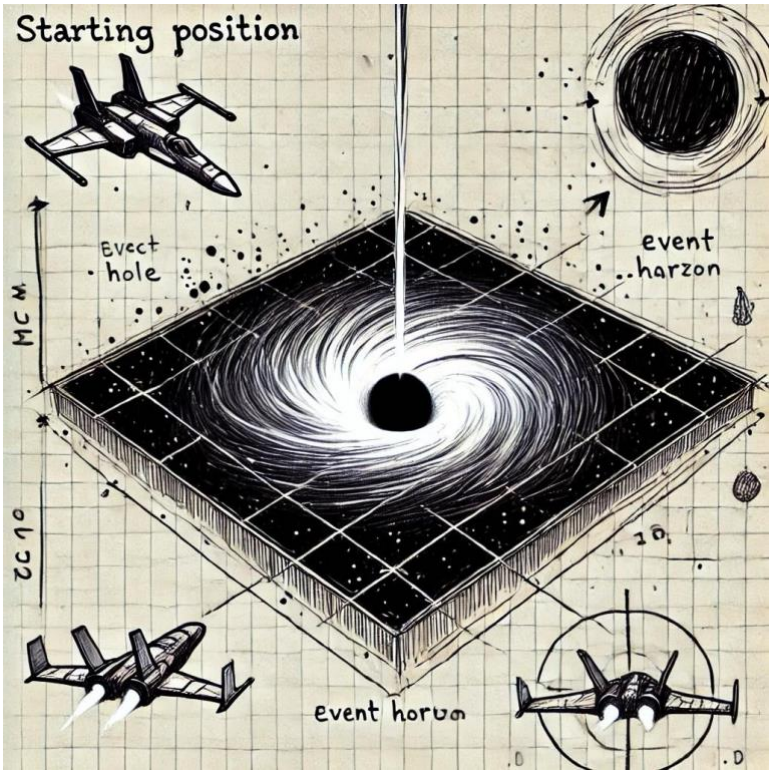
Erwan Le Doeuff
M1 CompuPhys

# Conrad




## Technical report

October 6th, 2024

Python

### Description of the project structure and list of the different files

Conrad's repository can be found on *Github*[1] and contains the project's folders, subfolders and files. Let's briefly introduce the project structure.

The `resources/` directory which contains the scope statements, some sketch illustrations AI-generated and this report. The following `src/` folder holds all the source files. Other files are needed to configure the installation of Conrad.

Diving now into the source directory, firstly with the main files,

- `main.py` : the entry point of Conrad.
- `Game.py` : this file contains all the game logic : the game loop, entity instantiation, and more.
- `params.py` : a configuration file.
- `StateManager.py` : class file. It serves a global store of game's state.
- `EntityPool.py` : contains a class that will hold all references to entity objects.

Then a bunch of files related to game's entities and classes. Their names are self-explicit.

- `IEntity.py` : a really important class/interface since all entities inherits from it.
- `Ship.py`
- `Player.py`
- `Target.py`
- `BlackHole.py`
- `Projectile.py`
- `HeavyProjectile.py`
- `LightProjectile.py`

Finally, two files that implement the controller of the interactive console : commands, etc.

- `CustomConsole.py`
- `custom_implementations.py`

Also, there is a folder called `assets/` containing the different assets of the game.

1. https://www.github.com/atomicwelding/conrad

# 1 Functional requirement of the program

Conrad had to be a mix between a simulation and a video game. As such, some gameplay elements were integrated within the post-newtonian simulation of massive objects orbiting a black hole. The whole is described here.

## 1.1 The game

### 1.1.1 Installation

The game has to be downloaded from *Github* and both `python` and `pip` must be installed. It is packaged as `pip` package ; as such, you'll need – once the game downloaded – to use the following commands in conrad's directory,

```
pip3 install .
```

```
cd src
```

```
python3 main.py
```

As an additional suggestion, install the utility `rlwrap`. This wrapper is built on top of the `GNU Readline` library and brings facilities in regards to command line edition : history, completion, … It thus makes the Conrad less dull by avoiding to type the same commands, again and again.

```
rlwrap python3 main.py
```

### 1.1.2 Mechanics

Welcome aboard! Once the game started, your terminal should greet you. It is now an interactive console which invites you to type in commands. A list of commands is provided if you type `list commands`.[2] This interactive console (let's say it's the cockpit of your ship!) will be in charge of transmitting your inputs to the game.

You first need to `set` the according initial conditions before `run`(ning) the game[3].

```
set distance X
```

```
set radial_velocity Y
```

```
set angular_velocity Z
```

```
run
```

Where $(X, Y, Z)$ are floating-point numbers. When `run` is executed, a graphical scene pops up. It displays a black hole and two ships. Yours is the blue one at the right. The other one at the opposite side is the target. The target begins with the same number of projectiles and initial conditions as yours. As Conrad is a turn-by-turn video game, some informations about the current turn are displayed at the top left part of the scene.

The game starts at your turn. Each time it is your turn, you will have three possibilities. Whether you `wait` for a better position or you decide to `shoot` a `heavy` or `light` projectile by providing the type of the missile and the angle at which you decide to shoot.

2. There's also a `help` command.

3. `set_default` command is providen to set cool values to play with.

Heavy projectiles assume elastic collisions. They can be used to propel yourself somewhere by altering your orbit, or to push your target. On the other hand, light projectiles are explosive and destroy the ship if they hit it.

The game ends when one of the two ships is destroyed, being touched by a light projectile or falling into the black hole. If the player can't play anymore (having no ammunation left), the game is forwarded until one ship is destroyed.

### 1.1.3 Configuring the game

Some parameters can be modified in the `params.py` file, such as the masses of ship or black hole, the number of projectiles per type, …

Still, **warning**. The game is still under development, and some quantities like the screen's size or the Schwarzschild radius of the black hole are intrinsically related to the way assets are displayed. Modifying such parameters could lead to strange graphical behaviors.

## 1.2 The simulation

The simulation part of the game comes from the physics engine. The most used framework to study black holes is general relativity. However, under the assumption of a weak gravitational field[4], you can apply some corrections to Newton's laws to approximate the behavior of an object orbiting a black hole – these are known as Post-Newtonian approximations. This will be elaborated further in section 2.

# 2 Internal structure of the program

## 2.1 A short theoretical approach

### 2.1.1 Differential equation

As stated in the section 1.2, the simulation is essentially classical with the addition of a relativistic correction force field term,

$$\vec{F_R}/m = -\frac{3}{2}R_S\dot{\theta}^2\vec{e_r}$$

To the classical gravitational force field,

$$\vec{F_G}/m = -G\frac{M}{r^2}\vec{e_r}$$

Resulting in the following second order differential equation,

$$\vec{a}\cdot\vec{e_r} = \left(-G\frac{M}{r^2}-\frac{3}{2}R_S\dot{\theta}^2\right)\vec{e_r}$$

$$\iff \qquad \ddot{r} = -\frac{GM}{r^2}+\left(r-\frac{3}{2}R_S\dot{\theta}^2\right)$$

4. As the gravitational potential field $\Phi$ is both proportional to $1/r$ (r being the distance from the black hole) and $M$ (the mass of the black hole), the assumption holds if you are either far from the black hole or if its mass is low.

Where $R_S$ is the Schwarzchild radius of the black hole and $M$ (resp. $m$) the mass of the black hole (resp. of the object orbiting the black hole). Initial conditions are given as an user's input.

Solutions to such differential equations can be approximated with various numerical scheme, one of them being the leapfrog algorithm as it tends to be stable with respect to energy conservation, given a small timestep.

### 2.1.2 Why a black hole has the name it does?

Recalling basics of orbital mechanics, the escape velocity $v_e$ is defined as

$$v_e = \sqrt{\frac{2GM}{r}}$$

If you reverse the equation by substituting $v_e$ with $c$ the speed of light and isolating $r$, you will derive the definition of the Schwarzschild radius,

$$R_S = 2GM/c^2$$

It is the distance from the black hole beyond which even light cannot escape. This was the main motivation to represent the black hole as a large black and plain circle of radius $R_S$ in Conrad.

### 2.1.3 Elastic collisions

The results of elastic collisions can be quite complex to derive in the general case. Therefore, I referred to sources[5] formulating the following result ;

Given two objects numbered respectively 1 and 2, with masses $m_i$, position $\vec{r}_i = (r_i, \theta_i)$ and velocities $\vec{v}_i = (\dot{r}_i, r_i\dot{\theta}_i)$ known at the moment of collision, collisions yield

$$(r_i\dot{\theta}_i)_{\text{new}} = \text{atan}\left[\frac{m_i - m_j}{m_i + m_j}\tan\{r_i\dot{\theta}_i\} + \frac{2m_j}{m_i + m_j}\frac{v_j}{v_i}\frac{\sin\{r_j\dot{\theta}_j\}}{\cos\{r_i\dot{\theta}_i\}}\right]$$

$$(\dot{r}_i)_{\text{new}} = \left[\left(\frac{m_i - m_j}{m_i + m_j}\dot{r}_i\sin\{r_i\dot{\theta}_i\} + \frac{2m_j}{m_i + m_j}\dot{r}_j\sin\{r_j\dot{\theta}_j\}\right)^2 + (\dot{r}_i\cos\{r_i\dot{\theta}_i\})^2\right]^{1/2}$$

## 2.2 Program structure

### 2.2.1 Dependencies

Conrad use a few external dependencies to operate. Namely, `pygame` to handle the visualization part and `numpy` for computational purposes.

5. *Chocs élastiques en 2 dimensions,* **Pascal Rebetez**, 2008.

### 2.2.2 Structure

The key logical components of the program reside in three files. To name them: `Game.py`, `IEntity.py` and `CustomConsole.py`.

`Game.py` implements the main `Game` class. This class serves as the core of the game logic, managing the overall structure and flow of Conrad. It handles various critical aspects, including game initialization, state management, entity updates and the main game loop.

It coordinates interactions between the `CustomConsole` and the different entities through the use of a global state store object, implemented in `StateManager.py`. This type of global state store is convenient for small-sized projects, as it eliminates the need to intricately link properties between components. However it scales poorly as the project grows, since it becomes difficult to determine which logical component has the right to modify the states or not. This issue is exacerbated in programming language which do not enforce strict encapsulation (*e.g.* `python`), thus allowing unrestricted access to state variables by any component.

The `CustomConsole` is operating on a different thread. This allows for decoupling the two interfaces: one dedicated to graphical output, and the other to input/output interactions. Additionally, it is necessary for `pygame` (due to the underlying `Simple DirectMedia Layer` library) to run on the main thread, which limits the interactive capabilities of the console. Specifically, when `pygame` is performing computations, entering commands is impossible in a single-threaded setup. All the commands are stored within a `python` dictionnary, with a one-line usage example and a string containing the "goal" of the command.

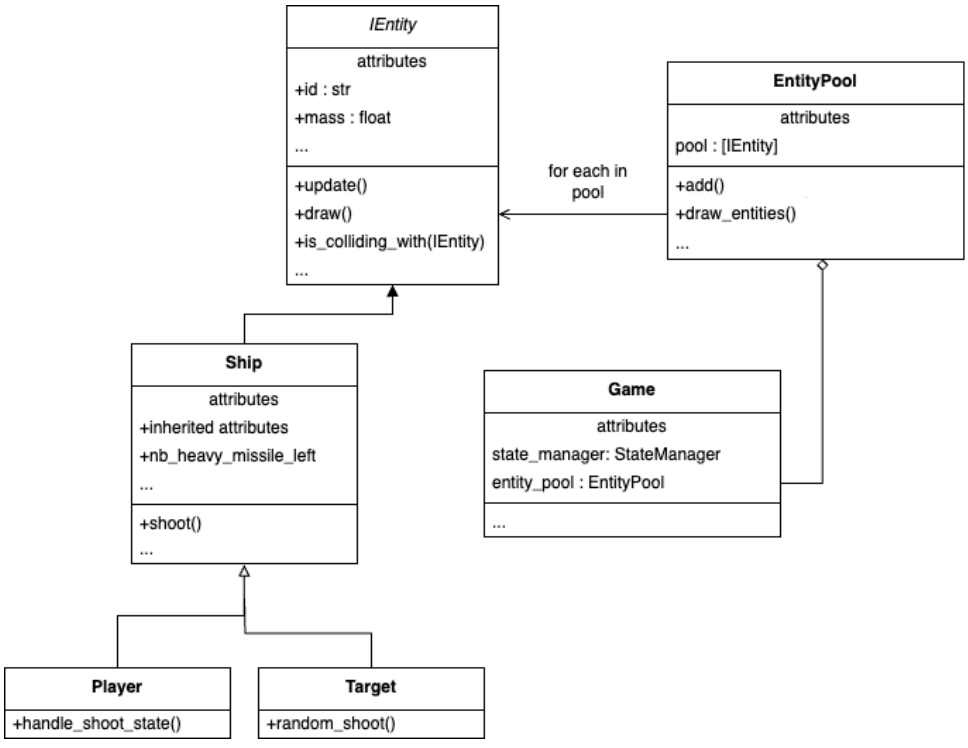



**Figure 1.** A sketch of what the UML class diagram would look like …

Finally, there is the abstract class `IEntity`. From black holes to projectiles, ships, and any other physical objects, all inherit from this class. It implements the update method using the leapfrog integration scheme, methods to draw the object or to handle collisions, etc. Once an object is created, a reference to it is stored in the `EntityPool` and most of method of the `Game` class that deal with entities simply involve calling `IEntity` methods for each object in the pool. `EntityPool` provides methods for that.

When a specific behavior is expected from an object, for example in the collision handler, the child class overrides the method from the parent class but still calls it, since IEntity's methods are kept very general.

# 3 Potential flaws

## 3.1 Bugs on the physical side …

Collisions sometimes exhibit strange behavior, and that's to be expected. Indeed, I assume the objects are rectangular when detecting collisions, but the calculations I perform assume a spherical shape.

Similarly, I am forced to give a certain “impulse” to the projectiles when they ared fired to prevent their hitbox from overlapping with the ship's hitbox.

## 3.2 “Better call patterns”

As mentioned earlier, using a global store can be convenient but problematic, especially when managing critical states. This can be improved with design patterns like the Observer pattern.

With this pattern, the state manager becomes a publisher (or subject) that notifies subscribers (observers) when a state changes. Observers receive the updated value via notifications, without directly accessing the state.

If observers need to request changes, they use predefined methods. In this case, the console or game can act as observers, and the state manager serves as the subject.

## 3.3 Testing numerical integration and scientific softwares

Traditional testing methods, like unit testing, often fail in the context of scientific software and integration schemes for several reasons. First, scientific software typically deals with complex simulations or numerical approximations, where there is no clear “right” answer. Unlike typical unit tests, it's challenging to define expected outputs for comparison.

Moreover, verifying that a numerical scheme has been correctly implemented can be quite challenging. What automated test could we design to ensure the algorithm behaves as expected? None.

This is because scientific algorithms often rely on theoretical models and complex approximations. Even small deviations in the results can be hard to interpret—whether due to coding issues or inherent numerical instability. Additionally, behavior can depend heavily on the language and hardware, as floating-point handling differs between `python` on x86 and `common lisp` on ARM, for example.

All of this sometimes gives the impression of being in the realm of guesswork. To avoid this feeling, one can rely on statistical analysis that I've not implemented for Conrad.

## 3.4 Studying the system

I wrote a small Jupyter notebook alongside Conrad, implementing the basic physics engine to study different trajectories. It's included in the project under the resources/ directory. This allowed me to analyze some typical trajectories and values for the game. However, I didn't have enough time to conduct a comprehensive quantitative study. I wish I had more time to understand the effects, such as the impact of gluing opposite faces together, which forces every orbit path to be closed. So many unanswered questions for now.