UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Centre de la Imatge i la Tecnologia Multimèdia

Audiophysic simulation library for video games [NoPhysicsLibrary]

by Martí Buxeda Sardans

Director: David de la Torre & David Font

Grau: 20º

Curs: 2022-23

Universitat: UPC - CITM

This page has been strongly thought to intentionally be left blank.

Index

Abstract

Nowadays, sound designers must hard-code any acoustic effect in their video game, because there is not any module that automatizes the real audio behavior depending on the physical environment. That is what NoPhysicsLibrary is: a C++ library that allows developers to create their own world, apply realistic physics to it, and play sfx that change depending on the environment. The report shows a detailed description of the physics module, which explains the iteration methods and integration calculations. Also, is included a detailed explanation of the formulas used to calculate the real-time effects that will be applied to the played audio files, such as spatialization, volume attenuation, sound occlusion, frequential attenuation, pitch variation, and time delay. Finally, an overview of the audio system's functionality is presented. The resulting library is published on a github repository and several stand-alone test scenarios are provided. Two demonstration videos are also presented as part of this work, featuring all the acoustic effects covered by NoPhysicsLibrary.

Keywords

Acoustics, API, Audio, Audiophysics, C, C++, Library, Physics, Programming, Research, Video Games

Links

- NoPhysicsLibrary in 1 minute
- NoPhysicsLibrary | Features Showcase (Project Validation)
- NoPhysicsLibrary Repository (Project Validation)
- MiniAudio Public Issue

Table Indexes

Figure Indexes

Glossary

- Declip: The operation to separate two overlapping bodies.
- Delta Time: Time between two consecutive application frames.
- SFX: short for Sound Effect
- Vector: A programming data structure which allows to store and iterate any amount of defined objects.
- Hard-code: A programming practice of writing direct results into your code, as opposed to making the program scalable to any possible circumstance.
- Sound Roll Off: The volume attenuation of a sound over the distance

1. Introduction

1.1 Motivation

The video game industry, as we know it nowadays, contains a diverse range of arts, but among the most essential are music and sound design. The sound effects, music, and overall ambience account for more than 70% of a player's overall experience, which can mean the difference between a good and a bad game.

There has been a notable evolution in audio systems over the years. We started with basic playback of Square Saws beeps produced by the self console, whose goal was to provide limited feedback to the player. The techniques applied to sound effects today have progressed to the point where we can audibly replicate reality, so the game experience can be coherent. This is the reason why it is interesting to create libraries that can handle logic on their own rather than hard-coding the reality yourself. Obviously, it is critical to make the API as generic as possible in order to give the sound designer the maximum development flexibility possible.

The primary challenge is recreating an audio response to the physical events of the video game. Game developers must comprehend acoustics in order to correlate physics events with the appropriate audio configuration. The challenge is to capture reality in 3D or 2D games using a spatial audio environment.

At this stage, the state-of-the-art solutions provide the ability to "hard-code" how the sound behaves. Some engines, for example, let you place spheres of sound rolloff, manage filters such as low-pass or reverbs, apply panning to the audio based on the listener's position... However, some features are missing, such as the time delay produced by the medium through which the sound travels.

To summarize, the primary issue in the industry today is that the coordination between physics and audio is extremely hard-coded. I would even risk claiming that game developers do not care or comprehend enough about this relationship. To the best of my understanding, there is no readily accessible library that can handle both physics and audio while allowing both systems to interact automatically with each other, with capabilities such as generating and processing audio solely based on the physics occurring in the scenario.

1.2 Objectives of the TFG

IN-SCOPE:

Creation of a physics engine:

- Bidimensional point mass integrator.
- Allows application of forces and momentum.
- Allows friction, gravity, aerodynamic, hydrodynamic and elasticity forces.
- Collision solver algorithm by rectangle intersections.
- Provide summarized information about physics state (Audio System Requirement)

Audio modulation as a function of physics effects & environment:

- Apply audio modulation sfx:
  - Pitch modulation.
  - Automated reverb.
  - Volume management.
- Apply physics effects onto audio:
  - Sound propagation.
  - Medium change.
  - Reflection.
- Implement a 2D spatial audio system:
  - Audio Source Panning.
  - Volume attenuation rolloff.
- Generate a useful library from the audio/physics engine.
- Develop a video game demo to show up the library capacity.

LIFE-SCOPE:

- Investigate other effects such as diffusion.
- Expand the project to high-level API.
- Generate my own audio library.

OUT-SCOPE:

- Create a graphics library (we will use SDL2)
- Develop an audio engine (we will use SoLoud)
- Develop an input system (we will use SDL2)
- Create a final video game (Gold Version)

1.3 Project Scope

The project aims to be for all developers who require assistance with the physics and audio of their video games. It will be a simple library to achieve high-quality results with basic programming.

It is also intended for research, owing to the library's high scalability and wide modularity. This library is generic and does not assume any value, which means that everything is configurable and that each tweak changes the results received.

The reason a self-built physics library was done rather than implementing another is straightforward. To convert the physics events into audio features, is crucial to fully understand the internal logic of the physics engine, and be able to modify and extend its characteristics.

The constraints set for this project are linked to the platforms and languages that can use this library. Apart from the self-built physics engine, the audio library will be drawn from pre-existing external APIs, limiting the project's specialization.

It should be noted that all logic outside of Audio and Physics will be delegated to other libraries. It fully outsources the development of graphics and event APIs for this project.

2. Art state

The most common techniques used in the industry are the use of filters and random pitch modulation to achieve audio modularity in game sfx, which improves player immersion by ensuring that the next sfx does not sound identical to the prior one.

Another popular technique is audio spatialization, which enables developers to place sound in specific areas to create an immersive environment. Reverb is used to create depth, and panning and level attenuation are used to create a left/right localized experience.

However, real-world acoustics produce other effects besides those already mentioned, which adding them as a library feature is one of the goals of this project. This generates the following query: Is there already a solution available on the market?

First and foremost, I contacted several people who work in the video game industry and have general knowledge of audio tools. I explained the concept of an acoustic library to them, and their response was unanimous. They had never used or heard of anything similar.

Following that, I began researching the current market on the internet. I found various High Level programs that had some of the characteristics of an acoustic library, but nothing that really solved my thesis.

WWise:

A multi-platform software that offers advanced features such as interactive music and adaptive audio, and integrates seamlessly with game engines like Unity and Unreal.

FMOD Studio:

Another popular game audio engine, known for its powerful sound design capabilities and user-friendly interface. It supports real-time mixing and monitoring, and better than WWise, can be connected to Unity, Unreal, and CryEngine.

Unity Audio:

A system that is internally implemented directly into the Unity game engine. It provides basic tools to create simple sound effects and background music.

Unreal Audio:

As Unity Audio, this system is built into the Unreal game engine and includes features really similar to the previous mentioned. Allows spatialization of the audio, using panning and volume attenuation, as well as doppler effect.

It is important to note that this market analysis is not focused on audio or physics APIs, but rather libraries that enable users to hard-code any kind of acoustic effects in video games.

Because of the hard specialization of each of them, it is difficult to find consistent APIs in the Low Level area. There is no fully generic audio system library that contains all of the features of the previously mentioned High Level programs.

However, there are some useful option that would do the trick, such as:

OpenAL:

It is a cross-platform API for representing three-dimensional, multi-channel positional music. Its API style and standards are designed to be similar to those of OpenGL.

FMOD:

The low level version has been designed to support interactive music, sound effects, and voice playback for games and other interactive applications.

SDL Mixer:

It is a GitHub-hosted example multi-channel audio mixer library. It has 8 tracks of 16 bit stereo audio and one channel of music. It allows 3D spatialization audio to the user, and managing audio filters.

IrrKlang:

IrrKlang is a simple 3D music library that supports WAV, MP3, OGG, and FLAC files. It has a variety of audio effects such as echo, delay, and distortion that can be heavily customized.

Wwise:

The low level version offers various features like real-time mixing, effects, and audio synchronization. It is very similar to the high level api, but without an UI to rely on.

All of these tools are extremely well designed and produce high-quality results, having been tested by numerous professionals and being used in a variety of famous video games.

Although they appear to be ideal, they require a high level of acoustic knowledge in order to recreate a realistic environment with an accurate depiction of real-life physics. And it is interesting because, in order to represent reality through audio, we need the assistance of physics. Surprisingly enough, all of the products listed above separate both modules, either because they're specific audio libraries or because the game engine in which they're built is designed that way.

If you think about it, the physics and audio should be two components joined by the hands, or at least, connected to an “Acoustics” module. One is immediately related to the consequences of the other, but it is not present in any of the market solutions available.

3. Project Management

In this part, I'll concentrate on TFG project management, as well as some cost and difficulty analysis.

3.1. DAFO

Strengths Is an underutilized market topic. Can be used for game development and study.	**Weaknesses** It is challenging to create a generic library from it. It requires a plethora of features in order to be marginally helpful.
Opportunities Because of its low-level structure, it can be changed to a high-level structure.	**Threats** It is difficult to locate the research and information needed to make it generic. There are numerous particular formulas, but none that are generic.

3.2. Risks and Contingency Planning

Risc	Solució
Unable to find required libraries	Adapt them to my needs and extend them if it is needed
Find libraries with constraints in which audio system they work on	Prioritize multi-platform APIs or adapt them by myself.

3.3. Costs analysis

The price per hour for library creation would be around 20€, plus a budget of 1000€ for testing the library. In this instance, my TFG started one year ago, and I have four months until the deadline. The average number of hours devoted per week has been 15h/w, resulting in a total cost of 19.000€.

It would further require an overhead of 30€/h to hire physics and audio department experts to monitor and ensure the quality of the library's performance and coherency with the real physics world. The deal would last approximately 200 hours (100 hours per each), for a total of 6000€.

The economic outcomes are determined using gross wage.

4. Methodology

The proposed methodology consists of three main stages: construction, expansion, and demonstration. These stages must be completed sequentially. They are completely independent of one another, but it is possible to revert to a previous stage if required.

4.1 Construction

This stage involves the general construction of the library's base code. This includes user experience concept testing as well as logic and mechanics implementation. This stage is divided into sub-phases.

- PHYSICS BUILDING: It is critical to begin building the physics engine because the core of the library logic is having audio react to game physics. This step encompasses all physics and code research.

- ACOUSTIC BUILDING: Following the physics engine, it will be coded with logic linked to both audio and physics, such as volume attenuation, delay response, spatialization, liquid filters, reverb..., as well as relevant research.

- AUDIO BUILDING: Finally, the audio engine will be implemented, allowing the user to playback music without spatialization or any physics relationship and/or consequence.

- LIBRARY TESTING: This stage entails having the user beta-test the library to see how intuitive it is and to identify any bugs or missing features.

- LIBRARY BUILDING: And finally, there will be research into how to construct the library and make it accessible online.

4.2 Expansion

Once the library is finished, the expansion process will begin. This stage involves increasing the library's internal presets, such as physics, scenario, and material presets.

It will also be necessary to create a documentation page and some user tutorials to engage the community to use the API on their projects.

Generating a good issue system on github is also an imprescindible topic to take into account, in order to use developers as library debug agents.

4.3 Demonstration

During this phase, I will build a simple game that showcases all of the library's features for anyone to test. Finally, the game will be created and uploaded to the official website of the library.

Also, a video explaining and showcasing every library feature will be edited and linked in this report.

4.4 Project Tools

It is also necessary to highlight the tools used throughout the project, which are listed below:

- Visual Studio 2019: Coding, programming, and development for the library.

- Discord: A meeting place with the project's tutors. In addition, we will set up a server with various chat channels. A "general" channel for communicating with the tutors. A "fonts" channel, where all the research web-pages will be listed so that they eventually can be linked in this report. Finally, an "ideas" channel where all types of new proposals for the library are proposed and discussed.

- GitHub: As a utility for creating a repository and storing code in the cloud. This will make it simpler to resolve conflicts and organize branches.

4.5 Meetings with tutors

During the building phase, we will meet with the relevant tutor (physics and audio) on a weekly basis, to monitor and guide me through the code development process.

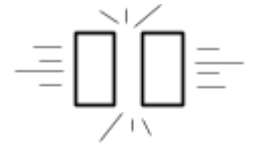
We will use an Agile approach, meeting with the fictional library user at each meeting and discussing the requirements to implement/solve for the next update.

5. Project Development

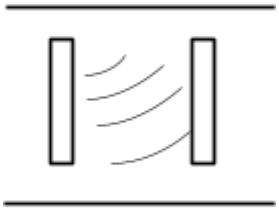
5.1 Library Structure

NoPhysicsLibrary is an API that unifies a physics engine with an audio engine, coining the name of Acoustics Engine. As a result, it is supported by three primary pillars: Physics, Acoustics & Audio.

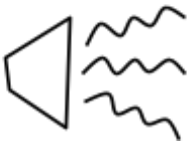
Using this library, the developer can simulate fundamental physics using squared hitboxes with a point-mass model, recreate music and sound effects. The innovative feature involves the physics engine cooperating with the audio playback, in order to modify the sounds to be coherent with the world created. It includes audio spatialization, sound traveling delay, dynamic audio filters, material creation, between other features. The previously mentioned pillars are encapsulated into different classes inside the library, where each class is tasked with a certain function:



The simulation of the laws of physics, management of the bodies, and resolution of all collisions fall under the responsibility of the physics class. Additionally, it retrieves data that is beneficial to programmers and can help in the development of the application.

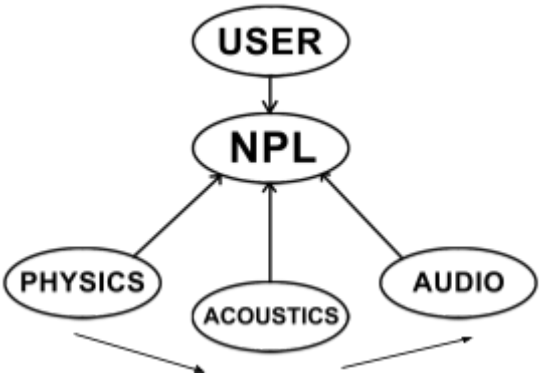


The acoustics class takes all the data from the physics engine and processes it into information that the audio engine can understand. It determines the listener logic, sound occlusion, pitch variation, frequential attenuation, audio spatialization and propagation delay.



Last but not least, the audio class is in charge of reproducing music and sound effects by using various filters on the data that has been processed by the acoustics class. The audio files loaded into the library are saved inside this class.

These classes are handled by the core class, NoPhysicsLibrary, so the end-user is not required to deal with them directly.



All of the pillar classes are sequentially called during the game's execution, and this ensures that each one is independent of the others. All the methods required to create, configure, and retrieve data from the world are contained inside.

The main object used by the API is called Body. All of the data that NPL needs to iterate the physics, acoustics, and audio is contained within Body class. It contains tools for changing Body parameters, playing sounds, applying forces or momentums to the Body, and retrieving useful data.

5.2 Developer Workflow

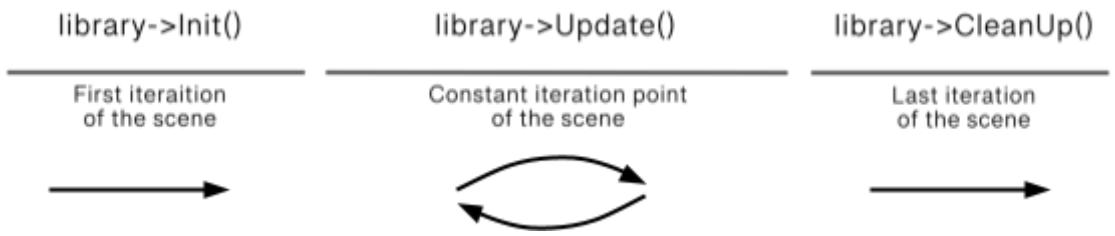
I decided to expose all the library's functionalities through only three specific objects in order to maintain a straightforward and understandable workflow for the developer. By doing so, the library's available methods are constrained to three classes and are easily accessible by adding a "->" after the variable name. One of the main problems with other libraries is that their functions are global, so users are unaware of their names and must read the documentation in order to locate and comprehend them. NoPhysicsLibrary solves this problem via the previously mentioned encapsulation.

Due to the library's dual focus on research and game development, it is essential to maintain a usable API workflow:

5.2.1 NoPhysicsLibrary Class

This core class defines the entire Acoustic World, containing all the bodies and materials of the scenario. If multiple worlds are required, it can be created more than once, but it is not necessary. The variable should be initialized dynamically, which is highly recommended.

It is advised to build the fundamental framework for each frame iteration before beginning to implement any feature of the library itself. The developer's code must call the following functions, in order:



The Init() method must be called on the first iteration of a scene, and always before any other function. In order for the library variables to function properly, it enables and initializes them. A float parameter named "pixelsPerMeter" is also requested. As the name suggests, the game's developer must specify how many pixels make up a full meter.

The Update() function must be called each iteration, after the Init(). The physics, acoustics, and audio engine are iterated inside. It requires a target delta time.

The CleanUp() function is required because it removes all of the data stored in the NoPhysicsLibrary variable. In the last iteration of the scene, it must be called. The developer can then call Init() again to restart the library variable. It can be used to restart the world so that a new one can be created.

It is interesting to note that this library only iterates the audio and acoustics engines when a Body produces sound. In order to maximize API flexibility, it is completely optimized to only iterate physics if the user does not require any type of audio.

Inspecting the library structure further reveals that each engine iterates within particular functions:

The Step Physics process applies gravity, friction, restitution, hydrodynamic forces if in a liquid, and aerodynamic forces if in a gas. Finally, it integrates all of the forces to determine the final acceleration, velocity, and position. This procedure will be expanded upon further in this report.

Following the iteration order, the Solve Collisions process involves two major steps:

1. Detecting all the overlaps between the bodies of the scenario, and storing all the relevant data to process it in the next stage.

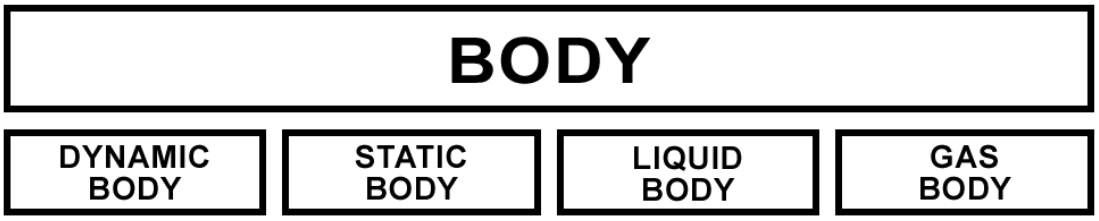
2. Using the previously acquired information to declip each overlapping body in the best way. The internal logic of this process is really complex and is left out of this report for clarity purposes.

The Simulate process begins when any Body makes a sound. It extracts data from the selected audio file and, if a listener is present, it begins to calculate all of the necessary acoustic data so that the audio module can apply it to the sound. If no listener is present, no information is calculated, and the audio system plays the sound completely dry.

The Audio engine's Playback process then takes all of the information that the acoustics engine has processed, applies it to the desired audio file, and plays the sound.

5.2.2 Body Class

The Body class is the library's fundamental unit. It is, by definition, a rectangular spatial entity with mass and size that interacts with the world that has been created. The base Body, in fact, cannot be instantiated because it is a virtual class. The developer will work with the following derivatives:



- **Gas Body**: This Body type defines the world's gas/atmospheric environment. It influences the sound propagation velocity as well as the lift force applied to the bodies within this gas. It cannot be moved by forces. If there isn't any created, sound cannot propagate through a gas, causing inaudibility when the emitter and receiver are in void.

- **Liquid Body**: A Liquid Body is a fluid rectangle in the physical world. It affects the velocity of sound propagation, applies filters to the sound based on viscosity, and influences the movement of the body due to buoyancy forces.

- **Static Body**: A solid, immobile body in the world. It implies a static collision for bodies and, depending on its young modulus, occludes sound.

- **Dynamic Body**: A movable solid Body in the world. Developers have the ability to apply forces and momentum to this Body. It occludes sound as the static Body, depending on its young modulus.

The only way to create these bodies is to use the NoPhysicsLibrary variable's "CreateBody()" method. This method requires a rectangle as parameters. To specify which type of Body is required, the developer must place a "->" in front of the function, and four more functions with the name of each Body will be displayed, with their specific parameters to configure.

In terms of audio playback, all bodies can emit sound, but an audio file must be loaded into the library first. LoadSound() is a method on the NoPhysicsLibrary variable that accepts an external file path. That same function will return true or false whether the sound has been correctly loaded or not.

Every Body contains a Play() method that accepts an integer and a float. The integer represents the index of the loaded audio file in order, and the float represents the decibels of the sound between 0dB and 120dB. This data is saved in a list. Following that, the acoustics engine will detect the presence of a sound and will begin its logic.

It is necessary to know who is listening to the sound emitted in order to spatialize the audio. In the absence of a listener, the library interprets that there must be no spatialization or acoustic effects on the audio and thus plays it directly.

To assign a listener, use the Config() method found within the NoPhysicsLibrary variable. This function does not accept any parameters and may appear useless at first glance. To see all the configurable parameters, the developer must use a pointer arrow after the function name, such as "Config()->". This will bring up a slew of functions for changing the world's internal parameters. There will be one called "Listener()" among them, that accepts a Body pointer. The developer must send their preferred Body to become the world's listener.

As a result of the physics of the world, the library will automatically compute the logic to spatialize and modify the audio.

5.2.3 Material Class

Each body comes with a default material, which can be changed or modified later on. The material class utility loads specific parameters inside itself, and then it can be assigned to a body. The sounds emitted in the world will be modified by the variables inside.

To load materials, the material class provides three main functions:

- BuildSolid(): This function takes as parameters specific variables for Solid Bodies, such as the young modulus, the absorption coefficient and density.

- BuildLiquid(): This function takes specific variables for Liquid Bodies as parameters, such as viscosity, bulk modulus, the absorption coefficient and density.

- BuildGas(): This function takes specific variables for Gas Bodies as parameters, such as the heat ratio, the pressure, the absorption coefficient and the density.

Each Body has a SetMaterial function that allows the developer to assign their preferred material.

5.3 Behind the Physics Logic

The Physics class is in charge of determining body behaviors and resolving collisions. Its logic starts with a function called "StepPhysics()" in the Update() method of the NoPhysicsLibrary variable.

The main framework of the physics course can be found in that function:

```
void NoPhysicsLibrary::StepPhysics(const float dt)
{
        bodies.Iterate<Physics*, float>
        (
                [](Body* b, Physics* p, float dt)
                {
                        p->Step(b, dt);
                },
                physics, dt
        );

        physics->SolveCollisions(&bodies);
}
```

Iterate through each body to integrate its forces and momentums to determine its final position, velocity, and acceleration.

To declip each body, find all collisions in a single frame.

Let's take a look at the physics class's Step function. This function requires two parameters: the current Body pointer to be processed and the delta time as a const float.

The internal workings of the Step method are as follows:

```
void Physics::Step(Body* b, float dt)
{
        if (b->Class() != BodyClass::DYNAMIC_BODY) return;
        DynamicBody* body = (DynamicBody*)b;

        if (!body->HasPhysicsUpdatability()) return;

        ApplyNaturalForces(body);

        body->ApplyForce(globalGravity * b->Mass()
                , InUnit::IN_METERS);

        body->ApplyForce(body->gravityOffset * b->Mass()
                , InUnit::IN_METERS);

        body->SecondNewton();

        body->FirstBuxeda();

        body->Backup();

        Integrate(dt, b);
}
```

It filters non-dynamic bodies. Only bodies to which we can apply a force must be processed.

The user can disable the integration of a dynamic body. So we filter out the non-updatable ones.

Apply forces related to liquids and gases, such as aero/hydro drag, buoyancy, etc…

Applies the global gravity to the body. It can be setted on the “Config()” function.

Applies the local gravity offset to the body. It can be setted inside each Dynamic Body.

“The sum of all the forces applied to a body is equal to the body's acceleration times its mass.”

“The sum of all the momentums applied to a body is equal to the body's velocity times its mass.”

Saves the information of the body, so next frame can be used to know the body's direction.

Integrates all the information to acquire the final acceleration, velocity and position.

Now that the Step function's pipeline has been described, I'll go over some functions in detail:

ApplyNaturalForces(): This function englobes the application of forces to liquids and gasses, which are referred to as ApplyHydroForces() and ApplyAeroForces(), respectively:

> ApplyHydroForces(): It iterates through all of the created liquids in the scene, checking for collisions with the Dynamic Body we're processing. If it collides, the following forces are applied:

- Hydro Drag: - ½ * body density * body velocity^2 * sunken area (opposite direction of the body direction)
- Hydro Lift: ½ * body density * body velocity^2 * sunken area * lift coefficient (vertical axis from the hydro drag direction)
- Buoyancy: body density * sunken area * gravity * Buoyancy coefficient (force that makes the body emerge from the water)

> ApplyAeroForces(): It iterates through all of the generated gasses in the scene, checking for collisions with the Dynamic Body we're analyzing. If it collides, the following forces are applied:

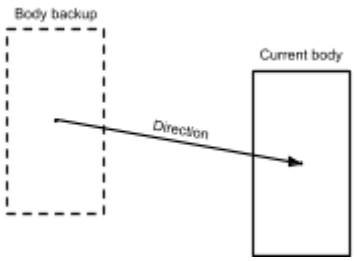
- Aero Drag: - ½ * body density * body velocity^2 * area * drag coefficient (opposite direction of the body direction)
- Aero Lift: ½ * body density * body velocity^2 * area * lift coefficient (vertical axis from the aero drag direction)

SecondNewton(): Iterates through all of the applied forces in that specific frame of the processed body. Finally, it divides the whole sum of the forces by the mass to obtain the body's final acceleration.

FirstBuxeda(): Iterates through all of the applied momentums in that specific frame of the processed body. Finally, it divides the total sum of the momentums by the mass to obtain the body's final velocity increment.

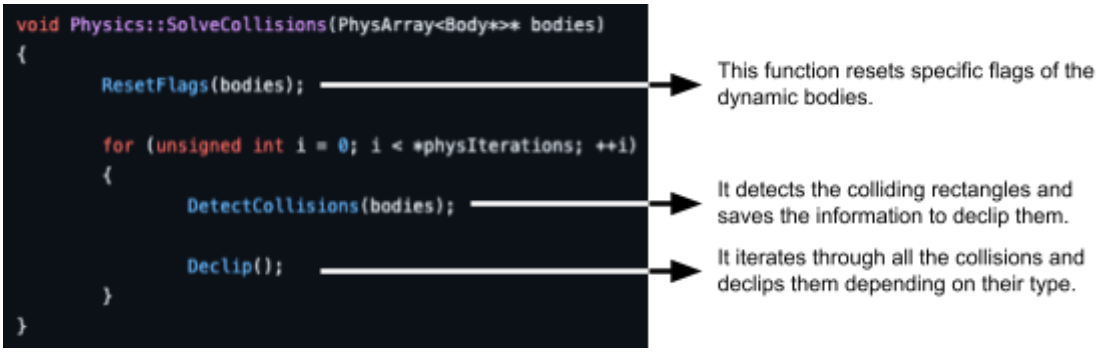
Fun fact about the name of the function: as the specific formula of "the sum of all momentums equals velocity times mass" did not have a "creator," I put my surname on it, as if I were its inventor.

Backup(): It generates a "BodyBackup" class that stores the body's rectangle, velocity, acceleration, total momentums, and total forces. It is named BodyBackup backup and is saved within the body itself. It is used to declip the bodies in the event of a collision and to calculate the body's direction, as shown here:



Integrate(): It integrates the acceleration to obtain the velocity, allowing the final position to be determined and fixed to the body. I employed the verlet's integration approach.

Let's look at the SolveCollisions() method next. This function only requires one parameter, a pointer to the vector of all the bodies in the scene. Let us go over how it works internally:



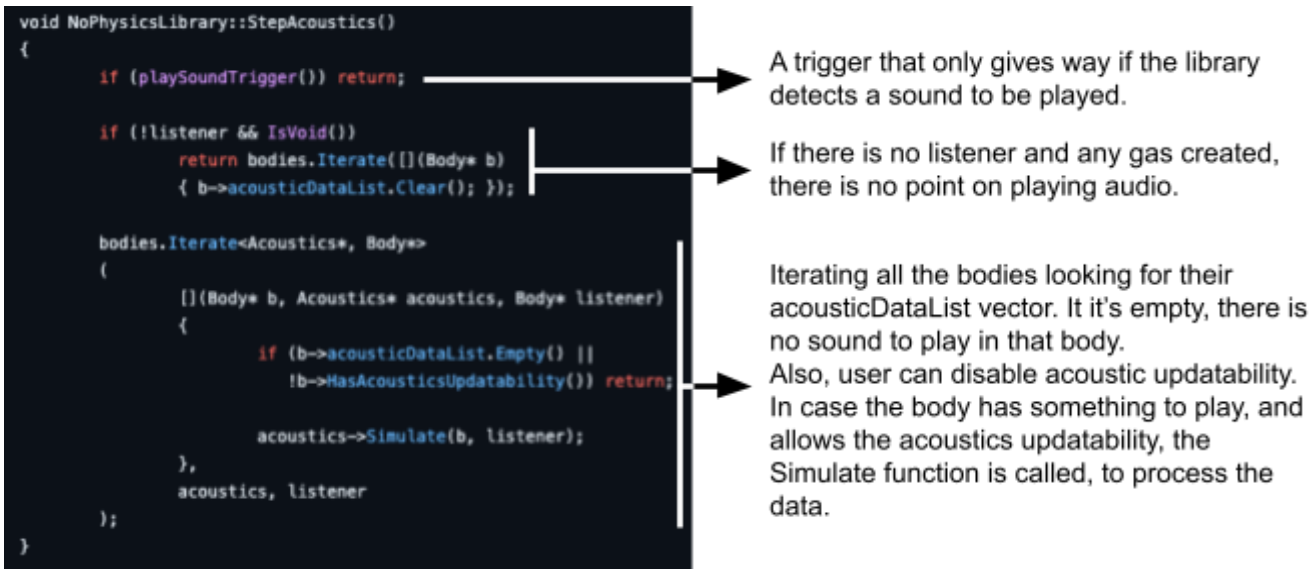
These functions employ a highly complex algorithm to sort through all collisions while also attempting to declip them. The for loop that encircles them is important to note. The "physIterations" variable specifies the number of collisions solved per frame. The higher it is, the more accurate the world's collision solving will be, but the computational time will be longer.

The number of Physics Iterations can be set on the main NoPhysicsLibrary variable's "Config()" function.

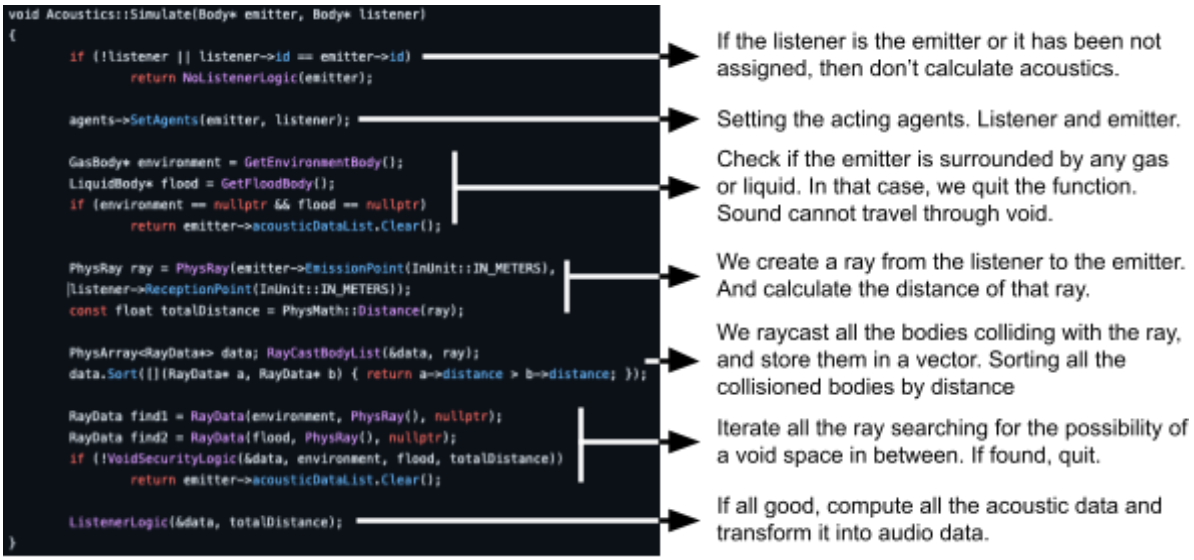
5.4 Behind the Acoustics Logic

The Acoustics class is in charge of converting the world's acoustic data into audio data that the Audio system can interpret and play. Its logic begins with a function called "StepAcoustics()" in the Update() method of the NoPhysicsLibrary variable.

The main structure of the acoustic class is found within that function:



The simulate function contains all of the logic for converting acoustic data to audio data:



We will only look at the ListenerLogic() and NoListenerLogic() functions in this code. However, for the sake of the reader, the others are briefly described below:

GetEnvironmentBody(): It iterates through all of the gas bodies in the scene, looking for the largest one that collides with the emitter. If there is no gas body, the emitter is more likely to be surrounded by a void. The next function will be required to check and ensure it.

GetFloodBody(): This function follows the same logic as the previous one, but instead of iterating the gasses, it iterates the scene's liquid bodies. If neither function returns a body, the sound will not be able to propagate.

RayCastBodyList(): This function employs a complex algorithm to raycast from the emitter to the listener in order to detect all obstacles in the path of these agents. It requires two parameters: a vector reference to RayData (the struct that stores information about each obstacle) and the ray between the emitter and the listener. Check out the source code to learn more about how it works. Note that the complexity of this code is substantially high.

VoidSecurityLogic(): Using a complex iteration algorithm, this method checks over all the obstacles to ensure that there is a path between them without passing through a void section. If the void section is discovered, it returns false and exits. It requests four parameters: a reference to RayData's vector, the environment gas, the flood liquid, and the total distance. Check out the source code to learn more about how it works. Note that the complexity of this code is substantially high.

Before discussing the NoListenerLogic() and ListenerLogic() functions, the AcousticData and SoundData classes are presented below.

> AcousticData: This class is created when the player invokes the "Play()" function from within any body. The Play() function requests an index, which represents the sound to be played, as well as the decibels of that sound. This is the data contained within this class. Each body is equipped with a vector of AcousticData classes.

> SoundData: this class is created by the acoustics Simulate() function to receive all of the AcousticData class's processed data. The following parameters are stored in this class:

- The Sound Index (unsigned int)
- The Panning (float -1 to 1)
- The Volume (float 0 to 1)
- The Delay Time (float)
- The Frequency Cutoff (float)
  The Resonance (float)
- The Pitch (float)

When there is no listener assigned or the emitter is the self listener, the NoListenerLogic() function is called. We can't compute any kind of raycast in either case, so the only information we want to save in the new SoundData variable is the sound index and volume of the body's AcousticData variables.

```
soundDataList->Add(new SoundData(data->index, 0, data->spl / maxSPL, 0, 22000, 2, 1));
```

/\ *vector [2]* of all the playing sounds index pan volume delay freq & res pitch

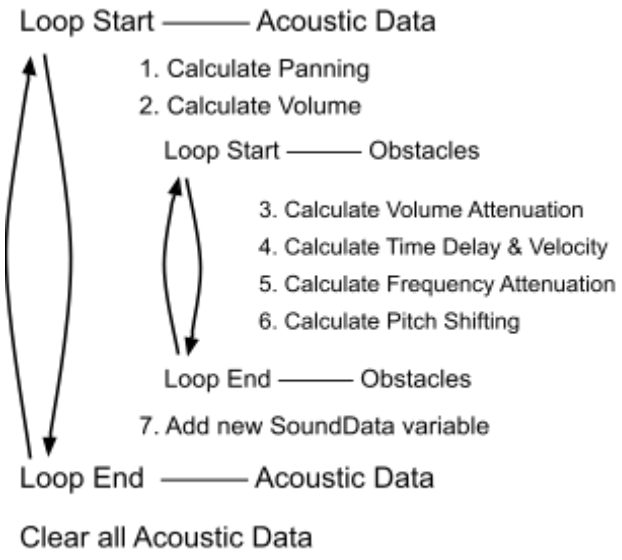
Now it is time to tackle the final big boss of the library, the mighty ListenerLogic() function:

This function, like NoListenerLogic(), iterates through all of the AcousticData variables in the emitter's list.

Then it begins calculating all of the parameters available in this library.

Some of the parameters to calculate are dependent on the obstacles between the emitter and the listener, so in the loop, it is necessary to loop again for each obstacle.

Check this graph to understand its functionality:



Within that loop, the panning is first calculated using the following formula:

*Panning = Direction * (Distance / Panning Range) ^ Panning Factor*

Where:

- Direction: Is an integer whose value can only be 1 or -1. It depends on which side the listener is from the emitter.
- Distance: Is a float that represents the total distance between the listener and the emitter
- Panning Range: The maximum distance which represents the 1 or -1 of the result. It can be configured in the "Config()" function of the main variable.
- Panning Factor: Is a float factor that defines if the pan value is linear or curved. It can be configured in the "Config()" function of the main variable.
- Panning: The final result, which varies from -1 to 1, being -1 full left sound and 1 full right sound.

The volume is then calculated as a function of distance using the classic volume-distance attenuation formula of acoustics theory:

*Final SPL = Sound SPL - (20 * Log(Distance * Attenuation Factor))*

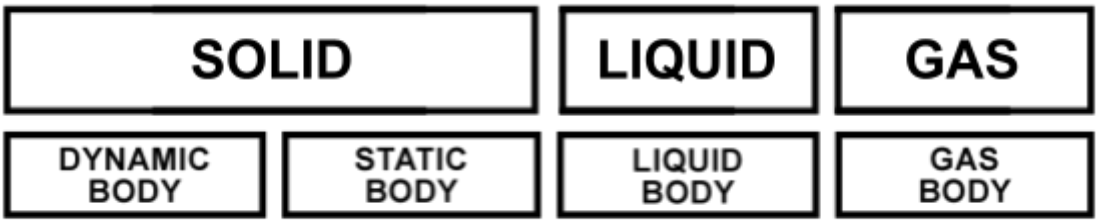
Final Volume = 10^(Final SPL / Max SPL) / Max Volume

Where:

- Sound SPL: Is a float whose value represents the Sound Pressure Level in decibels of the sound being played.
- Distance: Is a float that represents the total distance between the listener and the emitter.
- Attenuation Factor: Is a float factor that allows to exaggerate or minimize the volume attenuation effect.
- Final SPL: The final Sound Pressure Level.
- Max SPL: A const float defining the maximum loudness of a sound. Defined at 120dB.
- Max Volume: A const float defining the maximum volume of a sound. Defined at 10.
- Final Volume: The final volume of the sound, ranged from 0 to 1.

The accuracy of the second formula is very low. In the future, it is planned to remove the Maximum Volume const float, which makes little sense. In addition, the formula for converting the logarithmic scale of the Final SPL (dB) into a linear value ranging from 0 to 1 occasionally produces unexpected results. It should be revisited in the future.

All of the obstacles have been iterated and processed at this point. Each type of obstacle has its own set of formulas because each one describes a different type of body in a specific state of matter. Look at the diagram below:



In the case of Volume Attenuation, we require a separate formula for solid bodies and another for gasses and liquids:

Solids —

*Abspt Att = Abspt Coeff * Distance*

*Trnsm Att = 20 * Log(1 / Absorption Coeff)*

Final Vol Att = 10^((Abspt Att + Trnsm Att) / Max SPL) / Max Volume

Liquids & Gas –

*Final Vol Att = 10^(e^(Abspt Coeff * Distance) / Max SPL) / Max Volume*

Where:

- Distance: Is a float that represents the total distance between the entrance and the exit of the sound in that specific obstacle.
- Abspt Coeff: Is a float that represents the absorption coefficient of a body.
- Abspt Att: Is a float that represents the final Absorption Attenuation in dB.
- Trnsm Att: Is a float that represents the final Transmission Attenuation in dB.
- Max SPL: A const float defining the maximum loudness of a sound. Defined at 120dB.
- Max Volume: A const float defining the maximum volume of a sound. Defined at 10.
- *Final Vol Att*: Is the final volume attenuation, which represents the volume in linear scale to subtract from the current sound volume.

It should be noted that these formulas are a simplification of the actual phenomenon. It is planned to expand them in order to obtain a more realistic approach. There is also the same issue as in the volume calculus depending on distance. It is necessary to improve the way logarithmic data is transformed into lineal data.

Following the calculation of volume attenuation comes the calculation of time delay, for which we have different formulas depending on the type of obstacle:

Solids -> *Time Delay = Distance / √(Young Modulus / Density)*

Liquids -> *Time Delay = Distance / √(Bulk Modulus / Density)*

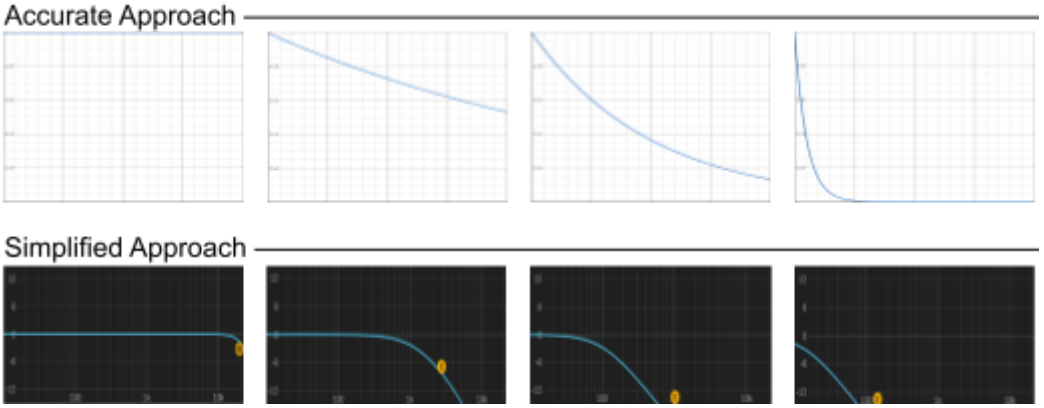
Gasses -> *Time Delay = Distance / √(Heat Ratio * Pressure / Density)*

Where:

- Distance: Is a float that represents the total distance between the entrance and the exit of the sound in that specific obstacle.
- Young Modulus: Is a float value that measures the resistance to elastic deformation of the solids. It is usually measured in GigaPascals.
- Bulk Modulus: Is a float value that measures the resistance to compression of the liquids. It is usually measured in GigaPascals.
- Heat Ratio: Is a float value that measures the heat capacity at constant pressure to heat capacity at constant volume.
- Pressure: Is a float value that measures the pressure which a gas is submerged in. It is measured in Pa.
- Density: Is a float value that represents the density of the obstacle. It is measured in Kg/m^2
- Time Delay: Is the final value, which represents the delay in seconds.

It is worth noting that the formula is essentially the classic "Time = Distance / Velocity." This means that the formula containing only the square root is calculating the transmission velocity in that obstacle. Because we will need the velocity parameter in future calculations, we will keep it in a separate variable for future use.

Then we start calculating the Frequential Attenuation. There are different formulas for various body types. Let's start with the formula for the liquid. The most accurate one has been oversimplified. Because I couldn't get the raw data of the volume per frequency with the library I was using, SoLoud, I couldn't directly apply that formula. I took the approach of applying a low pass filter to the sound in order to match the results of the accurate formula.



The following formula is used to obtain an accurate frequential attenuation curve in liquids:

$$Attenuation = e^{((-2\pi * Freq * Viscosity) / Density)}$$

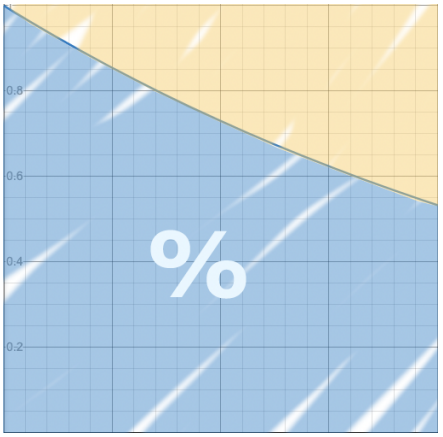
Where:

- *Freq*: Is a float that represents the specific frequency of the sound that we want to attenuate.
- *Viscosity*: Is a float value that measures the resistance to deformation of a liquid. It is usually measured in Pascals / Second.
- Density: Is a float value that represents the density of the obstacle. It is measured in Kg/m^2
- Attenuation: Is the final value, which represents the attenuation in dB of that specific frequency

I needed a simplification because I couldn't access the information about the volume per frequency of my sounds.

The solution I came up with was to roughly calculate the area below the curve to get a ratio from 0 to 1, which I'll be transforming into the cutoff frequency and resonance of the low pass filter.

Finally, the ratio must be converted from logarithmic to linear.



The formula for solids is also different because there is no "viscosity" for solid bodies. In this case, it is calculated as follows:

$$Wavelength = 2 * (Distance / (1 - Absorption\ Coeff))$$

$$Final\ Cutoff\ Ratio = (Velocity / Wavelength) / 22000$$

Where:

- Distance: Is a float that represents the total distance between the entrance and the exit of the sound in that specific obstacle.
- Absorption Coeff: Is a float that represents the absorption coefficient of a body.
- *Wavelength*: Is a float that represents the length of one loop of a sound wave.
- Velocity: A float that represents the velocity of the sound inside the obstacle.
- Final Cutoff Ratio: Is the final value, which represents the frequency at which the lowpass filter must be cutted. It ranges from 0 to 1, but by multiplying by 22000, which is the maximum Hertz a human can listen to, we acquire the cutoff in Hertz.

Let's finish with the pitch shifting formula, which uses 343 m/s as the reference for the sound velocity to determine the current pitch:

*Final Pitch = (Current Pitch * Rel %) + ((Velocity / 343) * %)*

Where:

- Current Pitch: Is a float that represents the current pitch of the volume. It starts by 1, but it can be modified along the different obstacles.
- %: Is a float that represents the relationship between the distance from the entrance and the exit of the obstacle, divided by the total distance, from emitter to listener.
- *Rel %*: Is the inverse of the %, calculated as follows: “Rel % = 1 - %”
- Velocity: A float that represents the velocity of the sound inside the obstacle.
- Final Pitch: Is the final value, which represents the pitch of the emitted sound.

It is important to note that all of the previous features, which were computed in the ListenerLogic() function, can be activated or deactivated by configuring the library. There are several functions to manipulate these features by accessing the "Config()" function, such as:

- PanRange(float): Default value is 10. The maximum distance which represents the 1 or -1 of the result.
- PanFactor(float): Default value is 1. It defines if the pan value is linear or curved.
- GlobalVolumeAttenuationFactor(float): The default value is 1. It is used to exaggerate or reduce the volume attenuation spatialization formula.
- SoundDelay(bool): Default value is true. It is used to activate the time delay feature of the acoustics module.
- SoundOclusion(bool): Default value is true. It is used to activate the sound occlusion feature of the acoustics module.
- FrequentialAttenuation(bool): Default value is true. It is used to activate the frequential attenuation feature of the acoustics module.
- PitchVariationFactor(float, bool, bool): Default value is 1, true, true. It is used to reduce or accentuate the pitch shift of the sounds. Also, you can deactivate specifically upwards or downwards pitch.

Now that we have all of the information computed and ready, we can create a new variable of type SoundData and store it in the vector of all the sound data's to play at that frame:

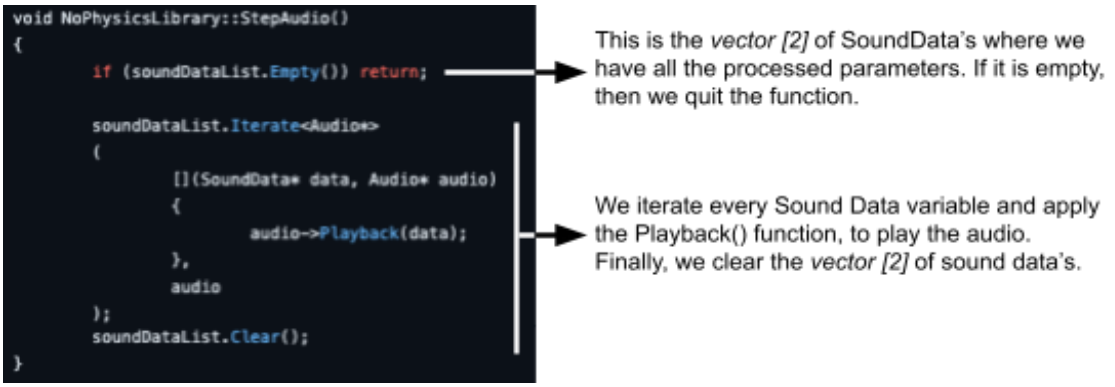
```
soundDataList->Add(new SoundData(aData->index, pan, volume, timeDelay,
FINAL_FREQ(cutoff), FINAL_RES(res), FINAL_PITCH(pitch, *pitchVariationFactor)));
```

cutoff * 22000 2 * (res + (0.2f * res)) / 1.2 (factor * pitch) + (1 - factor)

5.5 Behind the Audio Logic

The Audio class is responsible for configuring the sound with the various parameters in the SoundData variables in order to play the desired audio. Its logic begins with a function called "StepAudio()" in the Update() method of the NoPhysicsLibrary variable.

The main structure of the audio class is found within that function:

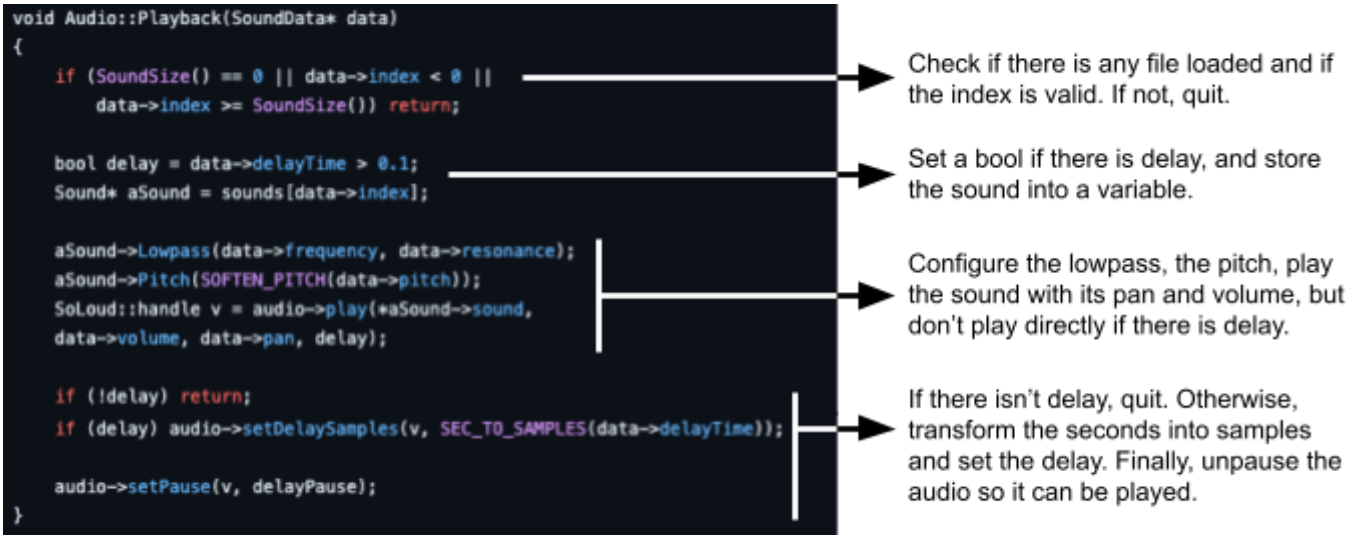


As previously stated, SoLoud is used as an audio library to play the sounds of this API. When the user calls the "Load()" function from the main class, the audio class calls the Load() method.

If the file already exists, the audio class creates a new struct called Sound and stores it in a vector of Sound. This struct contains the following data:

- SoLoud::Wav: This is the actual audio file decompressed.
- SoLoud::BiquadResonantFilter: This is the reference to the lowpass filter variable of the sound.
- PitchShiftFilte: It is a custom-made filter that inherits from the SoLoud::FFTFilter. It allows you to pitch up/down the sound.

This struct also includes some functions for configuring the SoundData variables' data. Finally, consider the Playback() function:



5.6 Project Changes

During the development of the project, some issues were encountered. This has led to some changes with respect to the initial structure of the NPL library and the planification of the project.

5.6.1 Change 1: MiniAudio

MiniAudio is a low-level audio library written in C++. It gives the developer the ability to load, play, modularize, and record. Initially, NPL used this library to manage the loading and playback of audio files. The documentation mentioned several implemented filters for modifying audio data, such as flangers, delays, and reverbs... And it was the primary reason I chose MiniAudio. Two issues prompted the change in planning:

When I was working on the sound delay feature with the sound propagation velocity, I needed a way to "delay" the audio for a specific number of seconds. I attempted to build a "delay node" that repeats the input signal based on a roll off factor and the time between repetitions.

My plan was to have the delay repeat once but with a specific amount of time between the original and the repetition. Then, completely remove the "dry signal" (original sound) and allow the wet signal (repetition) sound. This way, the library would only play the delayed repetition .

Once everything was in place, I noticed that MiniAudio's "Dry & Wet" functionality was incorrect. Its behavior was not the expected one, as previously explained. My only option was to contact the MiniAudio developer and inform him of the issue. He noticed this and deleted MiniAudio's Dry & Wet functionality.

The issue conversation is completely public and it can be checked on MiniAudio’s github page, attached in this document in the links section.

Without this option, I came up with another solution. I was able to program a timer to delay the sounds as needed. I was still with MiniAudio at the time.

Later, I was working on developing the reverb feature, and when I tested it in MiniAudio, it wasn't working as expected... So I decided to completely revamp the library.

Now I am using SoLoud as an audio engine. It has an audio delay feature built in a “Play” method. It also includes a reverb, which behaves strangely.

5.6.2 Change 2: Collision Types

After I finished developing the physics engine with rectangles, I began modifying it to accept circles. Given the structure of my library, the only easy way to implement that feature was to use templates, as any other option would necessitate a complete overhaul of the library workflow.

After hours of intense thinking and programming, I concluded that I didn't know enough to do it (I won't say impossible).

My main issue is that C++ does not permit the return of different variable types within the same function.

The template approach is extremely powerful, but it also requires a significant rethinking of the entire API structure. I eventually decided to drop the circle collider feature.

Apart from that, for each new shape I want to include, it supposes an exponential growth on the “matrix” of collisions between types.

It is not only to collide, for example, circle with circle, it is also circle with all the other shapes.

Knowing this information, it was a matter of time to decide whether I continue the work with or without other collision shapes.

6. Project validation

To validate the overall work during this months, you can access the main github page where the repository has been building up.

Also, I created a video showcasing all the features that NoPhysicsLibrary can bring to the users that want to use it. This video’s purpose is to validate the project and demonstrate the library usage.

All the relevant links can be found in the **Links** section.

7. Conclusions

The physics module and the audio system are two distinct entities in the videogame industry. As a result of this decision, developers and sound designers are forced to hard-code or, worse, discard the possibility to implement acoustics effects to their world. It is essential to have coherent audio feedback within the imaginary world that a videogame transports us to. Sonorization and music in videogames account for more than 70% of the player's immersion.

The solution is straightforward: "create a module that can connect them." However, as simple as it may appear, it is difficult to put into action. Despite the difficulties, I am proud to say that NoPhysicsLibrary is the world's first C++-based approach that combines physics and audio, by including an acoustics module in between. Its main purpose is to obtain data from the physics engine and convert it into valid data for the audio system.

Specifically, it features several effects such as spatialization, volume attenuation, sound occlusion, pitch shifting, frequential attenuation and travel delay. All these features are completely configurable, so the developer has full freedom to let their creativity fly.

The reader is encouraged to check out the final result of the library on its github repository. Two demonstration videos are also presented as part of this work, showing a simple world featuring all the acoustic effects covered by NoPhysicsLibrary.

7.1 Future Work

Even though the main structure and functionality of NoPhysicsLibrary is stable and functional, there is still a lot of future work to do with it. A lot of modules to be upgraded, new features to implement, a lot of debug testing and bug solving.

Below is a list with all the future implementations, upgrades and remodelings planned for this library, split in different sections.

> User-Related Future Work:

1. Development of a full documentation of the library.
2. Creation of a stable issue solving system on github.
3. Creation of extras, examples and templates for the community.
4. Expanding the library support platforms.

> Programming-Related Future Work:

5. Upgrading the physics declipping system (Collision Solver)
6. Implementing more collision shapes, such as circles or custom shapes.
7. Rework and upgrade of acoustic formulas to be more accurate over the resulting sound effects.
8. Bug-fixing the specific effects whose functionality is currently strange, such as Sound Occlusion, Pitch and volume attenuation.
9. Implementing the Doppler effect.
10. Implementing the Reverb effect. This feature requires a deep remodeling of the library to implement multiple-raycasting and spatial recognition. Given the expected workload and complexity, this could perfectly be the topic of another entire TFG project by itself.
11. Replace the current audio module (SoLoud) by a self-developed, in-house library that must be able to handle all the features that I require for the NoPhysicsLibrary.
12. Extend the library features to work with long and loopable audios.
13. Build a music and ambience playback system around the audio module.

> Testing-Related Future Work:

14. Remodeling and testing of the user experience of the library.
15. Adding more presets and materials.
16. Extensive debugging sessions to fix minor, non-critical bugs of the library.

8. Webgraphy

1) *Verlet Integration, Arcane Algorithm Archive*. Accessed 4 March 2023

2) Manager, C. (2022, May 30). *Sound Velocity Table*. Class Instrumentation Ltd. Accessed 8 March 2023

3) *Figure 1. Higher frequencies have higher attenuation on penetrating. . .* (n.d.). ResearchGate. Accessed 23 March 2023

4) ENGINEERING.com. (n.d.). *How is attenuation related to frequency?* Copyright 2000-2013 ENGINEERING.com. Accessed 12 April 2023

5) The Cherno. (2017, October 18). *Making and Working with Libraries in C++ (Multiple Projects in Visual Studio)* [Video]. YouTube. Accessed 19 April 2023

6) *Ultrasonic Testing*. (2016, January 1) Accessed 20 April 2023

7) Admin. (2022, February 11). *Sound Attenuation Calculator - Inverse Square Law | WKC Group*. WKC Group. Accessed 20 April 2023

8) *How to find nearest point on line of rectangle from anywhere?* (n.d.). Mathematics Stack Exchange. Accessed 3 May 2023

9) Freya Holmér. (2020, November 9). *Vectors & Dot Product • Math for Game Devs [Part 1]* [Video]. YouTube. Accessed 3 May 2023

10) Elert, G. (2021). *Aerodynamic Drag*. The Physics Hypertextbook. Accessed 12 May 2023

11) Collegedunia. (2022). Speed of Sound Formula: Solids, Liquids & Gases. *Collegedunia*. Accessed 17 May 2023

12) *Kinsler, L. E. (1982). Fundamentals of Acoustics. Wiley Publications. 480 pag.* Accessed 17 May 2023

13) Rossing, T. D., Fletcher, N. H., & Tubis, A. (2004). Principles of Vibration and Sound, 2nd edition. The Journal of the Acoustical Society of America, 116(5), 2708 pag. Accessed 18 May 2023