End-of-Degree Project

Bachelor's Degree in Informatics Engineering

2014/2015

X-GSD: Cross-Platform Game Skeleton (Data-Driven)

A basic data-driven C++ game engine built upon SFML

Official name (Spanish): Motor de videojuegos en C++ sobre SFML

Author:
Andrés Ruiz Bernabeu

Supervisor:
Jordi Joan Linares Pellicer

June, 2015

Abstract

The purpose of this end-of-degree project report is to explain some general concepts about video game programming, game engines and their utility, and how X-GSD is built around some of these ideas, including: types of game loop and simulation, resource management, scene graph, entity-component system, events and basic physics.

Nowadays, game engines have become very popular and a fundamental part for many game developers. Despite their benefits, this tendency comes with costs: As the most popular engines are proprietary, game developers become reliant on closed third-party software, which could lead —among others— to a deadlock of their projects in the worst case. This and more against-reasons regarding popularity of proprietary and big game engines are discussed along this work.

X-GSD stands for Cross-platform Game Skeleton (Data-Driven), a basic data-driven C++ game engine built upon SFML[1]. It is a totally free and open basic game engine which aims to solve some of the problems of proprietary game engines, or being a starting point for custom solutions. It can also serve as an academic example of how to start building the core of a game, or a game engine.

-

Keywords: Video games, game engine, programming, SFML, X-GSD.

[1]Simple and Fast Media Library

Contents

0 Abbreviations and Symbols

1. *X-GSD*: Cross-Platform Game Skeleton (Data-Driven), the name of the game engine developed along this work.

2. *OOP*: Object Oriented Programming. A programming paradigm based in the encapsulation of data and behaviour on logic structures called objects.

3. *C++*: Programming language invented by Bjarne Stroustrup on 1983. It is one of the most popular[1] programming languages, including games development.

4. *RAII*: Resource Acquisition Is Initialization, an idiom which encourages to manage all memory on constructors and destructors.

5. *SFML*: Simple and Fast Media Library. A multi-purpose modular cross-platform framework. It is the abstraction layer used by X-GSD to achieve cross-platform targeting.

6. *SDL*: Simple DirectMedia Layer. A very popular abstraction layer similar to SFML, but older and less object-oriented.

7. *API*: Application Programming Interface. This are the public operations and data which can be used to program a software application without knowing about the underlying implementation.

8. *SDK*: Software Development Kit. Software suite with a range of tools for developing on a particular platform or system.

9. *IDE*: Integrated Development Environment. Software that integrates different development tools (even a whole SDK) into a single application.

10. *FPS*: Frames Per Seconds, which refers to the rendering frequency of the portion of the screen where the game is shown.

11. *JSON*: JavaScript Object Notation. A widely-supported lightweight data-interchange format. [19]

12. *YAGNI*: You Ain't Gonna Need It. An idiom which warns to avoid anticipating the future by implementing features that *might* be needed, keeping the focus on the features that are actually needed at the present.

[1] LangPop. Programming language popularity. `http://langpop.com/`.

13. *WYSIWYG*:What You See Is What You Get, which means that the resulting product will look the same as it looked in the editor.

14. *AAA*: Common term to refer to video games mega-productions.

1 Introduction

Game development can be still considered as a very new area of knowledge and an emerging industry. Moreover, university studies related to video games and game development are still too young —when the author of this work began its bachelor's degree, university studies related to video games did not even exist in Spain— and there are no clear standards. The video games industry has been utterly hermetic for many years, especially the most technical parts as programming. Therefore, only those who were into it —working for some big company— had proven knowledge on which programming decisions are better for which scenarios, and companies would not allow their workers to reveal any information, making almost impossible to generate a collective and public knowledge base with regards to game development. This can be seen as one of the causes of game developers deciding to buy a proprietary game engine license, as these game engines handle big part of the game core programming itself and frees the programmer of this task to focus on developing games.

These proprietary game engines are very robust and helpful in general, plenty of tools and editors which make game development easier and faster. But most of these developers forget the price they pay for it —appart from the license, which generally does not fit everyone's budget—, which is becoming dependant on a closed and proprietary third-party technology. This may lead to face some problems such as:

- Targeting to a platform which is not compatible with this particular game engine.
- The need of low-level details control (for optimization or any other matters), which this particular game engine does not allow.
- A new version of the game engine is needed to target to a new platform, but this new version introduce drastic changes and deprecations. The project might stay on older platforms and engine's version, or need to invest time in deep refactoring.
- A distribution platform the team is using requires some mandatory updates (e.g. Apple required 64 bit compatible binaries for iOS apps[2]), but the game engine has not been updated yet and therefore the game or its updates cannot be published. May cause temporal deadlock of the project (or total if that update never happens).
- The proprietary company closes and/or the game engine gets discontinued. May cause total deadlock of the project.
- The programmer could become too much dependant to that particular engine workflow and may stop learning other technologies —a special word of caution on this: a job change can occur to a company where that particular engine is not used—.

[2] Apple Inc. 64-bit and ios 8 requirements for app updates. `https://developer.apple.com/news/?id=12172014b`.

In contrast, if using an open game engine and some of these problems would appear, the whole community of that game engine could take action to solve the issues as anyone can see, modify and compile the engine's code. This is also true if the programmer is using its own technology —although there may not be any community and the problem has to be solved by the programmer itself—. The point is the project does not totally depend on a closed proprietary third-party software which could —in the worst case— deadlock it.

This report of end-of-degree project shows how one could build its own technology or game engine, either by serving as an academic example or by being a possible basic solution to the problems stated before.

1.1 Previous research

Fortunately, with the arrival and growth of the Internet —as many other aspects of people's lives— this hermetic tendency of the games industry has been changing: free and open-source game engines and tools appeared on the web for anyone to look inside or freely download and use; on-line collaborative systems —see GitHub[3]— have revolutionised the way free software and open source is understood as anyone can very easily look how a particular part of a specific application or project has been made, and even try to enhance it and collaborate with the author of that project; There are also personal websites, blogs, social networks and wikis which are filled with information from people who want to share their knowledge about this topic all around the world. The combination of all these phenomenons with digital distribution have resulted in the appearance of the so-called "Indie developers" —small teams of developers that can now afford to get into the industry and create very innovative and/or artistic game experiences as opposed to big companies' AAA[2] games—. Maybe because of the origin and nature of these Indie developers, or because they do not have the pressure of a big company that forbids sharing information, they are generally very transparent and give very valuable information to the community —sales data, programming advices, reports detailing all the development process... Gamasutra[4] is a mine of them—. It is truly encouraging to see how much the games development scenario has changed, becoming more accessible. However, despite the rise of this knowledge-for-everyone epoch the Internet has brought, the fact is that there are no clear standards for game programming yet.

As it was stated above, the games development scenario has been changing and information sharing is more common. There are free and open game engines and tools to look into their code and some valuable documentation sources available for anyone. Therefore, it is a good starting point to look at what others have done in this field.

[2] Common term to refer to millionaire mega-productions.

[3] GitHub Inc. Github · build software better, together. `https://github.com/`, February 2008.

[4] UBM TechWeb. Gamasutra - the art & business of making games. `http://www.gamasutra.com/`, 1997.

1.1.1 Bibliographic review

The most valuable information sources for the development of this work are referenced in this section with brief descriptions and the reasons why they have been taken into account.

- SFML Game Development book [18]

  This introductory book to SFML is from where most initial key ideas have been taken and the main source of inspiration of this project. It is an almost perfect starting point for building a game from scratch —provided the reader has enough C++ knowledge—.

  The book covers the concepts of game loop, basic vector algebra, resource management, error handling, entities, scenes, input handling and many other topics which are out of the scope of this work, such as on-line multiplayer. It also recommends some best practises using C++11 new features, such as smart pointers in combination of RAII[3].

- A Tour of C++ (C++ in Depth Series) book [36]

  As the title suggests, the book does a review of C++ with special attention to the new features of the C++11 standard. Recommended to programmers who need to refresh their knowledge of C++ or are new to it, but have knowledge of OOP[4].

- Gaffer on Games website [11]

  This website features many technical interesting articles regarding game programming. The first two of them, *Integration basics*[12] and *Fix your timestep!*[10] have been specially useful to implement the core of X-GSD: The game loop and how objects advance in the simulation through integration.

- OpTank development blog [33]

  A blog with good game development design articles which were useful to implement the Entity-Component system of X-GSD. It also encourages good habits and principles like YAGNI[5]

- Game Programming Patterns book/website [25]

  This book has been a recent discovery and therefore most of the brilliant ideas and concepts of this book were not included on the project. It is a gem of game programming that any game developer should take a look at. The web version[26] is free. But please, if you find it really helpful, consider buying the book or eBook.

[3]Resource Acquisition Is Initialization, an idiom which encourages to manage all memory on constructors and destructors.

[4]Object Oriented Programming. A programming paradigm based in the encapsulation of data and behaviour on logic structures called objects.

[5]You Ain't Gonna Need It. An idiom which warns to avoid anticipating the future by implementing features that *might* be needed, keeping the focus on the features that are actually needed at the present.

- SFML website/forum [16]

  SFML website has a *Learn* section plenty of specific tutorials and the API documentation of different versions of SFML. The forum is a good source of information too as it has a good and active community and the developers of SFML participate very often to answer to doubts and read suggestions.

- SFML GitHub wiki [17]

  Another good source of official information and social collaboration with tutorials, FAQs and resources made by the community with the supervision of SFML authors.

1.1.2 State of the art

The term *game engine* may have many definitions depending of the point of view, but they all coincide in that game engines are software frameworks or tools which hide low-level programming and hardware details so that game developers can operate on a higher level of abstraction, resulting in better productivity.

The game engines and frameworks listed below are some of the most popular and accessible engines at the time of writing:

- Unity

  "Unity is a game development ecosystem: a powerful rendering engine fully integrated with a complete set of intuitive tools and rapid workflows to create interactive 3D and 2D content; easy multiplatform publishing; thousands of quality, ready-made assets in the Asset Store and a knowledge-sharing community.

  For independent developers and studios, Unity's democratizing ecosystem smashes the time and cost barriers to creating uniquely beautiful games. They are using Unity to build a livelihood doing what they love: creating games that hook and delight players on any platform." [38]

  Unity has become the most popular proprietary 3D game engine of today among indie developers. The main reasons would be the easiness of use thanks to a powerful WYSIWYG[6] and drag-and-drop editor, the availability of a free version, almost automatic cross-platform building, the Asset Store —a marketplace of any type of resource compatible with Unity, scripts included— and an affordable royalty-free license while other similar full-featured engines may cost more than a hundred times its price or have a high percentage of royalties —or at least that was when it came out. The market has been changing so much that Unity have become one of the most expensive engines right now—. New full 2D support has been added recently.

- Unreal Engine

  "Unreal Engine 4 is a complete suite of game development tools made by game developers, for game developers. From 2D mobile games to console

[6] What You See Is What You Get, which means that the resulting product will look the same as it looked in the editor.

blockbusters, Unreal Engine 4 gives you everything you need to start, ship, grow and stand out from the crowd." [9]

Epic Games Inc. is the company behind one of the most known proprietary 3D game engines of all-time, aging 17 years old at the time of writing (the first version was released in 1998). Countless games have been build upon this engine, most of them being big productions, retail games and many AAA's [39]. Unreal Engine features cross-platform targeting with high-end quality graphics and technologies. Written in C++, it also features *Blueprints*, a graphic programming and debugging tool for faster development iterations, in addition to a proprietary solution to C++ code hot-swapping.

Previous versions of the engine required a very expensive license —around a six or seven figure range of dollars, plus a percentage of royalties depending on the type of license— only affordable by big companies. Then, Epic Games adapted their business plan to target indies and general public too, giving access to the engine, tools and source code by paying a cheap monthly fee (USD 19) plus a little percentage of the revenue of a UE4-derived project or game in royalties (5%). After that, they removed the monthly fee and distributed the complete engine for free, but maintaining the 5% of royalties if a project reaches USD 3,000 of revenue.

- CryEngine

  "CRYENGINE is the leading all-in-one game development solution with truly scalable computation and benchmark graphics technologies for console, PC and mobile devices. By choosing CRYENGINE, developers can be assured that they're ready for the future of gaming, and empowered to create standout experiences for PlayStation®4, Xbox one™, Wii U™, Windows, Linux, iOS and Android."

  "CRYENGINE is the only game engine that provides multi-award winning graphics, state-of-the-art lighting, realistic physics, intuitive visual scripting, high fidelity audio, designer friendly AI, an efficient 3D stereoscopic solution across all platforms and much more - straight out of the box. DirectX 11 (DX 11) support significantly enhances the capabilities of CRYENGINE's powerful real-time renderer, allowing for some of the best visuals ever seen." [7]

  CryEngine is another proprietary high-end 3D graphics engine with cross-platform targeting. It is not as popular as Unity or Unreal Engine among small studios and indie developers, but several AAA games have used it proving its strength. Not only Crytek's engine pricing has been similarly prohibitive to Unreal Engine, but also competes with it in their new monthly pricing plans (USD 9 with no royalties, but some limitations) to reach indie and general public.

- ShiVa

"Cross Platform Development made easy. ShiVa3D is a 3D game and application development suite that comes in an easy to use, yet very powerful WYSIWYG (what you see is what you get) editor. Consider ShiVa the glue between your creative ideas, your art, your code, and the hardware you are targeting." [35]

This proprietary 3D graphics cross-platform engine is one of the most target-platform compatible —over 20 different platforms—. Scripts can be written in LUA or C++ depending on wether developers prefer easy and fast scripting or exploiting full potential of the hardware. ShiVa Technologies SAS provides a free version of their engine, and a much cheaper license pricing than Unity. However, updates and community seem inactive compared to the previously mentioned engines.

- GameMaker: Studio

  "GameMaker: Studio caters to entry-level novices and seasoned game development professionals equally, allowing them to create cross-platform games in record time and at a fraction of the cost of conventional tools!

  In addition to making game development 80 percent faster than coding for native languages, developers can create fully functional prototypes in just a few hours, and a full game in just a matter of weeks." [40]

  GameMaker: Studio is a cross-platform proprietary 2D game engine by YoYo Games —although the company has been bought by PlayTech recently— with great popularity among developers and many successful games have been developed with it. The engine features an easy to use graphical environment with many tools, a custom scripting language —GML, which stands for Game Maker Language— and a free limited version to try and start learning. License pricing vary depending on the exporting modules needed, but it can be considered as affordable.

- Cocos2D-X

  "Cocos2d-x is an open source game framework written in C++, with a thin platform dependent layer. It can be used to build games, apps and other cross platform GUI based interactive programs. The Cocos2d-x renderer is optimized for 2D graphics with OpenGL. It supports Skeletal Animation, Sprite Sheet Animation, Coordinate systems, Effects, multi-resolution devices, Textures, Transitions, TileMaps and Particles. It adopts a RenderQueue design. Performance: supports auto-batching, auto-culling and caching transform, games are running 1x 20x faster. C++ 11 features: take advantage of the awesome Lambda functions, override and final, template container, auto, threads, smart pointers and move semantics. " [6]

  Cocos2D-X started its development in 2010 as a C++ cross-platform port of the popular Cocos2D-iPhone game engine —which in turn is another port

of many from the original Cocos engine written in Python—. It is totally free, open-source and industry-proven. In 2013, Chukong Technologies acquired the engine leaving its original MIT license untouched, and invested in its development. This resulted in a great development boost, which added new features such as 3D —although the engine still has '2D' in its name—, enhanced the API and performance, and good documentation in English started being available. They also maintain a LUA binding so that one can decide to use C++ or LUA for scripting, a JavaScript/HTML5 port named Cocos2D-JS and some development tools such as Cocos IDE and Cocos Studio.

- Construct 2

  "Construct 2 is a powerful ground breaking HTML5 game creator designed specifically for 2D games. It allows anyone to build games — no coding required!" [34]

  Construct 2 is a proprietary HTML5-based[7] game engine for building 2D games with their own IDE and tools with special focus on the *"no coding required"* philosophy. Because of being HTML5-based, games built with this engine may run on web browsers or as an application —which in fact would be a web view which runs embedded web files instead of obtaining them from a server, simulating a native application—, and may require extra power and resources in exchange of the flexibility and portability it offers. Its license pricing is affordable for the quality and quantity of tools and features the engine provides.

Although the next game engines are not very popular yet, the author of this work believed that they deserved a mention because of their potential, and for being free and open-source:

- Blender [4]: Although this full-featured 3D suite has its popularity, it is not widely known that it includes an integrated game engine.
- Godot [29]: This engine released a robust 1.0 version by the end of 2014, featuring a complete Unity-like graphical environment to build cross-platform 2D and 3D games.
- Phaser [30]: A simple and easy-to-use HTML5 games framework for making 2D games in JavaScript or TypeScript.

1.2 Objectives

The main goal of the project is to build a free and open-source cross-platform game engine from scratch with basic features in order to serve as an academic example or as a starting point for custom solutions.

[7] Refers to a combination of web standard technologies, mainly HTML5 and JavaScript.

The objective is to show how games developers could build their own game engine from free technologies with the requirements they need, aiming for dynamic and data-driven architectures for flexibility. In addition, this may help the programmer to get autonomy and self-confidence to develop own solutions instead of constantly relying on closed third-party software and its associated problems.

To accomplish that objective, X-GSD will be designed and built with C++ and SFML, explaining every aspect of the engine and the reasons behind each decision. Finally, basic example games will be developed in order to show the engine capabilities.

2 Methodology

This work's title —X-GSD: Cross-platform Game Skeleton (Data-Driven), a basic data-driven C++ game engine built upon SFML— may sound weird and definitely does not look branded, but it encloses the main points of the engine in one sentence. This section will introduce and explain the motivation behind each of the features the title refers to, making emphasis on the used technologies to achieve it.

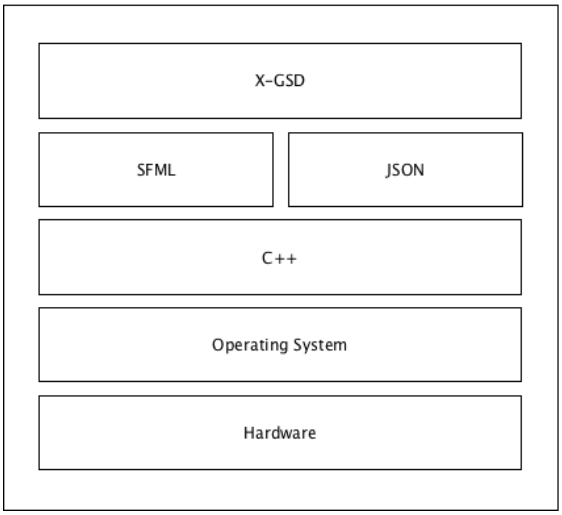

Figure 1: Stack of abstraction levels and technologies.

2.1 Cross-platform targeting

The X part of X-GSD stands for "Cross-platform", which in general terms is the ability to code once and produce a program that runs on several platforms. That is one of the most valuable features a game engine often features, as it can reduce the development time drastically. Thanks to the combination of C++ and SFML, this feature can be achieved at code level: the code is written once, but as C++ is a compiled programming language each target platform need its own executable. Other (interpreted[8]) programming languages such as Java achieve cross-platforming of applications at run level —the same executable runs on every Java-compatible platform—, which is possible thanks to the JVM (Java Virtual Machine). That virtual machine is responsible to translate Java bytecode —the intermediate code generated as a result of compiling Java code— to the specific hardware where it is running on.

It is obvious that the process involves an important overhead, and among others that is the main reason because of interpreted languages are not usually recommended for developing games where performance is critical. However, a common

[8]An interpreted programming language, in general, does not need the source code to be compiled, and the translation from high-level code to machine code is done at run time. However, there are some programming languages such as Java that do both compiling and interpreting: The source code is compiled to an intermediate bytecode, and then a *virtual machine* translates it to machine code at run time.

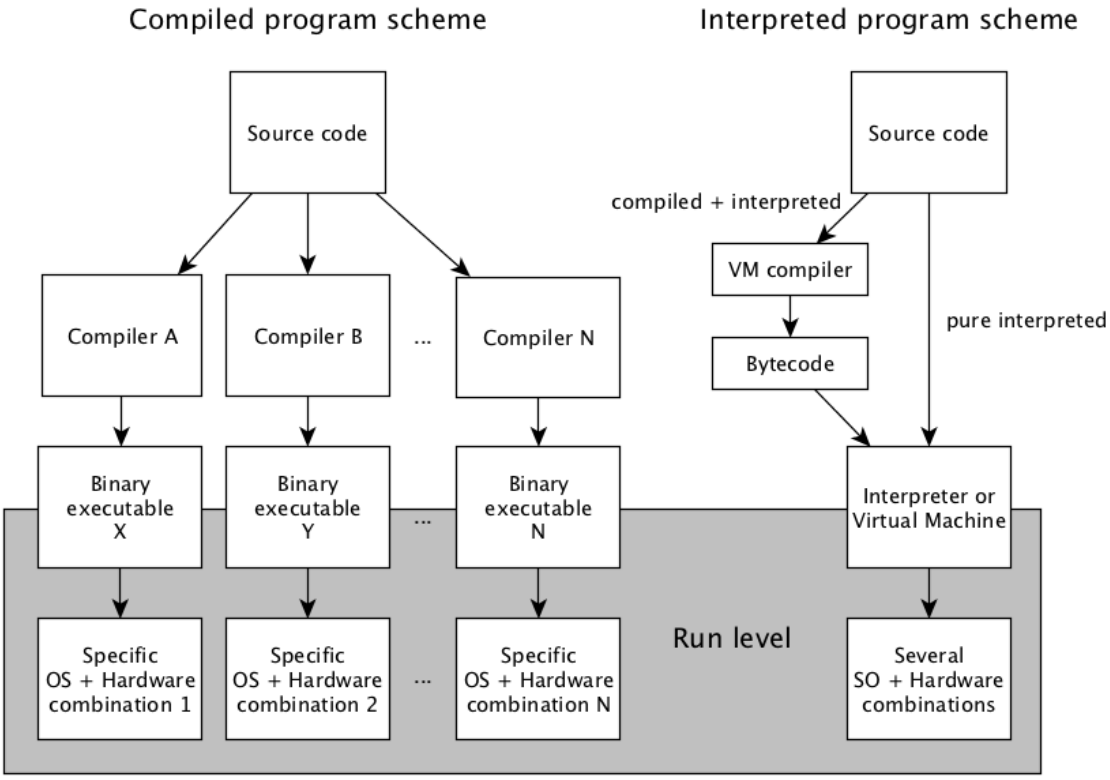

Figure 2: Comparison between compiled and interpreted code phases.

approach is to build the core and the most critical parts of the game —that usually is the engine— in a compiled programming language as C++, and let the user of the engine to "script" the game in a (partially or totally) interpreted language as C#, Java, JavaScript or LUA because —among other qualities such as automated memory management— these type of programming languages usually permit code *hot-swapping*[9] which translates in more flexibility, faster development iterations and better integration with IDEs or editor tools. However, building such systems can be complex and therefore —for simplicity— X-GSD is entirely built with C++.

The problem is that each target platform has its own API on some particular programming language. This translates in different ways of handling input devices, rendering, application and window behaviour, threading facilities, file systems, etc. And there is where SFML comes in: SFML provides a generic API which hides the platform-specific part to the programmer. Therefore, loading a texture, for example, becomes as easy as invoking a SFML procedure which does all the platform-specific work: look for the texture file on a particular file system, communicate with the graphics card and load the texture in its memory. And that line of code would produce the same result on Windows, Mac OS X, GNU/Linux, iOS or Android —the main platforms that SFML can target to—. The main reason to choose SFML instead of other well known libraries with the same purpose, such as the industry proven SDL[10], is that SFML has a very clear, organised and modular API with C++ OOP in mind, as opposed to SDL which exposes a C in-

[9]In software, hot-swapping refers to the ability to modify the code while running, where code changes materialize without the need of re-compiling and relaunching the program.

[10]Simple DirectMedia Layer

terface. Moreover, SFML has a very active development and community, whereas SDL looks stalled.

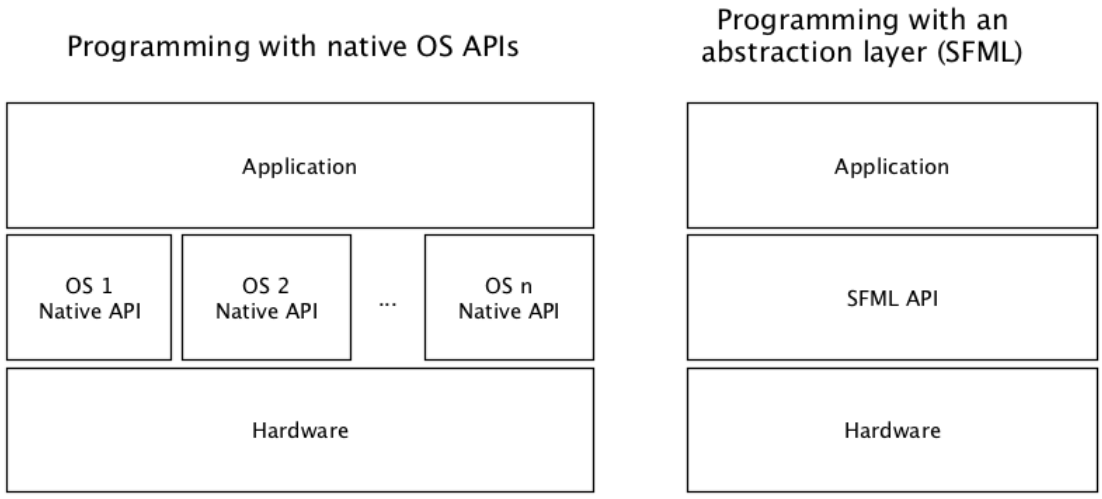

Figure 3: Comparison between programming with native APIs and with an abstraction layer API.

2.2 Data-driven engine

The "Data-Driven" part of X-GSD referrs to the ability of the code to produce different results depending on some input data. In an extreme case, the entire program flow would depend on data. For instance, a music player software produces different sounds as output —the music— depending on a particular input data —that song's file and its meta-data—, while the code of the music player remains unaltered. A more data-driven music player would accept, for example, a play list file that the music player would follow song per song.

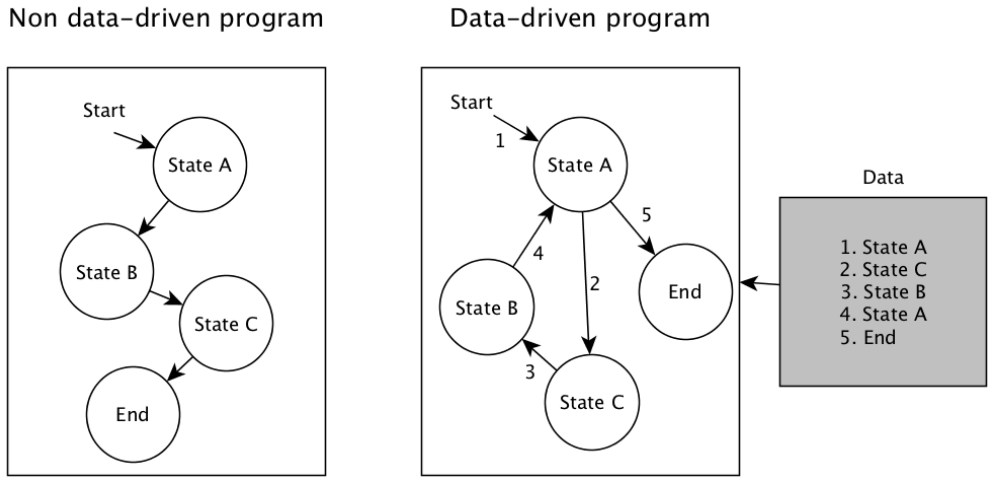

Figure 4: Comparison between non data-driven and data-driven programs.

In the game engines field, data-driven behaviour is an increasing demand by developers as it provides faster iterations for tuning particular parts of a game in

development —generally, there is no need to recompile after some changes—. This can be achieved by storing and loading the needed values from configuration files, which in turn provides and encourages decoupling —the game may be organised in some logic units the engine provides—. This is totally opposed to *hardcoding* values in code. Of course, this more dynamic behaviour involve an overhead, although this trade-off is reasonable in most cases.

With regard to X-GSD, this feature is achieved by a structure of *Scenes*, *Entities* and *Components* which are stored in and loaded from a JSON[11] file per scene and a general configuration file of the game.

Scenes are a set of a name, a *scene graph node*, some managers (resources, physics and controllers) and other properties. They can be considered as a container of entities and their resources. When a scene is loaded from a JSON file, every entity described on it is created and attached to the scene, and every specified resource is loaded into memory; after a basic fade-out/fade-in transition (if enabled), the previous scene is deleted from memory and the new scene begins.

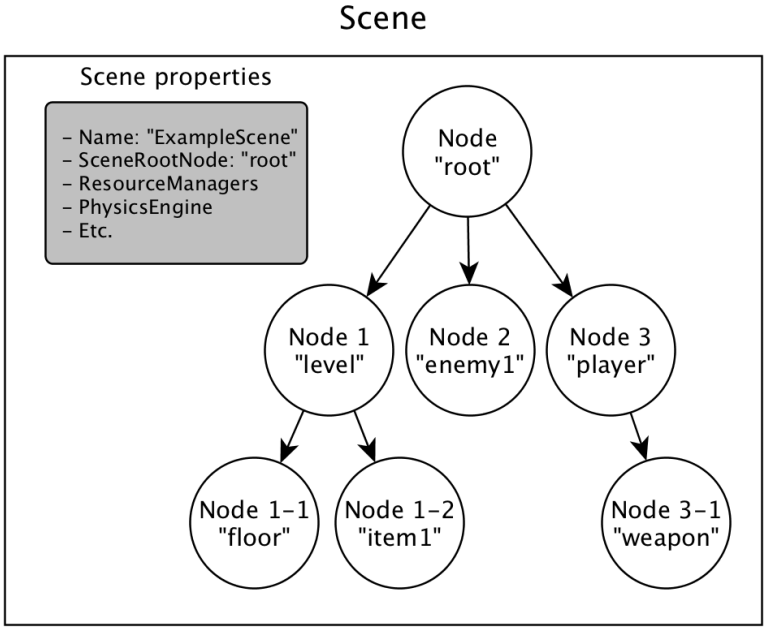

Figure 5: Representation of a scene and the scene graph.

Entities can be seen as the fundamental units of the scene. They are a combination of a name, a scene graph node —which in turn stores a collection of zero or more scene graph nodes or Entities as children and spatial information such as position, scale and rotation— and a collection of zero or more components.

Components give particular properties and behaviour to entities, so they can have visual representation —a sprite[12]— by attaching a ComponentSprite to it; or physics-related components —ComponentCollider and ComponentRigidBody— and even custom components that the user of X-GSD can write to add custom behaviour to the entity, known as controllers —the "scripting" part of the engine—.

[11] JavaScript Object Notation. It is a lightweight data-interchange format.[19]
[12] Graphical object of the scene (the character, a bullet, etc.)

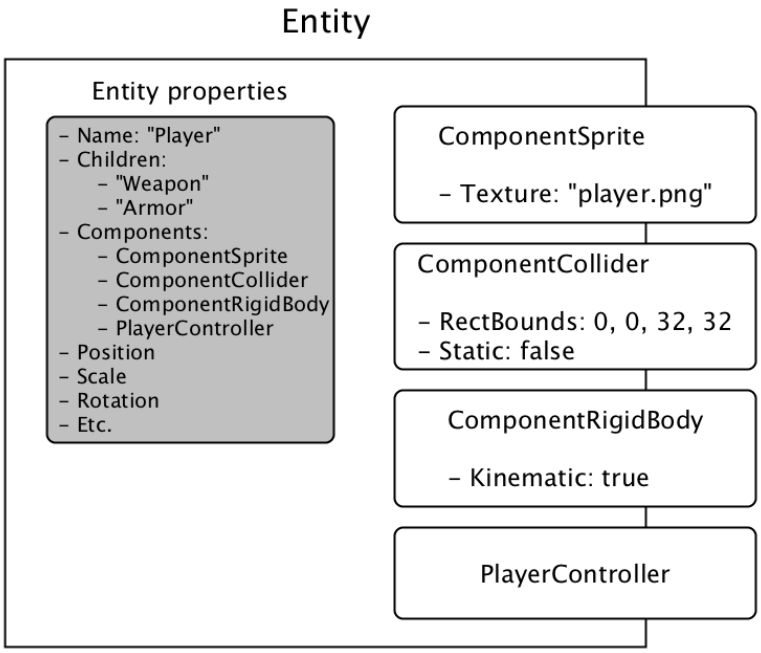

Figure 6: Representation of an Entity and its Components.

2.3 Free software and open-source

Free software and open-source have an important role nowadays: They encourage transparency, free and accessible learning, collaboration and altruism. Thanks to these philosophies, countless projects and technologies have seen the light of day for the benefit of the whole Internet community, which at the same time has an influence on people's lives. X-GSD would not have been possible without SFML or a similar free and open-source project, which in turn is made with C++ —also free of use—.

Because of that, and as stated before, X-GSD is released in a permisive zlib/png license —the same used by SFML— with full access to source code. Therefore, anyone can look into X-GSD classes and files to investigate how it is built and/or modify it freely, be it to enhance or completely change a particular part of the engine, be it because of simple curiosity.

At its current state, this game engine is functional with basic features. The project is available on a GitHub repository[2] so that anyone can collaborate and improve the engine by adding functionality, enhance some parts, optimise algorithms, etc.

2.4 Example game

Once all the X-GSD development is explained, a little example game will be developed so that one can see how to use X-GSD easily, covering all the needed parts:

- How to create the configuration file and its options.
- The game's entry point in the `main.cpp` file and how to start its execution with the invocation of a game *run method*.
- The creation and loading of scene files (with some resources and entities)

- How to write custom components (the controllers previously mentioned) to add logic and behaviour to the game.

3 Development

This section will present each part of X-GSD, class by class, explaining their interface and functionality of each and some annotations about the corresponding code if it takes place. Some fragments of code (include guards, header includes, namespaces, etc.) will be omitted for better readability and focus on the interesting parts. Code fragments will start with a single-line comment that indicate the source file from which code has been extracted, if any (the .hpp extension hints a header file, which generally corresponds to the interface or API; The .cpp extension is for implementation files) and can contain a "..." code ellipsis indicators. For more details, full source code is available as attached files of this work or on the GitHub repository[2]. It is plenty of comments and comes with some examples of use in order to make it easier to dive into it and understand each little part. Note that the presented fragments of code have a line numbering as visual aid, but it may not coincide with the source code's as these code fragments may be adapted and some parts omitted.

3.1 Game class

This class is the responsible to load the game's initial configuration, keep the game running in a specific game loop, handle resources and subsystems such as physics engine or events, and serve as a global point of access to these resources and subsystems. In order to achieve this functionality, the Game class is implemented similarly to the infamous *singleton pattern*[27] —but for a good reason: it offers accessibility while avoiding similar global state in other classes[13]—.

```
1  // Game.hpp
2
3  class Game
4  {
5  // Methods
6  public:
7    static Game&            instance() { return globalInstance; }
8
9    ...
10
11 private:
12 // Private constructor to ensure the static globalInstance is the only one
13                           Game();
14   ...
15
16 // Variables (member / properties)
17 private:
18   static Game             globalInstance;
19
20   ...
21 };
22
23
24 // Game.cpp
25
26 Game Game::globalInstance; // Static initialization of the Game globalInstance
```

[13] *"The goal of removing all global state is admirable, but rarely practical. Most codebases will still have a couple of globally available objects, such as a single Game or World object representing the entire game state."*[27]

The static method `instance()` returns a reference to the Game `globalInstance`. `Game()` constructor is set as `private` in order to ensure that the class cannot be instantiated besides the global instance. Any other class including Game's header file can get the game's instance simply by calling `Game::instance()`.

```
// Some class .cpp or .hpp

#include <X-GSD/Game.hpp>

// Retrieve the global instance, store its reference and do something with it
Game *myGame = Game::instance();
myGame->someMethod();

// Or simply use it directly
Game::instance()->someMethod();
```

3.1.1 Interface review

The state of the Game class is given by the following member variables:

```
// Game.hpp

class Game
{
// Typedefs and enumerations
private:
  typedef std::unordered_map<std::string, std::string> StringDataStore;

  ...

// Variables (member / properties)
private:
  static Game             globalInstance;

  xgsd::HiResDuration     mTimeSinceStart;
  sf::RenderWindow*       mWindow;
  bool                    mVSync;
  xgsd::Scene*            mScene;
  sf::Event               mEvent;
  xgsd::PhysicsEngine     mPhysicsEngine;

// Resources managers
  xgsd::FontManager       mFontManager;
  xgsd::TextureManager    mTextureManager;
  xgsd::SoundManager      mSoundManager;

#ifdef DEBUG
// Statistics
  bool                    mDebugRendering;
  bool                    mEnableStatistics;
  sf::RectangleShape      mStatisticsBackground;
  sf::Text                mStatisticsText;
  xgsd::HiResDuration     mStatisticsUpdateTime;
  std::size_t             mStatisticsNumFrames;
  std::size_t             mStatisticsNumSimulationSteps;
#endif

public:
  StringDataStore         DataStore;
};
```

From top to bottom:

- `mTimeSinceStart` holds the amount of time the game has been running.

- `mWindow` points to a `sf::RenderWindow`, an SFML class which represents the portion of the screen where the game will render and which have information about resolution/size of the window, video mode, etc.
- `mVSync` is a boolean which stores the vertical synchronization option.
- The `mScene` member variable is one of the most important of Game class. This `xgsd::Scene*` points to the current scene of the game. The Scene class contains all the scene-related behaviour and state which make possible to modularize the game in scenes, switch between them, load specific resources, etc. It will be explained in detail in its own section.
- `mEvent` is an `sf::Event`, another SFML class which represents a system event —generally, input-related such as mouse, keyboard or joystick input, or window-related such as closing or resizing the window—.
- The `mPhysicsEngine` member, of type `xgsd::PhysicsEngine`, is an object which handles basic collision detection.
- Resource managers of the Game class are useful to store global resources. For example, if the same font is used in all or a fair amount of the game's scenes, it is better to place it in `xgsd::FontManager` of this class to avoid loading, unloading and reloading the same font scene after scene. For more information, see (Appendix 6.4) or source code.
- All the variables between `#ifdef DEBUG` and `#endif` preprocessor directives will only be available if `DEBUG` symbol has been defined. They have statistic purposes such as displaying the frames per second in debug mode.
- Finally, `DataStore` is a `StringDataStore` which is a type definition of `std::unordered_map<std::string, std::string>`. This unordered map can be used to pass basic information in form of `std::string`s between objects of the game. For example, in a game with two levels and a class for each, level1 scene could store points and remaining lives before loading level2 scene so that the latter can know that information:

```
1 // level_1_scene.cpp
2 // Store player's lives in a "lives" record of the DataStore (from level1 scene)
3 Game::instance().DataStore["lives"] = std::to_string(mLives);
4
5 // level_2_scene.cpp
6 // Load player's lives from "lives" of DataStore (in level2 scene)
7 mLives = std::stod(Game::instance().DataStore["lives"]);
```

The behaviour of the Game class is defined with these methods:

```
1 // Game.hpp
2
3 class Game
4 {
5 ...
6
7 // Methods
8 public:
9   static Game&            instance() { return globalInstance; }
```

```
10
11    void                    runFixedDeltaTime(int stepsPerSecond = 60);
12    void                    runVariableDeltaTime();
13    void                    runSemiFixedDeltaTime(int simulationFrequency = 60, int stepLimit = 3);
14    void                    runFixedSimulationVariableFramerate(int simulationFrequency = 60);
15
16    Scene&                  getSceneManager() { return *mScene; }
17    ControllersManager&     getControllersManager() { return mScene->getControllersManager(); }
18    PhysicsEngine&          getPhysicsEngine() { return mPhysicsEngine; }
19    FontManager&            getGlobalFontManager() { return mFontManager; }
20    TextureManager&         getGlobalTextureManager() { return mTextureManager; }
21    SoundManager&           getGlobalSoundManager() { return mSoundManager; }
22    FontManager&            getLocalFontManager() { return mScene->getLocalFontManager(); }
23    TextureManager&         getLocalTextureManager() { return mScene->getLocalTextureManager(); }
24    SoundManager&           getLocalSoundManager() { return mScene->getLocalSoundManager(); }
25    sf::RenderWindow&       getWindow() { return *mWindow; }
26    HiResDuration           getRunningTime() { return mTimeSinceStart; }
27
28    void                    broadcastEvent(const Event& event);
29
30  #ifdef DEBUG
31    bool                    isDebugRenderingEnabled() { return mDebugRendering; }
32    void                    updateStatistics(const HiResDuration& elapsedTime);
33  #endif
34
35  private:
36  // Private constructor to ensure the static globalInstance is the only one
37                            Game();
38    void                    loadConfigurationFromFile();
39    void                    update(const HiResDuration& dt);
40    void                    render();
41    void                    handleEvents();
42
43  ...
44  };
```

All methods starting with 'get' —known as *getters*— return a reference to a subsystem such as `Scene`, `ControllersManager`, `PhysicsEngine`, resource managers (global, which belong to the Game; and local, which belong to the current scene) or the game's `sf::RenderWindow`. The method `getRunningTime` returns the amount of time the game has been running. There is also a `isDebugRenderingEnabled` method in debug mode to know whether debug rendering is active (for example, print on screen rendering information such as FPS, or visualize colliders of entities which have one).

`broadcastEvent` can be invoked from any class in order to inform to Entities belonging to the scene graph (and their Components/Controllers, if any) that a certain event occurred. It can be an `xgsd::Event` (a wrapper class of `sf::Event`. See Appendix 6.3) of type `xgsd::Event::EventType::System` —the system events explained before— or a custom event of type `xgsd::Event::EventType::Custom`, which are user-defined events such as *PlayerDied* or *GameWon*.

The rest of the methods (starting with 'run', `updateStatistics` and those defined as `private`) need a more detailed explanation, which will be developed below.

Starting with `loadConfigurationFromFile`, this is the first method called by Game when it is instantiated. That is, it is called in its constructor:

```
1  // Game.cpp
2
3  // Constructor
```

```
Game::Game()
: mVSync(false)
, mTimeSinceStart(0)
{
    // Load configuration (window properties, first scene to load...)
    loadConfigurationFromFile();

    ...
}
```

In this method, a JSON file named `gameconfig.json` will be read in order to get the initial configuration of the game. These are the JSON elements to load:

- `windowName` (as string). Default value: empty string.
- `windowSize` (as object, containing `width` and `height` as integers). Default value: 800, 600.
- `fullscreen` (as bool). Default value: false.
- `vsync` (as bool). Default value: true.
- `keyRepetition` (as bool). Default value: false.
- `mouseCursorVisible` (as bool). Default value: true.
- `debugFont` (as string). Default value: empty string. Only in Debug mode.
- `icon` (as string). Default value: empty string.
- `initialScene` (as string). No default value: Valid initialScene is required.

Here is a valid example of the `gameconfig.json` file:

```
{
  "windowName" : "Example Game",
  "windowSize" : {
    "width" : 720,
    "height" : 480
  },
  "fullscreen" : true,
  "vsync" : true,
  "keyRepetition" : false,
  "mouseCursorVisible" : false,
  "debugFont": "fonts/PressStart2P.ttf",
  "icon" : "textures/ballSpriteRed.png",
  "initialScene" : "scenes/TitleMenuScene.json"
}
```

The following `gameconfig.json` file is also valid. Note that `initialScene` is the only configuration parameter that is compulsory as it cannot have a default value.

```
{
  "initialScene" : "InitialScene.json"
}
```

Loading these elements from the JSON file is done with the JsonCpp[5] library. Implementation details can be found in `Game.cpp` source code file, but it is not explained here as it is out of the scope of this work to explain the use of JsonCpp library.

Note that `debugFont`, `icon` and `initialScene` are file paths, not only file names. The specified path is relative to the resource path, and this resource path must be returned by the function `resourcePath`, defined in `ResourcePath.hpp`. As the implementation depends on the operating system, the user must provide a different implementation for each targeted operating system. Boost's filesystem library[6] can be useful to write cross-platform code regarding file system and paths. SFML's website provide Xcode templates for SFML projects, which also generate an Objective-C implementation for Mac OS X and iOS. This implementation has been used in the development of the example games for simplicity.

3.1.2 The Game Loop

The game loop is the main loop that keeps the game running, performing all the needed operations in a specific order. It is responsible of invoking update, render and input handling procedures among others.

Generally, the update procedure manages all the logic of the game —physics and scripts included—, the render procedure is responsible to draw objects on screen, and the input handling procedure checks if there is any input in order to take appropiate actions.

For example, supposing that all entities are stored in `entitiesContainer`, one could implement the following methods:

```
1  // Game.cpp
2
3  void Game::update(const xgsd::HiResDuration& dt)
4  {
5      // Update calls here
6
7      // Call each entity's own update method
8      for (auto &entity : entitiesContainer) {
9          entity.update(dt);
10     }
11 }
12
13 void Game::render()
14 {
15     // Draw calls here
16
17     // Clear the window before drawing a new frame
18     mWindow->clear();
19
20     // Call each entity's own draw method
21     for (auto &entity : entitiesContainer) {
22         entity.draw();
23     }
24
25     // Display the frame on screen
26     mWindow->display();
27 }
28
```

[5] Baptiste Lepilleur. Jsoncpp. `https://github.com/open-source-parsers/jsoncpp`.

[6] Beman Dawes and Rene Rivera. Boost filesystem library. `http://www.boost.org/doc/libs/1_36_0/libs/filesystem/doc/index.htm`.

```
29 void Game::handleEvents()
30 {
31     // Ask mWindow for events and process them if any
32     while (mWindow->pollEvent(mEvent))
33     {
34         // Create an event wrapper and use it after
35         Event eventWrapper(mEvent);
36 
37         // Perform different actions depending on the event type
38         switch (mEvent.type)
39         {
40             // The window has been closed
41             case sf::Event::Closed:
42                 mWindow->close();
43                 break;
44 
45             // A key has been pressed
46             case sf::Event::KeyPressed:
47                 // Do something or check for specific key code
48                 break;
49 
50             // A joystick button has been pressed
51             case sf::Event::JoystickButtonPressed:
52                 // Do something or check for specific button code
53                 break;
54 
55             // Etc.
56         }
57     }
58 }
```

However, the user of X-GSD would need to modify these methods to make custom update, render and input handling, which would be unpractical —not to mention that a game engine or any other framework is generally made to handle these implementation details without the user needing to know about them, presenting only an interface—. In order to solve this, X-GSD defines these methods in different levels: Game, Scene, Entity and Component (or Controller, which is a user-defined Component). Thanks to that structure, these methods can be invoked in cascade, that is: the Game calls update on the Scene, which in turn calls update on each of its children scene nodes or Entities; and finally, each Entity calls update on its Components. Therefore, the default implementation is to call the next level's update method until it reaches the end of the chain. In most cases, that end point will be a Component, which can implement a custom update method —generally, that is how scripting is done: The user of X-GSD can create Controllers inheriting from Component and implementing a custom update method—. The same applies to render and input handling.

```
1  // Game.cpp
2 
3  void Game::update(const xgsd::HiResDuration& dt)
4  {
5      mScene->update(dt); // Update the scene
6  }
7 
8  void Game::render()
9  {
10     mWindow->clear(); // Clear the window before drawing the new frame
11     mScene->render(); // Draw the scene
12     mWindow->display(); // Display the frame
13 }
14 
15 
16 void Game::handleEvents()
17 {
```

```
18      // Ask mWindow for events and process them if any
19      while (mWindow->pollEvent(mEvent))
20      {
21          // Create an event wrapper and use it after
22          Event eventWrapper(mEvent);
23
24          // Propagate the event to the scene
25          mScene->handleEvent(eventWrapper);
26      }
27  }
```

In the case of `Game::handleEvents`, a previous filter can be done as shown in the first code fragment so that only the desired events propagate to the scene and its entities.

As it was stated before, the game loop is responsible of calling these procedures in a specific order. For example, a basic game loop may proceed in this order:

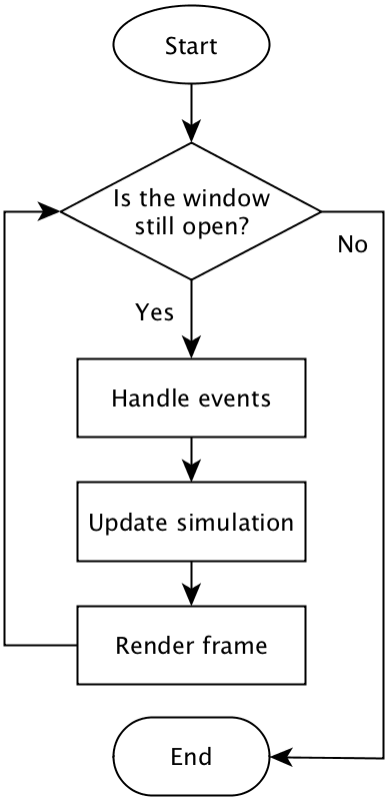

Figure 7: Basic game loop execution flow.

X-GSD provide four different game loop variations with different features and purposes which will explained below. The following implementations are based on the Glenn Fiedle's great article *Fix your timestep!*[10].

3.1.3 Basic fixed delta time

Updates at a fixed delta time, that is, the game's time advances in discrete delta time periods. The game's time must be synchronized with the real time in order for it to behave correctly. A simulation frequency value can be provided (if not, default value 60 is used). The implementation (simplified, without debug code) is the following:

```cpp
// Game.cpp

void Game::runFixedDeltaTime(int simulationFrecuency)
{
    HiResDuration simulationFixedDuration(ONE_SECOND/simulationFrecuency); // Simulation time step

    ...

    while (mWindow->isOpen())
    {
        ...

        handleEvents();
        update(simulationFixedDuration);
        render();
    }
}
```

First of all, the `HiResDuration` and `ONE_SECOND` types (and others as `HiResTime` and `HiResClock`, used later in other loops) are defined in `Time.hpp` as typedefs of `std::chrono` for easier time handling. For more information, see Appendix 6.2 or source code.

For better understanding of this first type of game loop, the key concepts will be explained. In the same way, a deep explaining of this fist type of game loop is needed to understand the other game loop variations.

- The game's simulation time —or *game time*— may not be equivalent to the real time. The game time advances at each update call by the amount of time specified in that call. That amount of time is generally called *time step* or *delta time*, although its meaning may vary depending on the type of game loop.

- Each loop iteration is composed of three important procedures: `update`, `handleEvents` and `render`. As all three procedures are called one after the other, being `render` the responsible of producing the visual result —the *frame*—, it is also common to use the term 'frame' for referring to a loop iteration. Each procedure costs a certain amount of time to complete, and the sum of the three is known as *frame time*.

- It is common to measure the amount of frames produced in one second and call it *FPS*, which stands for *Frames Per Second*.

- For simplicity when comparing with FPS, the game's simulation time step will be indicated in frequency (measured in Hertz, from the SI[14], abbreviated Hz) instead of time: 1 Hz = 1 cycle / 1 second. For example, a time step —or delta time— of 0.05 seconds can be expressed as a frequency of 20 Hz (1 cycle / 0.05 s = 20 Hz).

- Because of how this loop is coded, every iteration the game's simulation advances one step in the prefixed time and a frame is rendered. Therefore, the game time depends directly on the relation between the simulation frequency and the FPS —if expressed in frequency—, or the time step and the frame time —if expressed in time—.

[14]International System of Units

- *Vertical Synchronization* is a graphics-level property which can be enabled to limit the FPS to the display rate (if a display can output 60 Hz, the graphics card will limit the FPS to 60).

It is easy to mislead what the parameter of the `runFixedDeltaTime` method is, but it is required to understand it correctly before proceeding to explaining the game loop variations. As it was stated above, the game's simulation time may not match the real time because it is directly related to the frame time. The key concept is that the method's parameter determines the time that the game will advance in each iteration, and not the interval in which the update method will be invoked.

A parameter value of 30 will indicate that the simulation frequency is 30 Hz, so 1 cycle / 30 Hz = 0.0333 seconds is the amount of time that the game's world will advance each loop iteration. However, the time that takes to complete a full iteration —the frame time— is unknown in most cases because it depends on several factors such as the computation power of the machine, the available resources, the complexity of the game (number of entities and operations, etc.), some operating system settings or even hardware configurations, among others. Suppose a set of circumstances that make the iteration to complete in 0.0083 seconds (120 Hz or FPS, approximately). Then, the loop will complete 120 iterations each second, and each iteration will invoke update with a time of 0.0333 seconds, four times the time interval in which the update method is called, which results in a game's time advancement of four times the real world's, making the game to look accelerated. On the opposite, if FPS are lower than the simulation frequency, it would slow down.

Vertical Synchronitzation can fix half of the problem: For example, if a given scenario produces 500 FPS, it will be limited down to 60 FPS, so that the simulation frequency can be easily synchronized setting it to 60 Hz, making the game to run at a normal speed. However, it should not be considered as a good solution as this technique might not be available in some devices, or the operating system and even the user may disable it. Furthermore, the other half of the problem, the game running at lower FPS than the simulation frequency, can still occur. The need for other game loop type or variation in order to solve these problems becomes obvious.

Although this fixed delta time loop presents such problems, it can be used in some scenarios: With Vertical Synchronization activated, it can be suitable for mobile platforms where system events (notifications, updates, incoming call...) could break the game experience. As with this kind of time step the game would slowdown, the player can react to these system events interruptions.

3.1.4 Variable delta time

The main motivation behind this variation of the first game loop is to achieve a normal speed of simulation regardless of the machine in which the game is run. The key concept is that the delta time becomes variable instead of being a fixed value, and the game time will advance in different amounts of time depending on the frame time. The simplest method to achieve this is to measure the frame time and pass that time to the update method:

```cpp
// Game.cpp

void Game::runVariableDeltaTime()
{
    HiResDuration lastRenderDuration(0); // Will contain the frame time which is the delta time
     in this kind of loop

    HiResTime lastTimeMeasure = HiResClock::now(); // Initial time measure
    HiResTime newTimeMeasure;

    while (mWindow->isOpen())
    {
        newTimeMeasure = HiResClock::now(); // Time measure at the beginning of the iteration
        lastRenderDuration = newTimeMeasure - lastTimeMeasure; // Calculation of the last
     iteration (or frame) duration
        lastTimeMeasure = newTimeMeasure; // Update the last time measure

        handleEvents();
        update(lastRenderDuration); // Pass the last frame time to advance the simulation in that
     amount
        ...
        render();

        ...
    }
}
```

This way, each frame or loop iteration will last a specific time which is measured and passed to update as its argument. Therefore, if a frame only takes 0.002 seconds (that would be $500Hz$ or FPS), the simulation will advance in only 0.002 seconds too (the simulation frequency would be $500Hz$, coinciding with the FPS). On the contrary, if a frame takes much more time, then the simulation will advance in that much time too, making possible for the game to run always at the same simulation speed independently of the machine and circumstances in which the game runs.

Although this loop seems to be bulletproof for any kind of scenario, the ugly truth is that it is in fact very *dangerous* to use it for the physics part of the game. In order to illustrate the problem, suppose a scenario in which the game runs at 60 or more FPS and where the player shoots a bullet against an enemy, and each frame the physics engine will check if the objects collided[15]. For simplicity, the enemy will be considered as a sphere of 5 meters of radius. The bullet has a velocity of, for example, $400m/s$ and the enemy is standing at a distance of $20m$ from the player. The frame after the bullet was shot, update is invoked with the last frame time, which is 0.016 seconds. The update call propagates to scene children, with the bullet being one of them, and the bullet updates its position according to its velocity and the simulation time passed:

$$0.016s \cdot 400m/s = 6.4m$$

Then, the next frame lasts the same as before (0.016 seconds) and the bullet advances $6.4m$ again, having travelled $12.8m$ so far. A third frame and travelling of the same values would imply that the bullet would have travelled $19.2m$, which

[15]This is a simple example to illustrate the problem of this loop. In the reality, there are several collision detection methods for aiming different scenarios. For fast-moving objects such as bullets, it is common to use ray-casting methods instead of overlapping colliders.

would be considered as a hit because the enemy has a radius of $5m$.

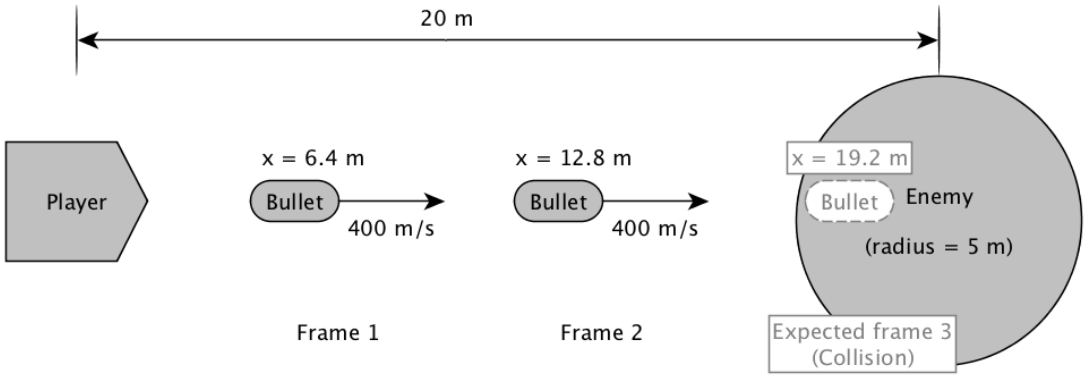

Figure 8: Movement of a bullet with variable delta time loop. Part 1

However, some random circumstance of the operating system (e.g. a software update notification) made that third frame to last longer than expected, and it takes $0.08s$ to complete. Then, update gets called with that time passed as its argument, and the simulation advances in $0.08s$. Therefore, the bullet travels more distance than before:

$$0.08s \cdot 400m/s = 32m$$

Then, the bullet's new position is at $12.8m + 32m = 44.8m$, which is far beyond the distance of $20m \pm 5m$ of radius that separated the player from the enemy, and the bullet did not hit him.

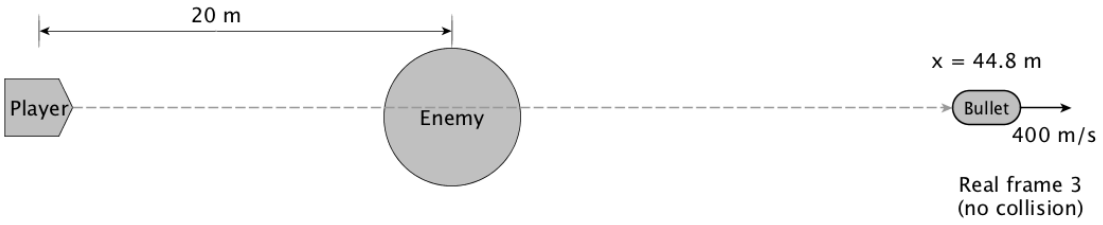

Figure 9: Movement of a bullet with variable delta time loop. Part 2

The consequence of that random performance drop is a collision not happening, which results in undesired behaviour of the game. Now suppose what would happen if another random event cause a performance drop and the delta time accumulates one second. That is just a simple example, but in a real game, a high delta time for the physics update could *break* the entire game, making the player and objects to fall through the floor or pass through walls, among other undesired effects. Therefore, this game loop variation can be considered even more unreliable than the previous, and other solutions become necessary.

Note that in this case Vertical Synchronization would only lower FPS and simulation frequency to the display's rate, but as the delta time is variable, the

game would still run at normal speed without needing to synchronize frequencies as it was the case with fixed delta time loop.

3.1.5 Semi fixed delta time

With the fixed delta time the problem was to simulate at normal speed in different machines and conditions, but physics behaved correctly —at least if a good enough simulation frequency was specified—. The variable delta time solved the problem of the speed, but lacked a consistent enough simulation. Taking the best of both worlds is the aim of the semi fixed delta time, which will be explained after the code:

```
1  // Game.cpp
2
3  void Game::runSemiFixedDeltaTime(int minSimulationFrequency, int stepLimit)
4  {
5      // stepLimit must be at least 1, or it would not update the simulation
6      assert(stepLimit > 0);
7
8      HiResDuration simulationFixedDuration(ONE_SECOND/minSimulationFrequency); // Simulation delta
       time
9
10     const short multipleStepLimit = stepLimit; // Limit of steps to consume the frame time.
       Needed to avoid the spiral of death effect. Tuning is recommended
11     short multipleStepCounter = 0;
12
13     HiResDuration lastRenderDuration;
14     HiResDuration simulationDuration;
15
16     HiResTime lastTimeMeasure = HiResClock::now();
17     HiResTime newTimeMeasure;
18
19     while (mWindow->isOpen())
20     {
21         newTimeMeasure = HiResClock::now();
22         lastRenderDuration = newTimeMeasure - lastTimeMeasure;
23         lastTimeMeasure = newTimeMeasure;
24
25         ...
26
27         while (lastRenderDuration > HiResDuration::zero() && multipleStepCounter <
       multipleStepLimit)
28         {
29             simulationDuration = std::min(lastRenderDuration, simulationFixedDuration);
30
31             handleEvents();
32             update(simulationDuration);
33
34             lastRenderDuration -= simulationDuration;
35             ...
36             ++multipleStepCounter;
37         }
38         multipleStepCounter = 0;
39
40         render();
41     }
42 }
```

The idea is that one can provide a simulation frequency —as it was possible with the fixed delta time—, while the frame time is also measured. Then, before updating, the minimum of the two times is selected to be passed as the argument of update. This way, if the FPS are high, the simulation frequency will be high as well; but on the contrary, if the FPS are low or a load spike or random event occurs and affect the game's FPS, the simulation is guaranteed to update at least

at a specific frequency. This can prevent the physics from breaking and leaving the game in an undesired state. However, if the iteration —especially the simulation update— takes too many time to complete, having a guaranteed minimum simulation frequency could lead to a worse slow down, which would result in longer iterations, and the cycle repeats making performance to drop heavily. To avoid this, a multiple simulation step limit can be added. With it, in the worse scenario the game would still run slower, but at least it would not fall into that undesired cycle.

To achieve this, another loop is needed (see line 27) in order to handle the process. Each iteration of the outer loop measures the time of the last iteration and enters the inner loop. This inner loop is responsible for:

- Check the remaining frame time and the multiple step counter against the limit.
- Selecting the minimum time between the last frame time and the provided fixed delta time.
- Handle events and update with the previously selected delta time.
- Subtract the simulated time from the frame time.
- Raise the multiple step counter.

Therefore, when the minimum simulation time that is selected coincides with the frame time, this inner loop will perform a single iteration and it will behave as the variable delta time in favorable conditions. On the contrary, if the frame time is higher than the provided simulation fixed time, the simulation will advance in that fixed time, subtract it from the frame time, raise the multiple step counter and repeat the loop until the frame time is zero or the counter reach the specified limit.

Although this game loop variation still has some drawbacks, it is arguably better than the fixed or variable delta time loops, and it will work well in most cases. It may also be appropriate for particular targets as mobile platforms for the same reasons given on the fixed delta time: Any condition that can affect performance may slow down the game instead of performing big simulation or physics steps, and therefore the player can react to these events.

3.1.6 Fixed delta time with variable rendering frame rate

The previously discussed game loop variation is one of the best and is suitable and good enough for a wide range of games. However, it does not feature a deterministic simulation —that is, each execution is guaranteed output the same exact results for the same exact input values, independently of the FPS—, which may be a needed or desirable behaviour. This is because the simulation duration passed as argument of the update call is not always a fixed time. The fixed delta time game loop provided this feature, but at the cost of strongly coupling FPS and simulation frequency which derived in other problems.

The last game loop variation that will be presented on this work is the fixed delta time with variable frame rate. This game loop achieves deterministic simulation while running at normal speed by decoupling simulation and rendering. In order to understand it better, one can think of `render` as a *producer* of time and of `update` as the *consumer* of that time. The loop will measure the time the renderer took to generate a frame and accumulate it, and then an inner loop will advance the simulation in fixed delta time steps until that time is consumed:

```
1  // Game.cpp
2
3  void Game::runFixedSimulationVariableFramerate(int simulationFrequency)
4  {
5      HiResDuration accumulatedRenderTime(0); // Accumulator of render time to be consumed by
       simulations
6      HiResDuration simulationFixedDuration(ONE_SECOND/simulationFrequency); // Fixed simulation
       delta time
7
8      HiResDuration lastRenderDuration; // Frame Time
9
10     HiResTime lastTimeMeasure = HiResClock::now();
11     HiResTime newTimeMeasure;
12
13     while (mWindow->isOpen())
14     {
15         newTimeMeasure = HiResClock::now();
16         lastRenderDuration = newTimeMeasure - lastTimeMeasure;
17         lastTimeMeasure = newTimeMeasure;
18
19         accumulatedRenderTime += lastRenderDuration;
20
21         while (accumulatedRenderTime >= simulationFixedDuration)
22         {
23             accumulatedRenderTime -= simulationFixedDuration;
24
25             handleEvents();
26             update(simulationFixedDuration);
27         }
28         ...
29
30         render();
31     }
32 }
```

For example, if the simulation frequency has been set to $120Hz$ and the game outputs 60 FPS, the inner loop will iterate and update the simulation twice each frame. On the opposite, with $60Hz$ for the simulation and 120 FPS, the inner loop would perform one iteration each 2 frames because time must be consumed in fixed delta time durations, and therefore the first attempt will not enter the inner loop but the time will be accumulated and consumed at the second attempt.

In the case of non exact divisions between simulation frequency and FPS, as for example $100Hz$ and 60 FPS, some *temporal aliasing*[16] can appear because of an alternation between the inner loop executing and consuming the accumulated time and not executing and accumulating time. Following with the same example, 60 FPS $\sim 0.0166s$, and $100Hz = 0.01s$. Therefore, each frame will add $0.0166s$ to the accumulator and the inner loop will run once and subtract $0.01s$ from the accumulator, leaving $0.0066s$ that cannot be consumed yet. Then, the renderer will produce another $0.0166s$, making a total of $0.0232s$, and the inner loop will

[16] An undesired visual effect of stuttering.

iterate twice, subtracting $0.01s$ each iteration and leaving $0.0032s$ as remainder, which will accumulate with subsequent remainders until it reaches $0.01s$ and gets consumed by the inner simulation loop.

In order to avoid the temporal aliasing problem, one can set Vertical Synchronization on and configure a simulation frequency which results in an exact division (60 FPS and $30Hz$ or $120Hz$, for example). The problem with this is the dependency on Vertical Synchronization. There is a better approach to correct that temporal aliasing, which consists in performing a linear interpolation between the previous and new state with the remainder time left in the accumulator. However, this is not implemented in X-GSD as it goes out of the academic scope of the project, and the previous solutions work fairly well for almost all scenarios.

3.1.7 The game's entry point

All C++ programs require at least one function called `main` from which the program will start. Therefore, the simplest manner to start the game is to include the Game class in the file which will contain the main function, and then invoke a run method of the four previously explained:

```
1  // main.cpp
2
3  #include <X-GSD/Game.hpp>
4
5  int main()
6  {
7       /* Invoke one of these
8     xgsd::Game::instance().runFixedDeltaTime();
9     xgsd::Game::instance().runVariableDeltaTime();
10    xgsd::Game::instance().runSemiFixedDeltaTime();
11    xgsd::Game::instance().runFixedSimulationVariableFramerate();
12    */
13
14     xgsd::Game::instance().runFixedSimulationVariableFramerate(30);
15 }
```

Excepting the `runVariableDeltaTime` method, an argument can be passed to these methods for tuning purposes. In the example above, a value of 30 has been passed as an argument indicating the desired simulation frequency. If no argument is provided, a default value is set (that value would be $60Hz$ in the example). See the previous sections for more details.

This instruction is the entry point of the game, and it will remain running in the specified loop until the end of execution.

3.2 Scenes, the scene graph and its nodes

Almost every game can be separated in scenes. A scene can be considered as an organization unit in which one can place objects —or Entities, which will be explained on the next section— with some relation that end up conforming a part of a game: The main menu, a battle, a mini game or a level are some examples of typical parts of a game which conform a scene.

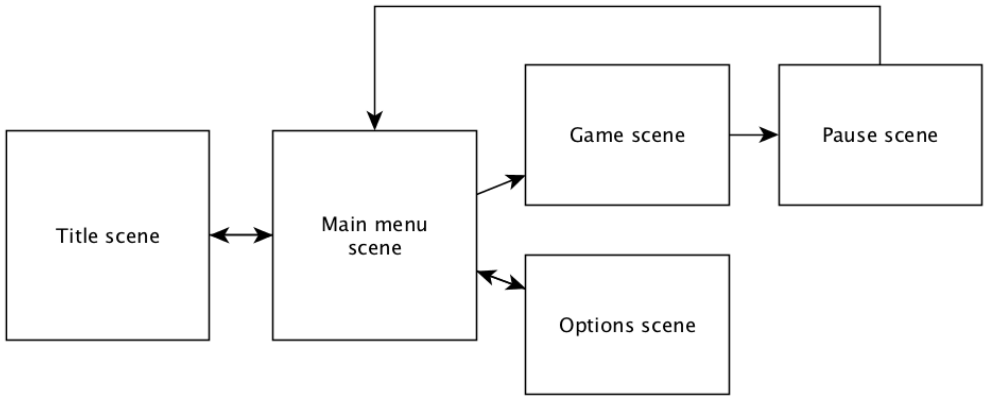

Figure 10: Example of a game organized in scenes.

3.2.1 Scene class

In X-GSD, a `Scene` is the owner of the root `SceneGraphNode` of the scene graph[17] and some properties (name, paused, camera, a reference to the game's window and others related to transitions) and managers for elements that are specific to that scene such as resources and physics. These resource managers are the same as the Game class, but the difference relies on their life cycle: The scene's resource managers will load/unload specific resources for that scene at its loading/unloading, while Game's resource managers will remain until the game ends. For more information about resource managers, see Appendix 6.4 or source code.

```
1  // Scene.hpp
2
3  ...
4
5      // Typedefs and enumerations
6  public:
7      typedef std::unique_ptr<Scene>    Ptr;
8
9  ...
10
11     // Variables (member / properties)
12 private:
13     std::string             mName;
14     SceneGraphNode::Ptr     mSceneGraph;
15     sf::RenderWindow&       mWindow;
16     sf::View                mSceneView; // Camera
17     std::string             mNextScenePath;
18     bool                    mSceneChangeRequest;
19     bool                    mPaused;
20
21     // Resources managers
22     FontManager::Ptr        mFontManager;
23     TextureManager::Ptr     mTextureManager;
24     SoundManager::Ptr       mSoundManager;
25
26     ControllersManager      mControllersManager;
27
28     PhysicsEngine&          mPhysicsEngine;
29
30     bool                    mTransitionEnabled;
31     TransitionState         mTransitionState;
32     HiResDuration           mTransitionTime;
```

[17] A scene graph, despite its name, is commonly implemented with a tree structure in which all the elements of a scene are stored or referenced in a specific order. It generally determines the simulation and rendering order of the objects belonging to the scene.

```
    HiResDuration           mTransitionDuration;
    sf::RectangleShape      mTransitionFadingRectangle;
};
```

The behaviour of the Scene class can be separated in the following methods:

```
// Scene.hpp

...

    // Methods
public:
    Scene(sf::RenderWindow& window);
    ~Scene();

    void                    update(const HiResDuration &dt);
    void                    render();
    void                    handleEvent(const Event &event);

    std::string             getName();
    FontManager&            getLocalFontManager()       { return *mFontManager; }
    TextureManager&         getLocalTextureManager()    { return *mTextureManager; }
    SoundManager&           getLocalSoundManager()      { return *mSoundManager; }
    ControllersManager&     getControllersManager()     { return mControllersManager; }

    void                    addNode(SceneGraphNode::Ptr node);

    void                    loadSceneFromFile(std::string jsonPath);

    bool                    isTransitionEnabled();
    void                    setTransitionEnabled(bool option);
    void                    pauseGame();
    void                    resumeGame();

private:
    void                    readJsonSceneFile(std::string jsonPath);
    void                    performSceneChange();
    void                    unloadScene();

    void                    updateTransition(const HiResDuration &dt);
    void                    renderTransition();

    ...
```

The constructor only requires a `sf::RenderWindow`, which is a reference of the game's window.

As it was mentioned earlier (section 3.1.2), `update`, `render` and `handleEvent` methods of Scene are invoked from the Game class with a specific order in the game loop. Then, the Scene invoke the same methods of its root SceneGraphNode —that is, they are propagated to the next level—. See the implementation:

```
// Scene.cpp

void Scene::update(const HiResDuration &dt)
{
    if (mPaused)
        mSceneGraph->onPause(dt);

    // If a scene change is requested, do not update
    else if (!mSceneChangeRequest) {
        mPhysicsEngine.checkCollisions();
        mSceneGraph->update(dt);
        mSceneGraph->performPendingSceneGraphOperations();
    }

    // Perform the scene change transition when requested
```

```
16      else if (mTransitionEnabled)
17          updateTransition(dt);
18  }
19
20  void Scene::render()
21  {
22      // If a scene change is requested, do not render the transition instead of the scene graph
23      if (!mSceneChangeRequest)
24          mWindow.draw(*mSceneGraph);
25      else if (mTransitionEnabled)
26          renderTransition();
27  }
28
29  void Scene::handleEvent(const Event &event)
30  {
31      mSceneGraph->handleEvent(event);
32  }
```

The `update` method checks if the game is paused, in which case it will invoke `onPause` on the SceneGraphNode instead of `update` so that nodes can perform tasks when the game is paused such as a pause menu. Because of a Scene could receive a request for changing to another scene, the update method also checks that condition. If neither paused nor requested to change the scene, the Scene will ask the physics engine to check for collisions, invoke `update` on the scene graph and perform some scene graph operations (both `checkCollisions` and `performPendingSceneGraphOperations` methods are explained later on Appendix 6.6 and Section 3.2.2). The last part will only be executed if a scene change occurs and the transition effect is enabled. In fact, this part is responsible to update the forementioned transition.

The `render` method also checks if the scene has been requested to change in order to render the transition effect. The rest of the time it will invoke the draw method of the window passing a pointer to the root scene graph node as an argument. This will result in SFML invoking `draw` on the root scene graph node, which is possible because the SceneGraphNode class inherits from `sf::Drawable` —any class that is intended to render something on the game's window must inherit from it—.

Finally, the `handleEvent` method just propagates the call to the scene graph.

Returning to the Scene's behaviour interface, the next methods are simple *getters* of the aforementioned managers. The `addNode` method simply attaches a SceneGraphNode (or derived classes, such as Entity) as a child of the root scene graph node (this will be explained in the next section).

The `loadSceneFromFile` method, along with the auxiliary private methods `performSceneChange`, `unloadScene` and `readJsonSceneFile`, is responsible of loading and switching to a new scene. If the transition is disabled, it immediately calls `performSceneChange`. Otherwise it will be called at transition end —but it will not be shown for simplicity. For more details on transition implementation, see source code—. In any case, `performSceneChange` will call `unloadScene` in order to release resources and unload the entire scene graph, and finally invoke `readJsonSceneFile` which will parse the new scene's json file and generate the new scene graph and load the needed resources. See the implementation:

```
// Scene.cpp

void Scene::loadSceneFromFile(std::string jsonPath)
{
    mSceneChangeRequest = true;
    mNextScenePath = jsonPath;

    if (!mTransitionEnabled)
        performSceneChange();
    else
        mTransitionState = out;
}

void Scene::performSceneChange()
{
    // Unload current scene's resources, nodes, values, etc.
    unloadScene();

    // Load the new scene's resources, entities, etc.
    readJsonSceneFile(mNextScenePath);

    mNextScenePath = "";
}

void Scene::unloadScene()
{
    mName = "";

    // Reset the scene graph
    mSceneGraph.reset(new SceneGraphNode);

    // Reset resources managers
    mFontManager.reset(new FontManager);
    mTextureManager.reset(new TextureManager);
    mSoundManager.reset(new SoundManager);
}
```

The `readJsonSceneFile` method reads a JSON file as the `loadConfigurationFromFile` from the Game class did: It uses the JsonCpp[21] parser in order to collect all the needed information. In X-GSD, a scene is described with the following JSON structure:

name (as string). No default value: a valid name for the scene is required.

entities (as list of objects). Default value: empty list.

- **name** (as string). No default value: a valid name for the entity is required.
- **parentEntityName** (as string). Default value: "root". Name of the entity to which this one will be added as child.
- **transform** (as object). Default value: a default-initialized transform.
  - **origin** (as object). Default value: default-initialized components.
    - **x** (as float). Default value: 0.
    - **y** (as float). Default value: 0.
  - **position** (as object). Default value: default-initialized components.
    - **relative** (as bool). Default value: false. If true, values are interpreted as a ratio of the view's width and height. For example, `x:0.5` is equivalent to `view.width*0.5`.
    - **x** (as float). Default value: 0.
    - **y** (as float). Default value: 0.

  - **scale** (as object). Default value: default-initialized components.
    - **x** (as float). Default value: 1.
    - **y** (as float). Default value: 1.
  - **rotation** (as float). Default value: 0.
- **components** (as object). Default value: null.
  - **ComponentSprite** (as object). Default value: null.
    - **globalTexture** (as bool). Default value: false. Determines whether the texture will be fetched from the local texture manager or the Game's global one.
    - **texture** (as string). No default value: a texture identifier (from 'textures' of 'resources') is needed.
    - **textureRect** (as object). Default value: a rectangle containing the complete texture. This can be used to select part of a texture such as a *spritesheet*.
      - **top** (as int). Default value: 0.
      - **left** (as int). Default value: 0.
      - **width** (as int). Default value: 0.
      - **height** (as int). Default value: 0.
  - **ComponentCollider** (as object). Default value: null.
    - **boundsRect** (as object). Default value: default-initialized components. It represents the bounding box of an entity which detect collisions.
      - **top** (as int). Default value: 0.
      - **left** (as int). Default value: 0.
      - **width** (as int). Default value: 0.
      - **height** (as int). Default value: 0.
  - **ComponentRigidBody** (as object). Default value: null.
    - **kinematic** (as bool). Default value: false. If true, forces do not affect this rigid body.
  - **controllers** (as list of objects). Default value: empty list.
    - **type** (as string). No default value: A valid user-controller identifier is needed.

resources (as object). Default value: null.

- **textures** (as list of objects). Default value: empty list.
  - **name** (as string). No default value: An identifier is needed.
  - **path** (as string). No default value: The path of the resource file is needed (relative to the resources folder).
- **fonts** (as list of objects). Default value: empty list.
  - **name** (as string). No default value: An identifier is needed.
  - **path** (as string). No default value: The path of the resource file is needed (relative to the resources folder).
- **sounds** (as list of objects). Default value: empty list.

- **name** (as string). No default value: An identifier is needed.
- **path** (as string). No default value: The path of the resource file is needed (relative to the resources folder).

Notice that it is a very basic scene description with few properties and obvious room for improvement, although it offers enough functionality to start making games with X-GSD.

Here is a valid example of a scene JSON file:

```
{
  "name": "AnExampleScene",

  "entities": [
    {
      "name": "ExampleEntity1"
    },

    {
      "name": "ExampleEntity2",
      "parentEntityName": "ExampleEntity1",
      "transform": {
        "origin": {
          "x": 0,
          "y": 0
        },
        "position": {
          "relative": false,
          "x": 100,
          "y": 100
        },
        "scale": {
          "x": 1,
          "y": 1
        },
        "rotation": 0
      },
      "components": {
        "ComponentSprite": {
          "globalTexture": false,
          "texture": "ExampleTexture"
        },
        "controllers": [
          {
            "type": "ExampleController1"
          }
        ]
      }
    }
  ],

  "resources": {
    "textures": [
      {
        "name": "ExampleTexture",
        "path": "exampleTexture.png"
      }
    ],
    "fonts": [
      {
        "name": "ExampleFont",
        "path": "exampleFont.ttf"
      }
    ],
    "sounds": [
      {
        "name": "ExampleSound",
        "path": "exampleSound.wav"
      }
```

```
    ]
  }
}
```

Returning to the `readJsonSceneFile` method, it will parse the specified JSON file collecting all the needed data for the creation of the scene. The explanation is focused on the implementation aside from JSON operations. Full implementation details can be found in `Scene.cpp` source code file.

The method is separated in three parts: Loading some properties of the scene and its resources, creating entities and components and finally performing some final operations and checks. The following code is plenty of comments to be self-explanatory (the "..." parts represent omitted code, mostly JSON-related):

```
// Scene.cpp

void Scene::readJsonSceneFile(std::string jsonPath)
{
    // Read the json file

    ...

    ////////////////////////////////////////////
    //  1.-  Fill scene info and load resources  //
    ////////////////////////////////////////////

    ...

    // Iterate through textures array from JSON
    for (auto texture : textures) {

        ...

        // Load texture into the scene's texture manager
        mTextureManager->load(name, path);
    }

    // The same procedure is done for sounds and fonts loading with the appropriate resource managers

    ...

    ////////////////////////////////////////////
    //  2.-  Create entities and components   //
    ////////////////////////////////////////////

    // Collection to store entities whose parent entity was not available at its time of creation. They will try to attach to its parents again after all entities load.
    std::vector<std::pair<Entity::Ptr, std::string>> delayedAttachmentEntities;

    // Temporary map to store entity-parent names to check and prevent circle parent reference between entities (a.parent = b, b.parent = a)
    std::unordered_map<std::string, std::string> entitiyParentRelations;

    // Register user-defined controllers
    ControllersRegistration controllersRegistration;
    controllersRegistration.registerControllers(mControllersManager);

    ...

    // Iterate through entities from JSON
    for (auto entity : entities) {

        ...

        // Create a new entity and fill its data
        Entity::Ptr newEntity(new Entity(name));
```

```
51
52          ...
53
54          // Look for X-GSD defined components (Sprite, Collider and RigidBody) in the JSON. If found, add them to the entity. For example, a ComponentCollider:
55
56          ...
57
58          // Create the component and fill its properties
59          Component::Ptr componentCollider(new ComponentCollider(...));
60          // Add it to the entity
61          newEntity->addComponent(std::move(componentCollider));
62
63          ...
64
65          // Iterate through controllers from JSON and add them to the entity (if any)
66          for (auto controller : controllers) {
67              ...
68
69              // Create the controller and add it to the entity
70              Component::Ptr newController(mControllersManager.getController(type));
71              newEntity->addComponent(std::move(newController));
72          }
73
74
75          // And finally, add this entity to the scene's root node or the specified parent entity
76          ...
77
78          if (parentEntityName == "root")
79              mSceneGraph->requestAttach(std::move(newEntity));
80          else {
81              // Look for the parent entity
82              auto parentEntity = Entity::getEntityNamed(parentEntityName);
83
84              if (!parentEntity) {
85                  // If the specified parent has not been found, store this entity in a temporal collection in order to retry later
86                  delayedAttachmentEntities.push_back(std::make_pair(std::move(newEntity), parentEntityName));
87              }
88              else
89                  parentEntity->requestAttach(std::move(newEntity));
90          }
91
92          // Store parent name for circular parent reference checking
93          if (parentEntityName != "root")
94              entitiyParentRelations[name] = parentEntityName;
95
96      } // End of entities loop
97
98
99      ////////////////////////////////////////////////
100     //  3.-  Perform final operations and checks   //
101     ////////////////////////////////////////////////
102
103
104     // Try to attach again the stored entities whose parent entities were not available
105     for (auto iter = delayedAttachmentEntities.begin(); iter != delayedAttachmentEntities.end(); ++iter) {
106         // Look for the parent entity
107         auto parentEntity = Entity::getEntityNamed(iter->second);
108
109         // If the parent entity is not found at this point, it does not exist. Throw an error
110         if (!parentEntity)
111             throw std::runtime_error("The specified parent entity ... does not exist");
112         parentEntity->requestAttach(std::move(iter->first));
113     }
114
115
116     // Check circle parent reference between entities (a.parent = b, b.parent = a)
117     for (auto iter = entitiyParentRelations.begin(); iter != entitiyParentRelations.end(); ++iter)
118     {
119         std::string entityName = iter->first;
120         std::string parentName = iter->second;
121
```

```
122         // If a circular parent reference is found, throw an error
123         if (entitiyParentRelations[parentName] == entityName)
124             throw std::runtime_error(... "Circular parent reference" ...);
125     }
126
127     // Ensure scene graph operations (attachments) get done
128     mSceneGraph->performPendingSceneGraphOperations();
129 }
```

With this, the Scene class has been covered, although many concepts need from the next sections to be fully understood.

3.2.2 SceneGraphNode class

This class takes the main ideas from the SceneNode class of the *SFML Game Development Book*[15], with several additions and adaptions.

The `SceneGraphNode` is the core element of the Scene and a key element of the game: All entities inherit from SceneGraphNode, and therefore they are in fact SceneGraphNodes with added functionality. Because of that, several methods are marked as `virtual`, which mean that derived classes can override SceneGraphNode's implementation.

The purpose of this class is to serve as a data structure to describe a node of a scene graph and provide some functionality: handle scene graph operations such as children attachment and detachment; continue with the `update`, `draw` and `handleEvent` method chains; and give flexibility for customization —that is, inheritance and method overriding—.

Notice the inheritance of this class: `sf::Drawable`, which forces the class to implement a SFML-standard draw method, and `sf::NonCopyable`, which disables copy constructor and assign operator for this class in order to avoid potential undesired behaviour.

Here is the interface:

```
1  // SceneGraphNode.hpp
2
3  class SceneGraphNode : public sf::Drawable, private sf::NonCopyable
4  {
5      // Typedefs and enumerations
6  public:
7      typedef std::unique_ptr<SceneGraphNode> Ptr;
8
9      // Methods
10 public:
11     SceneGraphNode();
12     virtual             ~SceneGraphNode();
13
14     void                requestDetach(SceneGraphNode* child);
15     void                requestAttach(Ptr child);
16     void                requestDestroy();
17     void                performPendingSceneGraphOperations(); // Attach, detach, destroy
18
19     void                onAttach();
20     void                onDetach();
21     void                onPause(const HiResDuration &dt);
22     void                update(const HiResDuration& dt);
23     void                draw(sf::RenderTarget& target, sf::RenderStates states) const override;
24     void                handleEvent(const Event& event);
```

```
25
26      sf::Vector2f        getWorldPosition() const;
27      sf::Transform       getWorldTransform() const;
28      SceneGraphNode&     getParent();
29      bool                isDestroyPending();
30
31  private:
32      virtual void        onAttachThis();
33      void                onAttachChildren();
34
35      virtual void        onDetachThis();
36      void                onDetachChildren();
37
38      virtual void        onPauseThis(const HiResDuration& dt);
39      void                onPauseChildren(const HiResDuration& dt);
40
41      virtual void        updateThis(const HiResDuration& dt);
42      void                updateChildren(const HiResDuration& dt);
43
44      virtual void        handleEventThis(const Event& event);
45      void                handleEventChildren(const Event& event);
46
47      virtual void        drawThis(sf::RenderTarget& target, sf::RenderStates states) const;
48      void                drawChildren(sf::RenderTarget& target, sf::RenderStates states) const;
49
50      void                attachChild(Ptr child);
51      Ptr                 detachChild(SceneGraphNode& child);
52      void                destroy();
53
54      // Variables (member / properties)
55  public:
56      sf::Transformable               mTransformable;
57
58  private:
59      std::vector<Ptr>                mChildren;
60      SceneGraphNode*                 mParent;
61
62      std::vector<SceneGraphNode*>    mPendingDetachments;
63      std::vector<Ptr>                mPendingAttachments;
64      bool                            mPendingDestruction;
65  };
```

Starting with the data of this class, the `mTransformable` variable is a public `sf::Transformable` —an SFML element that provide functionality on top of a transformations matrix[18]—, `mChildren` is a collection of children nodes and `mParent` is a reference to a node containing this node as a child (or `nullptr` if that is not the case). There are also two additional collections, `mPendingDetachments` and `mPendingAttachments`, and a flag variable called `mPendingDestructions` which are needed for safe handling of attachments and detachments.

The behaviour of this class can be divided in three main parts: methods to manage the relationships parent-child between nodes, methods related with the game loop and life cycle of the node and some getters.

As it was stated before, `update`, `draw` and `handleEvent` methods are present in this class too in order continue the propagation of the invocations from the Scene, which in turn get its invocations from the Game class —in the game loop—. Only one node will receive the invocations from the Scene, which is the scene root node that the Scene has as member variable, and it will propagate them through the entire scene graph. That is possible because, in this class, each of these methods will invoke two private methods: `-This`, which performs the corresponding opera-

[18] A transformation matrix is a widely-used representation of geometric transformations in computer graphics such as position, scale and rotation.

tions in this node, and `-Children`, which will invoke the same method on all the children of this node. For example, see the update method implementation:

```
// SceneGraphNode.cpp

void SceneGraphNode::update(const HiResDuration& dt)
{
    updateThis(dt);
    updateChildren(dt);
}

void SceneGraphNode::updateThis(const HiResDuration& dt)
{
    // Do nothing by default. Override this on a derived class or a custom Component controller
}

void SceneGraphNode::updateChildren(const HiResDuration& dt)
{
    for(Ptr& child : mChildren)
        child->update(dt);
}
```

The methods finishing with `-This` are empty by default, and they were marked as `virtual` on the interface file. Therefore, derived classes as Entity can override those methods to implement custom behaviour. If a derived class does not override and implement them, the invocation chain will stop at this point.
In the case of `draw`, there is a subtle difference:

```
// SceneGraphNode.cpp

void SceneGraphNode::draw(sf::RenderTarget& target, sf::RenderStates states) const
{
    // Apply transform of current node
    states.transform *= mTransformable.getTransform();

    // Draw the node and its children
    drawThis(target, states);
    drawChildren(target, states);
}
```

In order to render nodes in a parent-child hierarchy properly, the transformations matrix must be multiplied before performing the rendering to apply the transformations of previous nodes in the scene graph. Then it is stored and passed along through a `sf::RenderState` context variable to the next level of children nodes.

In addition to these methods, this class adds three more, `onAttach`, `onDetach` and `onPause`, with the same structure as the aforementioned: One public method which call two private methods `-This` —also marked as `virtual` so that it can be overridden and implemented in derived classes— and `-Children`. The first two methods get invoked at attachment/detachment time, which can be useful to perform initialization/cleaning actions that cannot be done in the constructor/destructor. For example, any operation that needed the parent node reference can be postponed to be done on attachment instead of the constructor. The case of `onPause` is more obvious: It will be called when the game is in pause state, so that some operations can still be done such as managing a pause menu or displaying an animation.

The second part of the behaviour of the SceneGraphNode class manages the relationships between nodes. It divides into public methods (those starting with `request-`), and private methods:

```
// SceneGraphNode.cpp

// public methods
void SceneGraphNode::requestAttach(Ptr child)
{
    mPendingAttachments.push_back(std::move(child));
}

void SceneGraphNode::requestDetach(SceneGraphNode* child)
{
    mPendingDetachments.push_back(child);
}

void SceneGraphNode::requestDestroy()
{
    mPendingDestruction = true;
}

void SceneGraphNode::performPendingSceneGraphOperations()
{
    // Perform delayed node attachment/detachment/destruction, when it is safe

    // Invoke the same method on children first
    for (auto childIter = mChildren.begin(); childIter != mChildren.end(); ++childIter) {
        (*childIter)->performPendingSceneGraphOperations();
    }

    // Perform delayed node attachments
    for (auto nodeIter = mPendingAttachments.begin(); nodeIter != mPendingAttachments.end();
     ++nodeIter) {
        attachChild(std::move(*nodeIter));
    }
    mPendingAttachments.clear();

    // Perform delayed node detachments
    for (auto nodeIter = mPendingDetachments.begin(); nodeIter != mPendingDetachments.end();
     ++nodeIter) {
        detachChild(**nodeIter);
    }
    mPendingDetachments.clear();

    // Perform delayed destruction of this node
    if (mPendingDestruction) {
        destroy();
    }
}

// private methods
void SceneGraphNode::attachChild(Ptr child)
{
    // Set this node as the parent reference for the new child and add it to the children
     collection
    child->mParent = this;
    child->onAttach();
    mChildren.push_back(std::move(child));
}

SceneGraphNode::Ptr SceneGraphNode::detachChild(SceneGraphNode& child)
{
    // Look for the child node to detach
    auto found = std::find_if(mChildren.begin(), mChildren.end(), [&] (Ptr& p) { return p.get()
     == &node; });
    assert(found != mChildren.end());

    child.onDetach();

    // Delete the parent reference, the node from the children collection and return it
```

```
64      Ptr result = std::move(*found);
65      result->mParent = nullptr;
66      mChildren.erase(found); % TODO: Update if changed in source code
67      return result;
68  }
69
70  void SceneGraphNode::destroy()
71  {
72      assert(mParent && "Attempted to destroy the root scene graph node, or a node which has not
         been attached as a child of another yet!"); % TODO: Cambiar por warning? Detectar si es el
         root de otro modo?
73      // Clear the collection of children and request the parent of this node to detach it
74      this->mChildren.clear();
75      mParent->requestDetach(this);
76      this->mParent = nullptr;
77  }
```

These public `request-` methods store the nodes to attach, references of nodes to detach or set the flag of pending destruction. But the real operations —calling the private methods— are delayed to be done all at once, when the node get its `performPendingSceneGraphOperations` method invoked (the Scene class is the responsible to do it. See the `update` and `loadSceneFromFile` methods implementation of Scene for more details). Delaying such actions provides consistency to the scene graph, so that it does not change in middle of an update.

Finally, this class offers some getters, of which these two require an explanation:

```
1   // SceneGraphNode.cpp
2
3   sf::Transform SceneGraphNode::getWorldTransform() const
4   {
5       sf::Transform worldTransform = sf::Transform::Identity;
6
7       // Navigate through the scene graph in reverse order (from this node to the root) and
         multiply their transforms
8       for (const SceneGraphNode* node = this; node != nullptr; node = node->mParent)
9           worldTransform = node->mTransformable.getTransform() * worldTransform;
10
11      // The transform is now in coordinates of the 'world' (the scene root node)
12      return worldTransform;
13  }
14
15  sf::Vector2f SceneGraphNode::getWorldPosition() const
16  {
17      return getWorldTransform() * sf::Vector2f();
18  }
```

Every node has its own transform, and the information that it contains is considered as *local* (for example, being at position x=2, y=3). However, if a node is child of another node, its final 'state' does not depend only on its own transform, but its parent's too, and the parent of its parent, and so forth. Continuing with the example, suppose that the previous node is attached to another with a local position of x=1, y=1. Then, although the previous node is *locally* at x=2, y=3, it would be at x=3, y=4 in *world* coordinates. Then, if a comparison of positions would be necessary for any reason (for example, collision detection), it is non-sense to compare local values. This problem is even worse if the nodes to be compared do not even pertain to the same parent-children hierarchy.

This may look tedious to solve. However, the fact is that using matrices for

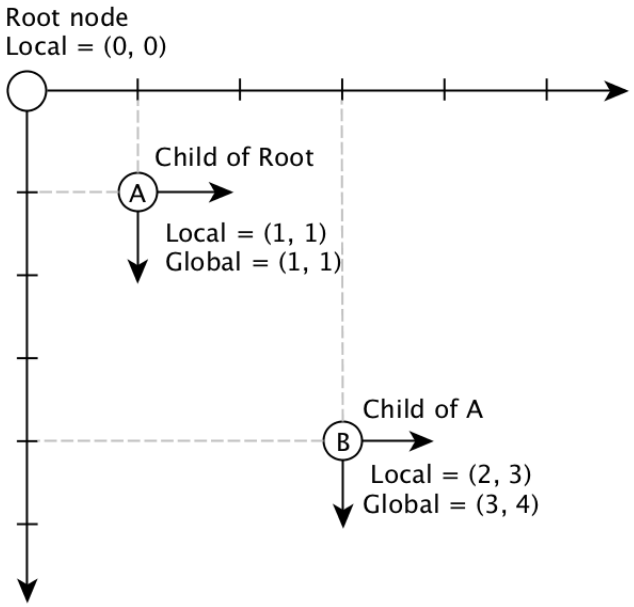

Figure 11: Local and global positions of nodes.

describing transformations has the purpose to facilitate these operations, as it can be seen in the `getWorldTransform` and `getWorldPosition` methods: By multiplying the transfoms until reaching the scene root node, the resulting transform is in coordinates of the *world*, and therefore comparisons become possible now.

3.3 Entity-Component system

In the previous section it was introduced that Entity is a derived class of SceneGraphNode, and therefore, it has the same functionality with some additions. Essentially, those are: having a name as an identifier and serving as a container of Components, which are the final links of the chain of invocations used all along X-GSD. Components are intended to implement the behaviour of different elements, such as rendering a visual representation (a Sprite) or the logic of a character (moving, jumping, shooting, etc.).

Entities and Components are two fundamental high-abstraction-level elements for the user of X-GSD in the sense that a user of X-GSD is unlikely to use bare Scene or SceneGraphNode objects, but instead use high-level options as creating JSON files describing scenes and the entities-components inside them. A user would also create Entity and Component-derived objects programmatically or implement a new type of Component (or Controller). For example:

- Describe a simple Scene in a JSON file:
  - Load needed resources
  - An Entity for the player's character with these Components:
    - A Sprite so that the character has a visual representation (using a texture specified at resources loading)
    - A Collider in order to detect collisions between the character and other elements (floor, enemies...)

        - A custom PlayerController where to implement the logic related to the character (controls, actions, movement, life...)
    - Some more Entities for several elements of the world (floor, walls, enemies...) with the needed Components for each.

- Create the PlayerController class, inheriting from ComponentController. Implement the character-related logic within this class. For example, a Shoot action:
    - Implement the action of shooting in a Shoot method, in which a new Entity (and the needed Components) representing a bullet is created programmatically.
    - Override the handleEvent method. In its implementation, check if some specific button or key has been pressed in order to invoke the Shoot method.

This example shows how a user of X-GSD can create a simple game without the need of (directly) using other X-GSD classes apart from Entity and Component-derived.

3.3.1 Entity class

As usual, here is the class inteface:

```
1  // Entity.hpp
2
3  class Entity : public SceneGraphNode
4  {
5      // Typedefs and enumerations
6  public:
7      typedef std::unique_ptr<Entity>    Ptr;
8
9      // Methods
10 public:
11     Entity(std::string name);
12     ~Entity();
13
14     void                    onAttachThis() override;
15     void                    onDetachThis() override;
16     void                    onPauseThis(const HiResDuration &dt) override;
17
18     void                    updateThis(const HiResDuration& dt) override;
19     void                    drawThis(sf::RenderTarget& target, sf::RenderStates states) const
        override;
20     void                    handleEventThis(const Event& event) override;
21
22     void                    addComponent(Component::Ptr component);
23     template <typename T>
24     std::unique_ptr<T>      removeComponent();
25     template <typename T>
26     T*                      getComponent();
27
28     void                    collisionHandler(Entity* theOtherEntity, sf::FloatRect collision);
29
30     std::string             getName();
31     static Entity*          getEntityNamed(std::string name);
32
33     // Variables (member / properties)
34 private:
35     std::map<std::type_index, Component::Ptr>       mComponents;
36     std::string                                     mName;
37
38     static std::unordered_map<std::string, Entity*> entities;
```

```
};
```

In order to provide the aforementioned functionality of an Entity, this class has to inherit from SceneGraphNode and hold a `mName` string and `mComponents`, a collection of `Component::Ptr` —an alias `typedef` for Component wrapped into `std::unique_ptr`—. This collection is a `std::map` in which the *key* is a particular `std::type_index`, and therefore an Entity can only hold one instance per type.

The behaviour of the Entity class divides in methods which override those marked as `virtual` in SceneGraphNode, methods for Component handling, the `collisionHandler` callback method and a pair of getters.

All overridden methods limit to propagate the invocation to all components of the Entity, if any. For example, this is the `onPauseThis` implementation:

```
// Entity.cpp

void Entity::onPauseThis(const HiResDuration& dt)
{
    // Perform update call of components/controllers
    for (auto iter = mComponents.begin(); iter != mComponents.end(); ++iter)
    {
        iter->second->onPause(dt);
    }
}
```

The `collisionHandler` method's implementation follows the same structure, although it does not override any method. This method is used as a callback by the PhysicsEngine if a collision occurs. More details will be given later in its section.

The `addComponent`, `removeComponent` and `getComponent` methods are responsible of handling the `mComponents` collection accordingly and notifying the Component of attachment/detachment events. For an in-depth explanation, read the comments of the implementation:

```
// Entity.cpp

void Entity::addComponent(Component::Ptr component)
{
    // Add this entity's reference to the component's pointer
    component->entity = this;

    // Notify the component that it has been added to an entity
    component->attachedToEntity();

    // Store the component according to its type
    mComponents[std::type_index(typeid(*component))] = std::move(component);
}


// Entity.hpp

template <typename T>
std::unique_ptr<T> Entity::removeComponent()
{
    // Retrieve the type_index of T
    std::type_index index(typeid(T));

    // Ensure that a component with type T exists in the collection
```

```
25      assert(mComponents.count(std::type_index(typeid(T))) != 0);
26
27      // Retrieve it from the collection and cast it back to the T polymorphic type
28      T *tmp = static_cast<T*>(mComponents[index].get());
29
30      // Remove it from the collection and return it
31      std::unique_ptr<T> removedComponent(tmp);
32      mComponents[index].release();
33      mComponents.erase(index);
34
35      // Notify the component that it has been removed from an entity
36      removedComponent->detachedFromEntity();
37
38      // Finally, return the component. If no ownership is claimed, it will be destroyed.
39      return removedComponent;
40  }
41
42  template <typename T>
43  T* Entity::getComponent()
44  {
45      // Retrieve the type_index of T
46      std::type_index index(typeid(T));
47
48      // If it exists in the collection, retrieve it, cast it back to the T polymorphic type and
        return it
49      if (mComponents.count(std::type_index(typeid(T))) != 0) {
50          return static_cast<T*>(mComponents[index].get());
51      }
52      // Otherwise, a nullptr is returned
53      else {
54          return nullptr;
55      }
56  }
```

Finally, the Entity class offer a simple mechanism of retrieving any Entity anywhere in code only by specifying its name. A static collection of entities (a map of string as keys and Entity pointers as values) is filled/emptied as instances of Entity are constructed/destructed.

```
1   // Entity.cpp
2
3   // Static initialization
4   std::unordered_map<std::string, Entity*> Entity::entities;
5
6   // Static method
7   Entity* Entity::getEntityNamed(std::string name)
8   {
9       // Look for an specific entity by its name in the entities collection
10      auto found = entities.find(name);
11      if (found != entities.end())
12          return found->second;
13      else
14          return nullptr;
15  }
16
17  // Constructor
18  Entity::Entity(std::string name)
19  : mName(name)
20  {
21      // Insert a reference of the new Entity in a collection of pairs name-entityPointer
22      auto inserted = entities.insert(std::make_pair(name, this));
23
24      if (inserted.second == false)
25          throw std::runtime_error("Insertion of entity with name [" + name + "] failed.");
26  }
27
28  // Destructor
29  Entity::~Entity()
30  {
31      // Delete this entity's pointer from the entities collection
```

```
32      auto found = Entity::entities.find(mName);
33      assert(found != Entity::entities.end());
34      Entity::entities.erase(mName);
35  }
```

Then, from any place one can retrieve a pointer to a specific Entity using the `getEntityNamed` static method:

```
1  // Some class .cpp or .hpp
2
3  Entity *playerEntity = Entity::getEntityNamed("player");
```

3.3.2 Component class

This class mainly serves as the base class from which to inherit when implementing a specific Component, such as ComponentSprite. The derived class must override the desired methods and implement their behaviour. If a method is not overridden, the Component's implementation of that method will be used instead, which is an empty method. The Component class, similarly to the SceneGraphNode class, inherits from `sf::NonCopyable` and therefore its copy constructor is disabled, making it less error-prone.

```
1   // Component.hpp
2
3   class Component : sf::NonCopyable
4   {
5       // Typedefs and enumerations
6   public:
7       typedef std::unique_ptr<Component> Ptr;
8
9       // Methods
10  public:
11      Component();
12      virtual ~Component();
13
14      virtual void            attachedToEntity();
15      virtual void            detachedFromEntity();
16
17      virtual void            onEntityAttach();
18      virtual void            onEntityDetach();
19      virtual void            onPause(const HiResDuration& dt);
20
21      virtual void            update(const HiResDuration& dt);
22      virtual void            draw(sf::RenderTarget& target, sf::RenderStates states) const;
23      virtual void            handleEvent(const Event& event);
24
25      // Callbacks for other common components
26      virtual void            collisionHandler(Entity* theOtherEntity, sf::FloatRect collision);
27
28      // Variables (member / properties)
29  public:
30      Entity*                 entity;
31
32  };
```

Regarding the data of the class, it only stores a pointer to the Entity holding this Component. The available methods are mostly the already known `update`, `draw`, `handleEvent`, `onPause` and `collisionHandler`; two renamed methods from Entity: `onEntityAttach` and `onEntityDetach` (called `onAttach` and `onDetach`

in the Entity class) and a new pair of methods with names `attachedToEntity` and `detachedFromEntity`. These last two get invoked when the Component is attached to or detached from an Entity, which should not be confused with `onEntityAttach` and `onEntityDetach` because these are invoked when a SceneGraphNode or Entity is attached to or detached from another SceneGraphNode or Entity —but it is interesting to have them in Component so that actions can be taken in response to these events—.

X-GSD already provides three Component-derived classes integrated with the engine: ComponentSprite, ComponentRigidBody and ComponentCollider. They offer very basic functionality with much room for improvement, but sufficient for the scope of this project and for building some example games. There exists also the option for the user to create their own custom components —the Controllers—, so that custom behaviour can be implemented and then added to entities.

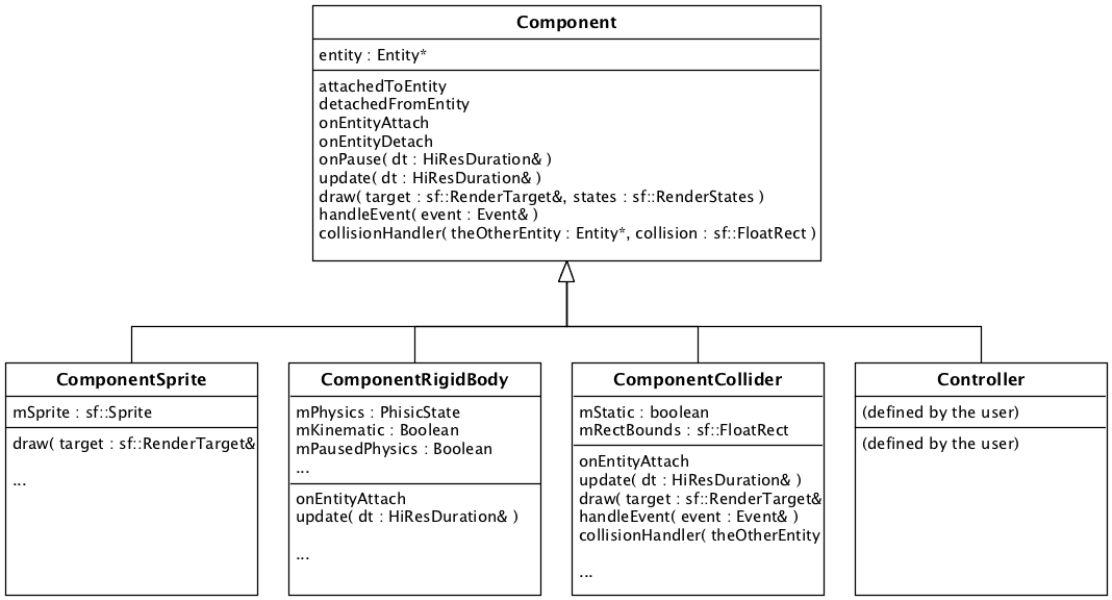

Figure 12: UML diagram of Component-derived classes.

3.3.3 ComponentSprite

The term *sprite* has already been explained along this work, but as a remainder: it is a graphical representation (in 2D) of some object of the game, such as the character. There is almost no game without graphics, and therefore a Component to implement such behaviour is compulsory.

SFML already provides this functionality with the `sf::Sprite` class, so the ComponentSprite class is a simple wrapper of it. Its interface is fairly simple: A private `sf::Sprite` of which some properties can be accessed through some getters and setters, two different constructors which vary in the number of arguments, and an override of the `draw` method.

```
1 // ComponentSprite.hpp
```

```
2
3  class ComponentSprite : public Component
4  {
5      // Methods
6  public:
7      ComponentSprite(const sf::Texture& texture);
8      ComponentSprite(const sf::Texture& texture, sf::IntRect textureRect);
9      ~ComponentSprite();
10
11     void                    draw(sf::RenderTarget& target, sf::RenderStates states) const
        override;
12
13     sf::FloatRect           getGlobalBounds();
14     sf::Color               getColor();
15     const sf::Texture&      getTexture();
16     sf::IntRect             getTextureRect();
17
18     void                    setColor(sf::Color color);
19     void                    setTexture(sf::Texture& texture);
20     void                    setTextureRect(sf::IntRect textureRect);
21
22
23     // Variables (member / properties)
24 private:
25     sf::Sprite              mSprite;
26 };
```

The `sf::Texture` argument of both constructors and the setter represents the texture —the picture from which the sprite will draw pixels—. One of the constructors and a setter expect a `sf::IntRect`, which is an SFML data structure to hold a rectangle (a point and a size), and in this case represents the *portion of the texture*[19] which will be used to render the sprite. If no texture rect is given, the entire texture is used for rendering the sprite. More advanced implementations could configure the texture of the sprite from a configuration file generated by a texture-packing software.

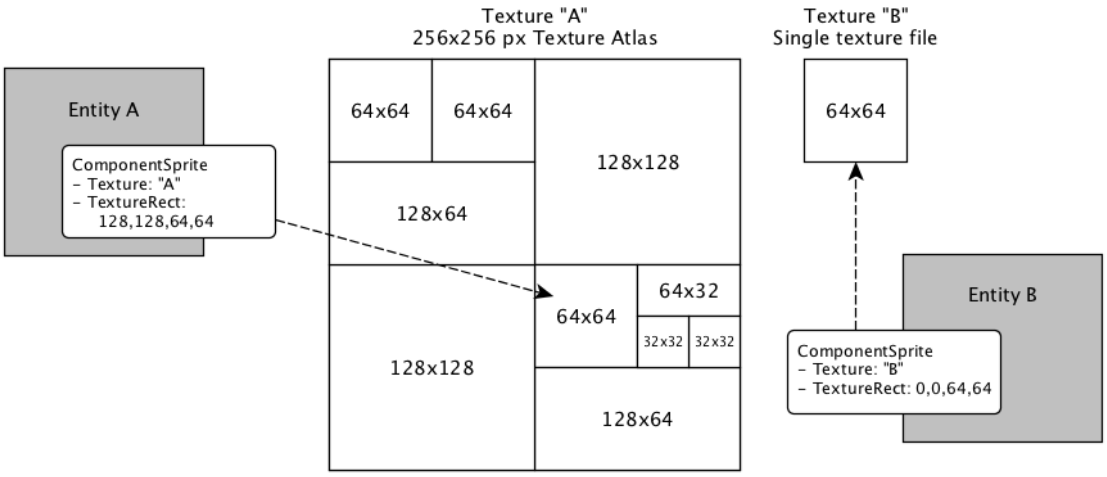

Figure 13: Texture atlas and single texture files.

Finally, `getGlobalBounds` return a rectangle in which the sprite is contained,

[19]In games development is very common to optimize the GPU operations by putting many textures together in the same file —called *texture atlas*, *sprite sheet* or *tile set* depending on their contents— so that the texture is uploaded to the GPU memory only once. Then, each object hold the coordinates and the size of the desired texture in the texture atlas.

and the `-Color` getter and setter let the user to access and modify the tint colour of the sprite —a colour which will be added to the *texels*[20]

The implementation will not be shown here as it simply consists on calling the same setters and getters of `sf::Sprite` on the `mSprite` private variable, excepting the `draw` method that finally uses the `sf::RenderTarget&` reference that has been passed around all these classes to render the sprite:

```
// ComponentSprite.cpp

void ComponentSprite::draw(sf::RenderTarget& target, sf::RenderStates states) const
{
    // Draw the sprite on the render target
    target.draw(mSprite, states);
}
```

3.3.4 ComponentRigidBody

This component can be added to an Entity with needs for physics simulation. That is, the Entity will get its position updated according to several physics properties such as mass, velocity and force. Such physics properties are represented by the PhysicState class (see Appendix 6.5). Besides, if the Entity also holds a ComponentCollider —the class is explained on the next section—, the PhysicsEngine (see Appendix 6.6) will track it and call its `collisionHandler` method if a collision occurs.

```
// ComponentRigidBody.hpp

class ComponentRigidBody : public Component
{
    // Methods
public:
    ComponentRigidBody(bool isKinematic);
    ~ComponentRigidBody();

    void                onEntityAttach() override;
    void                update(const HiResDuration& dt) override;

    void                returnToLastPhysicsState();

    PhysicState&        getPhysicsState();

    void                pausePhysics();
    void                resumePhysics();

    // Variables (member / properties)
private:
    bool                mKinematic; // If true, the Entity's position does not get affected by
     physics, but still triggers collisions
    bool                mPausedPhysics;

    PhysicState         mPhysics;

    PhysicState         mLastPhysicsState;
    sf::Transformable   mLastTransformable;
};
```

[20]Texture elements, similarly to the concept of *pixel* or picture elements, are the individual coloured square-dots that conform the texture.

The state of this class is primarily defined by the PhysicState `mPhysics` variable. There are also two boolean `mKinematic` and `mPausedPhysics` properties which indicate, respectively, if the rigid body is *kinematic* —this means the physics simulation will not be performed, normally because the Entity will be manually moved; but still can trigger collisions—, or if the simulation has been manually paused and therefore it should not be updated. The two remaining variables are another PhysicState called `mLastPhysicsState` and a `sf::Transformable` called `mLastTransformable` whose purpose is to store the previous PhysicState and transform, needed for the `returnToLastPhysicsState` method to work.

With regard to its behaviour: the constructor take an argument to indicate whether the rigid body is kinematic or not, and also set a default gravity force; `returnToLastPhysicsState` is an experimental method that returns the Entity to its previous physics state and transform; `getPhysicsState` returns a reference of `mPhysics` from which physics properties can be accessed and modified; and `pausePhysics` and `resumePhysics` methods are used to pause or resume the physics simulation. See the implementation:

```
1  // ComponentRigidBody.cpp
2
3  ComponentRigidBody::ComponentRigidBody(bool isKinematic)
4  : mKinematic(isKinematic)
5  , mPausedPhysics(false)
6  {
7      mPhysics.setForce(sf::Vector2f(0.f, 9.8f*20)); // Default force simulating gravity
8  }
9
10 void ComponentRigidBody::returnToLastPhysicsState()
11 {
12     mPhysics = mLastPhysicsState;
13     entity->mTransformable = mLastTransformable;
14 }
15
16 PhysicState& ComponentRigidBody::getPhysicsState()
17 {
18     return mPhysics;
19 }
20
21 void ComponentRigidBody::pausePhysics()
22 {
23     if (mKinematic)
24         DBGMSGC("Warning: Tried to pause the physics simulation of a kinematic
       ComponentRigidBody.");
25     else
26         mPausedPhysics = true;
27 }
28
29 void ComponentRigidBody::resumePhysics()
30 {
31     if (mKinematic)
32         DBGMSGC("Warning: Tried to resume the physics simulation of a kinematic
       ComponentRigidBody.");
33     else
34         mPausedPhysics = false;
35 }
```

The class also overrides two methods from Component, `onEntityAttach` and `update`. The first method checks if the Entity does already have a ComponentCollider in order to set it as *dynamic* —this will be explained on next section—. The latter method will advance the physics state and transform according to the

physics simulation, unless the ComponentRigidBody is marked as kinematic or its physics simulation is paused. That physics step is done with the help of the RK4 integration method of the PhysicsEngine class (see Appendix 6.6).

```
1  // ComponentRigidBody.cpp
2
3  void ComponentRigidBody::onEntityAttach()
4  {
5      // Check if the Entity has a collider attached
6      if (auto collider = entity->getComponent<ComponentCollider>()) {
7          collider->setStatic(false); // If it existed, make it dynamic now
8      }
9      else {
10         DBGMSGC("WARNING: a RigidBody has been attached to " << entity->getName() << " but it has
       no Collider. This may cause undesired behaviour.");
11     }
12 }
13
14 void ComponentRigidBody::update(const HiResDuration &dt)
15 {
16     // Do not update physics if is kinematic or physics simulation is paused
17     if (mKinematic || mPausedPhysics)
18         return;
19
20     // Advance the physics with RK4 integration
21     PhysicsEngine::integrateRK4(mPhysics, mLastPhysicsState, entity->mTransformable,
       mLastTransformable, dt);
22 }
```

Note that this class offers very basic functionality and has much room for improvement. For example, an easier API for accessing and operating with the physics; an override of the `collisionHandler` method for an automatic collision response modifying the PhysicState accordingly; or even improve its integration with the Scene's JSON loading so that more properties could be loaded from file.

3.3.5 ComponentCollider

The ComponentCollider class defines a component which can be added to an entity in order to give it a *collider* —a determined region of the space that reacts in a specific manner when another collider overlaps, or *collides*, with it—. The PhysicsEngine will periodically check for collisions and call `collisionHandlers`. The Collider can be *static* or *dynamic* depending on the use of the Entity:

- A ComponentCollider is marked as *static* (`mStatic = true`) if the Entity has no ComponentRigidBody. The main use of these type of colliders is for static elements of the game, such as the floor or some walls. Another common use is for "trigger areas", e.g. the player enters a specific room and a cutscene should start at that event. The PhysicsEngine will not check collisions between static colliders in order to optimize the collision detection process.

- A ComponentCollider is marked as *dynamic* (`mStatic = false`) if the Entity already holds a ComponentRigidBody. The main use of dynamic colliders is for dynamic elements of the game, such as the player, the enemies, bullets, etc. that may need to react when collisions happen between them (e.g. the player looses life if a bullet collides with him, and the bullet destroys). The PhysicsEngine will check collisions between dynamic colliders

(e.g. a bullet hits the player), and between dynamic and static colliders (e.g. the player walks over the floor, touches a wall, etc.).

The area of the collider is represented by an `sf::FloatRect`, which is a rectangle.

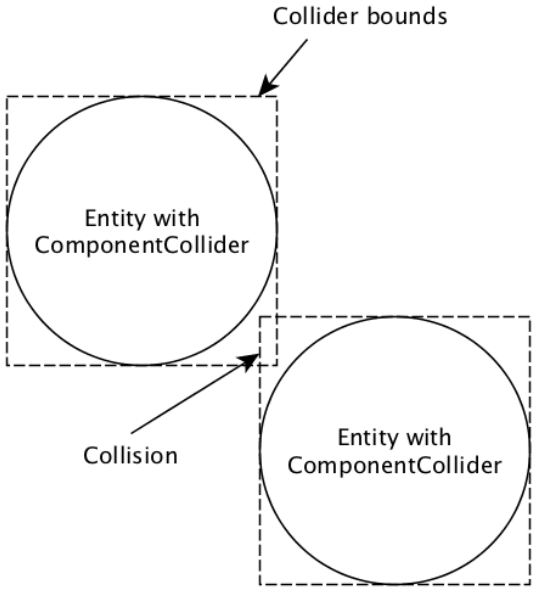

Figure 14: Two Entities with ComponentColliders.

As many other parts of X-GSD, this class is simple and enough for the scope of this work, with much room for improvement. For example, one could add circle-based colliders, composite colliders based on the composition of rectangles and circles, etc.

```
1  // ComponentCollider.hpp
2
3  class ComponentCollider : public Component
4  {
5      // Methods
6  public:
7      ComponentCollider(sf::FloatRect rectBounds = sf::FloatRect());
8      ~ComponentCollider();
9
10     void                onEntityAttach() override;
11     void                update(const HiResDuration& dt) override;
12     void                draw(sf::RenderTarget& target, sf::RenderStates states) const override;
13     void                handleEvent(const Event& event) override;
14
15     void                collisionHandler(Entity* theOtherEntity, sf::FloatRect collision)
        override;
16     void                setStatic(bool option);
17     bool                isStatic();
18     void                setRectBounds(sf::FloatRect rect);
19     sf::FloatRect       getRectBounds();
20
21     // Variables (member / properties)
22 private:
23     bool                mStatic;
24     sf::FloatRect       mRectBounds;
25
26     ...
27 };
```

Regarding the implementation, it is fairly simple: `update`, `draw`, `handleEvent` and `collisionHandler` overridden methods only perform debug actions, and therefore they have been omitted. That debug functionality includes a visual representation of the collider (see source code for details). The rest of methods are simple getters/setters or their implementation can be understood trough the comments in code:

```
// ComponentCollider.cpp

void ComponentCollider::onEntityAttach()
{
    // Check if the entity has a RigidBody component or not and decide the type of collider
     (static or dynamic). Add a pointer of this collider to the corresponding PhysicsEngine's
     collection of collider pointers.
    if (entity->getComponent<ComponentRigidBody>()) {
        Game::instance().getPhysicsEngine().addDynamicCollider(this);
        mStatic = false;
    }
    else {
        Game::instance().getPhysicsEngine().addStaticCollider(this);
        mStatic = true;
    }

    // Check if the entity has a Sprite component and mRectBounds has not been set yet
    if (mRectBounds == sf::FloatRect()) {
        if (auto sprite = entity->getComponent<ComponentSprite>()) {
            // Set the collider bounds to the sprite's
            mRectBounds = sprite->getGlobalBounds();
            ...
        }
    }
    ...
}

void ComponentCollider::setStatic(bool option)
{
    // If mStatic is equal to option, there is no need to do anything as the Collider is already
     the desired type
    if (mStatic != option) {

        // Swap the collider from one type to the other
        if (option) {
            Game::instance().getPhysicsEngine().addStaticCollider(this);
            Game::instance().getPhysicsEngine().deleteDynamicCollider(this);
        }
        else {
            Game::instance().getPhysicsEngine().addDynamicCollider(this);
            Game::instance().getPhysicsEngine().deleteStaticCollider(this);
        }
    }

}

ComponentCollider::~ComponentCollider()
{
    // Cleanup
    if (mStatic)
        Game::instance().getPhysicsEngine().deleteStaticCollider(this);
    else
        Game::instance().getPhysicsEngine().deleteDynamicCollider(this);
}
```

3.3.6 Controllers: User-defined Components

One with access to X-GSD's source code could implement new types of components inheriting from Component, overriding the needed methods and providing an im-

plementation. In fact, in order to give Entities some custom and specific behaviour —the *script*—, e.g. the logic of the game or the controls of the player, it should be done in a Component. Then that Component should be added to the desired Entities. However, that procedure is not ideal for a user of the engine, because it would require him/her to access the engine's source code and make some modifications, which would contradict some of the goals of a game engine such as making the user to forget about the inner game engine implementation details and focus on the game. For example, the `loadSceneFromFile` method of the Scene class would require specific code in order to load the new Component-derived class from JSON.

Controllers are user-defined components that can integrate with the engine without the need of that much trouble for the user. In order to provide the controllers integration with the engine, a ControllersManager class is necessary:

```
1  // ControllersManager.hpp
2
3  #pragma once
4
5  #include <X-GSD/Component.hpp>
6
7  #include <unordered_map>
8  #include <stdexcept>
9
10 namespace xgsd {
11
12     class ControllersManager
13     {
14         // Methods
15     public:
16         template <typename T>
17         void            registerController(std::string controllerName);
18
19         Component::Ptr  getController(std::string controllerName);
20
21         // Variables (member / properties)
22     private:
23         std::unordered_map<std::string, std::function<Component::Ptr()>> mControllerFactories;
24     };
25
26
27     // Template implementation
28     template <typename T>
29     void ControllersManager::registerController(std::string controllerName)
30     {
31         mControllerFactories[controllerName] = [] ()
32         {
33             return std::move(Component::Ptr(new T()));
34         };
35     }
36
37 } // namespace xgsd
38
39
40 // ControllersManager.cpp
41
42 #include <X-GSD/ControllersManager.hpp>
43
44 using namespace xgsd;
45
46 Component::Ptr ControllersManager::getController(std::string controllerName)
47 {
48     auto found = mControllerFactories.find(controllerName);
49     if(found == mControllerFactories.end())
50         throw std::runtime_error("Attempt to load an unknown controller: " + controllerName);
51
52     return std::move(found->second()); // Call the needed functor to constructor
53 }
```

This class contains methods to register and create ControllerComponents in a dynamic manner. The `registerController` template method can be used to store a pointer to the constructor method of that type of controller associated with a name string. Then, `getController` can be used to create controllers of the desired type.

There is another class, ControllersRegistration, that is provided with X-GSD but that should be with the game's project files. Is inside this class where the user should register their controllers:

```
// ControllersRegistration.hpp

#pragma once

#include  <X-GSD/ControllersManager.hpp>

// Include all controllers
#include "MenuController.hpp"
#include "Level1Controller.hpp"
#include "PlayerController.hpp"
#include "EnemyController.hpp"
...

class ControllersRegistration {

public:
  void registerControllers(xgsd::ControllersManager& controllersManager) {

    // Register the controllers
    controllersManager.registerController<MenuController>("MenuController");
    controllersManager.registerController<Level1Controller>("Level1Controller");
    controllersManager.registerController<PlayerController>("PlayerController");
    controllersManager.registerController<EnemyBallController>("EnemyController");
        ...
  }

};
```

The Scene class will call the `registerControllers` method with a controllersManager object from the Game class, and therefore they will be accessible from anywhere after that (including the scene loading from file method, so controllers can be added to entities in scene JSON files):

```
// Some-class.cpp

auto controllersManager = Game::instance().getControllersManager();
auto playerController = controllersManager.getController("PlayerController");
anEntity->addComponent(std::move(playerController));

// Or in one line:
anEntity->addComponent(Game::instance().getControllersManager().getController("PlayerController"));
```

Finally, there is a header file which packs the most common headers needed by controllers so that a user can include only that header and start coding:

```
// ControllersHeaders.hpp

#pragma once

#include <X-GSD/Component.hpp>
#include <X-GSD/ComponentSprite.hpp>
```

```
7  #include <X-GSD/ComponentRigidBody.hpp>
8  #include <X-GSD/ComponentCollider.hpp>
9  #include <X-GSD/Entity.hpp>
10 #include <X-GSD/Event.hpp>
11 #include <X-GSD/Game.hpp>
12 #include <X-GSD/Debug.hpp>
13 #include <X-GSD/Time.hpp>
14
15 #include <SFML/Graphics.hpp>
16 #include <SFML/Audio.hpp>
17 #include <SFML/System/Vector2.hpp>
```

This solution is far from ideal and lacks some features such as being able to configure some values of the controllers in scene JSON files or make modifications to scripts without recompiling. Dynamic libraries were considered, but their compatibility through different platforms and operating systems depend on their native APIs, and therefore the decision was not to use dynamic libraries for simplicity. However, one could implement that feature, or rely in a library like *Poco.sharedLibrary*[28]

With this, all aspects and elements of X-GSD have been finally covered.

3.4 Example game

The next section will explain step by step how to create an example game with X-GSD. Here is a list of the steps to be done:

1. Install dependencies and tools (SFML, IDE, compiler, etc.)
2. Create a new project (directory structure, X-GSD files, etc.)
3. Modify the contents of gameconfig.json
4. Create a TitleScene.json file and add the necessary resources
5. Create and implement a `TitleSceneController`
6. Modify the `ControllersRegistration.hpp` after each Controller creation
7. Create a GameScene.json file and add the necessary resources
8. Create and implement a `GameSceneController`
9. Create and implement a `PlayerController`
10. Create and implement an `AsteroidController`
11. Build and run!

First of all, SFML must be installed as it is the main dependency of X-GSD. There is a learn[7] section in the SFML website with instructions for installing the library and configure a IDE/compiler. X-GSD is developed with SFML 2.1, but there should not be a problem to use newer versions. However, if that is the case, it can be solved installing SFML 2.1, or looking into the error messages and trying

[7] Laurent Gomila. SFML. `http://www.sfml-dev.org/learn.php`.

to fix them.

The project creation depends on the platform, IDE and personal preferences. The tutorials in the SFML website also offer project templates for some IDEs in order to create new SFML projects easily. Once the project is created, the X-GSD files must be referenced or copied into the project. The recommended directory structure is the following:

```
GameProject
├── include/
│   ├── X-GSD/
│   │   └── X-GSD header files
│   ├── Game/
│   │   ├── Controllers/
│   │   │   └── User Controllers header files
│   │   └── ControllersRegistration.hpp
│   └── ExternalLibraries
│       ├── json
│       │   └── JsonCpp header files
│       └── Add more libraries if needed
│
├── src/
│   ├── X-GSD/
│   │   └── X-GSD source files
│   ├── Game/
│   │   ├── Controllers/
│   │   │   └── User Controllers source files
│   │   └── Main.cpp
│   └── ExternalLibraries
│       ├── json
│       │   └── JsonCpp source files
│       └── Add more libraries if needed
│
├── resources/
│   ├── configuration/
│   │   └── gameconfig.json
│   ├── fonts/
│   │   └── Font files
│   ├── images/
│   │   └── Image files
│   ├── sounds/
│   │   └── Sound files
│   ├── scenes/
│   │   └── Scene JSON files
│   └── Other needed resources
│
└── Other files needed for the project (ResourcePath.hpp)
```

If the decision was to reference X-GSD, it is still recommended to copy the `ControllersRegistration.hpp` file into `GameProject/include/Game/` and the `Main.cpp` file into `GameProject/src/Game` and use these. If not, modifying these would result in side effects for other projects using X-GSD also in referenced mode.

The next step is to create the `gameconfig.json` file in `GameProject/resources/configuration/`. For the example, this configuration will be used:

```
{
  "windowName" : "X-GSD Example Game",
  "windowSize" : {
    "width" : 720,
    "height" : 480
  },
```

```
  "fullscreen" : false,
  "vsync" : false,
  "keyRepetition" : false,
  "mouseCursorVisible" : false,
  "debugFont": "PressStart2P.ttf",
  "icon" : "playerShip.gif",
  "initialScene" : "TitleScene.json"
}
```

This configuration file will make the game to run in windowed mode of resolution 720x480, without vsync (if the OS allows that), no key repetition, invisible mouse cursor, a custom icon, a font for debug information on screen and an initial scene called `TitleScene.json`. The needed resources (the icon, font and scene) must be placed correctly in their directories as explained before.

The initial scene `TitleScene.json` must be created and added to the scenes directory:

```
{
  "name": "TitleScene",
  "entities": [
    {
      "name": "levelControllersContainer",
      "components": {
        "controllers": [
          {
            "type": "TitleController"
          }
        ]
      }
    },
    {
      "name": "logo",
      "parentEntityName": "root",
      "transform": {
        "origin": {
          "x": 128,
          "y": 128
        },
        "position": {
          "relative": true,
          "x": 0.5,
          "y": 0.35
        },
        "scale": {
          "x": 2,
          "y": 2
        }
      },
      "components": {
        "ComponentSprite": {
          "globalTexture": false,
          "texture": "titleLogoTexture"
        },
        "controllers": []
      }
    }
  ],
  "resources": {
    "textures": [
      {
        "name": "titleLogoTexture",
        "path": "logoSprite.png"
      }
    ],
    "fonts": [
      {
        "name": "mainFont",
```

```
        "path": "PressStart2P.ttf"
      }
    ],
    "sounds": [
     {
        "name": "selectionSound",
        "path": "selection.ogg"
     }
    ]
  }
}
```

This file declares some dependencies: A texture, a font, a sound and a Controller of type `TitleController`. All these must be added to the appropriate directories.

The the header and implementation files of the `TitleController` are the following:

```
// TitleScene.hpp

#pragma once

#include <X-GSD/ControllerHeaders.hpp>

using namespace xgsd;

/*
 Controller for the logic of the TitleScene.
 */
class TitleMenuController : public Component
{
  // Typedefs and enumerations
private:


  // Methods
public:
  TitleMenuController();
  ~TitleMenuController();

  void          onEntityAttach() override;
  void          update(const HiResDuration& dt) override;
  void          draw(sf::RenderTarget& target, sf::RenderStates states) const override;
  void          handleEvent(const Event& event) override;

  // Variables (member / properties)
private:
  sf::Text        mTextPressAnyKey;
  sf::RectangleShape    mBackground;
  sf::Sound       mMenuSound;
  HiResDuration     mPressAnyKeyTime;
};
```

```
#include "TitleController.hpp"


TitleMenuController::TitleMenuController()
: mBackground()
{
    // Load resources here (RAII)
}

void TitleMenuController::onEntityAttach()
{
    // Sound
    mMenuSound.setBuffer(Game::instance().getLocalSoundManager().get("selectionSound"));
```

```
    // Font
    sf::Font& mainFont = Game::instance().getLocalFontManager().get("mainFont");
    mTextPressAnyKey.setFont(mainFont);

    // String
    mTextPressAnyKey.setString("Press any key to start");

    // Size
    mTextPressAnyKey.setCharacterSize(14);

    // Positioning
    sf::Vector2f viewSize = Game::instance().getWindow().getView().getSize();
    sf::FloatRect textRect = mTextPressAnyKey.getLocalBounds();
    mTextPressAnyKey.setOrigin(textRect.left + textRect.width/2.0f,
                               textRect.top  + textRect.height/2.0f);
    mTextPressAnyKey.setPosition(viewSize.x / 2.0f, viewSize.y / 1.2f);

    // Color
    mTextPressAnyKey.setColor(sf::Color::Transparent);

    // Background
    mBackground.setSize(viewSize);
    mBackground.setFillColor(sf::Color(50, 100, 200));
}

void TitleMenuController::update(const HiResDuration& dt)
{
    // Update "Press any key" blinking animation
    mPressAnyKeyTime += dt;
    if (mPressAnyKeyTime > ONE_SECOND / 2) {
        if (mTextPressAnyKey.getColor() == sf::Color::White)
            mTextPressAnyKey.setColor(sf::Color::Transparent);
        else
            mTextPressAnyKey.setColor(sf::Color::White);
        mPressAnyKeyTime = xgsd::HiResDuration(0);
    }
}

void TitleMenuController::draw(sf::RenderTarget &target, sf::RenderStates states) const
{
    // Draw background
    target.draw(mBackground, states);

    // Draw text on top of everything
    target.draw(mTextPressAnyKey, states);
}

void TitleMenuController::handleEvent(const Event& event)
{

    if (event.type == Event::System) {

        auto systemEvent = event.systemEvent;

        // Perform appropiate actions
        if (systemEvent.type == sf::Event::KeyPressed ||
            systemEvent.type == sf::Event::JoystickButtonPressed)
        {

            // Pressing Esc will close the game
            if (systemEvent.key.code == sf::Keyboard::Escape) {
                Game::instance().getWindow().close();
                return;
            }

            // Any other key or button:
            mMenuSound.play();

            // Load Game scene
            Game::instance().getSceneManager().loadSceneFromFile("GameScene.json");

        }
    }
}
```

```
TitleMenuController::~TitleMenuController()
{
    // Cleanup
}
```

This Controller must be added to `ControllersRegistration.hpp`

```
#pragma once

#include  <X-GSD/ControllersManager.hpp>

// Include all of your controllers
#include "TitleController.hpp" // <- Added line

class ControllersRegistration {

public:
  void registerControllers(xgsd::ControllersManager& controllersManager) {

    // Register controllers this way
    controllersManager.registerController<TitleMenuController>("TitleController");  // <- Added
     line
  }

};
```

This is the result:

The following steps would be creating a JSON scene file for the game scene, implement a Controller for the logic of that scene, create and implement other two Controllers for the player and the enemies, modify the `ControllersRegistration.hpp` file to include these and finally build and run the game.

Figure 15: Example game running with debug rendering activated.

The code for the remaining steps will be omitted for simplicity. It can be found in the attached files of this work and the GitHub repository[2].

4 Conclusion

The game development field is relatively new and continuously evolving. With each new game and with every new technology or platform, many new programming challenges emerge. Game engines can help developers to worry less about technologies and platforms and focus more on the development of the game, working on a higher level of abstraction. However, being bound to a closed third-party game engine can be a problem —the reasons were explained at the beginning of this work—. Besides, the fact that game engines handle the low level details does not mean the programmer should not know about them: After all, a programmer who has understandings of the game engine internals will probably do better than a programmer who does not.

In one hand, the exercise of creating a game engine can help a programmer to understand how game engines work, to think of the vast amount of possibilities behind every single feature and to reason about each decision-making. In that sense it is clear that facing such challenge has a beneficial learning value for the programmer.

On the other hand, to create a game engine with plans for using it in real-world projects in mind can also be dangerous if the magnitude of the project and required features are not measured properly, or the team is too small or inexpert in this field, because it could lead to undesired results or be too much time-consuming.

One could consider creating its own game engine to meet the requirements of some project for several reasons. But if the game to be developed needs cutting-edge features and the team is not capable of developing them or it is not worthwhile, it may be wiser to use an industry-proven game engine. If that is the decision, one should then preferably look for an open-source engine so that one could modify it, be it for features addition, error fixing or even being able to react at some unfortunate event such as those listed at the introduction of this work without having to rely on external factors.

All along this work, a simple game engine has been successfully developed with the goals of cross-platform targeting, data-oriented programming, free software and simplicity, although it does offer few features with much room for improvement. Because of these and other reasons explained in the methodology section, C++, SFML and JSON were selected to develop the engine, X-GSD.

The structure of the engine consist of a continuous simulation loop combined with a simple scene graph implemented by a tree structure and a scene-entity-component organization hierarchy. Such structure can be seen as a mix of the one proposed at the *SFML Game Development Book*[14] and a organization hierarchy similar to some popular engines as Unity, which would be easy to understand and result familiar to some of the users of those engines.

Due to the dangers of creating a game engine stated above, some of the initial plans were cancelled: more components such as GUI-related ones and more features for the existing components as automatic reaction to collisions for ComponentRigidBody.

The game engine is working and available for free on a GitHub repository[2] with the hope of being useful for those interested in learning about games programming, or even grow with the contributions of different people.

5 Improvement proposals

Aiming to build a full-featured game engine by oneself is both hard and unrealistic. Even with this taken into account, several features were implemented in a very basic manner and others were cancelled due to lack of time. Besides, even the most complete engines continue evolving, adding features, enhancing several parts and fixing many errors.

With this in mind, a list of improvement proposals —with no specific order— has been elaborated:

- Graphical editor: Because positioning elements with a scene editor is always clearer and easier. The editor would read and write scene JSON files with the format presented in this work or an enhanced one.
- Provide more built-in Components: At least basic components such as GUI-related (texts, buttons, etc.), light sources and particle emitters.
- Enhance the ComponentSprite, ComponentCollider and ComponentRigidBody: There are details on what parts could be enhanced in their sections.
- Actions and callbacks for Entities, with easing options.
- Input bindings instead of hard-coded values for the controls of the game.
- Add a Z-index property to SceneGraphNodes. The renderer should get a collection of drawable references in the desired order of rendering. In order for this to work, drawables should be decoupled from the scene graph (may be in a separate binary tree, with references between them).
- Enhance the physics engine or integrate an open-source solution.
- Enhance the resources managing system (see [8]).
- Let the user to specify options and values of their controllers in scene JSON files (see [9]).
- Introduce a runtime-compiled C++ system (see [10] and [11] or a scripting language compatibility such as LUA or JavaScript for faster development iterations.
- Introduce threading/multi-core features (SFML does provide a Thread class).

[8] Sean Middleditch. Dangers of std::shared_ptr. `http://seanmiddleditch.com/dangers-of-stdshared_ptr/`.

[9] Stefan Reinalter. Schema-based entity-component data for fast iteration times. `https://molecularmusings.wordpress.com/2014/02/21/schema-based-entity-component-data-for-fast-iteration-times/#more-484`.

[10] Doug Binks, Adam Rutkowski, Matthew Jack, and Juliette Foucaut. Runtime-compiled c++. `http://runtimecompiledcplusplus.blogspot.co.uk/`.

[11] Stefan Reinalter. Using runtime-compiled c++ code as a scripting language: under the hood. `https://molecularmusings.wordpress.com/2014/05/10/using-runtime-compiled-c-code-as-a-scripting-language-under-the-hood/#more-487`.

- Network-related features (SFML also provides networking classes and utilities).

6 Appendix

Full source code is available as attached files of this work or on the GitHub repository[2].

6.1 Debug utils

This header file contains macros (based on a Late Developer's blog article[5]) for making debugging messages easier and only available if the `DEBUG` preprocessor symbol is defined. This is useful for *release* builds in which that symbol should be undefined so that debug instructions are not included in the final binary.

```
1  // Debug.hpp
2
3  #pragma once
4
5  #include <iostream>
6  #include <string.h>
7
8  #define INFO_MAX_LENGTH 40
9
10 // Macro for shortening the __FILE__ result. It hides the path and only shows the file name
11 #define FILE_WITHOUT_PATH (strrchr(__FILE__, '/') ? strrchr(__FILE__, '/') + 1 : __FILE__)
12
13 #define DEBUG_INFO(os) { std::string info = "DBG: "; info.append(FILE_WITHOUT_PATH);
       info.append("("); \
14 info.append(std::to_string(__LINE__)); info.append("): "); \
15 (os) << info; \
16 int infoLength = (int)info.length(); \
17 for (int i = 0; i < INFO_MAX_LENGTH - infoLength; i++){ (os) << " ";}\
18 }\
19
20 // Macro for debugging a certain variable to a specific output stream
21 #ifdef DEBUG
22 #define DBGVAR( os, var ) do { \
23 DEBUG_INFO(os)\
24 (os) << #var << " = [" << (var) << "]" << std::endl; } while( false )
25 #else
26 #define DBGVAR( os, var ) do { } while ( false )
27 #endif
28
29 // Macro for debugging with a message to a specific output stream
30 #ifdef DEBUG
31 #define DBGMSG( os, msg ) do { \
32 DEBUG_INFO(os)\
33 (os) << msg << std::endl; } while( false )
34 #else
35 #define DBGMSG( os, msg ) do { } while ( false )
36 #endif
37
38 // Macro for debugging a certain variable to standard output stream
39 #ifdef DEBUG
40 #define DBGVARC( var ) do { \
41 DEBUG_INFO(std::cout)\
42 std::cout << #var << " = [" << (var) << "]" << std::endl; } while( false )
43 #else
44 #define DBGVARC( var ) do { } while ( false )
45 #endif
46
47 // Macro for debugging with a message to standard output stream
48 #ifdef DEBUG
49 #define DBGMSGC( msg ) do { \
50 DEBUG_INFO(std::cout)\
51 std::cout << msg << std::endl; } while( false )
52 #else
53 #define DBGMSGC( msg ) do { } while ( false )
54 #endif
```

6.2 Time utils

This utility header file is for use in other X-GSD classes that require time handling. Defines some short names for `std::chrono` types:

- `HiResTime` defines a time point of a high resolution clock.
- `HiResDuration` is a time duration in nanoseconds. Used to store the difference between two `HiResTime` points.
- `HiResClock` is a high resolution clock that can be used to get the time and store it in `HiResTime` variables.
- `ONE_MILLISECOND`, `ONE_SECOND`, `ONE_MINUTE` and `ONE_HOUR` are time durations that can be used directly or mathematically operated to specify time durations.

```
1  // Time.hpp
2
3  #pragma once
4
5  #include <chrono>
6
7  /* This header defines some shortcurts for types of high resolution clock in std::chrono */
8
9  // One second, minute and hour in nanoseconds. You can operate with them to get the desired time
10 // Example: ONE_SECOND * 5 is 5 seconds, ONE_HOUR / 2 is 30 minutes
11 #define ONE_MILLISECOND
       (std::chrono::duration_cast<std::chrono::nanoseconds>(std::chrono::milliseconds(1)))
12 #define ONE_SECOND (std::chrono::duration_cast<std::chrono::nanoseconds>(std::chrono::seconds(1)))
13 #define ONE_MINUTE (std::chrono::duration_cast<std::chrono::nanoseconds>(std::chrono::minutes(1)))
14 #define ONE_HOUR (std::chrono::duration_cast<std::chrono::nanoseconds>(std::chrono::hours(1)))
15
16 namespace xgsd {
17
18     // std::chrono typedefs
19     typedef std::chrono::time_point<std::chrono::high_resolution_clock> HiResTime;
20     typedef std::chrono::nanoseconds HiResDuration;
21     typedef std::chrono::high_resolution_clock HiResClock;
22
23 } // namespace xgsd
```

6.3 Event wrapper class and CustomEvent

This Event class is a wrapper class of `sf::Event` which adds a `CustomEvent` type with two basic string fields: name (compulsory) and data (optional). They can be used to broadcast custom events or messages to the Entities in the scene graph. For example, an enemy can broadcast an *enemyDestroyed* event at its destruction so that any interested Entity can take actions or just ignore the event, such as a *levelController* that would decrement an enemies counter and win the game if that counter reaches zero.

```
1  // Event.hpp
2
3  #pragma once
4
5  #include <SFML/Window/Event.hpp>
6
```

```
7  namespace xgsd {
8
9      class Event
10     {
11     public:
12         ~Event() {}
13
14         struct CustomEvent
15         {
16             CustomEvent(std::string name, std::string data = "") : name(name), data(data) { }
17
18             std::string name;
19             std::string data;
20         };
21
22         enum EventType
23         {
24             System,
25             Custom
26         };
27
28         // Constructors
29         Event(sf::Event systemEvent) : type(System), systemEvent(systemEvent) { }
30         Event(CustomEvent customEvent) : type(Custom), customEvent(customEvent) { }
31
32         // Type of the event
33         EventType type;
34
35         union {
36             sf::Event   systemEvent;
37             CustomEvent customEvent;
38         };
39
40     };
41
42 } // namespace xgsd
```

6.4 ResourceManager class

Modified version from the *SFML Game Development Book*[14].

This class template is useful to store any type of resources that implement a `loadFromFile(std::string path)` method and give them identifiers. A STL map is used in combination with smart pointers to store the resources, and therefore memory management is safe and transparent to the user.

```
1  // ResourceManager.hpp
2  // Modified version of ResourceHolder.hpp from SFML Game Development Book
3
4  #pragma once
5
6  /*
7   ResourcePath.hpp is not provided with X-GSD because its
8   implementation is OS-dependent.
9   You can easily create your own as an envelop of the OS specific
10  function to reach the resource folder.
11  You can also use the implementation provided with SFML templates
12  for specific platforms-OSs (for example, they provide one for
13  Xcode in their downloads section of SFML webpage).
14  */
15 #include "ResourcePath.hpp"
16
17 #include <SFML/Graphics/Font.hpp>
18 #include <SFML/Graphics/Texture.hpp>
19 #include <SFML/Audio/SoundBuffer.hpp>
20
```

```
#include <map>
#include <string>
#include <memory>
#include <stdexcept>
#include <cassert>

namespace xgsd {

    /* Class template to store and manage resources: Textures, sounds, fonts, etc.

     You can create instances of this resource manager, each of
     which will store their own resources and have no relation or
     communication.

     Local version instances at scenes ensures that each scene will load / free
     from memory their required resources on creation / destruction
     of the scene (RAII), while the general (Game's instances) resources will last
     until the game ends or manual unload is invoked.
     */
    template <typename Resource, typename Identifier>
    class ResourceManager
    {
    public:

        typedef std::unique_ptr<ResourceManager<Resource, Identifier>> Ptr;

        void            load(Identifier id, const std::string& filename);

        template <typename Parameter>
        void            load(Identifier id, const std::string& filename, const Parameter&
     secondParam);

        void            unload(Identifier id);

        Resource&       get(Identifier id);
        const Resource& get(Identifier id) const;

    private:
        void            insertResource(Identifier id, std::unique_ptr<Resource> resource);

    private:
        std::map<Identifier, std::unique_ptr<Resource>>         mResourceMap;
    };

    // Specific resource managers (textures, fonts,  audio...)
    typedef ResourceManager<sf::Texture, std::string>           TextureManager;
    typedef ResourceManager<sf::Font, std::string>              FontManager;
    typedef ResourceManager<sf::SoundBuffer, std::string>       SoundManager;



    ////////////////////////////
    // Template implementation //
    ////////////////////////////


    ////// LOAD //////

    template <typename Resource, typename Identifier>
    void ResourceManager<Resource, Identifier>::load(Identifier id, const std::string& filename)
    {
        // Create and load resource
        std::unique_ptr<Resource> resource(new Resource());
        if (!resource->loadFromFile(resourcePath() + filename))
            throw std::runtime_error("ResourceManager::load - Failed to load " + resourcePath() +
     filename);

        // If loading successful, insert resource to map
        insertResource(id, std::move(resource));
    }

    template <typename Resource, typename Identifier>
    template <typename Parameter>
    void ResourceManager<Resource, Identifier>::load(Identifier id, const std::string& filename,
     const Parameter& secondParam)
```

```
93      {
94          // Create and load resource
95          std::unique_ptr<Resource> resource(new Resource());
96          if (!resource->loadFromFile(resourcePath() + filename, secondParam))
97              throw std::runtime_error("ResourceManager::load - Failed to load " + resourcePath() +
        filename);
98
99          // If loading successful, insert resource to map
100         insertResource(id, std::move(resource));
101     }
102
103
104     ////// UNLOAD //////
105
106     /* Use unload for special resource management. For general purpose, see description at the
        beggining of this file. */
107     template <typename Resource, typename Identifier>
108     void ResourceManager<Resource, Identifier>::unload(Identifier id)
109     {
110         auto found = mResourceMap.find(id);
111         assert(found != (mResourceMap.end()));
112         mResourceMap.erase(found);
113     }
114
115
116     ////// GET //////
117
118     template <typename Resource, typename Identifier>
119     Resource& ResourceManager<Resource, Identifier>::get(Identifier id)
120     {
121         auto found = mResourceMap.find(id);
122         assert(found != mResourceMap.end());
123
124         return *found->second;
125     }
126
127     template <typename Resource, typename Identifier>
128     const Resource& ResourceManager<Resource, Identifier>::get(Identifier id) const
129     {
130         auto found = mResourceMap.find(id);
131         assert(found != mResourceMap.end());
132
133         return *found->second;
134     }
135
136     ////// INSERT //////
137
138     template <typename Resource, typename Identifier>
139     void ResourceManager<Resource, Identifier>::insertResource(Identifier id,
        std::unique_ptr<Resource> resource)
140     {
141         // Insert and check success
142         auto inserted = mResourceMap.insert(std::make_pair(id, std::move(resource)));
143         assert(inserted.second);
144     }
145
146 } // namespace xgsd
```

6.5 PhysicState class

This class defines a basic physics state, with velocity, force and mass. It is needed by the ComponentRigidBody class and the PhysicsEngine class.

```
1  // PhysicState.hpp
2
3  #pragma once
4
5  #include <SFML/System/Vector2.hpp>
```

```
#include <SFML/Graphics/Transformable.hpp>

#include <cassert>

namespace xgsd {

    // Needed struct by ComponentRigidBody and PhysicsEngine classes
    class PhysicState
    {
        // Methods
    public:
        PhysicState();// Constructor

        void                    setVelocity(sf::Vector2f velocity);
        void                    setForce(sf::Vector2f force);
        void                    setMass(float mass);

        sf::Vector2f            getVelocity() const;
        sf::Vector2f&           getVelocityRef();
        sf::Vector2f            getForce() const;
        float                   getMass() const;

        // Variables (member / properties)
    private:
        sf::Vector2f            mVelocity;
        sf::Vector2f            mForce;
        float                   mMass;
    };

} // namespace xgsd
```

The implementation:

```
// PhysicState.cpp

#include <X-GSD/PhysicState.hpp>

using namespace xgsd;

// Constructor
PhysicState::PhysicState()
: mVelocity()
, mForce()
, mMass(1.f)
{
}

void PhysicState::setVelocity(sf::Vector2f velocity)
{
    mVelocity = velocity;
}

sf::Vector2f PhysicState::getVelocity() const
{
    return mVelocity;
}

sf::Vector2f& PhysicState::getVelocityRef()
{
    return mVelocity;
}

void PhysicState::setForce(sf::Vector2f force)
{
    mForce = force;
}

sf::Vector2f PhysicState::getForce() const
{
    return mForce;
}
```

```
39
40 void PhysicState::setMass(float mass)
41 {
42     assert(mass > 0);
43     mMass = mass;
44 }
45
46 float PhysicState::getMass() const
47 {
48     return mMass;
49 }
```

6.6 PhysicsEngine class

This class provide a simple collision detection mechanism based on axis-aligned bounding boxes intersection, taking advantage of the `intersects` method of SFML's `sf::Rect<typename T>` class. For more advanced collision detection techniques, see these two articles: *N Tutorial A - Collision Detection and Response*[22], *N Tutorial B - Broad-Phase Collision*[23].

The integration methods of this class have been implemented with the help of this *Integration Basics* article[12].

```
1  // PhysicsEngine.hpp
2
3  #pragma once
4
5  #include <X-GSD/Time.hpp>
6  #include <X-GSD/PhysicState.hpp>
7  #include <X-GSD/ComponentCollider.hpp>
8
9  #include <SFML/System/Vector2.hpp>
10 #include <SFML/Graphics/Transformable.hpp>
11
12 #include <set>
13
14 namespace xgsd {
15
16     // Needed struct for RK4 method.
17     struct Derivative
18     {
19         sf::Vector2f dp; // dp/dt = velocity
20         sf::Vector2f dv; // dv/dt = acceleration
21     };
22
23     class PhysicsEngine
24     {
25         // Typedefs and enumerations
26     public:
27         typedef std::unique_ptr<PhysicsEngine>    Ptr;
28
29         // Methods
30     public:
31         void    checkCollisions();
32
33         void    addStaticCollider(ComponentCollider* collider);
34         void    addDynamicCollider(ComponentCollider* collider);
35         void    deleteStaticCollider(ComponentCollider* collider);
36         void    deleteDynamicCollider(ComponentCollider* collider);
37
38     private:
39         Derivative static       evaluateRK4(const PhysicState& initialPhysics,
40                                             const sf::Transformable& initialTransf,
41                                             const HiResDuration& dt,
```

```
                                        const Derivative& d);

        sf::Vector2f static     accelerationRK4(const PhysicState&
                                            physics);

        // Class methods
    public:
        void static             integrateEuler(PhysicState& physics,
                                      PhysicState& lastPhysics,
                                      sf::Transformable& transf,
                                      sf::Transformable& lastTransf,
                                      const HiResDuration& dt);

        void static             integrateRK4(PhysicState& physics,
                                    PhysicState& lastPhysics,
                                    sf::Transformable& transf,
                                    sf::Transformable& lastTransf,
                                    const HiResDuration& dt);

        // Variables (member / properties)
    private:
        std::set<ComponentCollider*>    staticColliders; // Colliders whose Entity does not have a RigidBody
        std::set<ComponentCollider*>    dynamicColliders; // Colliders whose Entity has a RigidBody
    };

} // namespace xgsd
```

The implementation of the collision detection mechanism is plenty of comments explaining each step. For better understanding of the integration methods, see the aforementioned *Integration Basics* article.

```
// PhysicsEngine.cpp

#include <X-GSD/PhysicsEngine.hpp>

#include <SFML/Graphics/Rect.hpp>

using namespace xgsd;

// Very basic collision detection algorithm based on axis-aligned bounding boxes intersection
void PhysicsEngine::checkCollisions() {

    // Check between dynamic colliders first (entities which have a collider and a rigidBody)
    for (auto iterD = dynamicColliders.begin(); iterD != dynamicColliders.end(); ++iterD) {

        // If the entity containing this collider is pending of destruction, skip it
        if ((*iterD)->entity->isDestroyPending())
            continue;

        sf::FloatRect intersection;

        // Get the global bounds rect of one entity
        sf::FloatRect rectD = (*iterD)->entity->getWorldTransform().transformRect((*iterD)->getRectBounds());

        // Check between dynamic colliders (entities which have a collider and a rigidBody)
        for (auto iterD2 = std::next(iterD); iterD2 != dynamicColliders.end(); ++iterD2) {

            /* The starting point of the iterator is the next item of iterD because dynamic colliders
             must check collision between them avoiding repetition (1-2 is the same as 2-1). Also,
             it avoids self-collision detecton (1-1, 2-2, etc).
             */

            // If the entity containing this collider is pending of destruction, skip it
            if ((*iterD2)->entity->isDestroyPending() || (*iterD)->entity->isDestroyPending())
                continue;

```

```
36              // Get the global bounds rect of the other entity
37              sf::FloatRect rectD2 =
    (*iterD2)->entity->getWorldTransform().transformRect((*iterD2)->getRectBounds());
38
39              // Check collision with SFML and call collisionHandler of both entities
40              if(rectD.intersects(rectD2, intersection)){
41                  (*iterD)->entity->collisionHandler((*iterD2)->entity, intersection);
42                  (*iterD2)->entity->collisionHandler((*iterD)->entity, intersection);
43              }
44          }
45
46          // Check between dynamic (entities which have a collider and a rigidBody) and static
    colliders (entities which have a collider but no rigidBody)
47          for (auto iterS = staticColliders.begin(); iterS != staticColliders.end(); ++iterS) {
48
49              // If the entity containing this collider is pending of destruction, skip it
50              if ((*iterS)->entity->isDestroyPending() || (*iterD)->entity->isDestroyPending())
51                  continue;
52
53              // Get the global bounds rect of the other entity
54              sf::FloatRect rectS =
    (*iterS)->entity->getWorldTransform().transformRect((*iterS)->getRectBounds());
55
56              // Check collision with SFML and call collisionHandler of both entities
57              if(rectD.intersects(rectS, intersection)){
58                  (*iterD)->entity->collisionHandler((*iterS)->entity, intersection);
59                  (*iterS)->entity->collisionHandler((*iterD)->entity, intersection);
60              }
61          }
62      }
63  }
64
65  void PhysicsEngine::addStaticCollider(xgsd::ComponentCollider *collider)
66  {
67      auto inserted = staticColliders.insert(collider);
68      assert(inserted.second);
69  }
70
71  void PhysicsEngine::addDynamicCollider(xgsd::ComponentCollider *collider)
72  {
73      auto inserted = dynamicColliders.insert(collider);
74      assert(inserted.second);
75  }
76
77  void PhysicsEngine::deleteStaticCollider(xgsd::ComponentCollider *collider)
78  {
79      auto found = staticColliders.find(collider);
80      assert(found != staticColliders.end());
81      staticColliders.erase(collider);
82  }
83
84  void PhysicsEngine::deleteDynamicCollider(xgsd::ComponentCollider *collider)
85  {
86      auto found = dynamicColliders.find(collider);
87      assert(found != dynamicColliders.end());
88      dynamicColliders.erase(collider);
89  }
90
91  // Simple integrator. Cheap, but accumulates a lot of error as time advances.
92  void PhysicsEngine::integrateEuler(PhysicState& physics,
93                                     PhysicState& lastPhysics,
94                                     sf::Transformable& transf,
95                                     sf::Transformable& lastTransf,
96                                     const HiResDuration& dt)
97  {
98      float dtValue = ((float)dt.count()/ONE_SECOND.count());
99
100     // Save last physics state and transformable
101     lastPhysics = physics;
102     lastTransf = transf;
103     transf.setPosition(transf.getPosition() + physics.getVelocity() * dtValue);
104     physics.setVelocity(physics.getVelocity() + (physics.getForce() / physics.getMass()) *
     dtValue);
105 }
106
```

```
107 // More accurate than Euler, but a bit more resource-consuming.
108 void PhysicsEngine::integrateRK4(PhysicState& physics,
109                                  PhysicState& lastPhysics,
110                                  sf::Transformable& transf,
111                                  sf::Transformable& lastTransf,
112                                  const HiResDuration& dt)
113 {
114     float dtValue = ((float)dt.count()/ONE_SECOND.count());
115 
116     // Save last physics state and transformable
117     lastPhysics = physics;
118     lastTransf = transf;
119 
120     Derivative a,b,c,d;
121     Derivative initial;
122 
123     a = evaluateRK4(physics, transf, HiResDuration::zero(), initial);
124     b = evaluateRK4(physics, transf, dt / 2, a);
125     c = evaluateRK4(physics, transf, dt / 2, b);
126     d = evaluateRK4(physics, transf, dt, c);
127 
128     sf::Vector2f dpdt = 1.0f / 6.0f *
129     ( a.dp + 2.0f * (b.dp + c.dp) + d.dp );
130 
131     sf::Vector2f dvdt = 1.0f / 6.0f *
132     ( a.dv + 2.0f * (b.dv + c.dv) + d.dv );
133 
134     transf.setPosition(transf.getPosition() + dpdt * dtValue);
135     physics.setVelocity(physics.getVelocity() + dvdt * dtValue);
136 }
137 
138 // Auxiliar function for RK4 integrator
139 Derivative PhysicsEngine::evaluateRK4(const PhysicState& initialPhysics,
140                                       const sf::Transformable& initialTransf,
141                                       const HiResDuration& dt,
142                                       const Derivative& d)
143 {
144     float dtValue = ((float)dt.count()/ONE_SECOND.count());
145 
146     PhysicState physics;
147     sf::Transformable transf;
148     transf.setPosition(initialTransf.getPosition() + d.dp * dtValue); // Add the velocity to
        position
149     physics.setVelocity(initialPhysics.getVelocity() + d.dv * dtValue); // Add the acceleration
        to velocity
150     physics.setForce(initialPhysics.getForce());
151     physics.setMass(initialPhysics.getMass());
152 
153     Derivative output;
154     output.dp = physics.getVelocity();
155     output.dv = accelerationRK4(physics);
156     return output;
157 }
158 
159 // Auxiliar function for RK4 integrator
160 sf::Vector2f PhysicsEngine::accelerationRK4(const PhysicState& physics)
161 {
162     return physics.getForce() / physics.getMass();
163 }
```

References

[1] Apple Inc. 64-bit and ios 8 requirements for app updates. https://developer.apple.com/news/?id=12172014b.

[2] Andrés Ruiz Bernabeu. X-gsd github repository. https://github.com/AndresRuizBernabeu/X-GSD.

[3] Doug Binks, Adam Rutkowski, Matthew Jack, and Juliette Foucaut. Runtime-compiled c++. http://runtimecompiledcplusplus.blogspot.co.uk/.

[4] Blender Foundation. Blender. http://www.blender.org.

[5] Neil Butterworth. C++ debug macros. https://latedev.wordpress.com/2012/08/09/c-debug-macros/.

[6] Cocos2d-x.org. Cocos2d-x. http://www.cocos2d-x.org/products#cocos2dx.

[7] Crytek GmbH. Cryengine. http://cryengine.com/features.

[8] Beman Dawes and Rene Rivera. Boost filesystem library. http://www.boost.org/doc/libs/1_36_0/libs/filesystem/doc/index.htm.

[9] Epic Games Inc. Unreal engine. https://www.unrealengine.com/what-is-unreal-engine-4.

[10] Glenn Fiedler. Fix your timestep! http://gafferongames.com/game-physics/fix-your-timestep/.

[11] Glenn Fiedler. Gaffer on games. http://gafferongames.com/.

[12] Glenn Fiedler. Integration basics. http://gafferongames.com/game-physics/integration-basics/.

[13] GitHub Inc. Github · build software better, together. https://github.com/, February 2008.

[14] Laurent Gomila. Github of the SFML game development book, resourceholder class. https://github.com/LaurentGomila/SFML-Game-Development-Book/blob/master/02_Resources/Include/Book/ResourceHolder.hpp.

[15] Laurent Gomila. Github of the SFML game development book, scenenode class. https://github.com/SFML/SFML-Game-Development-Book/blob/master/03_World/Include/Book/SceneNode.hpp.

[16] Laurent Gomila. SFML. http://www.sfml-dev.org/learn.php.

[17] Laurent Gomila. SFML github wiki. https://github.com/LaurentGomila/SFML/wiki.

[18] Jan Haller, Henrik Vogelius Hansson, and Artur Moreira. *SFML Game Development*. Packt Publishing, June 2013.

[19] json.org. Introducing json. http://www.json.org.

[20] LangPop. Programming language popularity. http://langpop.com/.

[21] Baptiste Lepilleur. Jsoncpp. https://github.com/open-source-parsers/jsoncpp.

[22] Metanet Software. N tutorial a - collision detection and response. http://www.metanetsoftware.com/technique/tutorialA.html.

[23] Metanet Software. N tutorial b - broad-phase collision. http://www.metanetsoftware.com/technique/tutorialB.html.

[24] Sean Middleditch. Dangers of std::shared_ptr. http://seanmiddleditch.com/dangers-of-stdshared_ptr/.

[25] Robert Nystrom. *Game Programming Patterns*. Genever Benning; 1 edition, November 2014.

[26] Robert Nystrom. Game programming patterns. http://gameprogrammingpatterns.com/, 2014.

[27] Robert Nystrom. Game programming patterns. singleton. http://gameprogrammingpatterns.com/singleton.html, 2014.

[28] Günter Obiltschnig. Poco class sharedlibrary. http://pocoproject.org/docs/Poco.SharedLibrary.html.

[29] Okam Studio. Godot game engine. http://www.godotengine.org/.

[30] Photon Storm Ltd. Phaser. http://phaser.io.

[31] Stefan Reinalter. Schema-based entity-component data for fast iteration times. https://molecularmusings.wordpress.com/2014/02/21/schema-based-entity-component-data-for-fast-iteration-times/#more-484.

[32] Stefan Reinalter. Using runtime-compiled c++ code as a scripting language: under the hood. https://molecularmusings.wordpress.com/2014/05/10/using-runtime-compiled-c-code-as-a-scripting-language-under-the-hood/#more-487.

[33] Stefan Schindler. Optank development. http://www.optank.org/game-development-design/.

[34] Scirra Ltd. Construct 2. https://www.scirra.com/construct2.

[35] ShiVa Technologies SAS. Shiva. http://www.shivaengine.com/.

[36] Bjarne Stroustrup. *A Tour of C++ (C++ in Depth Series)*. Addison Wesley, September 2013.

[37] UBM TechWeb. Gamasutra - the art & business of making games. http://www.gamasutra.com/, 1997.

[38] Unity Technologies. Unity. http://unity3d.com/unity.

[39] Wikipedia. List of unreal engine games. http://en.wikipedia.org/wiki/List_of_Unreal_Engine_games.

[40] YoYo Games Ltd. Gamemaker: Studio. https://www.yoyogames.com/studio.