Python Numerical Project : Space Battle Around a Black-Hole Technical Report

Master CompuPhys - Python Numerical Project

Léo Bechet

year 2023/2024

Program version : Alpha-2
Languages : Python, GLSL

Table of contents

1. Preface

This project was realized in VisualStudio Code, using the Colorful Comments extension by Parth Rastogi. Though not necessary, it is recommended to install it as it will improve readability by adding color formatting to comments.

Version Alpha-1 was a proof of concept that was built upon and refined to finally obtain Alpha-2. Concerning the different versions requirement, the reviewer will find that all Alpha version features are implemented, only the out of bound return with periodic boundary conditions was implemented for the beta version (togglable) and a graphical engine based on shaders has been implemented, as it was found to be a better debug tool than the available matplotlib plots, as well as providing persistence of the displayed trajectories when removing data from the simulated objects position list. However no checks have been implemented to keep the simulation running once the inventory of ammo has been expanded. It is up to the player to keep the simulation going until he thinks he has lost.

2. Files description

a. Architecture

Files list and architecture :

```
[4.0K]  Source directory
├── [  36]  requirements.txt
├── [4.0K]  ressources
│   ├── [ 12K]  GameDisplay.py
│   ├── [6.0K]  GameLogic.py
│   ├── [2.2K]  GameSettings.py
│   ├── [4.0K]  PhysicsEngine
│   │   ├── [8.5K]  collision.py
│   │   ├── [ 494]  config.py
│   │   ├── [ 13K]  engine.py
│   │   ├── [ 10K]  engine_rev2.py
│   │   ├── [3.7K]  game_loop.py
│   │   ├── [ 198]  __init__.py
│   │   └── [  51]  physicalConstants.py
│   ├── [ 18K]  render_mode.py
│   ├── [4.0K]  shaders
│   │   ├── [ 194]  blackhole_frag.glsl
│   │   ├── [ 173]  blackhole_vert.glsl
│   │   ├── [ 362]  blit_frag.glsl
│   │   ├── [ 166]  blit_vert.glsl
│   │   ├── [ 213]  circle_frag.glsl
│   │   ├── [ 224]  circle_vert.glsl
│   │   ├── [ 360]  darken_frag.glsl
│   │   └── [ 117]  darken_vert.glsl
│   ├── [7.5K]  test.py
│   └── [4.0K]  utilities
│       ├── [  37]  __init__.py
│       └── [ 187]  sys_defined.py
├── [  36]  run.sh
└── [ 114]  setup.sh
```

b. Descriptions

requirements.txt
Contains all necessary python libraries used in the project. Used by setup.sh.

ressources
Directory with the source code. used to organize and minimize the number of files at root.

GameDisplay.py
Contains all the functions called when graphical views are required by the game. Does some minor input formatting in some special cases

GameLogic.py
Collection of functions used for the game logics, such as win and loss conditions

GameSettings.py
Configuration file for the game

PhysicsEnine
Directory of the engine

init.py
File responsible for converting a directory in a package. Contains all necessary imports of functions.

collision.py
Contains all functions related to collisions detection and management

config.py
Configuration file for the physics engine.

engine.py
Earlier version of the engine. Only present for historical purposes, it has no use.

engine_rev2.py
Version of the engine used in the game. Contains the physics simulation, and includes collisions.

game_loop.py
Contains the loop used to run the game and simulate the game until the next turn.

physicalConstants.py
A collection of physical constants used in the engine.

shaders
directory containing all shaders used in the renderer. Names of those files are self explanatory as they only have one use.

test.py
A playground file used to test and debug the simulation and new features.

utilities
directory imported as a package. Contains one only function used to clear the screen. The function is implemented such that it should not be platform dependent.

run.sh
launches the game from the current directory

setup.sh
Running this file should install all necessary dependencies.

3. Goals and usage

The program is presented as a video game where you have to hit a target with different projectile types, with a focus on simulated physics. The simulation implements collisions and integrates Newton's universal laws of attraction with a first order relativistic correction. The engine will be the focus of this paper, however a section will be dedicated to the renderer and the game mechanics implementation.

a. User's guide

- Installation

You may encounter some errors during installation. Installing requirements for **moderngl** may result in some crashes of the installation. Installation trials have been made on a windows machine, but require installation of Microsoft C++ build tools, only available in a special Visual Studio installation. Running the project on windows has been aborted for this reason, and no support can be provided. MacOS has not been tested either due to a lack of funding to get the required platform.

Linux Troubleshooting :

- No errors have been reported when trying to install required libraries using pip. Zll libraries were up to date as of 05/10/2023.
- Issues regarding **swarts** and **iris** during **libGL MESA-LOADER**.
  This issue arises as **conda** does not import the same **libstdc++.so** file as the ones on your system. To fix this issue you must copy the files back to your **conda** install. A script is provided in the root of the project called **fix_iris_swrast_error.sh**. Running it will only output warning messages, but commands are present in the file as comments. First navigate **/home/$USER/miniconda/lib** which is your **conda** install **lib** directory. From here you can run the last 5 commented commands that will, make a backup directory and backup the **conda** lib files, copy the right files back in your install and rename to the right name. Running the

project using **run.sh** should now work. If you get any other errors please contact me : leo.bechet@edu.univ-fcomte.fr

Using the game itself is pretty self explanatory. Prompts will guide the user on inputs and provide examples.

- Reusing the simulation

The PhysicsEngine is a package that can be imported in any python file, so long as it is in the same directory. Importing the package is as easy as using `import PhysicsEngine as e` . One important aspect of the engine is the **object** class. An **object** can be declared using `e.object(type, r, theta, vr, vtheta, mass, radius)` , Where r and theta are the position of the object in polar coordinates, and **vr** and **vtheta** are its speed on r and theta. The **object** class is the main building block of the simulation as it is the placeholder for all properties of an object. We recommend you to check the jupyter notebook in the **ressources** directory, to get a feel on how to implement the simulation in your own environment.

The engine also comes with an integrator that can be used like so : `nbody_coupled_integrator(obj, blackhole, deltat)` where **obj** is the object in the simulation, **blackhole** is an **object**, which only the mass is used, thus passing an arbitrary class with only a **.m** accessible property set to the mass of the blackhole will suffice. The function does not return anything but updates the object's properties. be aware that some values of the **object** will not have been updated. You can run `object.UpdateVariables(self, last_deltat)` to update all the variables.

Extracting data from the simulation is as easy as it comes. An **object** class possesses 3 accessible lists containing its position at each time step : **object.r_list** contains all r coordinates, **object.theta_list** has all theta coordinates and finally **object.v_list** has all speed on r. Note that the speed on theta is not available, but can be computed using **object.GetTheta_p(deltat) .** That is due to the fact that the lists on the **object** are meant to store data computed using the integrator. Position data can be graphed using common plotting libraries.

Documentation is provided with the project in the form of multiple HTML pages that can be open using a traditional browser.

- ## Using the game

Using the game has been made easy using prompts with indications on formatting. Though safeguards are in place to avoid unrecoverable formatting errors during execution, they are very basic and not everywhere, thus the user shall be careful with the inputs he sends.

The ship initial vector selection currently has no type checker, be careful on that part. In case of doubts, type **default** and choose one of the 4 default starting positions provided.

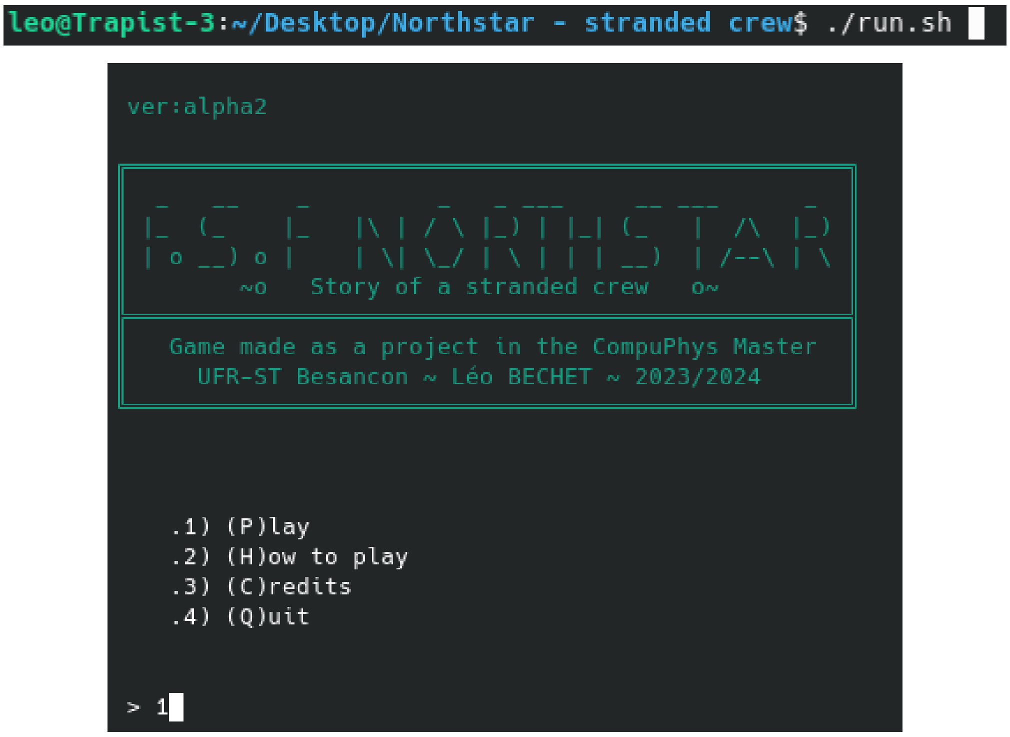
A known quirk of the program is that spamming the enter key during the simulation will skip input selection a number of times equal to the number of keypresses.

As for any python programs, using the **ctrl-z** key combo during code execution will stop the program. Please note that the OpenGL window will not close until the terminal closes.

You can edit game parameters in **GameSettings.py**, and simulation parameters in **PhysicsEngine/config.py**

<u>Typical walkthrough :</u>

Starting the game using start.sh

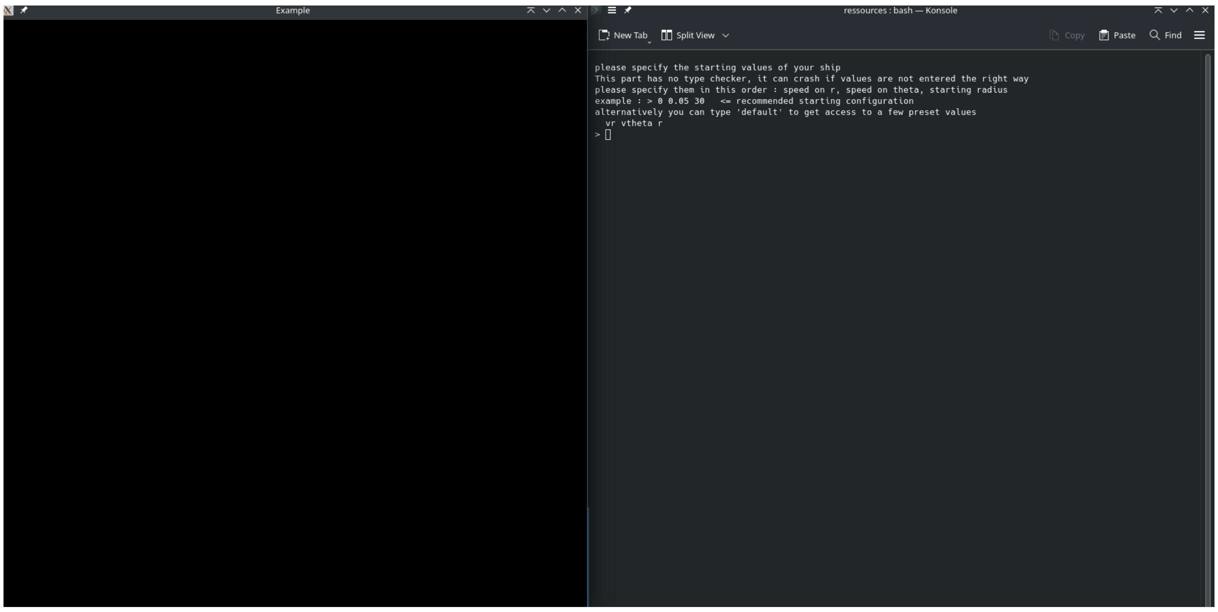


```
leo@Trapist-3:~/Desktop/Northstar - stranded crew$ ./run.sh
```

```
ver:alpha2

 _   __    _         _   _ ___     __ ___      _
|_  (_    |_   |\ | / \ |_) | |_| (_   |  /\  |_)
| o __) o |    | \| \_/ | \ | | | __)  | /--\ | \
       ~o   Story of a stranded crew   o~

  Game made as a project in the CompuPhys Master
    UFR-ST Besancon ~ Léo BECHET ~ 2023/2024



   .1) (P)lay
   .2) (H)ow to play
   .3) (C)redits
   .4) (Q)uit


> 1
```

Fig 1. Starting game command and game menu

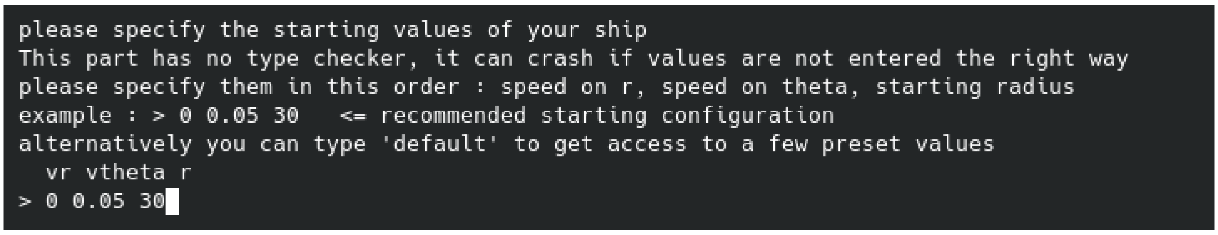
Watching the cinematic is optional, but will introduce you to colors used and mechanics, here we will skip it for the sake of rapidity.



```
please specify the starting values of your ship
This part has no type checker, it can crash if values are not entered the right way
please specify them in this order : speed on r, speed on theta, starting radius
example : > 0 0.05 30   <= recommended starting configuration
alternatively you can type 'default' to get access to a few preset values
  vr vtheta r
> 
```

Fig 2. Possible arrangement of the two windows of the game

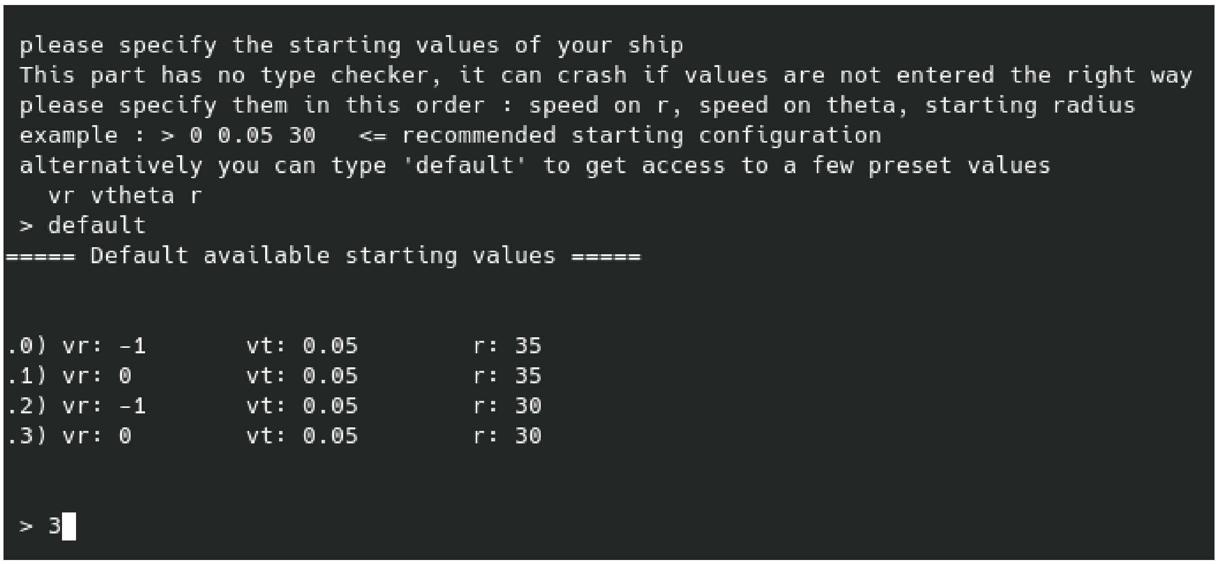
A black window appears, you can put it anywhere, arrange your screen as you wish, but bear in mind that it is best to not resize the window and keep it at its native size.



```
please specify the starting values of your ship
This part has no type checker, it can crash if values are not entered the right way
please specify them in this order : speed on r, speed on theta, starting radius
example : > 0 0.05 30   <= recommended starting configuration
alternatively you can type 'default' to get access to a few preset values
  vr vtheta r
> 0 0.05 30
```

Fig 3. Possible starting conditions

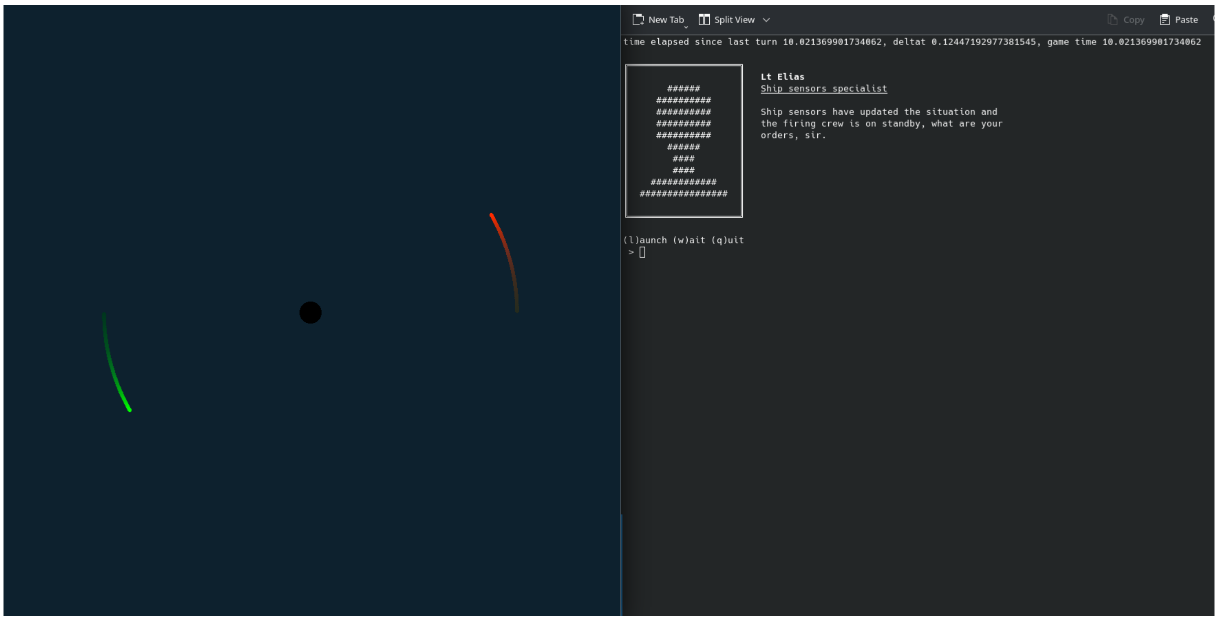
Starting values can be entered as such, or you can use one of the default presets.



```
 please specify the starting values of your ship
 This part has no type checker, it can crash if values are not entered the right way
 please specify them in this order : speed on r, speed on theta, starting radius
 example : > 0 0.05 30   <= recommended starting configuration
 alternatively you can type 'default' to get access to a few preset values
   vr vtheta r
 > default
===== Default available starting values =====


.0) vr: -1      vt: 0.05        r: 35
.1) vr: 0       vt: 0.05        r: 35
.2) vr: -1      vt: 0.05        r: 30
.3) vr: 0       vt: 0.05        r: 30


 > 3
```

Fig 4. Default values presets choice

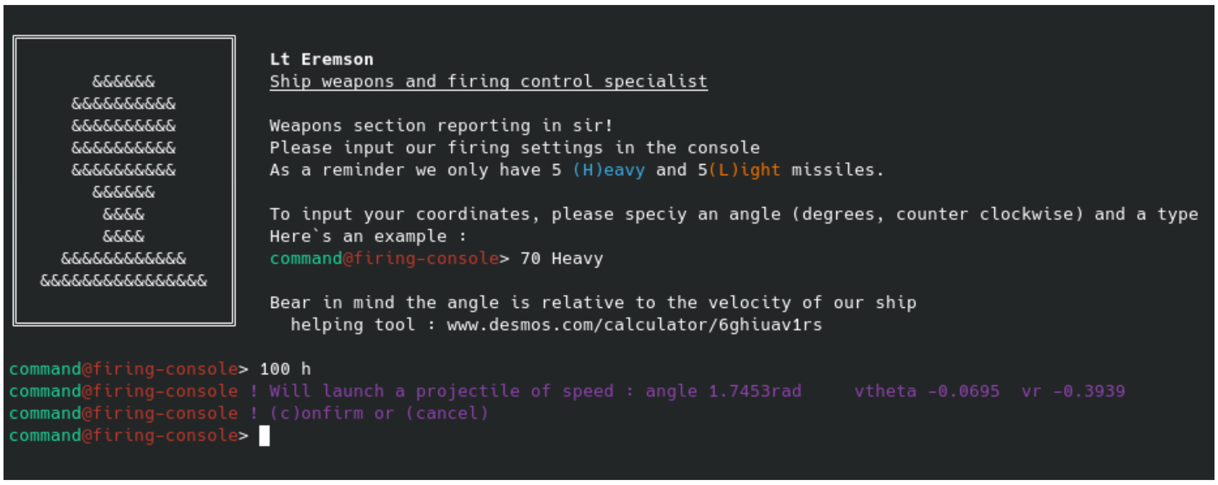
Here we chose preset number 3



```
time elapsed since last turn 10.021369901734062, deltat 0.12447192977381545, game time 10.021369901734062

      ######          Lt Elias
    ##########        Ship sensors specialist
    ##########
    ##########        Ship sensors have updated the situation and
    ##########        the firing crew is on standby, what are your
      ######          orders, sir.
       ####
       ####
   ############
 ################

(l)aunch (w)ait (q)uit
 > 
```

Fig 5. Beginning of the simulation and turn prompt

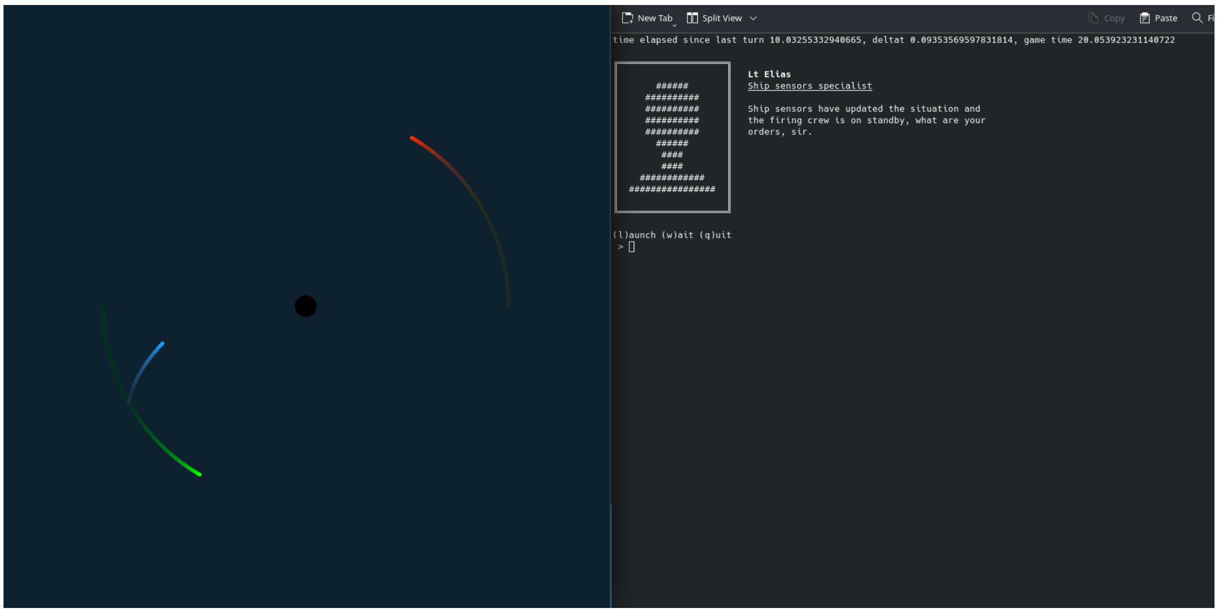
The simulation starts and we are now presented with a choice in the console. We have the option to quit by entering “q”, or to launch a missile using “l”. “w” is a dummy option, pressing enter with nothing or some random text will trigger the wait function. Waiting is equivalent to waiting a turn. Here we chose to launch a missile.



```
      &&&&&&          Lt Eremson
    &&&&&&&&&&        Ship weapons and firing control specialist
    &&&&&&&&&&
    &&&&&&&&&&        Weapons section reporting in sir!
    &&&&&&&&&&        Please input our firing settings in the console
      &&&&&&          As a reminder we only have 5 (H)eavy and 5(L)ight missiles.
       &&&&
       &&&&           To input your coordinates, please speciy an angle (degrees, counter clockwise) and a type
   &&&&&&&&&&&&       Here`s an example :
 &&&&&&&&&&&&&&&&     command@firing-console> 70 Heavy

                      Bear in mind the angle is relative to the velocity of our ship
                        helping tool : www.desmos.com/calculator/6ghiuav1rs

command@firing-console> 100 h
command@firing-console ! Will launch a projectile of speed : angle 1.7453rad     vtheta -0.0695  vr -0.3939
command@firing-console ! (c)onfirm or (cancel)
command@firing-console> 
```

Fig 6. Missile launch interface

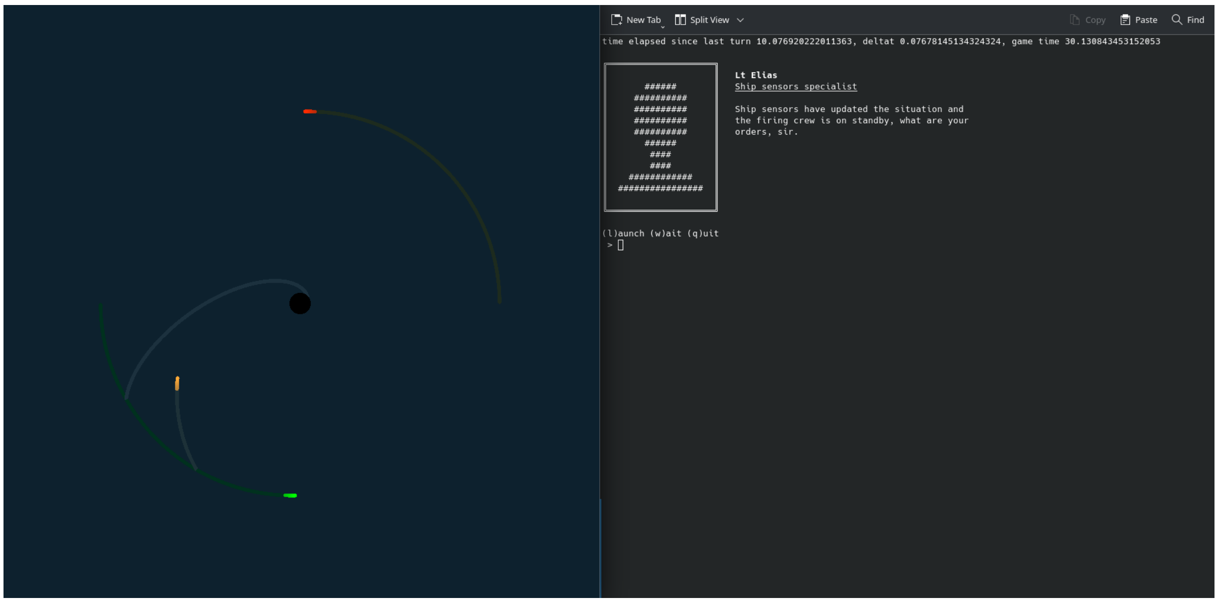
The remaining ammo is displayed in a prompt, and we are set in firing mode. The current prompt tells the game to fire a missile at 100° with the type heavy. Pressing enter resumes the simulation with the added projectile after we confirm. We can however cancel the action.



```
time elapsed since last turn 10.03255332940665, deltat 0.09353569597831814, game time 20.053923231140722

      ######          Lt Elias
    ##########        Ship sensors specialist
    ##########
    ##########        Ship sensors have updated the situation and
    ##########        the firing crew is on standby, what are your
      ######          orders, sir.
       ####
       ####
   ############
 ################

(l)aunch (w)ait (q)uit
 > 
```

Fig 7. Example of the simulation after the launch of an heavy projectile

The simulation has then resumed with a new projectile in blue. Firing a yellow projectile using the same prompt, but replacing the “h” by an “l” will lead to this result in the next turn.



```
time elapsed since last turn 10.076920222011363, deltat 0.07678145134324324, game time 30.130843453152053

      ######          Lt Elias
    ##########        Ship sensors specialist
    ##########
    ##########        Ship sensors have updated the situation and
    ##########        the firing crew is on standby, what are your
      ######          orders, sir.
       ####
       ####
   ############
 ################

(l)aunch (w)ait (q)uit
 > 
```

Fig 8. Example of the simulation after the launch of a light projectile

Our heavy projectile has sadly fallen into the black hole and is now out of the simulation. The sudden change in color for the trails are an artifact and a result of the renderer implementation. Its limitations and potential ameliorations are discussed in the relevant section. The game will now loop indefinitely until one of the win or loss conditions are met, or the player quits. In total there are 7 different endings, each with a different end prompt. Well it looks like you are battle ready,we wish you luck on your future battles, commander!

b. Dysfunctions, resource usage, dependencies

Currently, no major dysfunctions have been found, however extensive testing would be necessary to explore every nook and cranny of the program.

The program was developed on a KDE Neon Linux distribution based on Ubuntu using Python 3.10.6, moderngl 5.8.2, moderngl_window 2.4.4 and PyGLM 2.7.0. Systems specifications are 8go of RAM and an I7-7500U CPU with base clock a 2.9GHz and 3.45GHz turbo.

Ram usage is heavily dependent on how the simulation is used. The **object**s of the simulations are all equipped with lists containing all past indices, but as long as the last one is in them, it can run fine. Resetting those lists every couple of steps will drastically reduce RAM usage. The game implements a renderer for objects. Said renderer is in a very early stage but will work fine for basic visualization. VRAM usage is pretty low as it only requires one texture and a few uniform values. Its inner workings will be discussed in another part.

The required Libraries to run the simulation is **numpy** and **math**. Using the renderer will require **PyGLM** (please note that a similar package called **glm** can conflict with PyGLM, rendering the program inoperable.), **moderngl** and **moderngl_window.** A **setup.sh** script is provided to install all necessary libraries. Please note that the user may need to change **python** to **python3** or vice-versa depending on its environment variables. Default setup uses **python3.**

4. Internal structure of the program

UML diagrams are generated using **pyreverse**

a. UML Diagrams

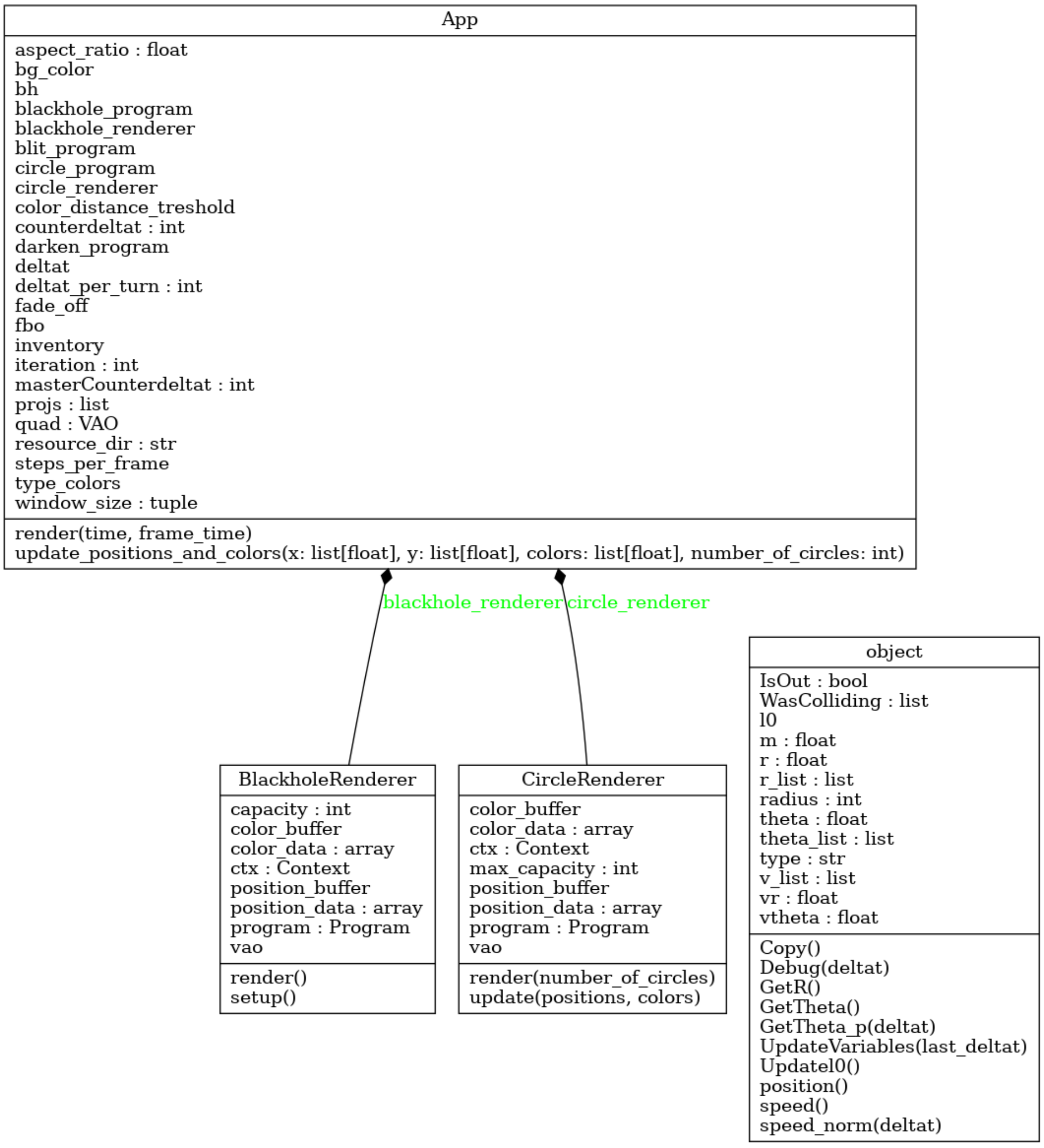


Fig 9. UML DIagram of game specific classes

The 3 linked classes are those of the renderer (see dedicated section), the engine **object** class is also specified

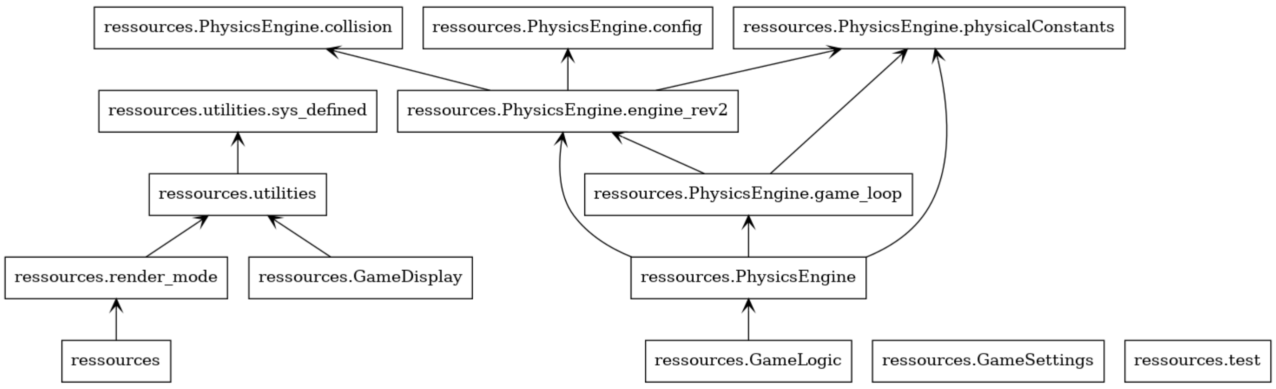

ressources.PhysicsEngine.collision
ressources.PhysicsEngine.config
ressources.PhysicsEngine.physicalConstants
ressources.utilities.sys_defined
ressources.PhysicsEngine.engine_rev2
ressources.utilities
ressources.PhysicsEngine.game_loop
ressources.render_mode
ressources.GameDisplay
ressources.PhysicsEngine
ressources
ressources.GameLogic
ressources.GameSettings
ressources.test

Fig 10. UML diagram of all modules

Above are the UML diagrams for the classes and modules in the game. Only one class is created in the engine, the **object** class. For the modules diagram, please start from **ressources.PhysicsEngine** if you wish to only have those from the engine. Renderer classes and modules are on the left of the diagrams.

b. Physical Models and theoretical computation algorithms

The models we chose to use do not take into account the clock shift effects and neglects the gravitational interactions between objects. The resulting equation is as follow :

$$\ddot{r} = -\frac{G*M}{r^2} + (r - \frac{3}{2} * R_s)\dot{\theta}^2 \qquad (1)$$

This is nothing more than a combination of Newton's law of universal attraction and a first order relativity correction. The conservation of energy in the system is introduced by a conservation of the angular momentum. We also define l_0, which can be determined using the initial conditions.

$$r^2\dot{\theta} = \frac{L_0}{m} \qquad (2) \qquad\qquad l_0 = \frac{L_0}{m} \qquad (3)$$

The pseudo code used to compute the radius and angle of the position after a time step is decomposed into two distinct integrators. The first one is based on Euler's method and is as follows :

```
For $k$ from $i$ to $i+N-1$ do
    $\theta_{k+1} \leftarrow \theta_k + \frac{\ell_0}{r_k^2}\Delta t$
End for
```

Fig 11. Pseudo code of the integrator for the angle

However one might spot that the radius is present in that formula. The radius is computed using the leapfrog integrator, which pseudo-code is as follows :

```
For $k$ from $i$ to $i+N-1$ do
    $r_{k+1} \leftarrow r_k + v_{k+1/2}\Delta t$
    $v_{k+3/2} \leftarrow v_{k+1/2} + a(r_{k+1})\Delta t$
End for
```

Fig 12. Pseudo code of the integrator for the radius

The implementation of those is discussed later in the paper.

Though inelastic collisions could be implemented, we will stick to elastic ones in this simulation. Implementation of the elastic collisions follows this detailed Stackoverflow posts and demonstrations : https://stackoverflow.com/a/35212639 . It has to be noted that multiple implementations of collisions have been tested, however, we found out very quickly that we did not take into account the angle between the two colliding objects, which would result in very strange behaviors that were simply impossible. The two equations for the collision between two objects. They give for each object the new speed vector in cartesian coordinates. Here, the dot product introduces a condition on the angle.

$$\mathbf{v}'_1 = \mathbf{v}_1 - \frac{2m_2}{m_1+m_2}\frac{\langle \mathbf{v_1}-\mathbf{v_2}|\mathbf{x_1}-\mathbf{x_2}\rangle}{\|\mathbf{x_1}-\mathbf{x_2}\|^2}(\mathbf{x_1}-\mathbf{x_2}) \tag{4}$$

$$\mathbf{v}'_2 = \mathbf{v}_2 - \frac{2m_1}{m_1+m_2}\frac{\langle \mathbf{v_2}-\mathbf{v_1}|\mathbf{x_2}-\mathbf{x_1}\rangle}{\|\mathbf{x_2}-\mathbf{x_1}\|^2}(\mathbf{x_2}-\mathbf{x_1}) \tag{5}$$

Implementation will be discussed later in the paper.

For the game, in case an object leaves the playing, 2 options are available. The object can either get destroyed and taken out of the simulation, or use the periodic boundary conditions. In the latter case, its radial speed is flipped (meaning instead of having a positive radial speed it will have a negative one), and we add half a turn to its angle, effectively sending it on the other side of the simulation.

c. Program structure and inner workings.

The provided program comes with two distincts parts. The **PhysicsEngine**, responsible for projectile simulations and game specific files for game logic, display and settings. One may wonder what the **utilities folder is for.** It only contains a function that clears the console screen, independent of the operating system using build checks.

PhysicsEngine

```
[4.0K]  PysicsEngine
├── [5.3K]  collision.py
├── [ 494]  config.py
├── [ 13K]  engine.py
├── [ 10K]  engine_rev2.py
├── [3.7K]  game_loop.py
├── [ 198]  __init__.py
└── [  51]  physicalConstants.py
```

Above is the list of the files available. **__init__.py** allows the import of the engine as a package rather than a module. It allows us to import everything from the same place, using **__init__.py** to only expose the required functions, thus avoiding potential conflicts. Most of the documentation and comments can be found in the **documentation/PhysicsEngine** directory provided with the project. Looking through the code will provide further comments on implementations. REquired dependency to run the engine is **numpy.**

i. collision.py

As its name states, this file contains all the collision implementation in two exposed functions. **DetectCollision()** and **update_colliding_objects()**. Two additional hidden functions allows conversions from polar to cartesian coordinates and vice-versa

- DetectCollision() :

    Used to detect collisions between 2 objects. Takes three arguments, a **list[object]**, containing all the objects in the simulation; an **integer** representing the iteration number, for debugging purposes; and a **float** representing the time step duration. Returns a **list[(int, int)]** of pairs of objects colliding.

    The program loops through each pair of objects and compares their distance to the sum of their radius. If that distance is smaller, then the objects are colliding. Further checks are made to ensure we do not detect ongoing collisions twice. In case the collision in a pair was already detected before, a check is made on the **object.WasColliding** list of each object. if one is detected in the other's list, collision is aborted and the pair is not added to the collision update queue. However if those checks are passed, the pair is added.

    There is room for improvement. Using parallelisation would reduce the current complexity of the function to O(n), where each pair check could be accomplished on each particle in the GPU, avoiding having the CPU looping for (n(n-1))/2 times.

- update_colliding_objects() :

    Used to update objects in a collision pair. Takes three arguments, **(int, int)** pair of objects considered to be colliding; a **list[object]** representing all objects in the simulation; and a **float** corresponding to the time step.

    The implemented collision method was discussed earlier in the physical models. This technique being implemented in cartésien coordinates, we project the polar vectors on cartesian coordinates before applying the method and converting back to polar. It has to be noted that updated objects will have their position updated using a multiplication by delta t. This is a limitation of this implementation since the first order relativistic correction and gravitational attraction are neglected during this time step for this object. it does induce an error in the simulation, hence the need to have the smallest deltas possible.

Room for improvement is also available here. The function has to be called multiple times, once per pair, to update the whole situation. Remaking another function with parallelization, using shaders, may become quicker in situations with a large amount of particles. However for low update count a CPU will be preferred as its base clock speed is much higher than the one of a GPU core.

ii. <u>config.py</u>

This file is a python file loaded in the engine that sets variables necessary to the simulation that can be customized. A default configuration is provided.

<u>engine-rev2.py</u>

Second installment of the engine. The first iteration was used to test implementation and behavior of equations.

- <u>object class</u>

  This class is the only one created and represents physical objects in the simulation. An object is created by invoking the class and creating an instance of it, using as input parameters, a type, the starting radius, angle, speed on the radius, speed on the angle; a mass and a radius. Documentation is available in the **documentation/PhysicsEngine** directory, especially under the **__init__** function description. Multiple functions associated with the object can be called in order to retrieve variables or update them. i.e. **UpdateL0()**

- <u>ComputeDeltaT()</u>

  Used to new delta for the next step. Takes two inputs, a **list[object]** containing all objects in the simulation; and the previous delta t as a **float.**

The simulation needs to have its time step updated at every step, the reason being that the closer you are to the blackhole, the faster you go and the smaller the timestep needs to be. A second reason is that objects traveling more than half their radius each step risk to pose problems in the collision detection, where they will not collide. Each delta is computed for each object, and only the smallest one is returned.

Parallelisation may be implemented here to reduce the complexity from **O(n)** to **O(1)**

- deltaless_deltat()

Same job as **ComputeDeltaT()** but does not require a delta to be entered. A delta is necessary to compute the derivative of the position on the angle as they are not recorded like the speed on the radius. An override argument can be implemented to force the return of the function.

Same possibility for improvement.

- compute_l0()

Used to compute the angular momentum of the object using its radius and angular speed, using the previously described equation.

- theta_next()

Function called in **nbody_coupled_integrator()** to compute the next angle of an object, using the the provided equation shown in the pseudo code of the calculation of theta

- acceleration()

Function called to compute the acceleration of an object using Newton's universal law of attraction, the first order relativistic correction and the conservation of the angular momentum

$$a(r) = -\frac{GM}{r^2} + (r - \tfrac{3}{2}R_S)\frac{\ell_0^2}{r^4}. \qquad (6)$$

- rk_next(), vk_next()

  Functions called in **nbody_coupled_integrator()** to compute the next radius and radial speed of an object, using the provided equations shown in the pseudo code of the leapfrog integrator.

- collision()

  Function called in the main game loop to check for collisions and update the objects colliding.

- nbody_coupled_integrator()

  Main function of the simulation, responsible for updating all positions of an object using the aforementioned functions. Takes 3 arguments, a **list[object]** containing all objects of the simulation; an **object** representing the blackhole of the simulation; and a **float** representing the time step.

iii. engine.py

Does not serve any purpose. It is a different implementation of the integrators that do not fit the requirements of the engine. only here for historical purposes.

iv. game_loop.py

This file contains a single function, responsible for updating. Takes 4 inputs, a **list[object]** containing all physical objects in the simulation; an **object** that is considered as the black hole of the simulation; a number of steps to do; and a **float** that is the initial time step for the simulation.

First, a check is made to see if **statusDebug** is set to true. If it is, debug logs will be printed in the console in the form of the iteration number, the current time step and an estimation of the remaining time. This debug is executed every step configured in the configuration file. Default is once every 100,000 steps.

If no objects are in the list of physical objects of the simulation, the simulation will instantly stop. You can comment the corresponding line to deactivate that behavior.

Object collisions are then detected using the **collisions** function described in **engine_rev2.py**. And the time step is calculated using **ComputeDeltaT()** from **engine_rev2.py.**

Then for each object, the new coordinates are calculated using **nbody_coupled_integrator().** Out of bounds checks are made in regards to the set **conf_outOfBoundMax** and **conf_outOfBoundMin** settings set in **config.py**. Defaults are, for the minimum out of bound, 1 as our unit is in Schwarzschild radius, and 50 as a max. 40 is the most appropriate size for the renderer, while guaranteeing a respectable playing area and a black hole visible, the max value is independent from the set **out_of_bound** from the game settings. 50 is selected to avoid any conflicts in checks when using the periodic boundaries return conditions. If an object is found to be out of bound from the simulation (not the game area), it is deleted from it for this run.

Remaining objects have their variables updated before being returned.

d. Game specific files

i. GameDisplay.py

This file contains all functions related to game display, such as screens for win and loss, information, firing prompts, game banner,... Only the **GameChooseStartShipPosition()** function is used for actual game logic by implementing the initial condition values inputs.

ii. GameLogic.py

Module containing game logic functions that are used in the game.

- ProjectileMatch()

Takes two arguments, a self object representing data of the simulation, namely the projectiles list and the inventory are used here; and array from a split string. This function is used to match the input string to the correct projectile type when firing in the game. 4 checks are made for each, to check for uppercase and lowercase combinations using a match syntax. Lowering the string would make the checks case insensitive and lower the number of checks, but has not been implemented yet.

- Shoot()

  Takes a split string of the input and a projectile type. Gets the default speed and mass of an object from the game settings and the firing angle set by the player, and returns its radial and angular speed as well as the projectile.

- CreateMissile()

  Takes 7 arguments, the speed of the missile on r and theta, as well as its type, the firing ship, and the time step and instantiates a physical object in the simulation with the missile properties. Collision tags are added to avoid detecting collisions on the next detection as the missile is instantiated on top of the ship.

- GameruleCollisions()

  Takes a **(int, int)** containing the indices of two colliding objects and a list. It is called for each collision to check the game rules and apply the right decision based on the set game rules. For example, the first match case checks if a Heavy type projectile comes in contact with a Light one, which would result in the destruction of both objects. All checks are made using match statements. A truth table is provided at the end of **GameSettings.py**. This function can be modified to fit other game rules.

e. GameSettings.py

File used to store all settings for the game. Modifying **steps_per_frame** may lead to instabilities, default is set to 1 and should stay that way. The aesthetics section of the configuration file may be changed, however it is in a very early development stage and changes may lead to instabilities. Current implementation of the renderer is bugged and therefore, **fade_off** and **color_distance_treshold** have been set to ease the readability and functionality of the renderer to the maximum. Future updates will revamp the system and allow for better controls on that part.

The file is quite commented and navigation should be pretty easy.

f. render-mode.py

This file contains the graphical implementation of the game, made in **moderngl. Moderngl** is a python wrapper library for **OpenGL** which is one of the most famous graphical libraries for computer graphics. Here,we do not use any geometry, the only object that will be in the OpengGL scene will be a quad on which we draw a rendered texture. As some of this goes beyond the scope of Python, we will only briefly discuss what is done in **moderngl** and in the shaders.

i. What's a shader?

A shader is a program used by a GPU to do calculations. They are mostly known for being used in computer graphics and video games, but can also be used for matrix calculus optimization, or convolutions (i.e. game of life implementations). Here we will use them to render a display a texture that will represent our simulation.

ii. Why use them here and not a simple matplotlib plot?

While a plot would be sufficient, implementing shaders will allow us to lower the RAM used by the program for the game. A plot necessitates multiple coordinates to create the path of the object, where the shader, using the technique described below, will display orbits without requiring more RAM than for one point per frame.

iii. Fade-out trails

The method we will use to render orbital trails is pretty simple. We first implement a ping pong frame buffer algorithm as described below.

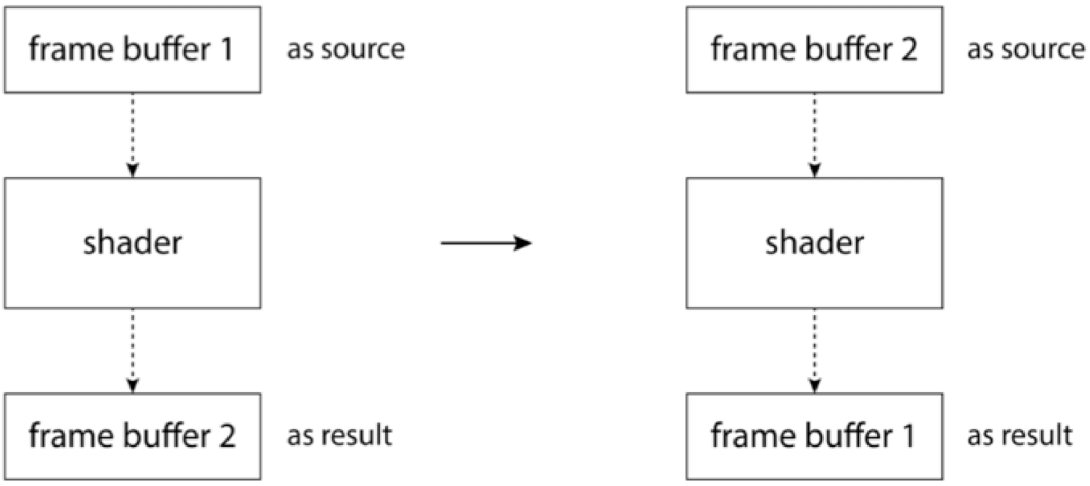


Fig 13. Ping Pong rendering technique visual diagram

This algorithm utilizes two frame buffers that are switched each frame. When rendering the first frame, we write data in frame buffer 2 using data from frame buffer 1 passed in a shader, and display the image in frame buffer 2. WHen rendering the second frame we do the opposite, we write data in frame buffer 1 using data from frame buffer 2 passed in a shader, and display the image in frame buffer 1. And we repeat that process of inverting frame buffers each frame. This technique allows us to pass data from the previous frame to the next. The reason we need to use two frame buffers is that the frame buffers are treated as **sampler2D** when getting data from it. A **sampler2D** is read only, which means we can’t write to it and hence why we use a second frame buffer and switch them after each render.

We can now get information from the last frame using this technique. In order to display a trail for the previous position, the trick is to render a frame each simulation step, and by lowering the opacity of the previous frame by a small factor, the previous circles we rendered will slowly fade away each frame, giving us a trail behind each object, see example below.

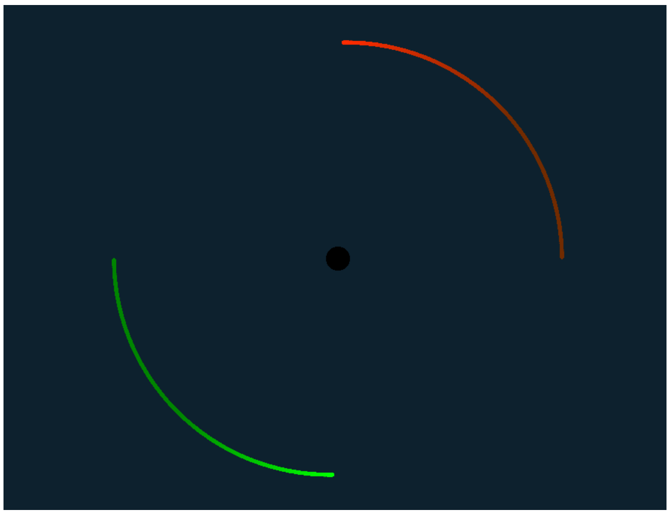


Fig 14. Rendering example 1

As we can see here, the objects leave a trail behind themselves, which allows us to preview their previous position. This technique does not require access to previous positions, and thus will consume less RAM. Those data are still logged inside each object, but can be either written to a file, or just deleted to lower the ram needed without losing the orbit previews. This method comes with a limitation when using variable time steps however.

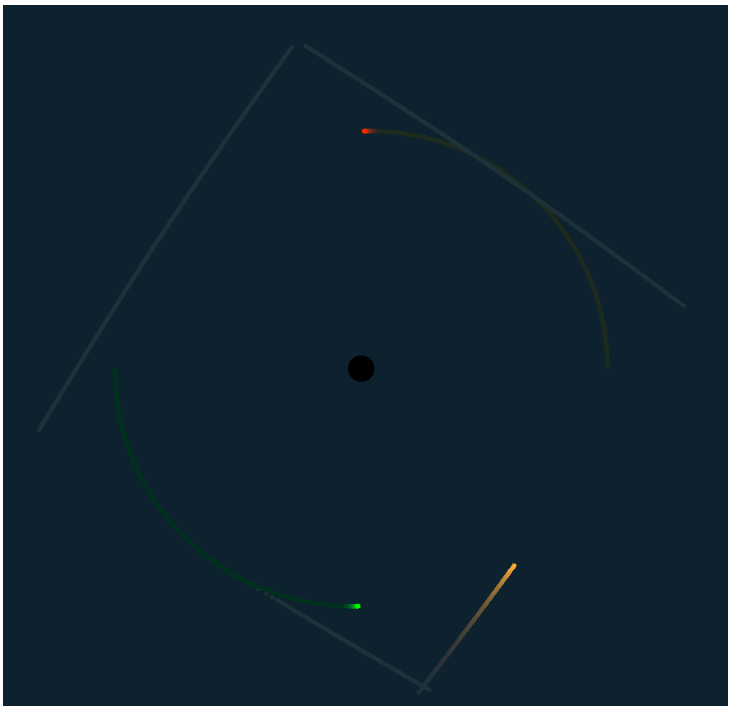


Fig 15. Rendering example 2

As you can see in this image, the orange object, which is much faster than the red and green one, has a trail of a normal size compared to the others. That is due to the time step variation inside the simulation. The fade-off happens every frame and is not proportional to the time spent. That could however be implemented later. You may also see that trails do not disappear here. There is an issue in the current implementation of that display. Colors can be defined as a 3D vector. Using traditional 3D formulas we can compute a “distance” between colors, and depending on that distance clamp them to the background color. However an issue in the shader prevents this from happening. This has not been troubleshooted yet but will be in future releases.

Though this is the base of the technique we use, a different approach has been implemented. Since we don’t need to access the data from the previous frame, but just darken it, a darken program has been made that simply lowers the opacity of the whole frame before rendering the next pass.

iv. How do we do that

The renderer is made of multiple parts. The first one is a **CircleRenderer** class. It is used to manage and draw circles on the texture. One limitation is that a fixed maximum amount of objects can be displayed. Drawing a circle needs position and color data. These are stored in two separate arrays present in the GPU VRAM. Those array sizes cannot be modified during execution, thus need to be instantiated with a length. The current max amount of circles that can be displayed is hard coded at 10,000. The data array is comprised of all position data for each circle following that schema : [$x_1, y_1, x_2, y_2, \dots , x_n, y_n$] and the color array follows the same schema : [$R_1, G_1, B_1, R_2, G_2, B_2, \dots , R_n, G_n, B_n$]. In order to not display the maximum amount of circle every time, we only populate the first items of the arrays and pass the number of circles to render, thus only the x-first circles, corresponding to active simulation objects, are rendered. The **CircleRenderer** class has a **Update()** function that allows us to pass the positions and colors of each circle and update the GPU buffers accordingly.

Rendering the black hole is done with an almost identical class as it does not require updating the arrays, but has to be rendered later in the frame. It only has **setup()** and **render()** functions that are used to set up and render the black hole.

All of that is used by an **App** class. The class is an extension of the **moderngl_window.WindowConfig** class. **WindowConfig** is a preconfigured class from the

moderngl_window package, used in tandem from **moderngl.** It allows users to create a window inside the operating system. The **__init__()** function is used to set up the window. We define here all starting values for the simulation, as well as defining programs that are using the shaders to render the scene. This part of the code is **moderngl** jargon. We will not take too much time on it.

To define a rendering program you need a fragment shader and a vertex shader. Fragment shaders define colors for each pixel of the window, and vertex shaders modify the geometry of the scene. All shaders used can be found in the **shader** directory. Four programs are declared, **blackhole_program, circle_program, blit_program, darken_program.** The blackhole and circle programs are used to display the black hole and circles, the blit program allows copying the data to the screen, and the darken program is the part that fades-off the previous frame, as discussed in previous sections. Passing variables between the GPU and the CPU is done using **program[“var_name”] = var** where **var_name** is the name of the variable as declared in the shader.

Four variables used in the simulation are defined here as well. **iteration** will count the total number of iteration in the simulation, **deltat_per_turn** is used to define how much time needs to at least pass before switching triggering a move for the player, **counterdeltat** counts the amount of time spent since the last player turn, and **masterCounterDeltat** is used to count the total amount of time spent since the start of the simulation.

The next section is used to define the simulation starting condition, such as initializing counters and the global time counters. Ship starting conditions are decided here as well using **GameChoseStraShipPosition().**

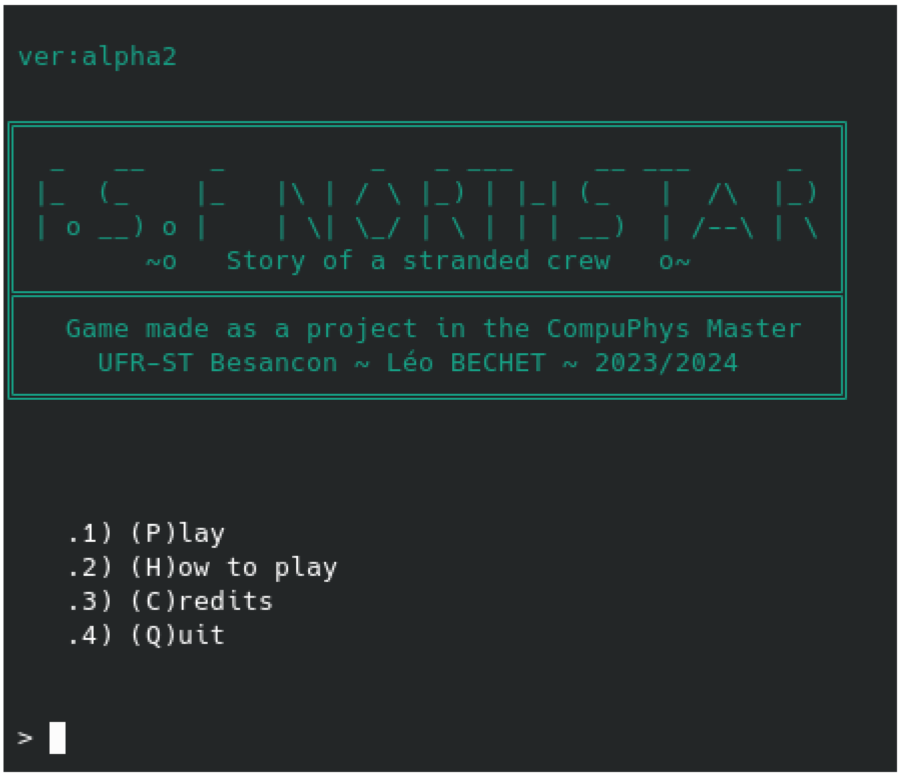
The **render()** function of the class is called every frame. depending on your screen refresh rate, it can be 25Hz, 60Hz, 120Hz,... This part of the code is well commented, reviewers wanting to understand the code are encouraged to read this part while following with the code. The first part tell **moderngl** what settings we use, and starts by darkening the last frame by rendering the last frame with a fade-off on the quad with the **darken_program** program. Simulation statements come in the next part. A time step is computed and time counters are incremented. The **game_physics_loop()** is called to advance the simulation by a time step, and collisions are then checked. Out of bound checks are then performed. If the ship or the target are considered out of the simulation, they are considered to have fallen in the black hole and the corresponding end is summoned. And the same is done if they are too far from the set out of bound max radius set in the game rules, and the periodic boundary condition is not active. Note that the out of bound of the game is under the set out of bound of the simulation. However if the boundaries are active, the ship is spun around and the radial speed is inverted. Note that the long time simulation for escaping objects has not been implemented yet.

Once the physics simulation has been done, we check if a user input is required, i.e. does the time since the last player turn is greater than the set time per turn. If it is, we clear the console screen and call the right GUI from **GameDisplay**. If the user inputs a “l”, it means he wishes to fire a projectile, we enter the firing loop accordingly. If the player is out of missile the game will display as such and continue the simulation for a step. Note that long time simulation once you run out of missiles has not been implemented, as such the player needs to simulate a few times and is the sole decider of whether he has lost or not. The program displays the firing interface and parses the order. If a parsing error is detected, it will restart the loop. A missile is only launched after the user inputs “c” to confirm, or the launch is aborted if he writes “cancel”. The simulation and input parts end here.

This next part is dedicated to rendering the new frame. Each projectile's coordinates are converted from polar to cartesian and appended to a list. The same is done with their colors. Colors can be customized in the **GameSettings** file. And finally the constructed lists are passed to the VRAM of the GPU to be used in rendering. Please note the order in which each step is rendered. We first render the circle to the frame buffer, use the blit program to render it on the screen, and finally render the black hole. If the black hole rendering was done before, particles rendered in the next frame would be displayed above it, which is not physically correct, as the black circle represents the event horizon.

v. <u>Game loop of render_mode</u>

This loop handles the menu. Multiple options are available before starting the renderer and the game itself



```
ver:alpha2

  _   __    _         _   _ ___     __ ___       _
 |_  (_    |_    |\ | / \ |_) | |_| (_    |  /\  |_)
 | o __) o |     | \| \_/ | \ | | | __)  | /--\ | \
        ~o   Story of a stranded crew   o~

   Game made as a project in the CompuPhys Master
     UFR-ST Besancon ~ Léo BECHET ~ 2023/2024



   .1) (P)lay
   .2) (H)ow to play
   .3) (C)redits
   .4) (Q)uit


>
```

Fig 16. Menu of the game, in the while loop

Four options are available, each check is here to display the right screen, and it is all bundled in a while loop to keep the game running until play is selected.

g. <u>Program diagrams</u>

Diagrams can be found in the **documentation/diagrams** directory.

5. Quality Approach

a. Technical tests and program reliability

A jupyter notebook can be found in the **ressources** directory. This notebook utilizes the PhysicsEngine to run tests on its stability, the efficiency of the collision algorithm, conservation of energy and precision of the integrator. Potential parallelization and optimization methods are presented. The web version of the notebook can be found here : github.com/lele394/PythonPool1/blob/main/ressources/PhysicsEngine-Precision_and_reliability.ipynb

The simulation has been tested using symmetrical systems. in different configurations to simulate different types of collisions. Elastic collisions have been visually debugged with the help of the renderer, and incoherent results have been hunted down and fixed (for the most part). Errors in checks for collisions and out of bound conditions should also have been eliminated.

No data saving has been implemented due to the scope of the game. It can however be easily done.

6. Outcomes and physical discussion

An example of use of the simulation can be found in the user’s guide. The use of the program is explained and can be repeated. Multiple examples can also be found in the PhysicsEngine reliability and testing notebook.

The only data needed for this program to operate is user input defining the starting conditions of the simulation. The engine can be customized to fit the player's needs.

No output data is provided due to the scope of the game, but saving data can be easily implemented as stated in the previous part.

It has to be noted that the physics engine provided is quite limited and far from perfect. Collision detection is very basic and only works with circles; collisions are considered elastic, and we can spot divergence from a real case. Such an example can be found in the Jupyter notebook, in the last example. The orbits do not represent something that we could normally see, see below.

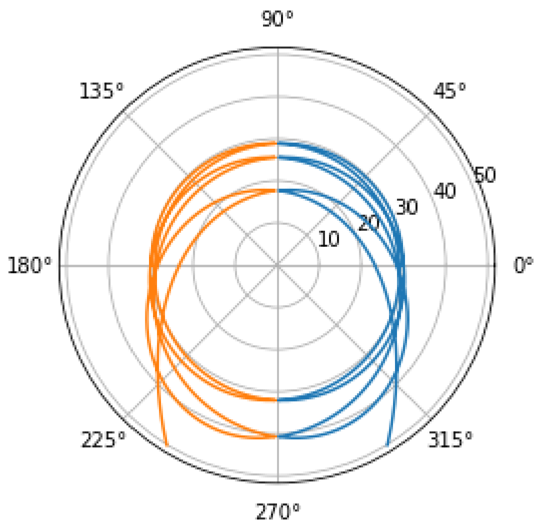


Fig 17. Not really realistic simulated orbits, at least I think they’re not too realistic.

It is strange, as the energy is conserved. As objects grow closer to the center, they are sent way farther out from the black hole.