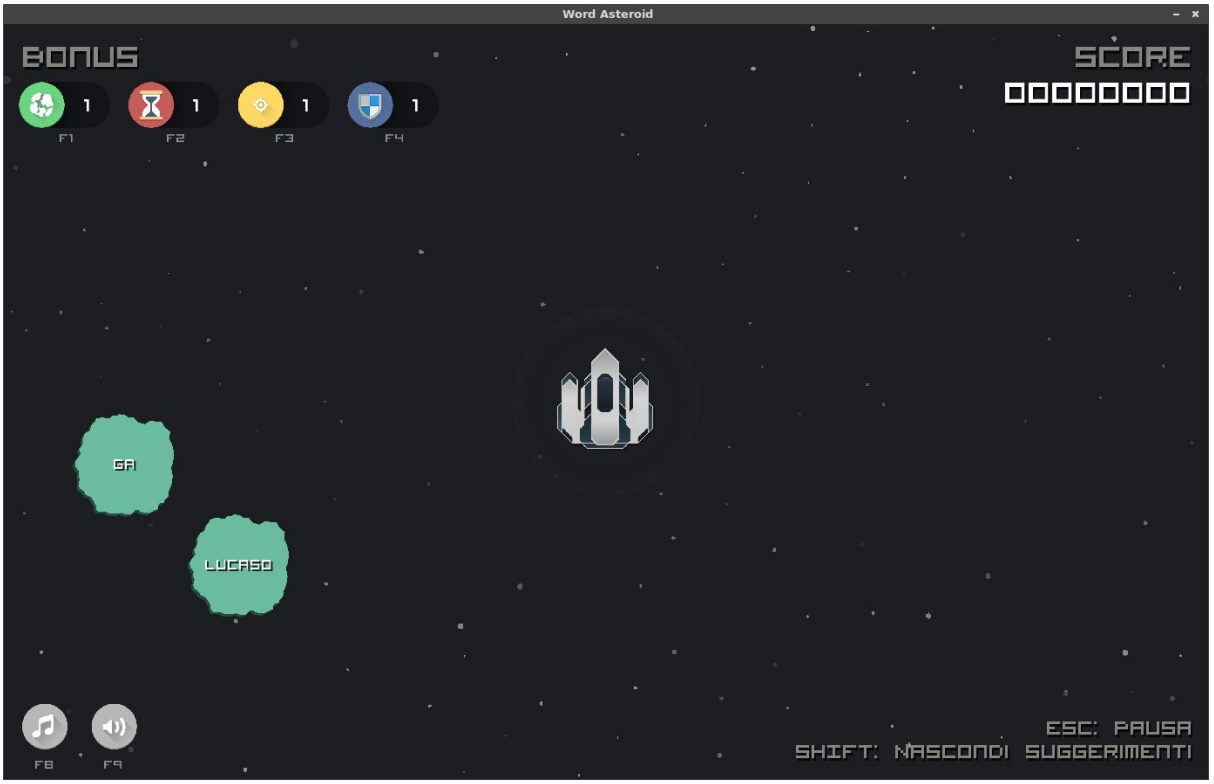
Word Asteroid on Password Extractor

Case study

This project born to satisfy some requirements encountered during my job as cybersecurity specialist, and to sensibilize the people to use a strong password.

The main problem that this project aims to solve, is the analysis of a large data of zipped archives. During my job, I encountered the necessity to explore huge public collections of email:password (hundreds of Gigabytes) in an efficient way. The standard `zgrep` command doesn't provide a multithreading interface, and the search in this archive takes usually more than 10 hours. With the solution that has been created for this project, the time to explore all the collections is drastically decreased.

With this idea in my mind, I created a word game which aims to encourage the people to use strong password playing with the public password present in the collections. Indeed, the difficulty of the game increases when strong password are extracted from the dump. While it decreases when weak password (as the infamous "password123") are taken from the collections. The game is called "Word Asteroid" because the players is a spaceship and the word are the asteroids to destroy.



The game shows to the player which are the already known password, but not the owners of them. The correlation between username:password is done asking the user his own email. While the game is focused to create the game to display to the user, some background

threads are looking for the correlation in the collections. When the user finishes playing, the results of the search are shown to the gamer.

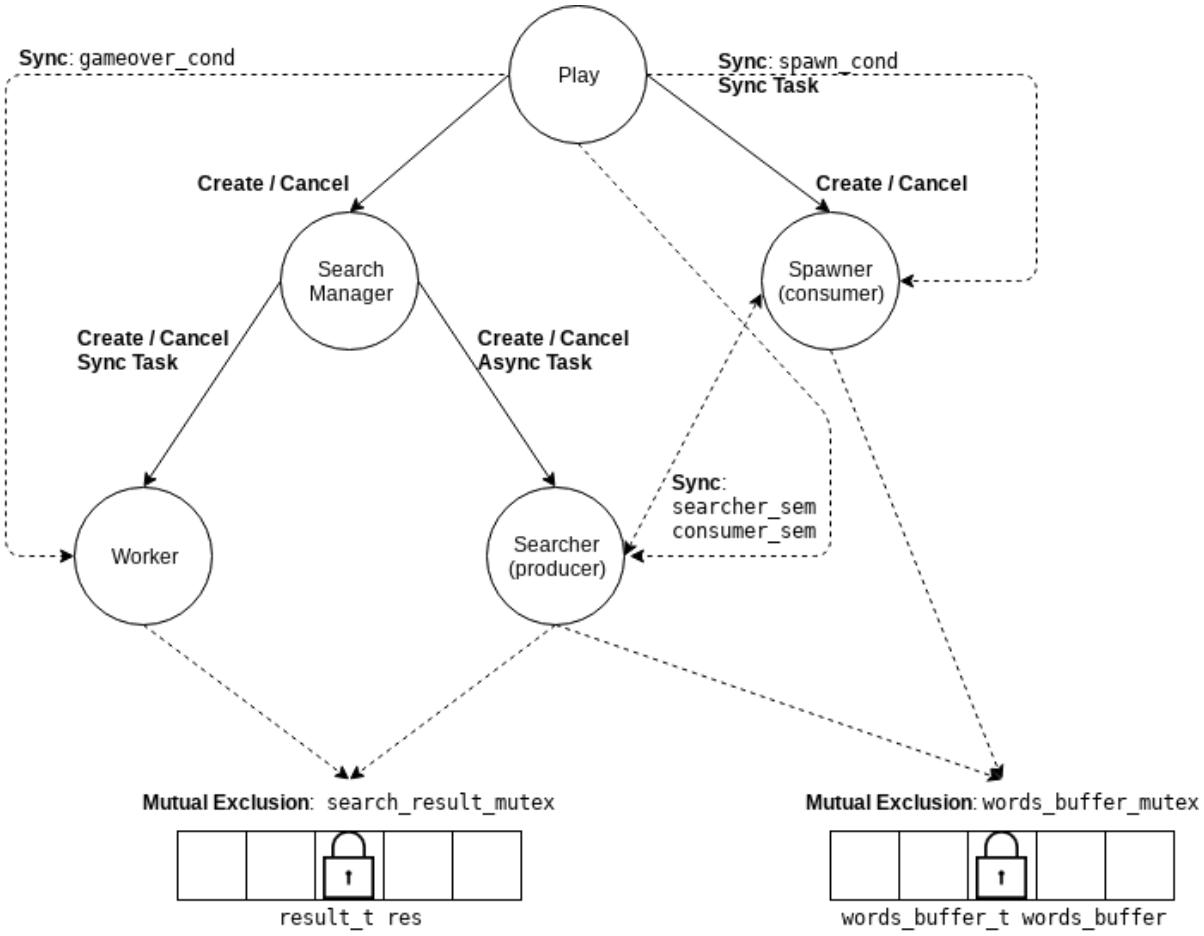
Multithreading Design

This chapter aims to explain the logic and the application of the multithreading in the project.

As already told in the introduction paragraph, there are two different jobs that the program is running concurrently.

1. Draw and refresh the game objects with the given input from the user;
2. Take the password from the collections to display to the user, which also represents the word that he is going to destroy. In the meantime, look for a match of the user's email in the collection.

All the data structures required for the synchronization are shown is the following picture and described below.



The entire game is managed by the main thread. Before the game is started, it creates two different threads:

1. **Spawner**: it is a consumer thread. It is synchronized with the producer thread using the private semaphore `consumer_sem`.
2. **Search Manager**: it manages the extraction of the password in the archives. This thread spawns a compile-time number of subthreads, which has different roles:
    a. **Worker**: these threads only look for a match in the collection. They are synchronized with the main game with the condition variable

`gameover_cond`. Every frame per seconds, they wait for a broadcast signal in order to go on with the research. If this signal does not arrive, the game is finished.

b. **Searcher**: these threads do the same job the worker, but they do also the job of the producer. They are synchronized with the private semaphore `producer_sem`.

The producer/consumer paradigm is applied to a shared buffer called `words_buffer`. This buffer contains the words candidate to be shown to the user, and the mutual exclusion is managed with the `words_buffer_mutex`. In this way, every time the consumer has to create an asteroid with a new word, it just simply take a ready word from the buffer. The job of the producer is to keep the buffer always full, in order to give to the consumer a word for each letter it decides to choose.

The worker and the searcher, which both of them manage the search in the collections, need to save their results. This is done with the shared buffer `res`, and the mutual exclusion is managed with the `search_result_mutex`.

The spawner plays a synchronous task that is scheduled with a dynamic timeout. The worker threads also play synchronous tasks with the fps timeout. The asynchronous task is done by the searcher thread because it takes a random time to fill the buffer and released the resource.

Multithreading Implementation

This chapter aims to show where the implementation of the multithreading is placed in order to give a guideline for the study.

The `play_wa` function, present in `section.cpp` file is the most important one.
It is the one that manages the game, thus all the threads. All the multithreading data structures are store in `thread_manager` variable, which is obviously global.

The following snippet shows the initialization of the `thread_manager`, and the creation of the spawner and the search manager.

```
init_thread_manager(thread_manager, email);

pthread_create(&thread_manager.searcher_thread, nullptr, start_searching, (void
*) &thread_manager.search_manager);

pthread_create(&thread_manager.spawner_thread, nullptr, spawn_asteroid_thread,
(void *) &match_vars);
```

The search manager runs the `start_searching` function presents in `archive_functions.cpp` file, that has the duty of the creation of the worker and searcher threads.
Both the threads are created as shown in the following snippet.

```
int err = pthread_create(&sm->threads[i % NUMBER_THREADS], nullptr, extract,
(void *) lk);
```

`lk` is a data structure which contains all the required information used by the subthreads. In detailed, it contains:

1. The path of the archive;
2. The email of the user, in order to check for the correct match;
3. The flag that tells to the subthread if it is a searcher or a simple worker.

Run the game

This chapter aims to provide a guide to run correctly the game.

The game requires the installation of the Allegro5.0 library from Github.
The steps to run correctly the game are the following:

1. Pull the Word Asteroid repository from Github.

```
$ git clone https://github.com/CorraMatte/WordAsteroid
```

2. Pull the Allegro5.0 repository from Github in the *wa* folder, then compile the Allegro library.

```
$ cd wa
$ git clone https://github.com/liballeg/allegro5.git
$ cd allegro5
$ mkdir build
$ cd build
$ cmake ..
$ make
```

3. Go back to the *wa* folder and run cmake.

```
$ cd ..
$ cmake -DCOLLECTION_FOLDER=\"/path/to/collection/\"
-DNUMBER_THREADS=4
```

4. Run make.

```
$ make
```

5. Run the auxiliary Python script to create the collection.

```
$ python --email test@email.it --nfile 4
```

6. Run the game.

```
$ ./wa
```

Game

This chapter aims to explain the main sections of the game.

The goal of the game is to destroy the asteroids present on the screen. In order to do it, it necessary to digit the entire word presents in the asteroid before they reached the shield of the spaceship.

The bonuses are the following:

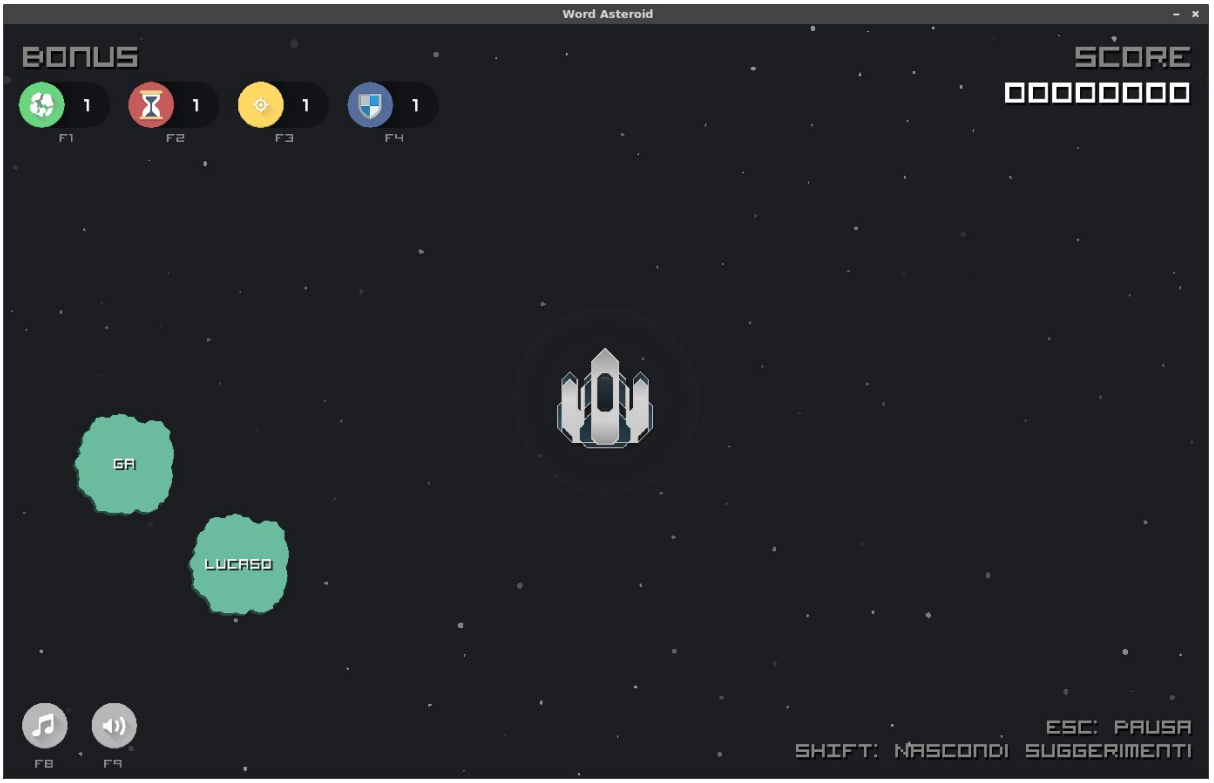
1. Atomic: destroy all the asteroids in the game;
2. Rallenty: slow the asteroids for a few seconds;
3. Fire: destroy the nearest asteroid;
4. Shield: destroy the asteroids that reached the shield.

The following pictures represent the main section of the game.

The main section - it is possible to start the game, or edit the settings (disable both game sounds and music).

Pre-game section - this section asks the email from the user.
The email could be the entire email or only a part of the email.

Play section - this section let the user play to the game.
Press ESC to show the instructions of the game.



End game - show the score of the player and the result of the search in the archive.