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

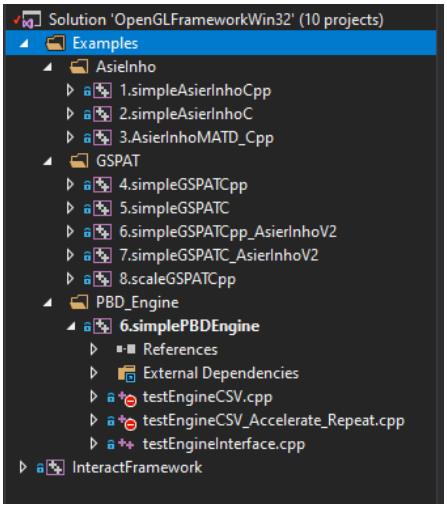
1 About this document:

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.

2 Project Overview:

The Instructions to get the latest version of the software can be found in the file "Code_Documentation_Resources\Documentation\4.SoftwareImplementation\ImplementationOverview.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\Unity Integration.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).

3 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.

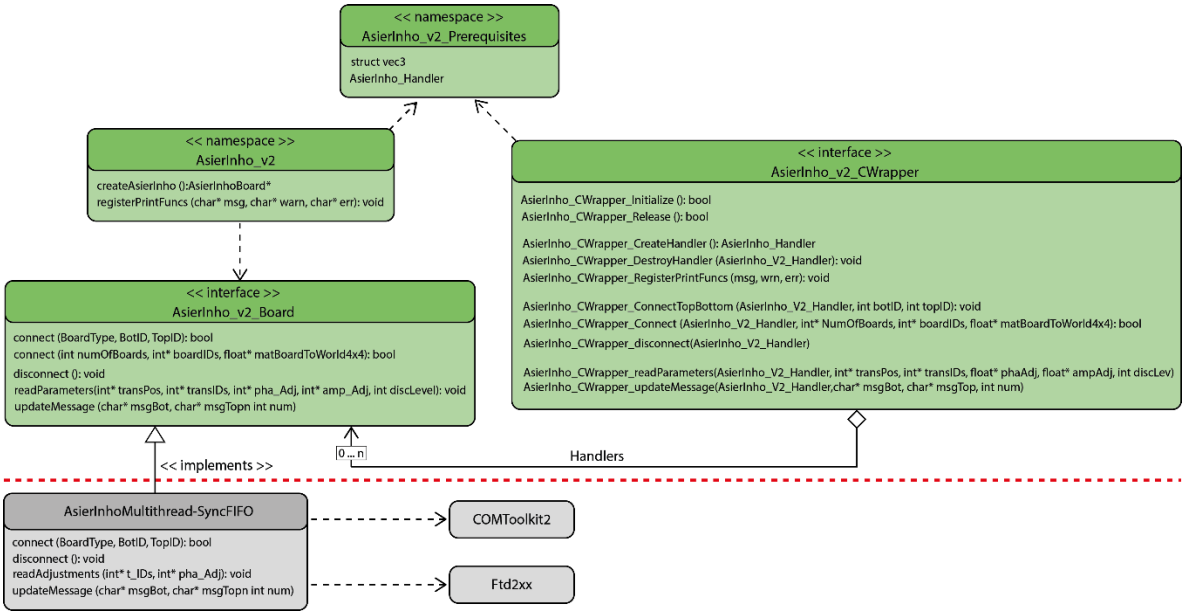


Figure 1: ***Class diagram of AsierInho_V2 DLL. Please note some methods are omitted for brevity.***

3.1 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.

3.2 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.*

3.2.1 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.

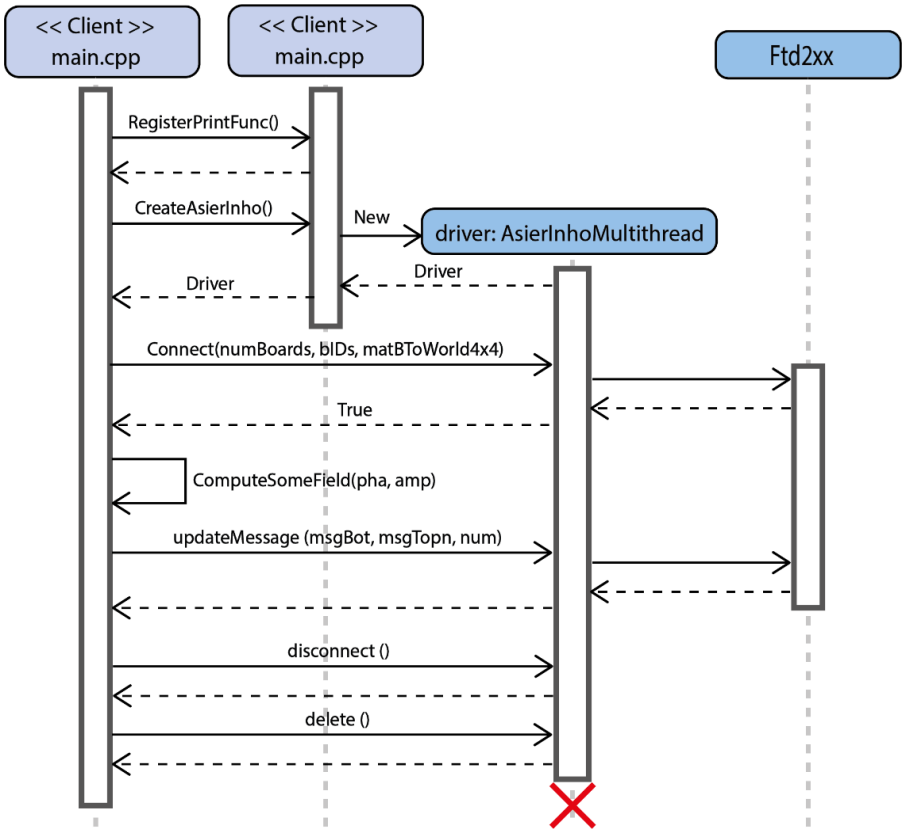


*Figure 2: **Initialization, usage and destruction of our controller, when used in NORMAL mode.***

3.2.2 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.

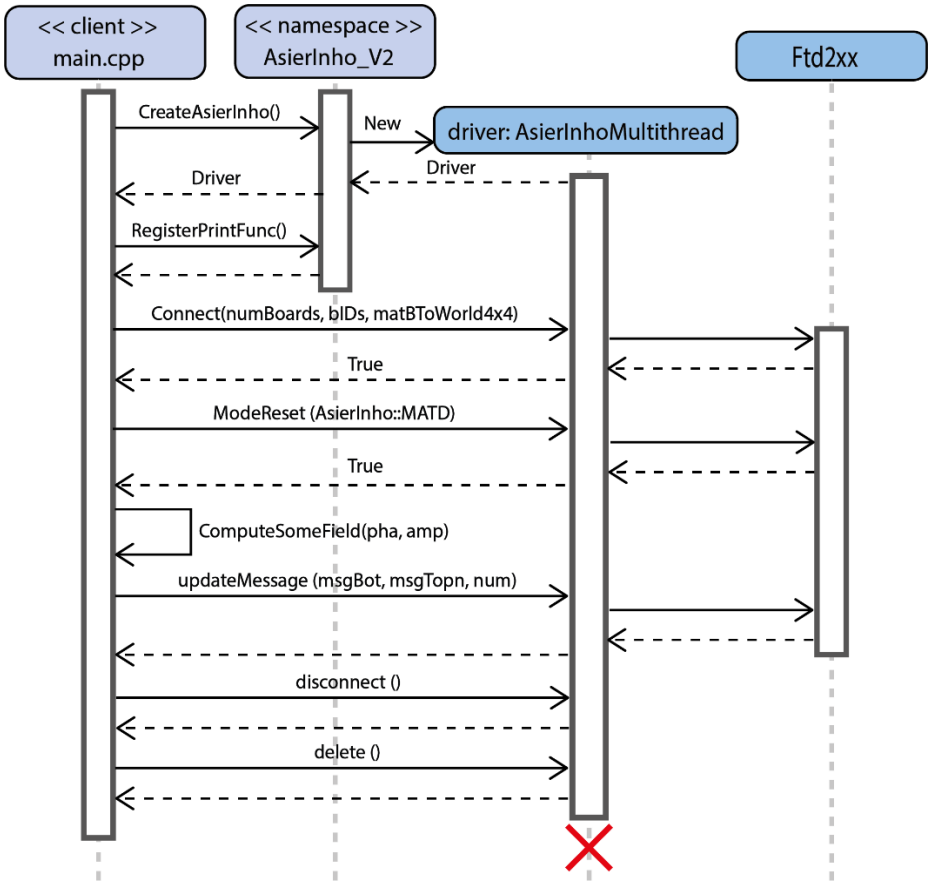


*Figure 3: **Initialization, usage and destruction of our controller, when used in MATD mode.***

3.3 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.

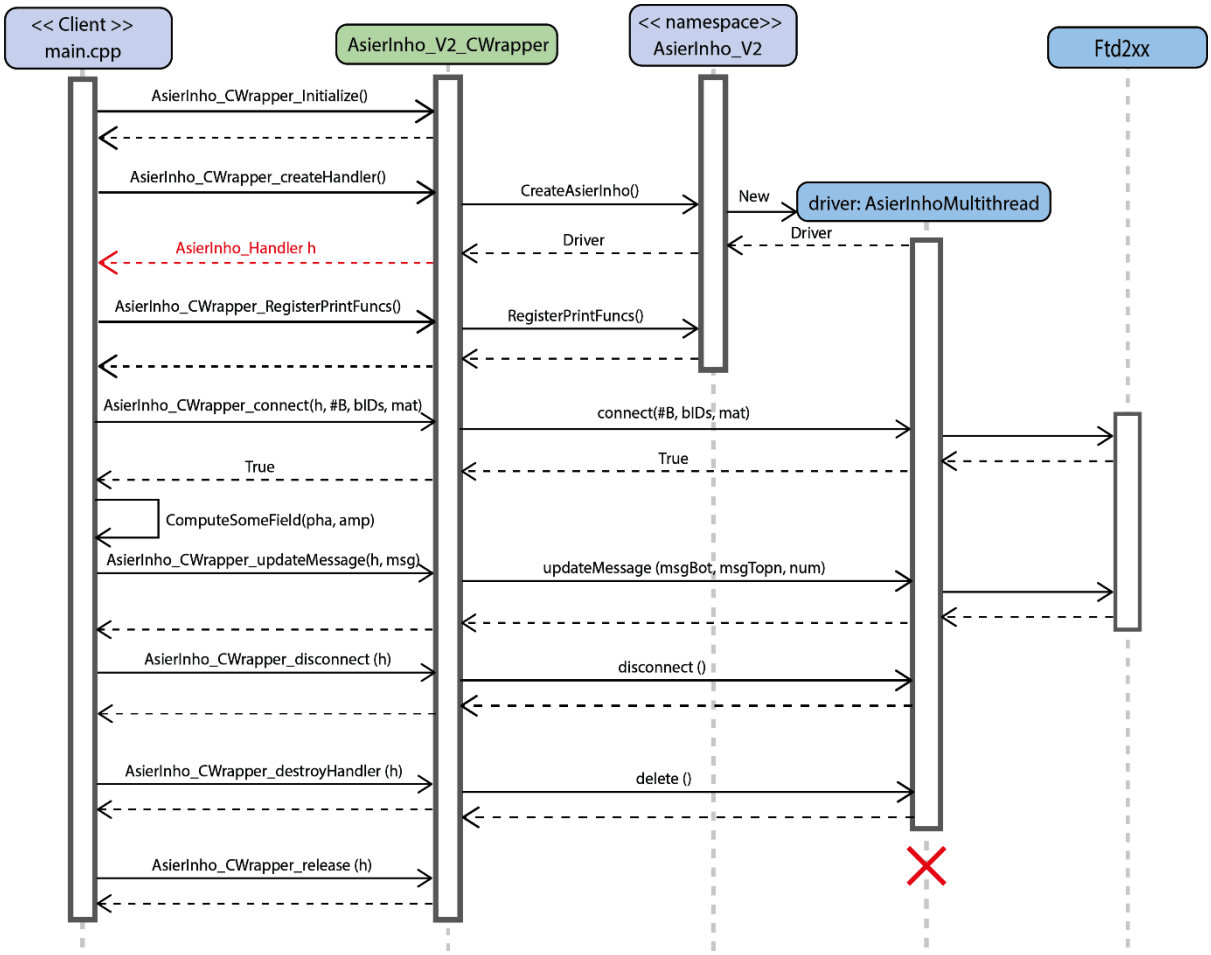


*Figure 4: **Initialization, usage and destruction of our controller used in NORMAL mode through the C interface. Please note the parallelisms with** Figure 2**, where the C++ interface was used.***

4 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.

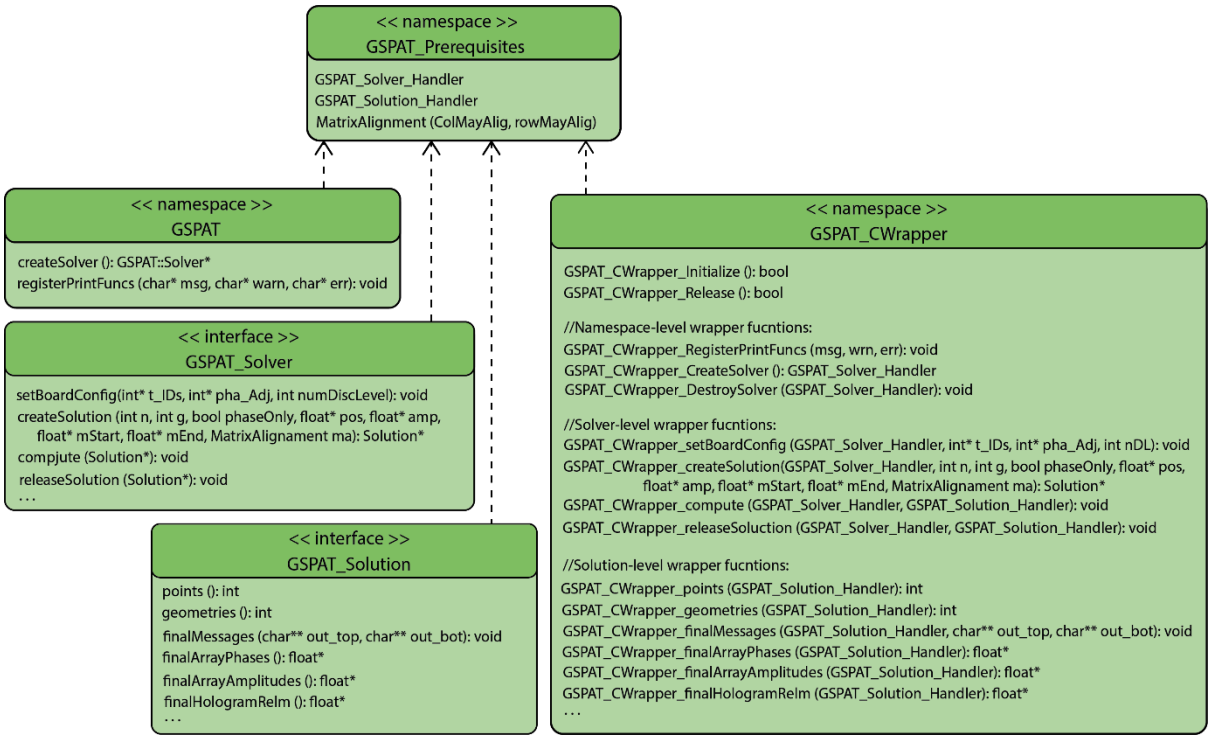


*Figure 5: **Overview of the interface classes exposed by the GSPAT DLL. Implementation classes are not shown here.***

4.1 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).

4.2 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.

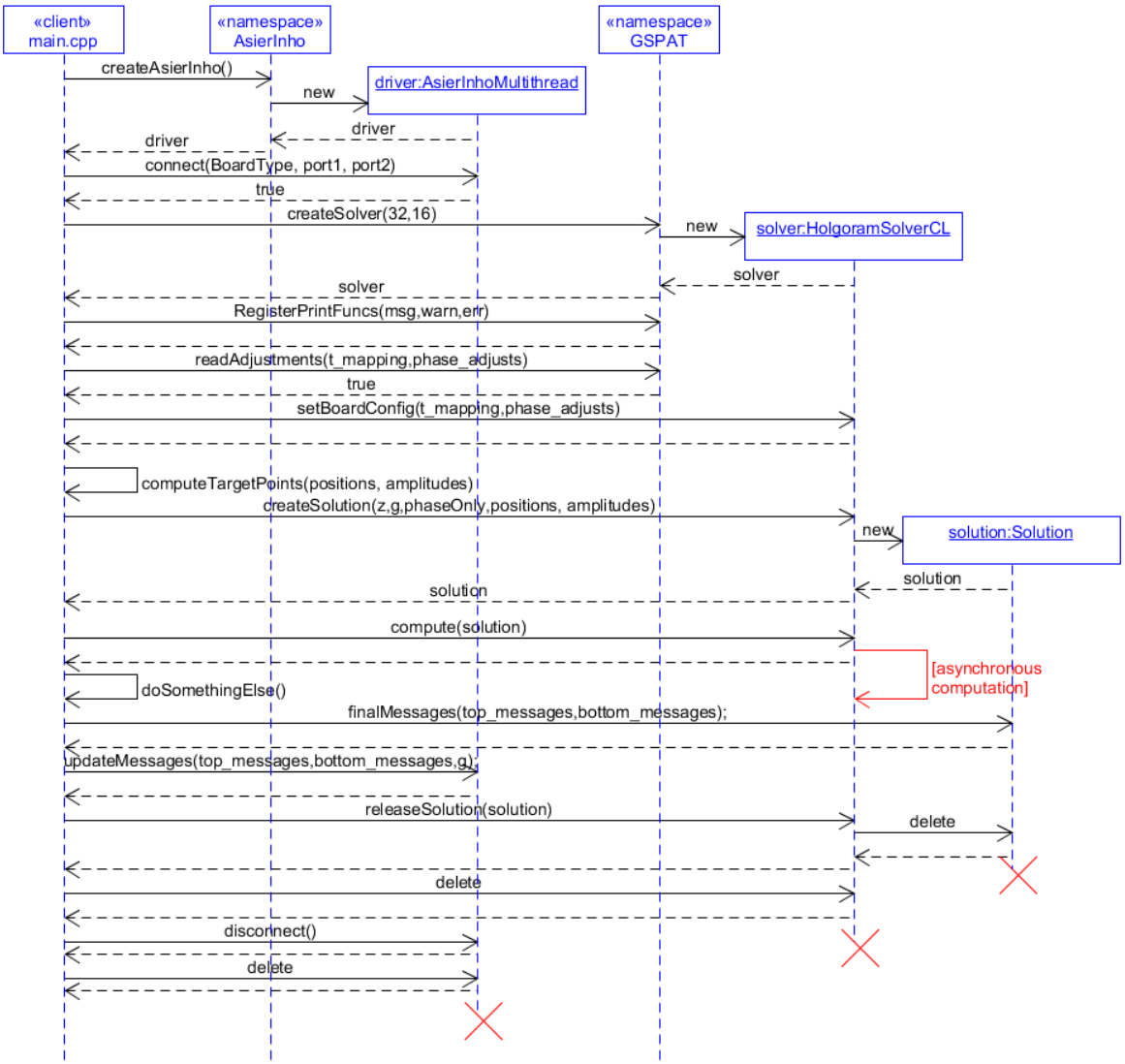


*Figure 6: **Initialization, usage and destruction of GSPAT through the C++ interface***

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*").

4.3 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*").

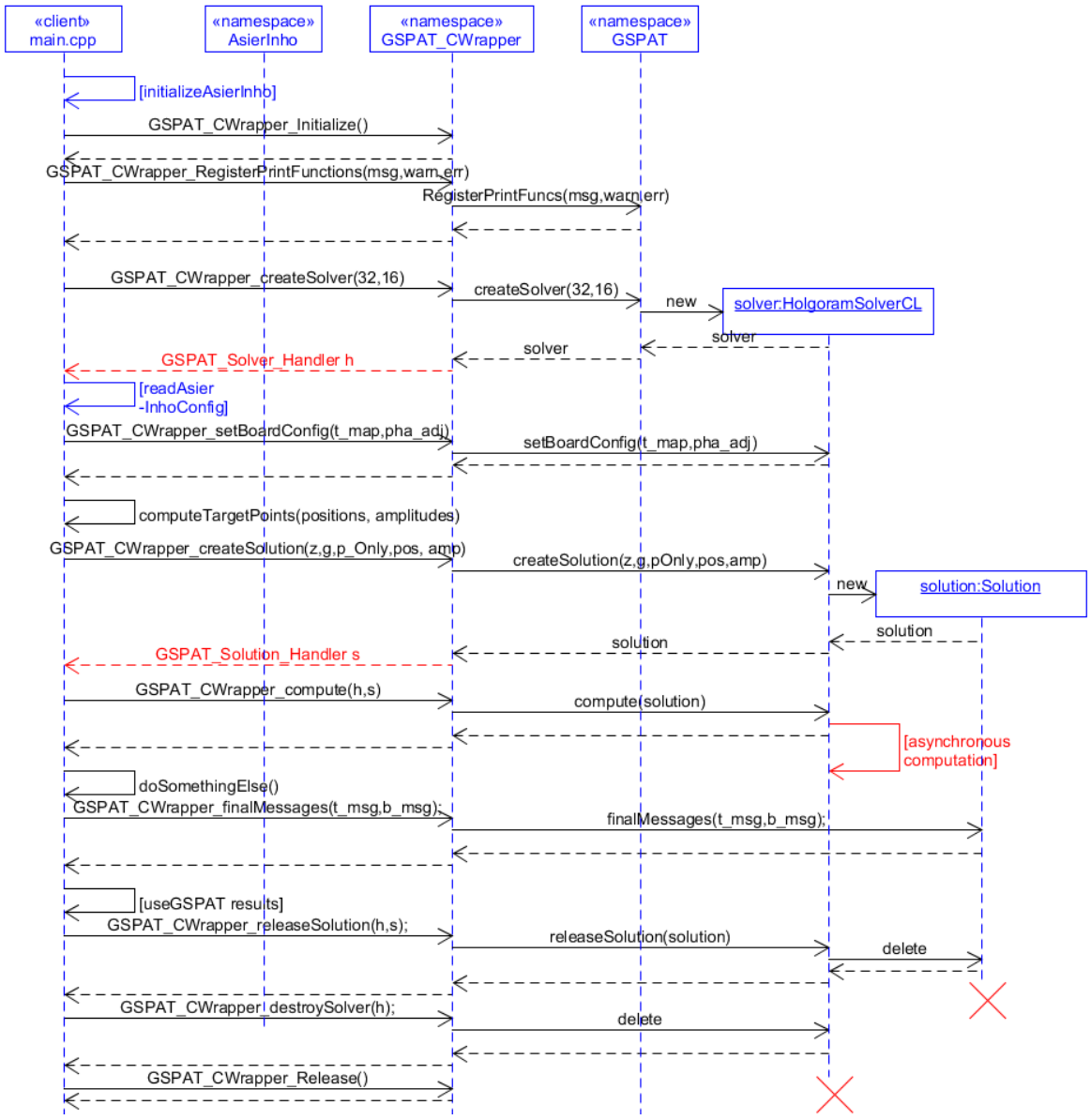


*Figure 7: **Initialization, usage and destruction of GSPAT through the C interface***

4.4 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.

4.4.1 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):

4.4.1.1 Content at low update rates:

This kind of content includes:

- **Point-based primitives:** We refer 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.

4.4.1.2 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.