1 Literature Review

1.1 Repository

According to the Merriam-Webster Dictionary (2017), a repository is "a place, room or container where something is deposited". A software repository is a computer, directory or server that stores all the source code for that software project. This is usually available on the Internet, but it can also be local to the developers.

Repositories are also related to the version control of the source code being produced. The definition of version control is "a system that records changes to a file or set of files over time so that you can recall specific versions later"(CHACON; STRAUB, 2014). This allows the user to compare versions, check updates, see who introduced (or removed) an issue and when and rollback to previous versions of the system (CHACON; STRAUB, 2014). The goal is to make it easy to return to states that were working, even after changes are made after a long time.

Modern version control systems allow developers to work on a distributed basis and to parallel their tasks, with the ability of *branching* the repository. Those *branches* are separated lines of development, that won't mess with the main one until they are merged (CHACON; STRAUB, 2014). This feature lets developers create and test new changes before submitting them to the project stable line of work, without affecting the final product.

1.2 Packages

In computer science, package can have multiple meanings, depending on the context being used. A Linux package means a bundle of files containing the required data to run an application, such as binaries and information about the package.

Most Linux distributions have their own package managers. Each expects and handle different types of files, but all of them have the common goal of making the installation easier. They download the package, resolve dependencies, copy the needed binaries and execute any post- or pre-configuration required by the system to install a package (LINODE, 2017). For example, Debian has *dpkg*, Red Hat has *rpm* and Arch Linux has *pacman* as default package managers.

Another installing method is compiling from scratch. This may be very handy if the user is more advanced or the package is not in the package manager's repository, however, in this case, the user will have to manually handle dependencies, download,

compile and do everything else the manager does.

1.2.1 CMake

Creating packages for multiple platforms requires a lot of time and effort, because it has to be compiled on each of the systems, with those system's libraries, binaries and architecture. In order to make this task easier, several cross platform building tools were created.

CMake was created to fulfill the need for "a powerful, cross-platform build environment for the Insight Segmentation and Registration Toolkit"(CMAKE, 2017). It is "a system that manages the build process in an operating system and in a compiler-independent manner"(CMAKE, 2017).

In a very clever way, CMake generates native compiling and configuration scripts (like makefiles for Unix and namespaces for Windows) and use them to build the package (CMAKE, 2017). It is also designed to be used with the native environment, unlike other building tools (CMAKE, 2017).

The building process is controlled by files named *CMakeLists.txt*. They can have commands to generate a building environment, set needed variables, compile dependencies, link required libraries and install the project. A project that has many subdirectories, each with their own rules, can have multiple CMakeLists.txt to create the final product.

1.3 Filesystem Hierarchy Standard

The Filesystem Hierarchy Standard was proposed on February 14, 1994 as an effort to rebuild the file and directory structure of Linux and, later, all Unix-like systems. It helps developers and users to predict the location of existing and new files on the system, by proposing how minimum files, directories and guiding principles (ALLBERY et al., 2015).

The Hierarchy starts defining types of files that can exist in a system. Whenever files differ in this classification, they should be located in different parts of the system: *shareable* files are the ones that can be accessed from a remote host, while *unshareable* are files that have to be on the same machine to be obtained. *Static* files are the ones that aren't supposed to be changed without administrator privileges, whereas *variable* ones can be changed by regular users (ALLBERY et al., 2015)

The root filesystem is defined then: this should be as small as possible and it should contain all the required files to boot, reset or repair the system. It must have the directories specified on Table 1 and installed software should never create new directories on this filesystem (ALLBERY et al., 2015).

Tabela 1 – Directories on the Hierarchy (ALLBERY et al., 2015)

Directory	**Description**
bin	Essential command binaries
boot	Static files of the boot loader
dev	Device files
etc	Host-specific system configuration
lib	Essential shared libraries and kernel modules
media	Mount point for removable media
mnt	Mount point for mounting a filesystem temporarily
opt	Add-on application software packages
run	Data relevant to running processes
sbin	Essential system binaries
srv	Data for services provided by this system
tmp	Temporary files
usr	Secondary hierarchy
var	Variable data

From the directories in Table 1, "`/usr`, `/opt` and `/var` are designed such that they may be located on other partitions or filesystems."(ALLBERY et al., 2015, p. 3). The `/usr` hierarchy should include shareable data, that means that every information host-specific should be placed in other directories.

About the `/var` hierarchy, FHS specifies "everything that once went into `/usr` that is written to during system operation (as opposed to installation and software maintenance) must be in `/var`."(ALLBERY et al., 2015, p. 30).

2 Methodology

This chapter explains what was and will be done within the duration of the whole project.

2.1 Project Overview

It has the main goal of creating a platform with all the games developed in the university's courses related to games. The games that will be available must have all their assets and required libraries in a single package that runs in Linux distributions without the need of installing any other package; they also must have a graphical installer for users without technical knowledge.

In order to achieve this goal, the games developed will be cataloged and cloned to a main GitHub organization (whenever possible). Two scripts will be created then, one to build games using SDL 1 and the other for SDL 2. The platform itself will be developed while all the other activities take place.

After that, a script will be generated to replicate the packaging system to all of the other games, making the necessary adjustments along the way. The games will be deployed to the website with all of their information and available installers.

The packaging scripts will be integrated and adapted to the platform, so that any student who posts a game will have the installers generated automatically.

2.2 Task Division

Because this is a shared work among courses, teachers and students, a special task division, illustrated in Figure 1, will be made to accomplish the established purpose.

Professor Edson and Mr. Faria were responsible for first cataloging the existing games. They will remain as helpers in the packaging system and main stakeholders for the team developing the website.

The team *Plataforma de Jogos UnB* from the courses *Métodos de Desenvolvimento de Software* (Software Development Methods) and *Gestão de Portfolios e Projetos* (Management of Portfolios and Projects) is in charge of creating the first version of the actual website with some of the features desired.

The scripts creation and their application on the games, their integration with the platform, as well as the evolution and maintenance of the platform after the courses are

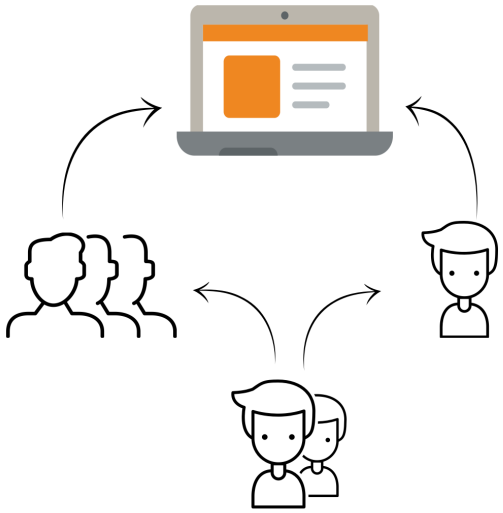

Figura 1 – Task Division

finished will be my responsibility.

2.3 Game Gathering

The games selected for this first part of the project were the ones developed in this Faculty throughout the last semesters, since the course *Introdução aos Jogos Eletrônicos* (Introduction to Electronic Games) has been created here in the Gama campus. Because this work is being mostly held at FGA, and all the games developed here are compiled and run on Linux distributions, these were selected as first games for the platform. The proximity with the students who created those games is another reason.

Professor Edson, that was ministering the course until last year, and Mr. Matheus Faria, the new teacher, first contacted the students and asked them to post their codes to GitHub. They cloned them into the fgagamedev GitHub organization.

After that, I was responsible for checking the status of the games, gathering information such as which of them compiled, which SDL version they used, which ones had licenses. Table 2 shows these initial results.

Out of 20 games created in *Introdução aos Jogos Eletrônicos*, 4 didn't have a known repository and 8 didn't have a license that allowed us to change them at that time. Mr. Faria and I were responsible for finding unknown games and getting the missing licenses. As result of this task, *The Last World War* was added and 5 other had licenses acquired as shown in Table 3.

Tabela 2 – Initial status of the selected games

Name	**Source?**	**License?**	**SDL**	**Compiles?**
Deadly Wish	y	n	2	n
Strife of Mythology	y	n	2	y
Travelling Will	y	n	2	y
7 Keys	y	MIT	2	n
Babel	y	GPL 2	2	y
Terracota	y	MIT	2	n
Dauphine	y	n	2	n
Imagina na Copa	y	n	2	y
Kays Against the World	y	n	2	y
Ankhnowledge	y	GPL 2	1	y
The Last World War	n	-	-	-
Post War	y	n	1	y
War of the nets	y	GPL 2	2	y
Jack the Janitor	y	GPL 3	1	y
Drawing Attack	n	-	-	-
Earth Attacks	n	-	-	-
Emperor vs Aliens	y	n	1	y
Ninja Siege	y	GPL 2	1	y
Space monkeys	y	GPL 2	1	n
Tacape	n	-	-	-

Tabela 3 – Final game status

	License	**SDL**	**Compiles**
Deadly Wish	GPL 3	2	n
Strife of Mythology	GPL 2	2	y
Travelling Will	MIT	2	y
7 Keys	MIT	2	n
Babel	GPL 2	2	y
Terracota	MIT	2	n
Dauphine	MIT	2	n
Imagina na Copa	MIT	2	y
Kays Against the World	n	2	y
Ankhnowledge	GPL 2	1	y
The Last World War	n	1	y
Post War	MIT	1	y
War of the nets	GPL 2	2	y
Jack the Janitor	GPL 3	1	y
Emperor vs Aliens	n	1	y
Ninja Siege	GPL 2	1	y
Space monkeys	GPL 2	1	n

2.4 Packaging

To create the installers for the games, professor Edson decided to use a folder structure that would be easy to understand to anyone familiar with Linux. Apart from the original directories in the repository, he added the folders *bin*, *dist*, *lib* and *linux*.

- *linux* would contain the scripts needed during compilation and also helpers to run the game after its building. When other OSs are added, there will be folders with the specifics for each one of them.
- *lib* initially consists of the source of the libraries used by the games. They are built inside this folder as well, according to the Operating System the package is being built for.
- *dist* includes the final installers, according to each supported OS. These can be distributed and installed in any supported computer.
- *bin* has the compiled libraries and the game executable.

I added the directory *Qt* to use the Qt Installer Framework without having to install it system-wide. Figure 2 shows the folders and their contents to a SDL 1 project.

2.5 Platform

The first version of the platform was developed using mixed development methods. During the first half of the semester, the Rational Unified Process was used. For the next part, Scrum and XP were chosen. This choice of development framework is because of how the courses are divided.

Throughout the RUP part of the development, the team created several documents to aid the development cycle, such as, vision, architecture document, class diagram, use case diagram, use case specification, test case specification.

These documents helped the team to understand the system requirements and how they should be implemented. The most experienced members also helped the others to learn the technologies to develop the website.

2.6 Tools

CMake is the chosen framework for generating the packages. It's suppose to help developers creating applications that run in several platforms, like Linux, Mac and Windows. It offers a lot options for that, like cross compilation and compilation directed to

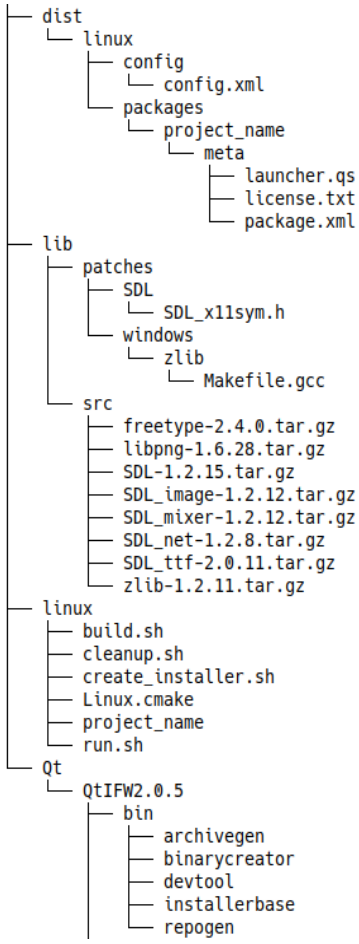

Figura 2 – Folder tree

each of them. It is distributed under OSI-approved BSD 3-clause License and the minimum required version is 2.8.

For the graphical installer, Qt Installer Framework has been selected. It is easy to use and offers a nice GUI with all the necessary steps for installing a package, like license agreement and path choice. This framework is distributed under LGPLv3 license and the version being used is 2.0.5.

For the website development, Django was picked because of the previous knowledge the group had with it. To make the front end of the application, Facebook's React was chosen for the flexibility it gives to the user interface. They are both very scalable, have a big support on the community and are released under the BSD 3-clause license. The versions being used are the last ones at the beginning of the project, namely, 1.11.1, for Django, and 15.5.4, for React.

Python is the language for the packaging script because it must integrate with the Django webapp and its powerful easy to use API. It will also be the language for any other needed scripts. The least required version is 3.4, and it's licensed under an OSI-approved open source license.

To develop the scripts, a virtual machine running Debian Jessie was used. The VM was powered by Vagrant, version 1.9, that allows easy environment virtualization. It also enables a developer to test in several Operating Systems, which is required for the nature of this project. The computer hosting the script and used to its development has an Intel Core i5-6200U 2.3 GHz processor. It also has 8 GB of RAM and an NVIDIA GeForce 940M graphic processor.

3 Partial Results

This chapter explains the results obtained so far with the project development

3.1 The Building Scripts

The building scripts were tested against half of the 16 available games. Table 4 shows which games were tested and the results of them. It was a success, because all the installers were created correctly for all of them (both GUI/Qt and *.deb*). For some games, even with the correct compilation, they wouldn't run as expected, due to logical errors in their source code.

The game *Space Monkeys* compiled correctly, however upon running it, the user couldn't do anything and the screens weren't precisely rendered. *War of the Nets*, even compiling without errors, had a segmentation fault after a few seconds with the game open.

Tabela 4 – Scripts results

Game	**Compiles?**	**.deb**	**GUI/Qt**	**Runs?**	**SDL**
Ankhnowledge	y	y	y	y	1
Post War	y	y	y	y	1
Jack the Janitor	y	y	y	y	1
Emperor vs Aliens	y	y	y	y	1
Ninja Siege	y	y	y	y	1
Space monkeys	y	y	y	n*	1
War of the Nets	y	y	y	n	2
Travelling Will	y	y	y	y	2

The building scripts are very similar, the only difference is that one builds games for SDL 2 while the other uses SDL 1 (and their respective libraries). They work by cloning the repository and copying the required files to use CMake as seen in Algorithm 1. This is the tool that actually compiles and builds everything. Because CMake generates a lot of files that are only used while it's running, professor Edson made a few more scripts to separate everything into folders and generate the installers.

A major concern when making this script was its generality. It should run successfully with as many games as possible, requiring only a few tweaks in the source code or folder structure of the repository, if any. Starting with *Jack the Janitor*, the example Professor Edson had made first, the building script assumed a lot of things, mostly due to my inexperience with games and the folder structure adopted by the students. For

Algorithm 1 Algorithm to build the games

Start
- $project \leftarrow$ INIT$(url, branch)$ ▷ Clone repository and set the project
- COPY_FILES$(default, project)$ ▷ Copy templates
- $project_{media\_dir} \leftarrow$ FIND_MEDIA ▷ Find the media folder
- $project_{source\_dir} \leftarrow$ FIND_SOURCE ▷ Find the source folder and *.cpp* files
- REPLACE_INFO$(default, project)$ ▷ Replace template defaults
- RENAME$(default, project)$ ▷ Rename some files
- BUILD$(project)$ ▷ Call the build script
- CREATE_INSTALLERS$(project)$ ▷ Create the installers

End

example, in the initial versions, I thought all the *.cpp* files would always be in a folder called *src*. Another assumption was that all media would be in a *media* folder.

Both of them proved me wrong as more games were tested, but were fairly easy to fix and keep the algorithm generic. For both folders, I had to modify the script to look for directories that would have similar names to those I thought were the rule. For example, *source* instead of just *src*, and *resources*, *res* and *sound* instead of just *media*. Even though this doesn't find all possible names, it follows a pattern found in most folder structures and all of the projects tested so far.

Another big premise was that all games would have their main file named *main.cpp*. Even in repositories with Portuguese file names, it never occurred to me that anyone would name those files in any other way. However, a few games, specially *Traveling Will* proved me wrong. Here, again, there was the option of looking for files named with similar words to *main*, but this was a terrible option in this case, because the file could have *any* name. Even if I looked for *principal.cpp* that would not guarantee anything.

One other option was parsing all the source files looking for the main function, but that would slow down the building process and would be error prone. Because there were so many options, I decided to keep the script looking for *main.cpp*, even if it required to manually change the repository.

3.2 Platform

The website as of now allows an administrator to upload a game, with its respective information, like supported platform, related media, and installers. The administrator has to manually add all the information related to a game, like developers who worked on it, awards won (if any), release date, version number, etc. Figure 3 shows part of the screen to add a game.

The general public can see a list of the available games, with their uploaded pictures. The home page also shows a slide with some pictures of highlighted games. By

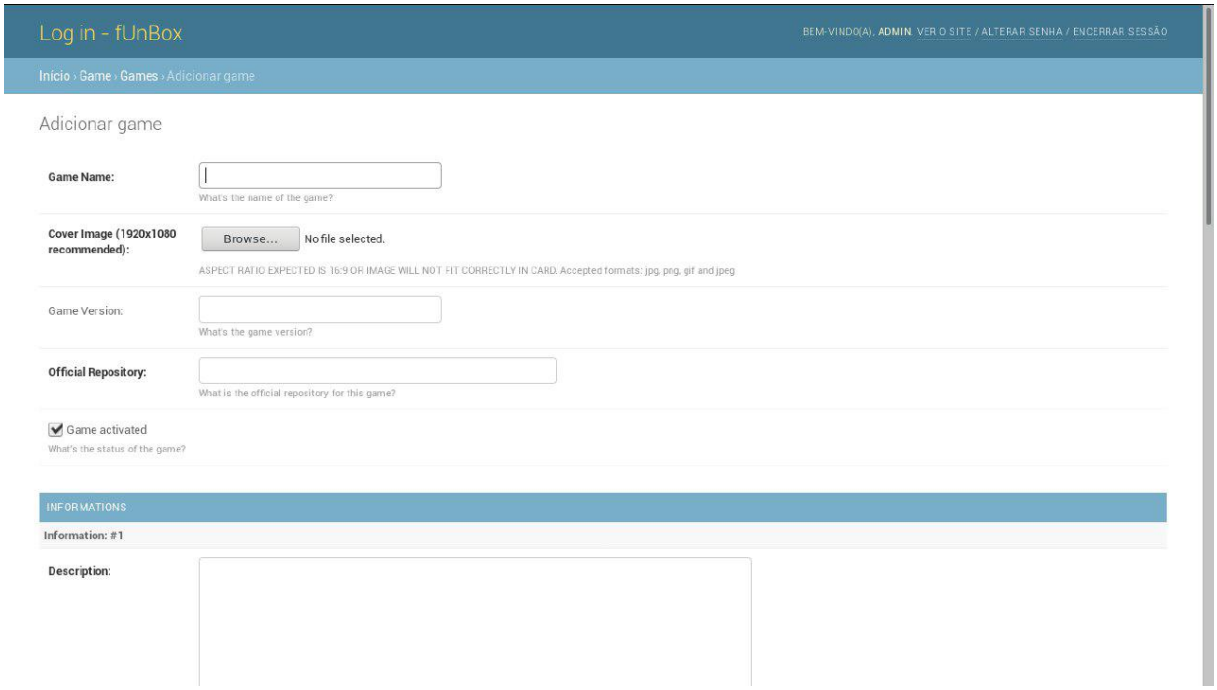

Figura 3 – Include new game

choosing one, it's possible to see its version, official repository, release date, description, among other information. It's also possible to download the game for the available Operating Systems or comment using a Facebook account, as shown in Figure 4

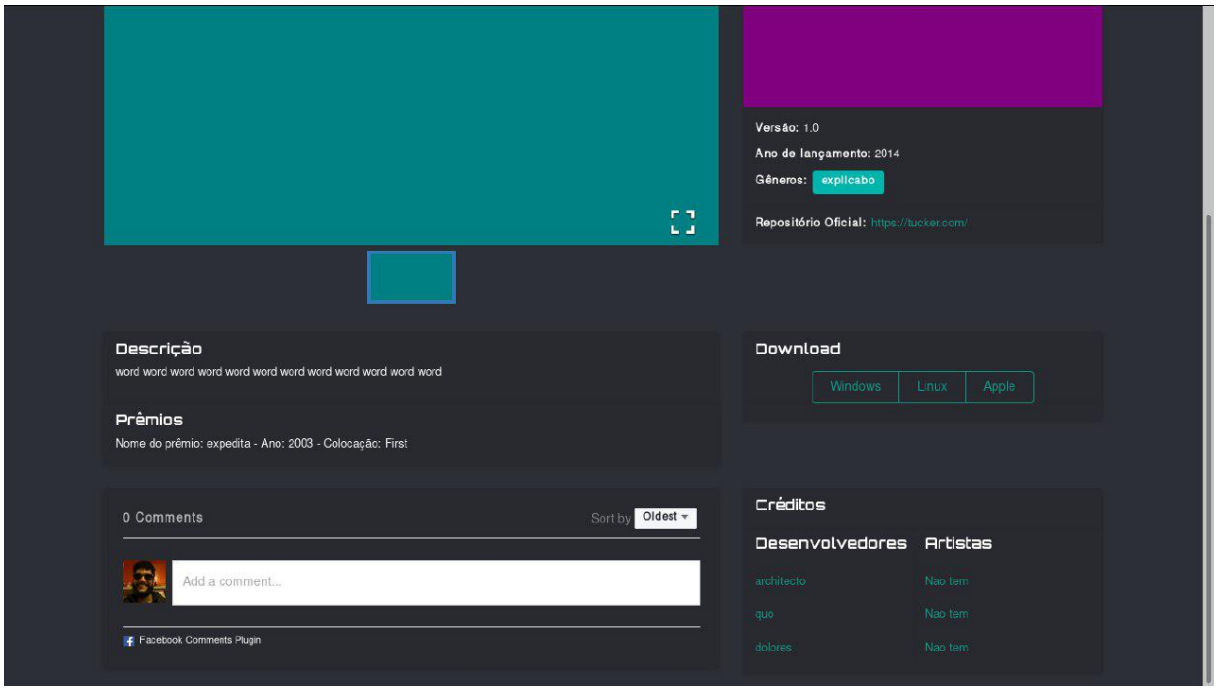

Figura 4 – Game detail

3.3 Known Issues

The building scripts have a few problems that will be fixed in the second part of the project:

- Choosing a destination folder different than the user's home directory causes the Qt installer to fail. This is likely happening because somewhere on the Qt templates the symbol ~ (abbreviation for home) is being used.
- When the installation works successfully, the game doesn't run properly (or at all) on some systems (tested on Arch Linux for now). Probably there are some missing libraries on the final package, but this is just a hunch and has to be proved.
- The games are running without sound, even on host systems that have full access to the sound card.

4 Future Work

This chapter explains what will be done in the remaining time of the project. It talks about some of the long term goals and the activities that will be carried to achieve them.

4.1 Schedule

For the next months it is expected to have the script for SDL 2 projects finished with support for Windows and MacOS. The known issues described on Section 3.3 have to be fixed for both SDL 1 and 2. The building scripts have to be tested against all the available games.

Another goal is to allow the user to link their GitHub repository to the website, letting them build the packages for the game automatically with GitHub hooks. The game will then, be available in the website without the need for a manual upload from an administrator.

It's also purpose of this next part of the semester to maintain and evolve the platform while the tasks related to the script are being held. Some selected issues will be resolved to add new features and fix bugs found on the system.

The following activities will be developed in the remaining time of the project. They are summarized in Figure 5.

1. **Literature Review** - Review what the literature has on packaging, CMake, game development.
2. **Add Lua support** - Some of the games, especially the ones built with SDL 2, have lua as a dependency library. This lib is not being being compiled in the current version of the building scripts.
3. **Add other Linux distros support** - Allow other users to install the games using their package manager. Generate at least *.rpm* packages in addition to the *.deb* already generated.
4. **Add Darcy's games** - Look for games developed at Darcy Ribeiro campus and run the building scripts on them.
5. **Add MacOS support** - Create install packages for Mac (Apple Systems).

6. **Add Windows support** - Make an installer (*.exe*) to run on Windows 10 (maybe with some backwards compatibility if possible).
7. **Integrate to platform** - Run the scripts though a request on the website
8. **Avoid duplication on save** - According to the GitHub repository of the team, models are being saved with duplicated data.
9. **Create games statistics** - Create endpoints to send game statistics, like views and download amounts.
10. **Submit issues to the GitHub repository** - Allow users to submit issues through the platform to the GitHub repository of the game.
11. **Integrate to GitHub** - Make the system receive GitHub signals.
12. **Allow user to build a game from a branch** - Let the user upload a game based on a chosen branch. This will in fact run the script to build the game, but without the need of a an administrator.
13. **Read game info from game branch** - Developers, description and even some images are usually available in the game repository. This task is to read that information to avoid the need of manual input of the administrator.
14. **Code refactoring** - Integrate scripts (SDL 1 and 2), make them more generic and efficient.
15. **Final adjustments** - Make minor improvements and fixes.
16. **Write Report** - Report progress and results.

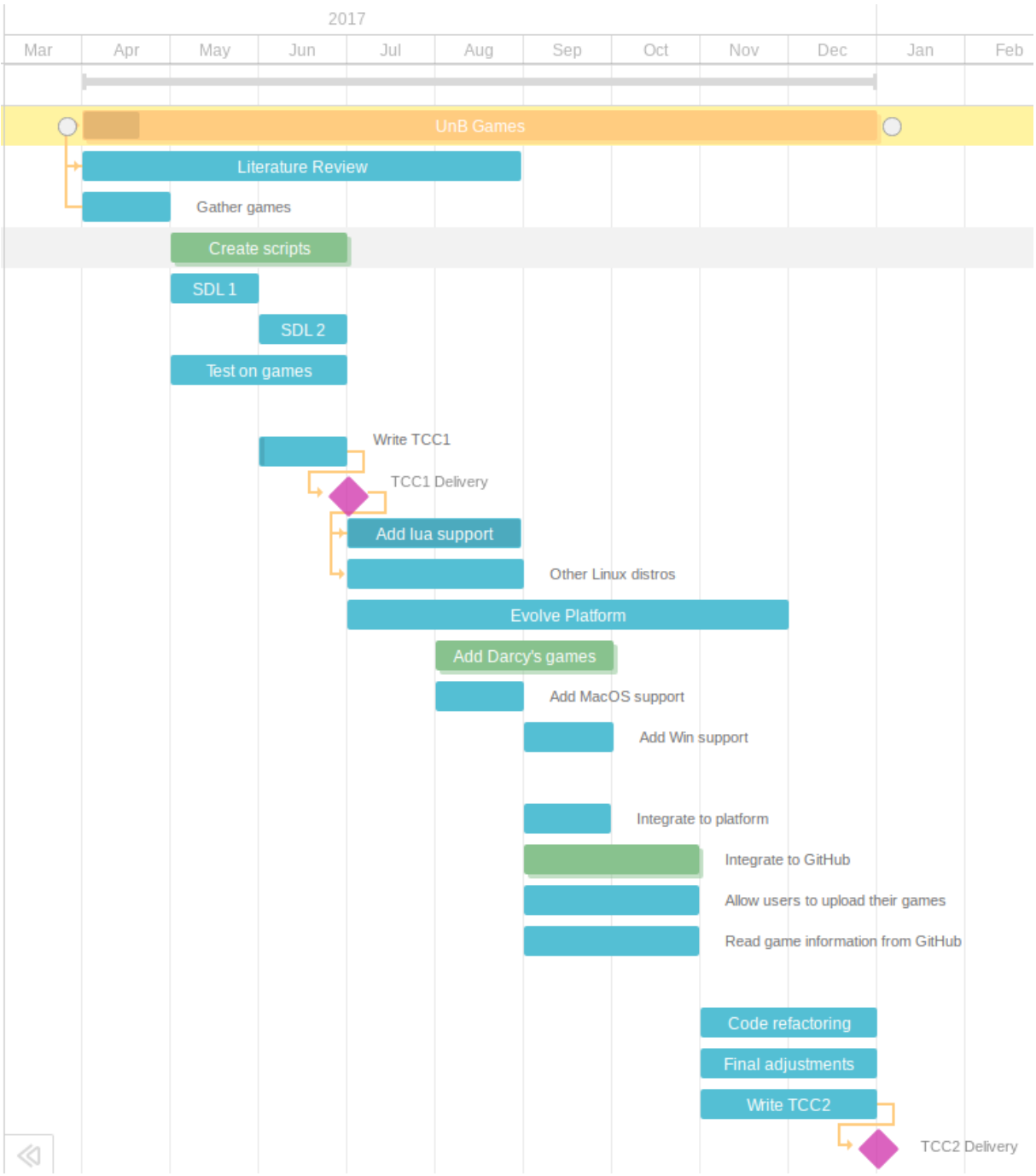

Figura 5 – Project Schedule

Referências

ALLBERY, B. S. et al. Filesystem hierarchy standard. 2015. Citado 2 vezes nas páginas 2 e 3.

CHACON, S.; STRAUB, B. *Pro git*. [S.l.]: Apress, 2014. Citado na página 1.

CMAKE. *CMake Overview*. 2017. Disponível em: <https://cmake.org/overview/>. Citado na página 2.

LINODE. *Linux Package Management*. 2017. Disponível em: <https://www.linode.com/docs/tools-reference/linux-package-management>. Citado na página 1.

WEBSTER, M. *Definition of repository*. 2017. Disponível em: <https://www.merriam-webster.com/dictionary/repository>. Citado na página 1.