444785)

[3.2 AsierInho\_V2: C++ interface, implementation and usage: 5](#_Toc63444786)

[3.2.1 Using the C++ interface: NORMAL operational mode and examples 5](#_Toc63444787)

[3.2.2 Using the C++ interface: MATD operational mode and examples 6](#_Toc63444788)

[3.3 AsierInho: C interface, implementation and usage: 7](#_Toc63444789)

[4 GS-PAT: 8](#_Toc63444790)

[4.1 Overall module structure: C++/C interfaces 9](#_Toc63444791)

[4.2 Using the C++ interface and examples 10](#_Toc63444792)

[4.3 Using the C interface and examples 11](#_Toc63444793)

[4.4 Using GS-PAT in practice: the client perspective 12](#_Toc63444794)

[4.4.1 Types of content, general considerations and typical challenges: 12](#_Toc63444795)

[5 Rendering Engine 14](#_Toc63444796)

[5.1 Overall module structure 14](#_Toc63444797)

[5.2 Using the C++ interface and examples 14](#_Toc63444798)

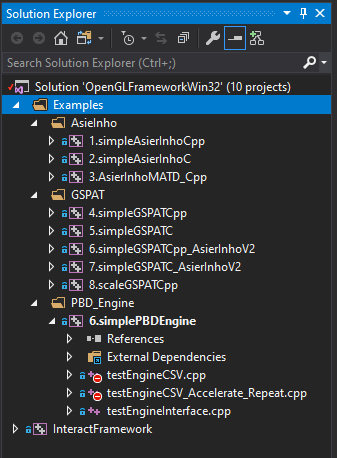
# About this document:

This document provides an overview of our C++ codebase, which is all compiled into a single Visual Studio C++ Solution. Even if under a single Solution, this actually comprises a huge amount of code, including our “core” components (*AsierInho* board controller/driver, *GSPAT* solver), as well as simulation software (compare to other solvers, compute Gor’kov, stiffness, etc), a small 3D framework in OpenGL and application examples.

This document will help you navigate this space, covering the “core” elements in the C++ codebase. These are the most important components for any developer creating PBD systems, and hence the first ones that you need to understand to build on what we have developed. The remaining elements (simulations, OpenGL framework and demos) will then be described in other documents.

# Project Overview:

The Instructions to get the latest version of the software can be found in the file “Code\_Documentation\_Resources\Documentation\4.SoftwareImplementation\ImplementationOve-rview.docx”. Please, make sure you have also completed the setup steps described in “*Documentation/SoftwareOverview.docx”* before continue reading this document. Once we have the “OpenGLFrameworkWin32.sln” open (use Visual Studio 2017 or 2019), we will find the following projects inside it:



* ***Examples***: There are actually a few different projects (9 at the times of writing) in this folder, providing very simple examples of usage of the core elements (*AsierInho*, *GSPAT and PBD\_Engine*). We refer to these during this document as demos supporting our explanations.
* **InteractFramework:** This is a compilation of other demos and interactive systems created using our core algorithms. Please note, this was the tool we used in the past for our demos, and it was built using low-level tools (e.g. OpenGL, instead of Unity). We are keeping it mostly for backwards compatibility, but those demos do not make use of latest features (e.g. GSPAT\_Rendering\_Engine, Unity) and most of the time you are not likely to use these.

Dependencies have been setup within the VS Solution already, so that only relevant parts of the code get compiled when you select a specific “*StartUp Project*”. Thus, you should be able to select the project you want to test (i.e. in the Solution Explorer, right click on the Project name and select *Set as StartUp Project*), compile and run it.

Our software relies heavily on 3 DLLs which are not part of the code you are seeing here. These DLLs are:

* *AsierInho\_v2* : the low level driver controlling the boards.
* *GS-PAT:* the solver that determines the data (phases and amplitudes) to send to the board.
* *PBD\_Engine:* a high-level library that makes it easier to work with GS-PAT.

The following subsections provide you with an overview of what each of these DLLs does and, more importantly, how you need to use them. The examples we mentioned above are actually discussed in these sections and used to illustrate how the DLLs are used. Besides, these demos are great to check that everything is working fine with your setup (e.g. driver, solver, etc), so **we highly recommend you to complete all the examples** (i.e. if you are having problems with your hardware, the first thing we will ask you to do is to run these examples anyways...).

Also, these examples are to be used from a C++ application. For details on how all these DLLs are integrated into Unity and how to make use of our hardware from Unity, we recommend you to go to “*Code\_Documentation\_Resources\Documentation\2. Unity Integration\UnityIntegration.docx”* after completing the steps in this document.

**Important note:** The projects have also been setup to use the libraries in your **LIBS\_HOME** folder (see “*Code\_Documentation\_Resources\Documentation\SoftwareOverview.docx*”). Actually, this folder also acts as your run-time environment: it holds the resources required for our demos (e.g. textures, data, .dll files), and the Solution copies relevant elements (e.g. OpenCL/OpenGL shaders, .dll files) to that folder when you change and recompile core projects. Thus, do not move this folder (or be sure to update the LIBS\_HOME variable if you do).

# AsierInho\_v2

This project produces a DLL encapsulating the driver we use to communicate with our board, which is implemented by *AsierInhoImpl\_v2*. The remaining classes in this module allow reusing of this implementation through both the C++ interface (used in some of our demos) and C interfaces (which can be integrated into Unity), as well as removing any external dependencies for our DLL clients.

The description of the module is structured as follows:

* We will start by providing an overview of the overall module structure in Section 3.1.
* In Section 3.2, we will provide examples describing its usage in both of its operational modes (GS-PAT and MATD) for the C++ interface, as well as describing most relevant implementation details.
* Section 3.3 the provides a description of the design and usage of the C interface.

Please note that the purpose of this document is to act as a quick guide to help you get started using *AsierInho,* as well as helping you understand most critical aspects related to its implementation (in case you need to maintain/extend it). We do not provide a comprehensive description of all methods and member variables used, as these are available in our Doxigen generated files.

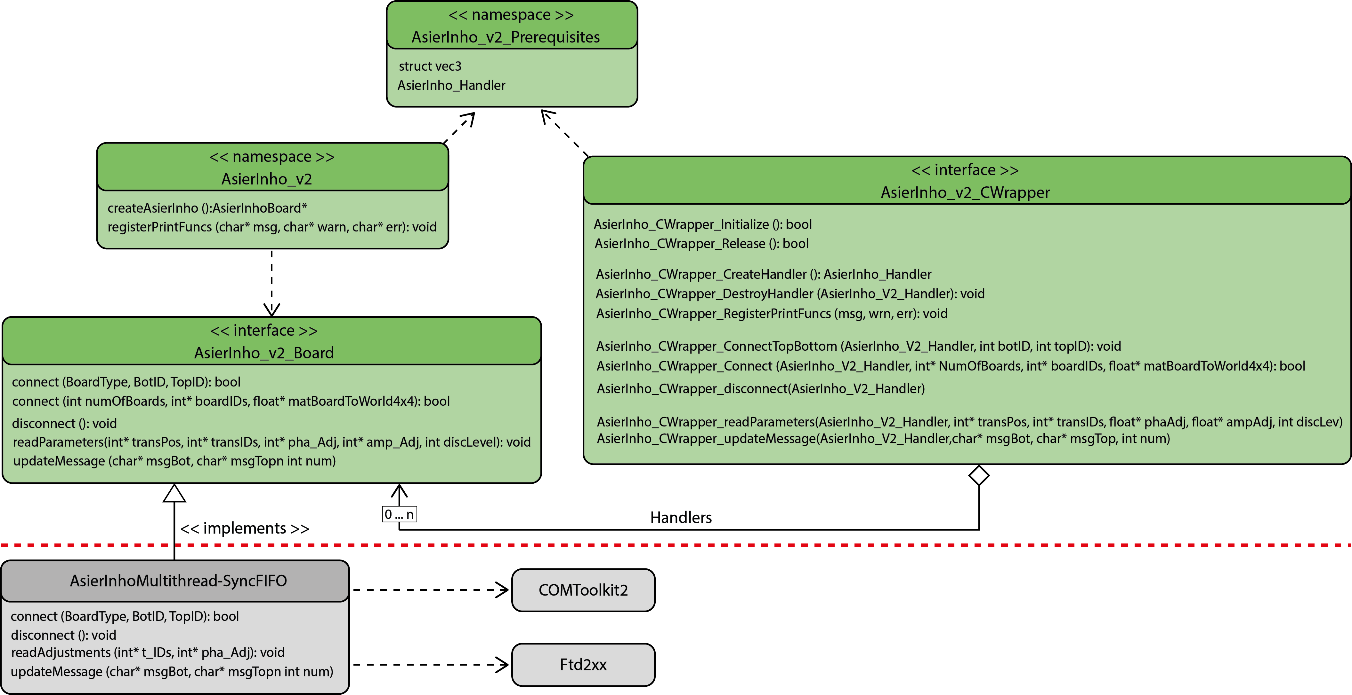


Figure 1: **Class diagram of AsierInho\_V2 DLL. Please note some methods are omitted for brevity.**

## Overall module structure

The classes within this module are shown in Figure 1. The elements related to the native C++ implementation are shown to the left, while the standalone C wrapper can be found to the right.

The horizontal red line also shows the division between the elements visible by DLL clients and those that remain encapsulated.

This division allows us to hide implementation details to clients and, more critically, related libraries (e.g. *COMToolkit, pthread* and *Ftd2xx*). This ensures that an *AsierInho\_V2* client will only need to include and link to our DLL, and not to any of the libraries its underlying implementation uses.

## AsierInho\_V2: C++ interface, implementation and usage:

The C++ implementation is the actual core implementation (other interfaces refer to these elements) and it uses a pure interface *AsierInhoBoard* (i.e. which is what C++ clients will retrieve and interact with), and the actual implementation *AsierInho\_V2\_Impl*.

The purely virtual interface ensures that changes to the implementation class will not affect clients (i.e. no need to recompile clients, just replace the .dll file). The implementation class and, more importantly, the external libraries it requires (i.e. COMToolkit, Ftd2xx and pthread) remain hidden to clients, ensuring that clients will only need to include and link to our DLL, but not to any other external libraries (i.e. this could cause issues, if clients ended up linking to a version of the external library different to the one used when building the DLL).

Finally, the basic *AsierInho\_V2* namespace contains a factory method, allowing clients to create instances of our implementation class without exposing it, as well as methods to allow us to configure how AsierInho\_V2 prints its notification, warning and error messages.

We will start describing how to make use of the different operational modes supported (NORMAL and MATD) and then describe the multithreaded behaviour of *AsierInhoMultithread\_SyncFIFO.*

### Using the C++ interface: NORMAL operational mode and examples

The default operational mode of our controller is NORMAL, which allows client applications to specify the phase and amplitude to be delivered to each transducer.

The typical usage of this mode is illustrated in the diagram above (Figure 2). The client can (optionally) register print functions, which will allow *AsierInho* to print its output messages to different targets (e.g. the print functions provided by the client could use *printf* to print to console, or they could print to files, UI components, etc). The client will start by creating an instance of an *AsierInhoBoard*, making use of our abstract factory. The initialization steps finish when the client connects to the board, providing the ID numbers of the boards in its setup.

During run-time, the client will be responsible for computing the sound-fields (i.e. phases and amplitudes) to be delivered to the board (i.e. method *computeSomeField()*). This can be done by using our GS-PAT solver, or by any other techniques the client wants to use. The target amplitudes and phases are then discretised (GS-PAT solver directly discretize both phase and amplitude) and sent to the board by calling *updateMessage()* (please note that this takes into account any corrections necessary, as well as transducer mapping and phase corrections – see document “*1.FPGA Firmware/2.Explanation of the Protocol.docx*”, Section 3.2).

Once finished, the client simply needs to disconnect from the board and delete the controller, and all ports and resources will be deallocated. While disconnection is not compulsory (it is done automatically in the destructor), **the client is responsible for deleting its *AsierInhoBoard* objects**.

A simple example, connecting to an *AsierInhoBoard*, using it to compute single levitation traps and releasing/deleting the controller can be found in our C++ solution (i.e. project “*Examples/AsierInho/ 1.SimpleAsierInhoCpp*”). It’s worth noting that *AsierInho\_V2 driver* supports two connect methods; i) “*connectTopBottom”* communicates and initializes a two-boards setup in a top-bottom configuration, and ii) “*connect”* method takes from one to multiple boards, supporting also a 4x4 transformation matrices to describe the position and orientation per board in world coordinates. By using the “*connect*” method, the effect of the board position and orientation is considered on the amplitude and phase computation, allowing with it a more dynamic boards-setup configurations. The use of these transformation matrices allow phase computations for PBD content up to 32 geometries per update, by considering initial and final transformation matrices and linearly interpolating in between.

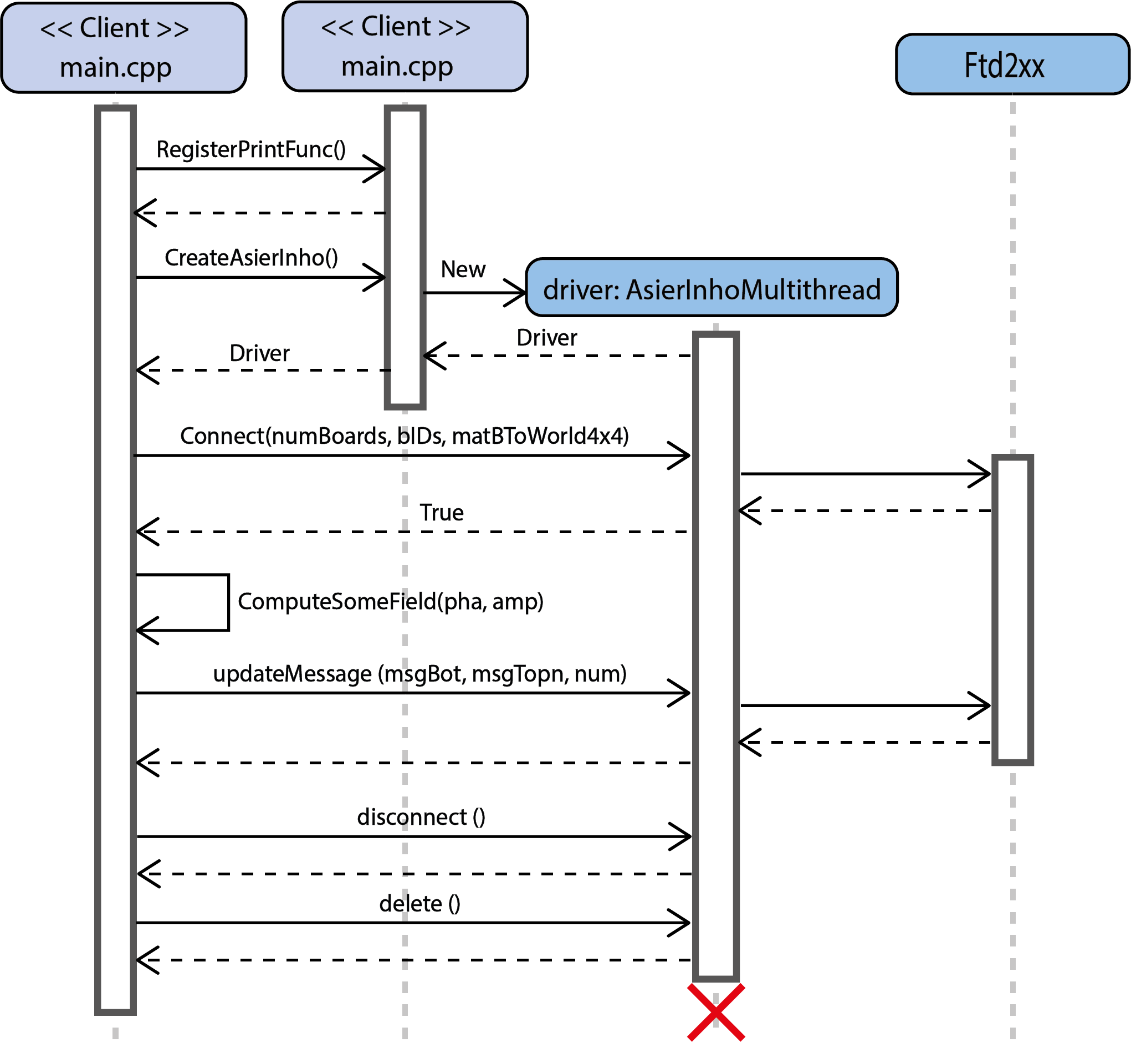


Figure 2**:** **Initialization, usage and destruction of our controller, when used in NORMAL mode.**

### Using the C++ interface: MATD operational mode and examples

Our firmware supports a second mode of operation, called MATD mode. In this mode of operation, single levitation traps or single focussing points are computed directly by the firmware. Hence, the client does not need to compute any sound-fields, but their applications are limited to single point/particles. As a major advantage, computation rates of up to 40KHz can be supported, allowing for parametric audio, with simultaneous visual and tactile content (see [**Hirayama, 2019**]).

The usage of this mode involves some steps similar to those we used in the NORMAL mode, as illustrated in Figure 3. Initialization and connection are similar to those used for NORMAL mode. After that, the client simply needs to change the operational mode (i.e. *modeReset(MATD)*) and specify the desired single trap to be created (i.e. in terms of its 3D position, amplitude, phase, colour, etc). It is worth noting that the client is responsible for computing the target positions and amplitudes at sufficient rates (e.g. 40KHz), as well as maintaining the timing between updates (i.e. use a real time clock to send them every 25 µs). The board will not apply updates any faster than once every 25 µs (*faster* updates are queued and executed one at a time, every 25 µs). However, if the client does not send updates in time (e.g. delays >25 µs), the board will retain the last update received until a new update is received. A simple example, connecting to an *AsierInhoBoard*, using it to compute single levitation traps in MATD mode and releasing/deleting the controller can be found in our C++ solution (i.e. project “*Examples/AsierInho/ 3. AsierInhoMATD\_Cpp*”).

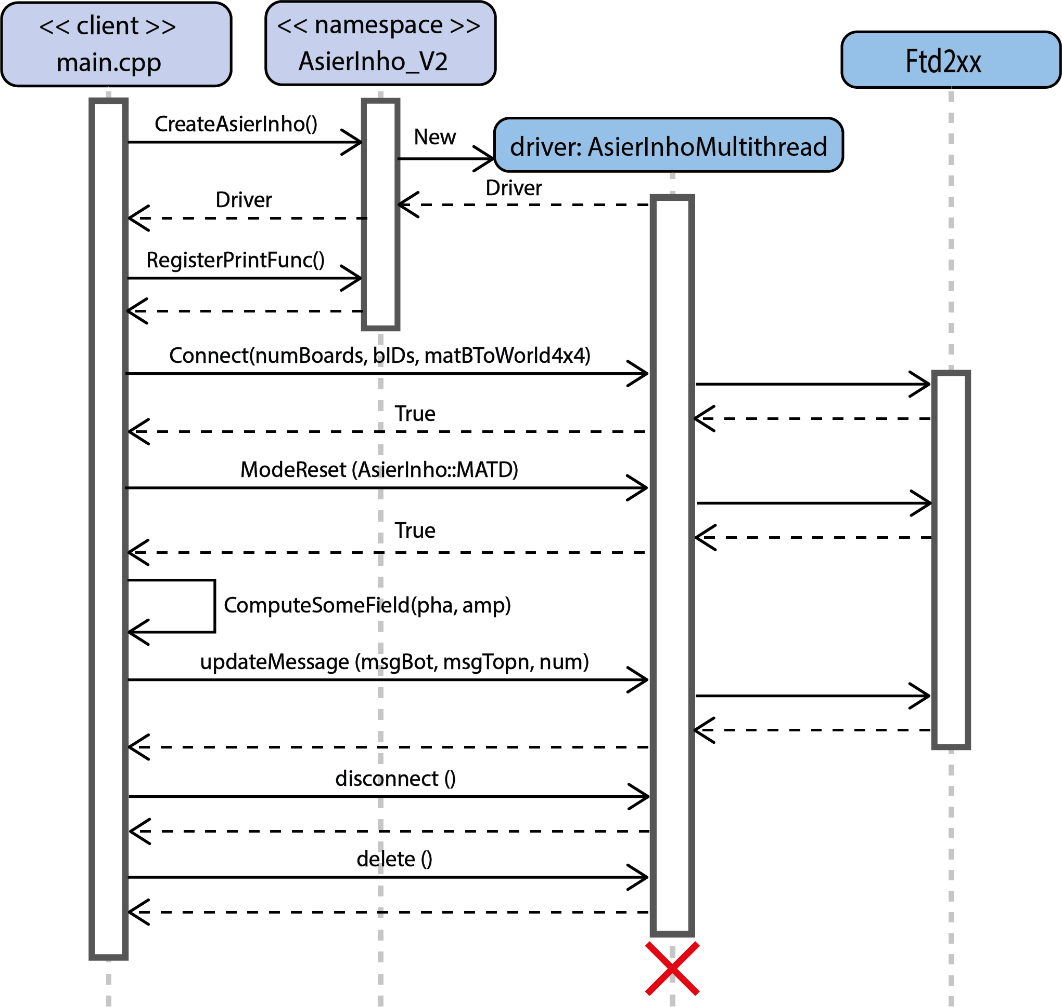
**NOTE**: The client can change modes as many times as required. Thus, it is possible to use GS-PAT and NORMAL mode in one part of the application, and then change to MATD, or vice versa. 

Figure 3**: Initialization, usage and destruction of our controller, when used in MATD mode.**

### AsierInhoImpl: Multithreaded implementation

The driver needs to deal with very high update rates (i.e. some GS-PAT applications require update rates >20KHz) while, at the same time, we wanted to avoid any delays between each board’s updates (i.e. we wanted both boards to update their phases/amplitudes with minimum delays relative to each other). At the same time, we wanted to minimise delays to the client thread (i.e. not locking the client thread while messages are sent through the port), so that clients can allocate their time to computing the required sound-fields.

To accomplish this, the driver is implemented as a multi-threaded system, using a working thread for each of the boards involved (i.e. one for the top and for the bottom), as shown in Figure 4.

In the diagram, blue boxes show activities related to the AsierInho DLL, while the green box represents the external client. Boundaries between threads are indicated with dashed lines. Also, the driver allocates resources shared with the worker threads, particularly one data stream for each of the boards (i.e. *top\_data\_stream*, *bot\_data\_stream*), where the driver copies the updates to be sent to the boards by each worker thread.

Binary semaphores (i.e. *pthread\_mutex\_t,* represented as black circles in the Figure 4) are used to implement a simple signalling mechanism between the threads. Each worker thread will wait for a signal to send a new update to its associated board (i.e. *top\_SEND*, *bot\_SEND*). In turn, they will notify the main thread when sending is finished (i.e. *top\_DONE*, *bot\_DONE*).

It is worth noting that the implementation can still momentarily lock the client thread, while worker threads finish sending previous messages.

The diagram shows the case of a call to *updateDiscretePhases.* Other methods available to send messages to the board in NORMAL mode (i.e. *updateDiscretePhasesAndAmplitudes, update-Messages*) follow the same structure. The methods used in MATD mode (i.e. *updateFrame*, *updateMultipleFrames*) do not use the worker threads (data sent from the main thread), as the size of such messages is very small, and we considered that the overheads of multi-threading did not justify using such an approach for the MATD mode.

A screenshot of a computer

Description automatically generated

Figure 4**: Threads and synchronization mechanisms used by the AsierInho implementation**

## AsierInho: C interface, implementation and usage:

The C interface is provided by the methods in *AsierInho\_CWrapper*, but this is simply a proxy allowing clients to access the functionality exposed by our previous classes, without making direct use of them. That is, C++ clients will directly interact with *AsierInhoBoard* objects by calling their methods (e.g. connect, disconnect). Instead, C clients will retrieve a *handler* (i.e. an object identifier), and will then provide this handler/identifier, when they want to interact with their board controllers (i.e. note how *AsierInho\_CWrapper\_connect* adds an extra argument – the handler- which identifies the object on which we wish to call connect).

Please note how the C wrapper simply mimics the interface of our C++ *AsierInhoBoard* (adding the handler as a first parameter). It also mimics the high-level methods (i.e. *createAsierInho*, *RegisterPrintFuncs*), which are again redirections to the C++ implementation. Please note that while a C++ client can implicitly destroy the *AsierInhoBoard* objects (i.e. call its destructor), such functionality would not be available to C clients (i.e. they cannot delete). The wrapper includes an explicit method to destroy handlers, but also provides and Initialize and Release method, which makes sure all resources related to the DLL (e.g. all handlers created, ports opened) get loaded/unloaded properly.

A simple example of its usage is provided in Figure 4, mimicking the behaviour described in section 3.2.1 for the C++ interface. The client must make an initial call to *AsierInho\_CWrapper\_Initialise* (to start the library) and a final call to *AsierInho\_CWrapper\_Release* (to deallocate all resources).

Otherwise, all the intermediate steps taken by the client are exactly the same than while using the C++ interface, but calling methods from the C wrapper. That is, each call to an *AsierInho\_CWrapper*\_*XX* method is forwarded to the equivalent method in the C++ interface (i.e. *AsierInho*, *AsierInhoImpl*). It is worth noting how the client does not receive an instance to the controller (*driver*, in the diagram), but instead received a *handler h*. This handler acts as a unique identifier for the underlying driver (implementation class) and is used as the first argument in all *AsierInho\_CWrapper\_XX* method calls that refer to the board. Finally, it is worth noting that the method *AsierInho\_CWrapper\_destroyHandler* is used to finalise the driver. Also, the call to *AsierInho\_CWrapper\_Release* will delete all driver instances created, even if the client did not destroy them (this differs from the C++ interface, where clients are responsible for deleting drivers).

A simple example, connecting to an *AsierInhoBoard* , using it to compute single levitation traps and releasing/deleting the controller can be found in our C++ solution (i.e. project “*Examples/AsierInho/ 2.simpleAsierInhoC*”). This implementation acts as a mirror for the C++ example provided in Section 3.2.1, as a way to illustrate the parallelism between the C++ and C interfaces.

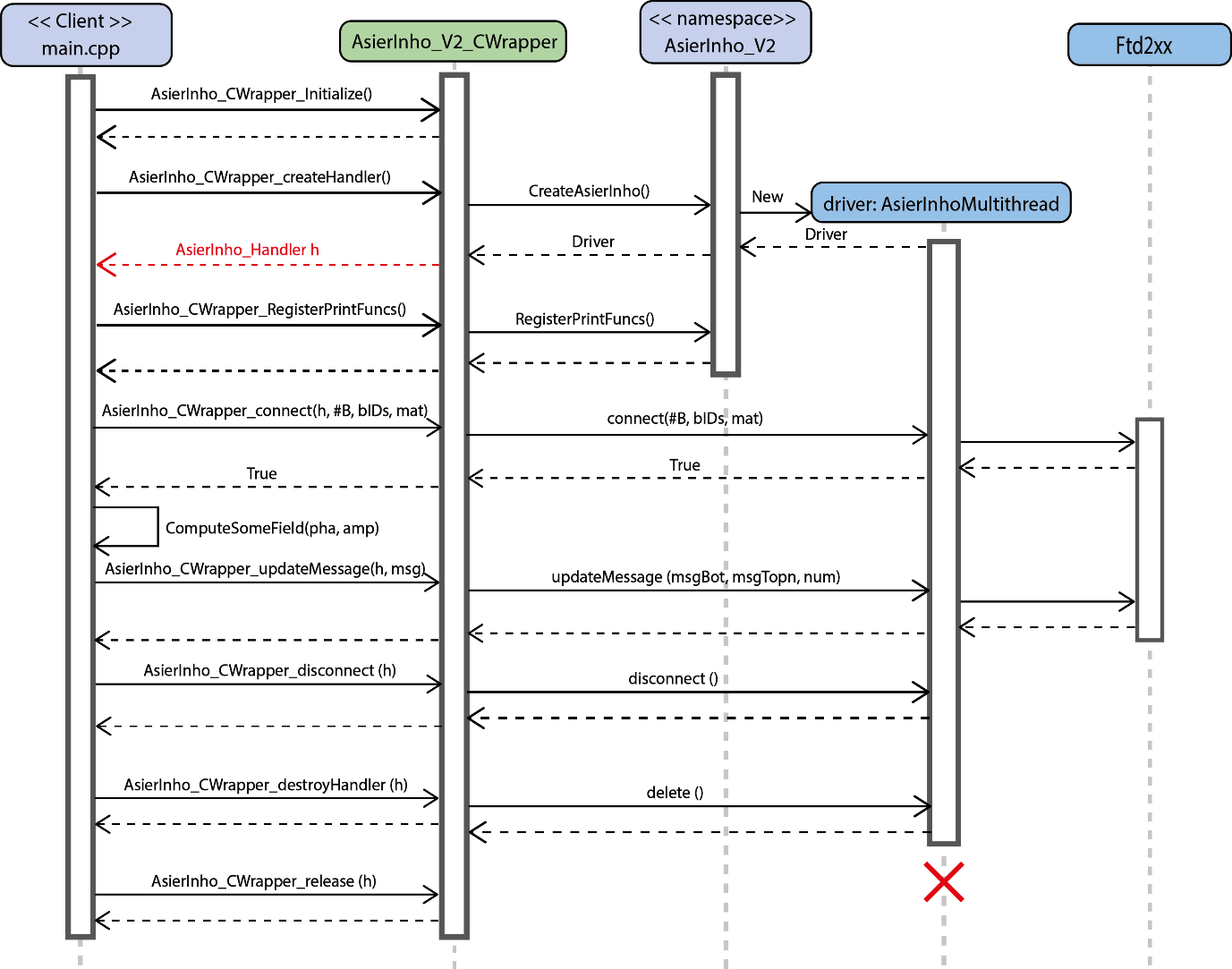


Figure 4**: Initialization, usage and destruction of our controller used in NORMAL mode through the C interface. Please note the parallelisms with** Figure 2**, where the C++ interface was used.**

# GS-PAT:

As introduced earlier, this project contains our GPU solver (GS-PAT), which allows for high performance computation of multi-point levitation/tactile points, with variable control of each point’s amplitude. High performance is enabled by the use of OpenCL (GPU computation), but also by using concurrent computation (i.e. the solver can compute in parallel up to 32 multi-point fields, which we usually refer to as ‘geometries’).

Again, the solver is encapsulated into a DLL, with interfaces for C++ and C. The following subsections will provide a description of the module, its interfaces and how to use them (i.e. useful if you only want to use our solver), but also its internal implementation (i.e. in case you want/need to extend, modify or improve it). We do not provide a comprehensive description of all methods and member variables used, as these are available in our Doxigen generated files.

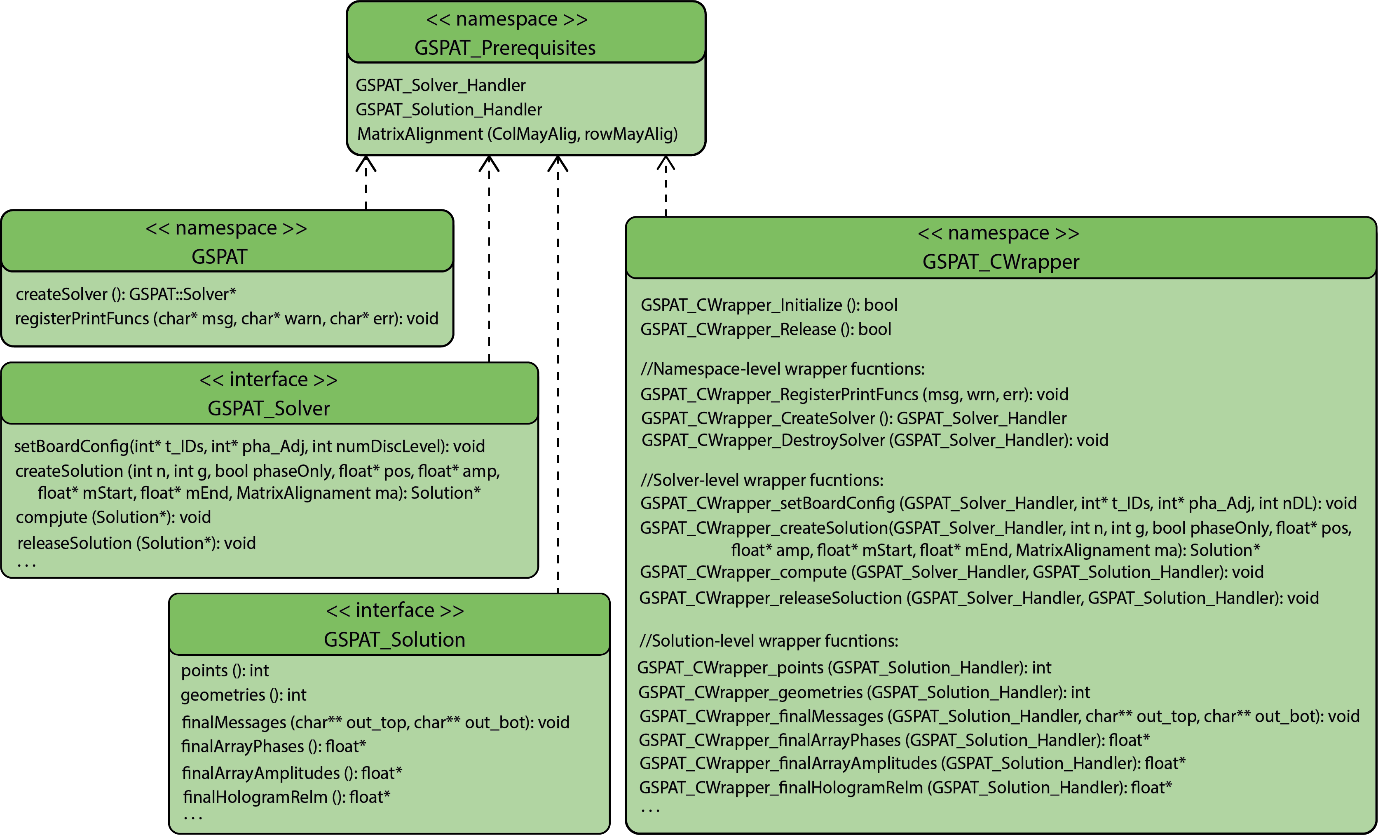


Figure 5**: Overview of the interface classes exposed by the GSPAT DLL. Implementation classes are not shown here.**

## Overall module structure: C++/C interfaces

This section only provides an overview of the interface classes exposed by the DLL ( i.e. element in the “*GSPAT/include*” folder). Their design and the way we present them in Figure 5 are very similar to those of the *AsierInho* DLL. The top-most file contains any common definitions required to use the DLL. The classes related to the C++ interface are placed to the left-hand side (they are all purely virtual interfaces) and the C interface wrapper is shown to the right.

* *GSPAT\_Prerequisites* provides basic type definitions required to use the DLL. This avoid making use of types defined in other external libraries.
* The namespace *GSPAT* defines global functions, providing high level functionality, such as retrieving instances of our solver or registering functions that GSPAT will call to notify the client about warning, errors or general notifications (i.e. for debugging).
* *GSPAT::Solver* encapsulates our multi-point phase-retrieval algorithm. The solver can be seen as a computing pipeline that clients can trigger to compute their target sound fields.
* *GSPAT::Solution* represents one instance of computation (i.e. a instruction) run through the solver (i.e. the pipeline). When clients need to compute a sound-field, they will create a *Solution*, configure it (e.g. position/amplitude of their points) and run it through the pipeline. They can then use the solution to read their results (e.g. messages to send to *AsierInho*).

Unlike the overview provided for AsierInho (see Section 3.1), the implementation classes are not shown in Figure 5 (i.e. we are only showing the elements above the “red line” in Figure 1), given their higher number and complexity. However, the design rationale of the GSPAT DLL is the same than before (i.e. implementation classes encapsulate dependencies with external libraries; factory methods are used to attain instances of implementation classes without exposing them).

## Using the C++ interface and examples

The C++ implementation is the actual core implementation of GSPAT (other interfaces refer to these elements) and, like *AsierInho* previously, it exposes pure interfaces (i.e. which is what C++ clients will retrieve and interact with), and encapsulates actual implementation classes.

A close up of a map

Description automatically generated

Figure 6**:** **Initialization, usage and destruction of GSPAT through the C++ interface**

The typical usage of *GSPAT* is illustrated in Figure 6. This illustrates the case of a client using *GSPAT* to compute a multi-point levitation field and *AsierInho* to recreate it using an actual device.

The initialization and connection to an *AsierInhoBoard* are similar to those described earlier. The client can (optionally) register functions for GSPAT to notify about potential errors or warning. The client then creates a GSPAT solver using our abstract factory (i.e. *GSPAT::createSolver*), reads the configuration of the device used (i.e. *AsierInhoBoard::readAdjustments*) and uses this to configure the solver (i.e. *Solver::setBoardConfig*). Please note that the solver creation step returns an instance of *HologramSolverCL*, which was not shown in Figure 5. This is actually the current implementation class for the *GSPAT::Solver* interface.

The following steps illustrate the process followed by the client to compute each of the target sound-fields required and their delivery to the device, and they are usually repeated in the client’s main loop. In each iteration, the client will retrieve a new *GSPAT::Solution* (i.e. *solver-> createSolution(…)*), and use the solver to *compute* it. This is an asynchronous call, and the control is returned to the client’s thread as soon as the required commands have been issued, but the computation might not be necessarily finished yet. The client can use this time to perform other tasks (i.e. *doSomethingElse* could be used to render content to the display, read sensor inputs, etc.). The client then retrieves the final sound-field (i.e. *solution->finalMessages(…)*), and can use the resulting buffers (*messages\_top*, *messages\_bottom*) to update the board (i.e. *driver->updateMessages(…)*). It is worth noting that the call to retrieve the final sound-fields will lock the client thread until the computation is finished. Also, the buffers returned are managed by the Solution, so the client should not delete them. Once the buffers have been used, the user must simply release its solution (buffers should not be accessed after the solution is released), and can start the next iteration.

The final steps in the diagram illustrate the disconnection process. The client must simply destroy its solver (this will deallocate any internal resources in the CPU and GPU), and destroy its board controller as explained before.

A simple example following this behaviour – connecting to an *AsierInhoBoard* , using it to compute single levitation traps and releasing/deleting the controller- can be found in our C++ solution (i.e. project “*Examples/GSPAT/4.simpleGSPATCpp*”).

## Using the C interface and examples

The C interface for *GSPAT* follows a very similar philosophy than that in *AsierInho*. C clients will interact with the solver using the methods in *GSPAT\_CWrapper*, which are simply a proxy for the C++ implementation. Like before, C clients will retrieve a *handler* (i.e. an object identifier) to interact with specific objects, but *handlers* can now refer to solvers (i.e. *GSPAT\_Solver\_Handler*) or to solutions (i.e. GSPAT\_Solver\_Handler). Please note the red arrows in Figure 7, which highlight the moment that the different types of handlers are created.

Otherwise, the usage of the interface remains very similar to that of the C++ interface (see Figure 7, which mimics the behaviour of the example in Figure 6), with the exceptions mentioned when we described *AsierInho\_CWrapper* (i.e. explicit methods to destroy objects, releasing the DLL deallocates all resources).

A simple example of usage of GSPAT\_CWrapper can be found in our C++ solution (i.e. project “*Examples/GSPAT/5.simpleGSPATC*”).

A screenshot of a social media post

Description automatically generated

Figure 7**:** **Initialization, usage and destruction of GSPAT through the C interface**

## Using GS-PAT in practice: the client perspective

The examples above provide an overview of how to make use of the solver to compute a given sound-field. However, the solver offers a wide range of possibilities for the creation of several types of (visual, tactile and audio) content, or the possibility to compute several sound fields in parallel. Also, the need to retain high update rates for the board while maintain consistent timing between updates raise individual challenges that anyone using GS-PAT needs to understand to be able to create the kind of multi-modal content in our demonstrators.

This section will try and cover all these aspects, related to how to make best use of GS-PAT.

### Types of content, general considerations and typical challenges:

GS-PAT allows us to deal with different types of content, with different capabilities but also different challenges related to their creation. We will broadly divide them according to their temporal requirements, that is, according to whether high update rates are needed (e.g. >10KHz) or lower rates are sufficient (e.g. hundreds of Hz):

#### Content at low update rates:

This kind of content includes:

* **Point-based primitives:** Werefer here to visual content enabled by multiple, independent beads, such as a cube with beads at its vertices or *LeviProps* [Morales, 2019]. Also, each bead does not need to move at high speed (i.e. no PoV content). These cases do not pose significant challenges. The client can use G=1 and solutions can be computed using only phases (*phaseOnly=true*). The relatively low update rate (e.g. hundreds of solutions per second) allows for the position of each point to be recomputed and updated in every frame.
* **Multi-point tactile feedback:** This refers to tactile shapes like those described in [Long, 2014]. Given our current top-bottom setup, their creation is not different than that of point-based primitives: in every frame, the client only needs to specify the position of the intended tactile points (i.e. in theory, no levitation signature should be applied; In practice, the user hand will occlude one of the boards, only receiving pressure contribution from the board facing his/her palm. Thus, even if a signature is applier, this is occluded by the user’s hand itself).

This kind of content is the easiest to generate. Clients normally will not need to make use of parallel computation (e.g. G=1), and they can simply integrate GSPAT in their application loop (e.g. a game engine, like Unity), computing a sound-field per frame with point positions matching the intended location of the levitation traps or tactile points required.

#### Content at high update rates:

This kind of content includes any content in which any of the parameters of the sound-field (e.g. position, amplitude of any of its points) needs to change over time at high rates (e.g. >10KHz):

* **Single/multi-point POV content:** This includes visual content creates by one or many particles tracing paths at high speeds (i.e. revealing the shape in <0.1s), so that the eye stops seeing the particles and sees the shapes that these are tracing/revealing. As per [Hirayama,19] optimum speeds can be achieved for update rates of the particle position above 10KHz. The creation of such PoV content is not inherently different than any other visual content (i.e. levitate and move particles), but the computation and delivery of >10K updates per second adds challenges to the way GSPAT is used and the client structures and delivers updates. This will usually involve parallel computation of several geometries (G>1) and, typically, dedicated threads to issue commands to the board at such high rates, while retaining good synchronization (i.e. one update every 0.1 ms). We discuss how to deal with these challenges bellow.
* **Single/multi-point tactile shapes:** These refers to the multi-point spatio-temporal modulation presented in [**Martinez,2020**], creating tactile shapes by rapidly scanning (one or more) focus points on the user’s skin. Again, the creation of such content is not inherently different than the creation of PoV content above sharing similar challenges.
* **Audible content:** Unlike the previous two categories, creating audible sound does not require high update rates of the point positions. Audible sound falls within this category because it involves high update rates of the **amplitude** of the points. Audible sound is created by modulating the amplitude of a given target point over time, according to the input audio signal (i.e. please note, the signal needs to be adapted for delivery using approaches from parametric audio, such us the single-sided band method we used in [**Hirayama,2019**]). The update rate used influences the maximum audible frequencies that can be reproduced (e.g. updating a 10KHz allows for audio component of up to 5KHz). The use of audible sound shares the challenges from other types of high update rate contents (parallel computation of solutions; synchronization) but they also require the solver to use variable amplitudes (e.g. *phaseOnly=false*), in order to specify the amplitude of the target audio sources in Pascals, retaining consistent amplitudes over time.

The definition of content using high update rates entails challenges that involve dealing with high update rates, use of parallel computation and (potentially) use of variable amplitudes. The following subsections deal with these challenges and how they affect the way we define content.

### Managing high update rates and computing solutions in parallel:

#### Dealing with high update rates: High-level (Content) and Low-level (Primitive) definitions

In many cases, updating your content definitions at the rates required (>10KHz) while keeping a consistent timing (delivering each update every 0.1 ms) imposes a very serious challenge to client applications.

In order to help with this, GS-PAT’s philosophy is to divide the definition of (visual/tactile/audio) content in two levels of granularity:

* **Low-level definitions (Primitives)**: This level provides a **fixed** **definition** of your content, **pre-sampled at the high update rate** you are using (e.g. 10KHz). That is, for any high-update rate content (PoV visual, tactile, audio), the client will hold pre-computed buffers describing the content’s positions/amplitudes over time (e.g. one sample per update cycle of, for instance 0.1ms). When it comes to position buffers, these should **define a whole cycle** (e.g. last position aligns with first position in the buffer), and the **positions are in local coordinates** to the content. This will avoid the need to recompute any of these positions at high update rates.
* **High-level definitions (Contents)**: This level simply describes the current location/ orientation of our content in the levitator using a 4x4 transformation matrix. That is, while the *Primitive* defines the positions in local coordinates, the (high-level) *Content* will allow us to move it inside the levitator. The advantage is that while the *Primitive* stays fixed (and keeps up with our high update rate), the *Content* can be dynamically changed/moved at lower rates (e.g. hundreds of Hz), compatible with the interactive rates that the client application (e.g. a rendering engine, like Unity) can deliver.

This decomposition is no different to how a 3D model is rendered by modern GPU pipelines. *Vertex buffers* (e.g. equivalent to our Primitives) define all the points/faces in our model, but they remain fixed over time. The renderer then simply updates the *Model matrices* (i.e. Content) to move/orient contents in space in real time. Please note OpenMPD later builds on this philosophy to automatically manage rendering at such high rates, but that is only possible thanks to this behaviour of the solver.

#### Computing solutions in parallel

The philosophy above is reflected on the way *GSPAT::Solutions* are created and computed. The *position* and *amplitude* buffers can stay precomputed, and the client can provide a transformation matrix to define their position in space (i.e. two matrices *mStart* and *mEnd* are used, as explained below).

This meets two related needs. First, the client application might struggle to run at high update rates (10KHz), and be better suited to run cycles at rates of a few hundred frames per second (e.g. 500fps). Second, the solver can provide sound-fields at such high rates (e.g. 10KHz), but to do so it will need to compute “batches” of several geometries in parallel. That is, the solver might provide 10KHz updates per second by running it at 500Hz and computing 20 sound-fields each time (500 x 20 = 10KHz).

This is the reason why two matrices per solution and content are used (i.e. *mStart* and *mEnd* instead of one matrix per content, such as in an OpenGL renderer). In the example, the client will issue 20 geometries to be computed in parallel. The first geometry will refer to the initial positions/ amplitudes of the contents, while the last geometry will refer to their positions/amplitudes 20 updates later. Each update is separated by 0.1ms, for a total difference of 2ms between the first and last geometry. The matrix *mStart* describes the transformation (position/orientation) to be applied to the first geometry, while *mEnd* describes the position/orientation to apply to the last geometry (2ms later). The geometries in the middle will use matrices interpolated between *mStart* and *mEnd*, providing a smooth transition over time and avoiding sudden jumps between “batches” (i.e. these would happen if only a matrix was used, updated at 500Hz).

Even if two matrices are needed, this is easily managed in practice in a very similar way to how this is done in OpenGL. In every “batch” computation, the client can set *mEnd* to the current matrix to be applied to the content (i.e. matrix *M(t)* as per your animation, input device, etc). Matrix *mStart* can then be set to the matrix computed in the previous frame (i.e. matrix M(t - 2 ms) ). This will ensure continuity between “batches” at the only expense of your transformations to be reached by the last geometry (e.g. 2ms later).

**NOTE**: The potential of the solver to compute at high rates lies in its ability to compute many sound-fields (i.e. point geometries) in parallel. We identified a peak for G=32 geometries, with further increases in G producing a plateau in the total number of sound-fields per second computed. The client can compute/use any number of parallel geometries, between 1 and 32.

### Solving for variable/fixed amplitudes (phaseOnly):

The flag *phaseOnly* allows us to specify if we want the solver to compute only transducer phases or both phases and amplitudes, and we will need to use one or another, depending on the content that we want to recreate:

* **Fixed amplitudes (*phaseOnly=true*):** Computing only phases implies that all transducers will use maximum power, and is the most common approach to use when we want to levitate a large number of particles or have them moving as fast as possible. In this case, we can still specify variable amplitudes for our target points (e.g. traps), and the solver will tend to produce an output that respects the relative value of those amplitudes (e.g. if we define one target point with amplitude 1 and a second point with amplitude 0.5, the final peak pressure delivered for the first point will be twice as high as that of the second). Thus, when operating with *phaseOnly=true*, it is common to specify the pressure of the target points in a normalized [0,1] range (it is the relative amplitude between points that matters, not the actual magnitude specified).
* **Variable amplitudes (*phaseOnly=false*):** The use of parametric audio relies on accurate control of the amplitude (pressure) of the target points, particularly for those that work as audio sources. In these cases, it is important that the reconstructed amplitudes remain consistent over time (and related to the magnitude of the audio signal). It is also important that we retain the timing of the audio signal (i.e. play a 10KHz audio signal, updating exactly every 0.1ms). This can be done making use of variable amplitudes (i.e. *phaseOnly=false*). In these cases, the target amplitudes can no longer be specified using arbitrary units (like before), and we need to specify the amplitude of our target points in Pascals (Pa), and using a Pa value that is within the capabilities of our setup.

While working with fixed amplitudes is relatively simple (i.e. we use *normalized* amplitudes [0,1]), the use of variable amplitude brings a significant issue: we need to work out what a feasible Pa value is. This depends on our setup but also, on the number of target points in our field (feasible target amplitudes tend to decrease with the number of points) and on their positions (amplitudes attenuate more or less depending on the relative location of each point).

For these cases, we used the results from our evaluation in [Martinez,20] to determine feasible target amplitudes, according to the number of points used, which we summarize in Table \_\_. These describe mean amplitudes that the solver achieved across 1000 random geometries for each number of points tested ({2,4,8,16,32} points), thus providing conservative estimates of the amplitudes that should be feasible. It must be said that this is only a conservative guidance, and one that is specific for our solver configured for a reference pressure of 8.2 Pa (this is the default reference pressure and the one that you are most certainly using). Also, these values should be considered as “peak values” (i.e. the value equivalent to you amplitude 1, in the *phaseOnly* case).

**Table 1: Reference peak pressures to use in variable amplitude mode, according to the number of points in the sound-field.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **numPoints** | 2 | 4 | 8 | 16 | 32 |
| **Peak Pressure** | 13000 Pa | 9400 Pa | 6800 Pa | 5000 Pa | 3800 Pa |

A second challenge, as mentioned before, is in retaining good timing between the audio signal and its delivery through the device. That is, the step above will make sure that the amplitude of the reconstructed signal follows the evolution of amplitudes in our audio file. However, we still to make sure that each of the samples is delivered at the correct time, as to also retain the timing of the audio signal. In order to do this, we make use of a real time clock to read the results from the solver and deliver them to the board at fixed rates. This will be described in detail in section 5.\_\_\_\_\_

### Encoding Solutions (GSPAT::Solver::createSolution):

The solver allows several ways of specifying solutions and understanding these is important to make best use of the solver. Once these aspects have been explained in the previous subsections, we here provide an summarized overview of the parameters we can setup when creating solutions (i.e. *Solver::createSolution()*):

* *numPoints*: Points in each geometry. All geometries share the same number of points.
* *numGeometries*: Number of geometries *G* to be computed in parallel. Computing geometries in parallel is crucial to go from computing a few thousand updates per second, to >10KHz. The solver shows optimum performance for *G*=32, which is set as the maximum possible value.
* *phaseOnly*: Specifies if the solver should solve only phases (A=1, for all transducers) or phase and amplitude.
* *positions*: buffer describing the positions of each point in each geometry. The buffer must contain all "numPoints" points for Geometry 0, then "*numPoints*" points for *Geometry 1*, etc. That is, for a Solution computing G=32 geometries of Z=6 points each, the position buffer must encode the 6 positions of the points for the first geometry, followed by the 6 positions for the second geometry, all the way until all geometries are encoded into the buffer. Please remember, positions use homogeneous coordinates, requiring 4 floating point numbers per point; amplitudes only require one floating point number per point.
* *amplitudes*: Specifies the amplitude of each point in each geometry. If "phaseOnly"==true, amplitudes can be described in a homogeneous range (i.e. [0,1]). However, if phase and amplitude is used, the target amplitudes must be described in Pascals.
* *mStart*: buffer storing Transformation matrices describing the position/orientation of each of the numPoints for the first geometry.
* *mEnd*: Transformation matrix describing the position/orientation of each of the numPoints for the last geometry.

### Format of final messages and other types of output (phases, amplitudes, complex):

The GSPAT pipeline includes stages that result in messages directly ready to be sent to our *AsierInho* driver (see Section 4.5), and these can be accessed by calling *Solution::finalMessages()*, as shown in Figure 7 and Figure 8.

Messages are stored in two output buffers (i.e. *messages\_top*, *messages\_bottom*), each of them containing the updates to be sent to each of the boards (already formatted according to the mapping and phase adjustments of the board, as specified in *Solver::setBoardConfig()*). The number of messages stored equals the number of simultaneous geometries G computed, and they are stored in a sequential manner. As explained in document *“1.FPGA Firmware/2.Explanation of the protocol.docx”*, each message/package occupies 512 bytes.

The client should still be careful to retain adequate timing between updates. That is, if operating at 10KHz, the client should trigger the sending of each message each 0.1ms. This usually requires the use of a dedicated thread (independent of the rendering/application logic), reading the solutions and sending them at the right times.

While the use of *Solution::finalMessages()* is the most common use for GS-PAT, the clients can retrieve results in other ways:

* *Solution::finalArrayPhases(): float\**: Returns the final array describing the phases of each transducer. This method is used for debugging purposes, and for backwards compatibility (e.g. clients that did not use the discretization in the GPU solver). Please note that while the Levitation signature has been applied, no other discretization steps are included (transducers mapping, phase adjustments).
* *Solution::finalArrayAmplitudes(): float\**: Returns the final array describing the amplitudes of each transducer. This method is used for debugging purposes, and for backwards compatibility (e.g. clients that did not use the discretization in the GPU solver). Please note that no discretization steps are included (transducers mapping, phase adjustments).
* *Solution::finalHologramReIm():float\*:* Returns the final array describing the complex field (Re/Im for each transducer). This method is retained for debugging purposes and can be used to visualize the fields (see HelperMethods). No Levitation signature has been applied (it is just a focusing hologram). No other discretization steps are included (transducers mapping, phase adjustments).

It is worth noting that by default these outputs are available only in the GPU. The pipeline does not automatically load them back to CPU buffers during execution (i.e. to improve performance). Hence, calling these methods will trigger an OpenCL command to read this buffers into main memory and the client thread will be blocked until reading is finished.

## GS-PAT: C++ Implementation and the OpenCL pipeline:

### Overview of implementation classes

The previous sections described the interfaces published to GSPAT clients and the way clients could use the GSPAT solver. Here we describe the actual implementation of our solver, which involves the classes shown in Figure 9.

* ***GSPAT::Solver*** and ***GSPAT::Solution*** are the pure interfaces described in Section 4.1
* ***HologramSolverCL*** provides the implementation of our *GSPAT::Solver*. It sets up the computing pipeline (in OpenCL) required to compute the required sound-fields, and allows clients to retrieve (and dispose of) *GSPAT::Solution*’s, to make use of such pipeline.
* ***HologramSolution***’s are like the instructions executed by our pipeline. Each *HologramSolution* provides the definition of the target points (levitation traps, focus points) that the client needs to create. The *HologramSolution* can then be run through our pipeline to compute the transducer activation that produces such target points. Besides the input (i.e. definition of target points) and output data (i.e. buffers to store transducer’s activation), this class contains all the intermediate data/buffers required to run it through the pipeline.
* ***SolutionPool:*** The solver has a finite number of solutions, which are reused over and over. *HologramSolution*’s contain a large amount of data (input/output/intermediate buffers) and allocating/deallocating them every time the client runs a computation adds a severe computing overhead to the system. Instead, *SolutionPool* keeps hold of the pre-allocated solutions, allowing retrieval and returning of the solutions in a thread-safe manner (this is required, given the asynchronous nature of our computation).
* ***OpenCLUtilityFunctions:*** Thisclass provides static functions for common tasks related to dealing with OpenCL kernels, such as loading and compiling OpenCL code from a file.

A screenshot of a cell phone

Description automatically generated

Figure 9**: Implementation classes in our GSPAT solver**

### *HologramSolverCL*: The GSPAT pipeline

*HologramSolverCL* provides an implementation for the algorithm described in [Martinez, 20], and we refer the interested reader to that paper for an explanation of the underlying mathematical explanation.

It is also worth noting that the class makes use of an external file “*hologramSolver2.cl*”, containing the OpenCL code that builds the core of this implementation. The description of the implementation must consider both elements (the C++ class *HologramSolverCL* and the file “*hologramSolver2.cl*”).

#### Initialization:

The solver itself (*HologramSolverCL*) contains the global OpenCL resources required to run the pipeline (e.g. OpenCL context, queues and kernels), as well as data buffers for the bits of information that will be shared among all solutions.

Kernels and global OpenCL elements are created during the object construction (see *initializeOpenCL()*). Shared data buffers include the following:

* *cl\_mem transducerMappings:* Describes the pin ID of each transducer in its respective board. Such data is obtained during *setBoardConfig* andused during the discretization stage to build *AsierInho* messages.
* *cl\_mem phaseAdjustments*: Describes the phase correction required by each transducer. Such data is obtained during *setBoardConfig* and used during the discretization stage to build *AsierInho* messages.
* *cl\_mem directivityTexture:* Our solver uses a piston model to define the directivity of our transducer sources. Such model describes the scalar response of the transducer at a point in space (in Pascals), as a function of the angle between the transducer’s normal and the point. Computing this angle in the OpenCL shader is (slightly) costly, and it is more convenient to compute the cosine of such angle (i.e. dot product). Thus, the directivity function is precomputed as a function of the cosine, and encoded into a 1D texture (*directivityTexture*), allowing the shader to access it as if it was a continuous variable. The texture is created during object construction (see *createDirectivityTexture()*) and used in stage *computeFandB.*
* *cl\_mem transducerPositions:* Buffer describing the position of each transducer in homogeneous coordinates (4 floats per position). Positions are automatically computed during object creation in *createTransducerPositionBuffer()*. The current implementation is hardcoded for top-bottom arrangements of 512 transducers.
* *SolutionPool\* solutionPool:* Contains the pool of solutions to be reused by solver’s clients.

#### Computing solutions (C++ and OpenCL perspectives):

The C++ side of computation is encapsulated in the *HologramSolverCL::compute()* method, which is decomposed in the stages shown in Figure 10:

A screenshot of a computer

Description automatically generated

Figure 10**: Overview of OpenCL commands/kernels (blue boxes) run at each stage of the pipeline and the synchronization events coordinating execution (black arrows).**

* *updateCLBuffers* simply copies the information in the *Solution* to OpenCL buffers in the GPU.
* *computeFandB* computes the forward and backward propagators required by our modified GS algorithm.
* *computeR* calculates the two-step propagation matrix, making use of the library *clBlas*.
* *solvePhases\_GS* optimizes the phases of each target point, using our iterative method.
* *computeActivation* uses the points’ final phases and amplitudes and matrices B to compute the final activation as an addition of individual points.
* *discretise* discretizes theactivation computed, packaging it into *AsierInho* messages (see “*1.FPGA Firmware/2.Explanation of the Protocol.docx*”).

In most cases, these methods simply trigger the execution of an OpenCL kernel (i.e. one our own kernels, a basic OpenCL kernel (e.g. such as *clEnqueueWriteBuffer*), or *clBlas* commands). Thus, the C++ implementation mostly deals with setting the input parameters of the kernel, as well as the synchronization events required.

Synchronization events are particularly relevant, and they are shown as black arrows in Figure 10 (e.g. *computeFandB* must wait until position data has been written). The method compute simply sets up the kernels and triggers them, returning control to the client as soon as possible (i.e. does not wait for kernels to be computed). The synchronization events are the ones that ensure that the order of execution between kernels respects data dependencies.

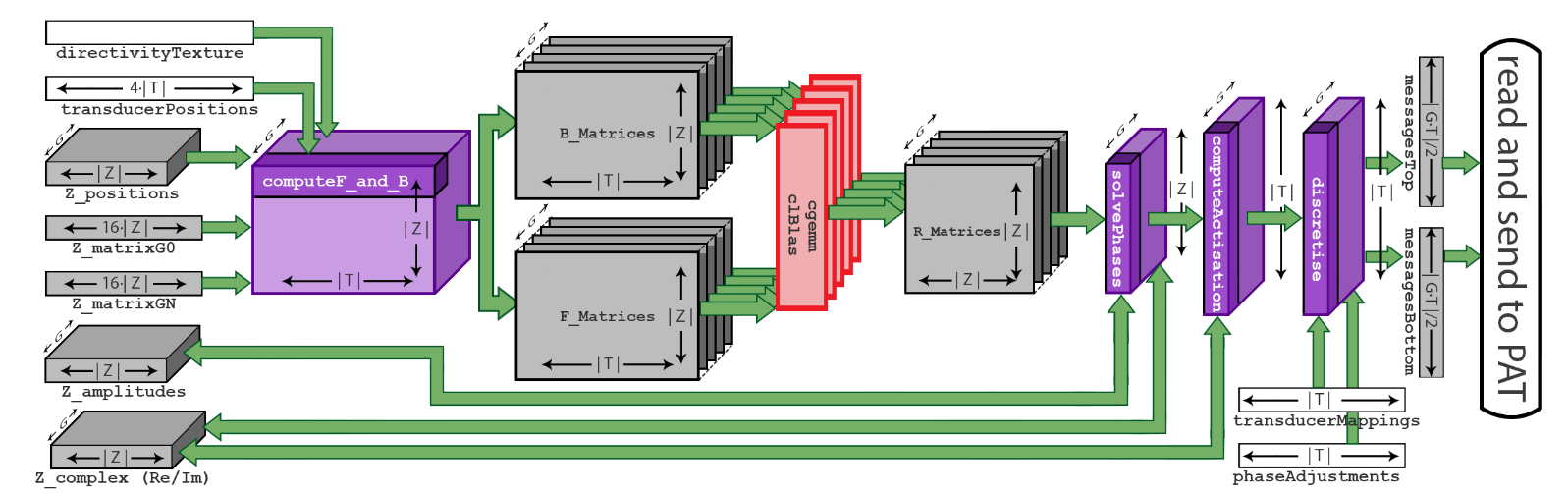


Figure 11: Overview dataflow in the current OpenCL pipeline, highlighting the data buffers…

While Figure 10 provides an overview of the OpenCL commands run by the pipeline and their synchronization events, Figure 11 provides an overview of the data flowing through the pipeline. This illustrates the role of each kernel in file “*hologramSolver2.cl*”, showing their input and output parameters. More specifically, white boxes identify shared data buffers (i.e. defined by the Solver), while grey boxes identify solution-level data buffers (note how most data is actually defined at a Solution level). Purple boxes identify kernels implemented in “*hologramSolver2.cl”* and red boxes represent calls to *clBlas* (i.e. OpenCL calls to read/write data omitted for brevity).

Dimensions of buffers are provided (where possible) in terms of number of target points (Z), number of transducers (T) and number of simultaneous fields to compute (G). When applied to kernels, the box dimensions identify global work-group sizes (total amount of computation), while shaded areas identify the size of the local work-group sizes (computed in parallel). Please note that the maximum local work-group size used is of size |T| (i.e. 512, in our current setup). This means that the GSPAT solver requires a GPU with support for (at least) this size.

* *[OpenCL Kernel] computeFandB* : This kernel computes our simple propagators (F) and normalised propagators (B). Each compute element only computes the activation of one transducer, for a given target point and geometry. The local work\_groups are arranged to cover all transducers for a given point and geometry. A reduction kernel is used within each local workgroup to compute the overall amplitude of the propagator, which is then used to compute the normalised propagator B (amplitude of exactly 1 Pa).
* *[OpenCL Kernel] solverPhases\_GS*: This kernel implements the iterative 2-step GS method. The number of iterations can be manually tuned by defining the value of NUM\_ITERATIONS in the first line of “*hologramSolver2.cl*”.
* *[OpenCL Kernel] computeActivation*: This kernel computes the final transducer activation required, using the results from the previous step and the *B* propagators. The results from this step still have **no levitation signature applied**. Also, the constraints related to using the *phaseOnly* flag in the solution are not applied yet. It is worth noting that, while Figure 11 only shows the final messages being computed (i.e. ready to be sent to *AsierInho*), this kernel computes other intermediate outputs which are also useful for simulation/debugging purposes:
  + *finalHologram\_Phases*: Contains the phases to be sent to the board, as floating-point numbers (not discretised yet; no levitation signature).
  + *finalHologram\_Amplitudes:* Contains the final amplitudes to send to the board, as floating-point numbers. (not discretised yet; *phaseOnly* flag ignored).
  + *finalHologram\_ReIM:* Contains the complex representation of the final hologram (not discretised, no lev. Signature: *phaseOnly* flag ignored).

*[OpenCL kernel] discretise*: This kernel uses the outputs from the previous step (particularly *finalHologram\_Phases* and *finalHologram\_Amplitudes*), the data related to the board configuration (transducer mapping and phase adjustments), and the constraints specified (*phaseOnly*), to discretise and package results into messages directly usable by *AsierInho*.