Technical Report

MISSION #2: THE AIMBOT™

by

Team Goku

Goku team:

Marc Gallardo - Gerard Romeu - Albert Espinosa - Marc Ariza
Marc San José - Angel González - Adrián Mirabel

Index

Introduction

What is an AimBot?

In the video game industry aimbot refers to a software mainly implemented in video games of the shooter genre.

Game development wise, this software is implemented to make IA controlled bots be accurate with their shots, regardless of terrain and other environmental conditions such as player/enemy position. Player wise, however this software is usually used to cheat.

The Aimbot that the development team has developed, however, will be based on the Monte-Carlo strategy, which its most famous implementation has been in the Worms Franchise.

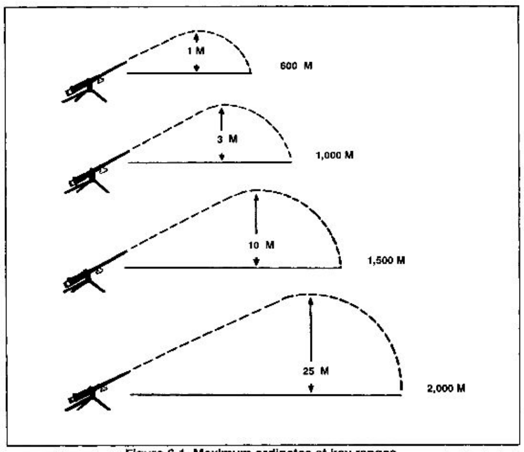


Figure 6-1. Maximum ordinates at key ranges.

Report Structure

The report will have the following structure:

Firstly, the basic equations to compute all the required calculus will be presented and explained

Secondly, the AimBot's main methods and variables, along with the Aimbot Core processes workflow, will be laid out and analyzed.

Lastly, this report will close up with the validation of the AimBot Core along with the project's conclusions.

Equations

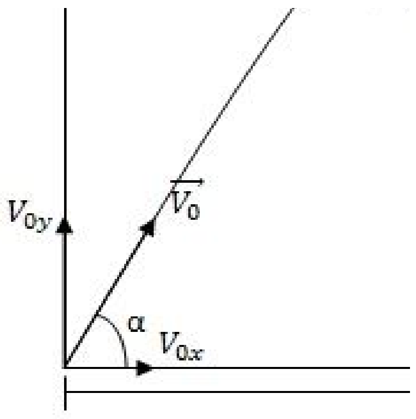
In the following section the equations used to implement the Monte-Carlo strategy will be presented and described.

Vector of the projectile velocity

To calculate the initial velocity of a particle acting as a projectile, the equations shown below were used. By using the equations, both the ‘X’ and ‘Y’ components of the initial velocity vector can be calculated while taking into account a throwing angle (represented as “Alpha” in the figure below).

$$Vox = Vo * \cos(\alpha)$$

$$Voy = Vo * \sin(\alpha)$$



Euler Integrator

The Euler integrator is a first order integrator as explained in this report

https://drive.google.com/file/d/1hA4uq0gxbkzDyzxeBFYB2SpZNvpwuoOb/view?usp=sharing

Development of the Code

The algorithm has been implemented in C++ programming language. In this language the algorithm is extremely fast and allows an easy debugging of the code. For the application of this algorithm in video games,this algorithm doesn't have a graphical view, however, this fulfils his main objectives and works efficiently.

The overall code can be summarized as the following:

1. **Initializing the Simulation World and its elements**

    a. Set configuration parameters for the World (gravity, world height, world width, number of frames, fluid velocity, fluid density…)

    b. Set configuration parameters for the projectile particle (position, speed, acceleration, mass, surface…)

    c. Set configuration parameters for the target particle (position (The position is randomized), speed and acceleration(The particle is not moving the speed and acceleration is (0,0,0))).

    d. Randomize wind velocity

2. Monte-Carlo Algorithm

a. Synchronize the projectile, the target and the number of iterations the algorithm is going to run for.

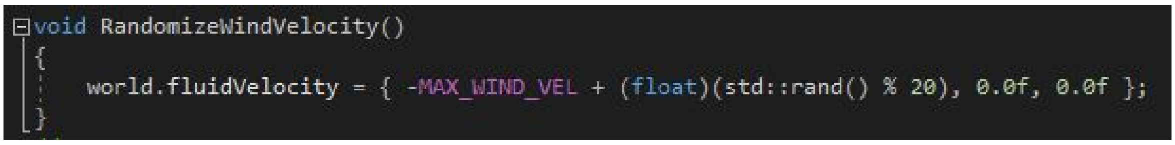
b. Randomize the projectile's velocity module and throwing angle.

c. Compute the PropagateAll() Method.

3. Propagate All function

a. Compute speed for the velocity vector.
b. Compute the Euler Integrator for the calculus of the projectile trajectory.
c. Checking if the projectile's hit the target.

The initialization stage declares all the variables for the projectile and the target properties (position, velocity, acceleration, surface…) , the world (height, width, fluid velocity, fluid density…) and the randomization of the variable fluid velocity to change the wind velocity each iteration.



```
void RandomizeWindVelocity()
{
    world.fluidVelocity = { -MAX_WIND_VEL + (float)(std::rand() % 20), 0.0f, 0.0f };
}
```

Once all the variations have been correctly initialized, the Monte Carlo algorithm is run. The algorithm starts to iterate from iteration index nº0 until the maximum number of iterations is reached.

The current fps should be synchronized with the global fps. However, in this particular case, the development team felt more inclined towards creating a timer. The timer adds 1/60 of a second each time an iteration is completed.

The algorithm propagates the projectile's state for 5 seconds in the Simulation World seconds (60 fps * 5 seconds = Number of iterations on the propagation loop of the AimBotEulerIntegrator() method) and at the end of that period of time, if the projectile has not hit the target, it is discarded and the next projectile will be initialized to the origin position and will have its initial velocity module and angle randomized.

After the aforementioned, the algorithm will have reached the PropagateAll() method. The first operation run by the method will be to calculate the projectile's velocity vector in the X and Y axis. After that,the projectile's state will be propagated by the AimBotEulerIntegrator() Method. Lastly, the CheckHit() function, which returns true on target hit, will be used to check whether or not the projectile has hit the target.

When the iteration is completed the results are represented in the screen.

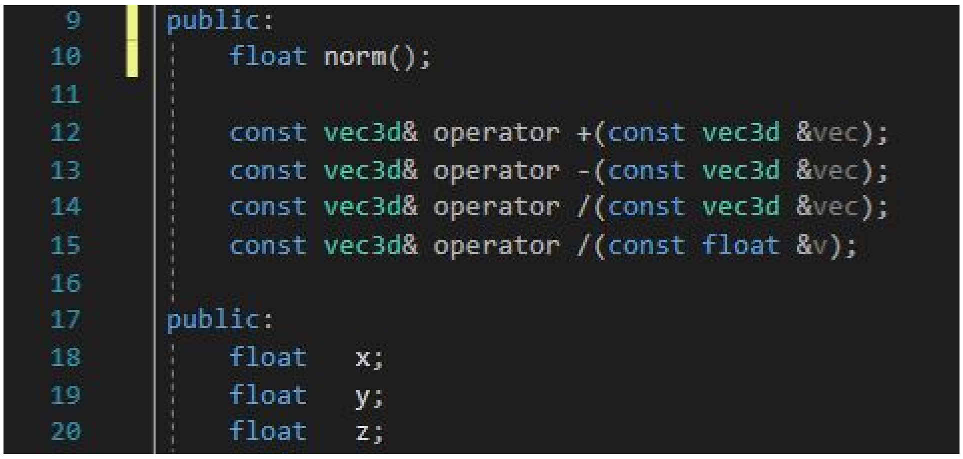
At this point there could be two possible outcomes. Either the projectile has hit the target and the algorithm stops, or the projectile has not hit the target and the algorithm continues to run until the target is hit or it reaches the maximum amount of iterations.

Methods and variables that compose the AimBot Core

To develop the AimBot core, diverse methods and structures (classes) were created. Both the aforementioned groups will be broken down down below:

- **Classes:**
  - **vec3d:** Declares 3D Vectors.

    Methods & Variables:



```
public:
    float norm();

    const vec3d& operator +(const vec3d &vec);
    const vec3d& operator -(const vec3d &vec);
    const vec3d& operator /(const vec3d &vec);
    const vec3d& operator /(const float &v);

public:
    float   x;
    float   y;
    float   z;
```

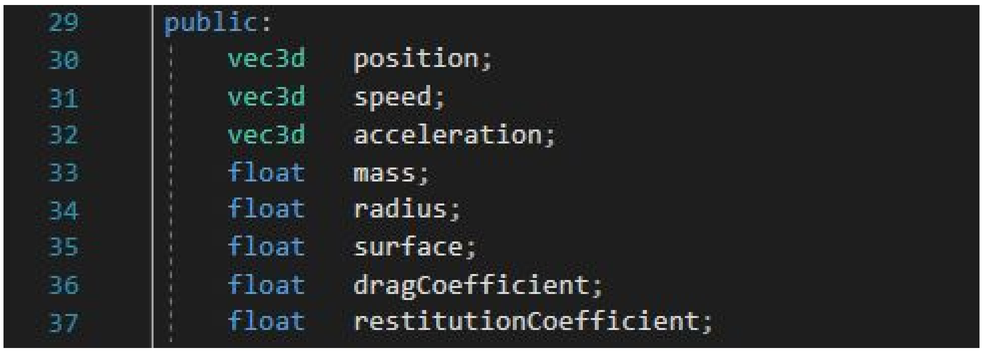
Methods:

- **norm():** Gets the norm of a vec3d.
- **operator +(...):** Allows to add a vec3d to another.
- **operator -(...):** Allows to subtract a vec3d with another.
- **operator /(vec3d&):** Allows to divide a vec3d by another.
- **operator /(float&):** Allows to divide a vec3d by a value.

Variables:

- **x:** X component of the vec3d vector (X Axis).
- **y:** Y component of the vec3d vector (Y Axis).
- **z:** Z component of the vec3d vector (Z Axis).

- **Particle:** Defines objects / particles and their properties. All particles will be considered spherical.



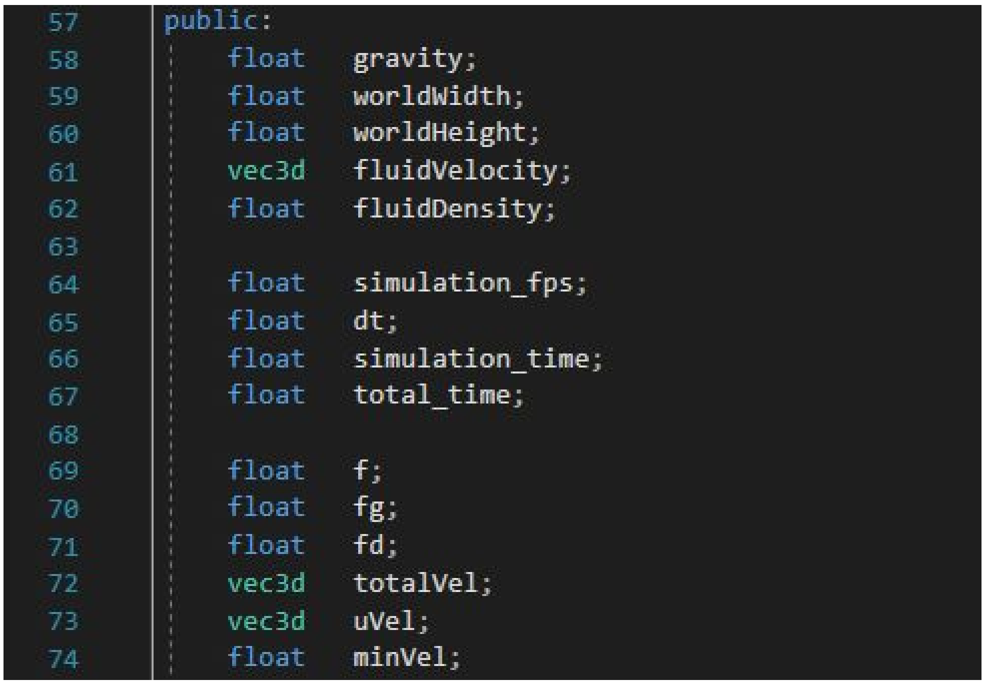
```
public:
    vec3d   position;
    vec3d   speed;
    vec3d   acceleration;
    float   mass;
    float   radius;
    float   surface;
    float   dragCoefficient;
    float   restitutionCoefficient;
```

<u>Variables</u>:

  - **position:** Position vector of a particle [m].
  - **speed:** Velocity vector of a particle [m/s].
  - **acceleration:** Acceleration vector of a particle [m/s^2].
  - **mass:** Mass of a particle [Kg].
  - **radius:** Radius of a particle [m].
  - **surface:** Front surface area of a particle [m^2].
  - **dragCoefficient:** Drag coefficient of a particle [NA].
  - **restitutionCoefficient:** Restitution coefficient of a particle [NA]

- **World:** Defines the simulation world's base properties and the forces that interact within it.

  Variables:



```
public:
    float   gravity;
    float   worldWidth;
    float   worldHeight;
    vec3d   fluidVelocity;
    float   fluidDensity;

    float   simulation_fps;
    float   dt;
    float   simulation_time;
    float   total_time;

    float   f;
    float   fg;
    float   fd;
    vec3d   totalVel;
    vec3d   uVel;
    float   minVel;
```

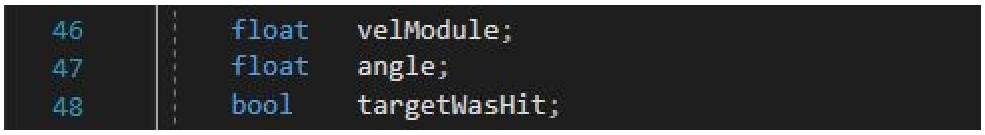
  - **gravity:** Gravity of the simulation world [m/s^2].
  - **worldWidth:** Width of the simulation world [m].
  - **worldHeight:** Height of the simulation world [m].
  - **fluidVelocity:** Fluid velocity in the simulation world [m/s]. (Air…)
  - **fluidDensity:** Fluid density in the simulation world [m/s]. (Air…)

  - **simulation_fps:** Average fps of the simulation.
  - **dt:** Timestep of the world [1/fps].
  - **simulation_time:** Time simulated for a specific Monte-Carlo [s].
  - **total_time:** Total time allocated to Monte-Carlo() [ms].

  - **f:** Total force applied on the projectile by the World [N]. [F = ma];
  - **fg:** Gravitational Force of the Simulation World [N]. [Fg = mg];
  - **fd:** Drag Force of the Simulation World [N].[Fd = 0.5*ϱ*v^2*Cd*A];
  - **totalVel:** Velocity variable of Fd [m/s].Only applied to the X Axis.
  - **uVel:** Unitary particle-wind velocity vector [m/s].
  - **minVel:** Threshold value for the minimum velocity allowed [m/s].

- **AimBotVariables:** Declares miscellaneous variables related with the AimBot.

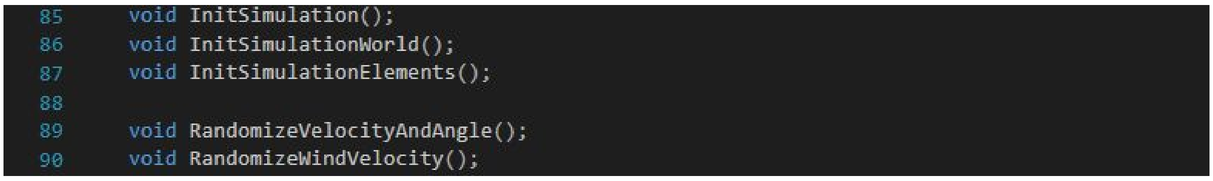
  Variables:



```
float   velModule;
float   angle;
bool    targetWasHit;
```

  - **velModule:** Randomized velocity module [m/s].
  - **angle:** Angle at which the projectile is thrown [º].
  - **targetWasHit:** Flag that keeps track of whether or not a projectile has hit its target.

- **Methods:**
  - **Variable Initialization Methods:**



```
void InitSimulation();
void InitSimulationWorld();
void InitSimulationElements();

void RandomizeVelocityAndAngle();
void RandomizeWindVelocity();
```

  - **InitSimulationWorld():** Initializes the variables of the Simulation World (gravity, boundaries...)

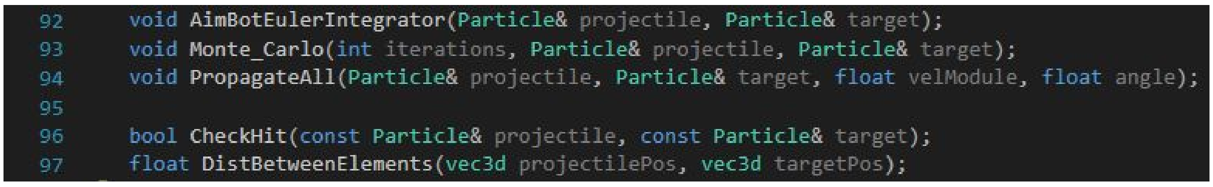
  - **InitSimulationElements():** Initializes the variables of the elements of the simulation (projectile and target).

  - **InitSimulation():** Calls both InitSimulationWorld() and InitSimulationElements().

  - **RandomizeVelocityAndAngle():** Randomizes the projectile's velocity vector module and throwing angle with the std::rand() method.

  - **RandomizeWindVelocity():** Randomizes fluidVelocity (wind speed) with the std::rand() method.

- **Main AimBot Methods:**



```
void AimBotEulerIntegrator(Particle& projectile, Particle& target);
void Monte_Carlo(int iterations, Particle& projectile, Particle& target);
void PropagateAll(Particle& projectile, Particle& target, float velModule, float angle);

bool CheckHit(const Particle& projectile, const Particle& target);
float DistBetweenElements(vec3d projectilePos, vec3d targetPos);
```

- **AimBotEulerIntegrator(...):** Euler Integrator adapted to take into account the drag force. Propagates the state of a projectile. Takes a Particle projectile and a Particle target as arguments (both being passed as reference).

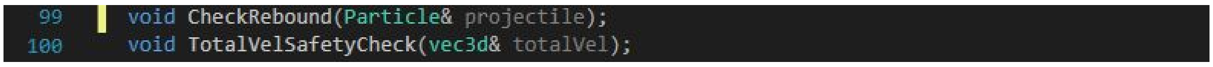
- **Monte_Carlo(...):** Calls the RandomizeVelocityAndAngle() and the PropagateAll() methods.Takes a number of iterations along with a Particle projectile and a Particle target as arguments (with the latter two being passed as reference).

- **PropagateAll(...):** Sets the velocity vector of the projectile according to the randomized velocity module and calls the AimbotEulerIntegrator() method. Lastly it checks whether or not the projectile has hit the target by calling the CheckHit() method. Takes a Particle projectile and a Particle target (both being passed as reference) and a velocity module along a throwing angle as arguments.

- **CheckHit(...):** Checks whether the projectile has hit the target or not by calling DistBetweenElements(), which returns the current distance between the projectile and the target. Takes a Particle projectile and a Particle target as arguments (both being passed as reference).

- **DistBetweenElements(...):** Calculates the distance between the two positions passed as arguments.

- **Supporting AimBot Methods:**



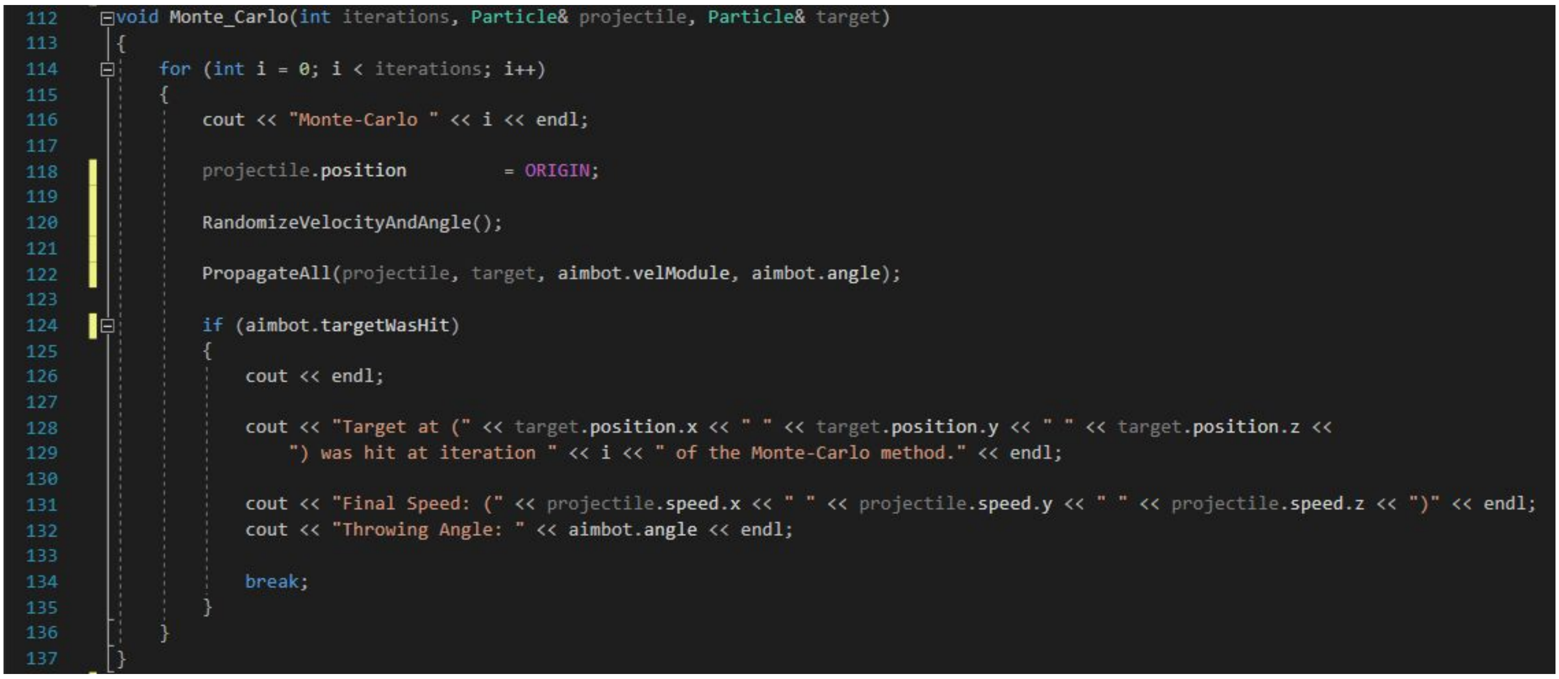
```
void CheckRebound(Particle& projectile);
void TotalVelSafetyCheck(vec3d& totalVel);
```

  - **CheckRebound(...):** Checks whether the projectile has collided against the simulation world's limits/walls or not. Takes a Particle projectile as argument. Passed as reference.

  - **TotalVelSafetyCheck(...):** Checks that the particle-wind unitary velocity vector is not below the minimum velocity threshold (0.0001 m/s). Passed as reference.

AimBot Core

The AimBot Core is contained within the aforementioned Monte-Carlo(...) method.



```
void Monte_Carlo(int iterations, Particle& projectile, Particle& target)
{
    for (int i = 0; i < iterations; i++)
    {
        cout << "Monte-Carlo " << i << endl;

        projectile.position         = ORIGIN;

        RandomizeVelocityAndAngle();

        PropagateAll(projectile, target, aimbot.velModule, aimbot.angle);

        if (aimbot.targetWasHit)
        {
            cout << endl;

            cout << "Target at (" << target.position.x << " " << target.position.y << " " << target.position.z <<
                ") was hit at iteration " << i << " of the Monte-Carlo method." << endl;

            cout << "Final Speed: (" << projectile.speed.x << " " << projectile.speed.y << " " << projectile.speed.z << ")" << endl;
            cout << "Throwing Angle: " << aimbot.angle << endl;

            break;
        }
    }
}
```

The Core's processes will be run as follows:

- The Monte-Carlo(...) method will receive a number of iterations, a Particle representing a projectile and a Particle representing the target as arguments.

- The number of iterations will define the amount of times the loop inside the aforementioned method will run for searching for a hit (Ex: 100 iterations → loop will be run 100 times).

- The projectile's position will be set (or reset after the 1st iteration) to Origin ({0.0f, 0.0f, 0.0f}).

- Both the projectile's velocity module and throwing angle will be randomized with the std::rand() function.

  The randomized variables will be limited to the following maximum and minimum values:
  - <u>Velocity Module Limits</u>: 0 m/s ~ 50 m/s.
  - <u>Throwing Angle Limits</u>: -180º ~ 180º.
  - <u>Wind Velocity Vector Limits</u>: -10 m/s ~ 10 m/s.
  - <u>Target Position Limits</u>: { 0m ~ 20m, 0m ~ 10m, 0m }

- The PropagateAll(...) method will receive the projectile, the target and the randomized velocity module and throwing angle. Then, inside the aforementioned method, the projectile’s velocity module will be set with the randomized values following formula presented below:
    - Vx = velocityModule * cos(angle);
    - Vy = velocityModule * sin(angle);
- Following that, the AimBotEulerIntegrator(...) method will be called while receiving the projectile and the target as arguments. Inside the aforementioned method, a loop will run for a specified amount of frames (in an environment with a stabilized framerate, the assigned amount of frames would be fps * simulation time [s]) within which:
    - The projectile’s state will be propagated while calculating the gravitational force and the drag force.
    - In each of the loop’s iterations the CheckHit(...) method will be called to check whether or not the target has been hit.
    - In case there was no detected hit, all the loops would keep running until a target was found or the Monte-Carlo(...) method reached the maximum iteration possible (Ex: Iterations = 1000 → Max iteration índex 999)
    - In case it was hit in any frame, the targetWasHit flag would be set to true and all the loops mentioned above would be broken, including Monte-Carlo’s. Before that, however, the values for the variables for which the projectile was initialized will be shown on the command console.

Core Validation

To validate the core, the following test was ran:

- Hypothesis: For a given target in any position within a set Simulation World (i.e 20m wide, 10m high, gravity, wind speed...), the core will be able to find one set of velocity module and throwing angle values with which the thrown projectile will hit the target.

- The projectile's initial position, velocity and acceleration vectors will be set to Origin ({0.0f, 0.0f, 0.0f}).

- The Target's position will be randomize for the purpose of this test and it will have a radius of 0.5m.

- As the AimBot has been developed for 2D, the only vector components that will be taken into account will be the X and Y Axis.

- The randomized variables will be limited to certain maximum and minimum values:
  - Velocity Module Limits: 0 m/s ~ 50 m/s.
  - Throwing Angle Limits: -180º ~ 180º.
  - Wind Velocity Vector Limits: -10 m/s ~ 10 m/s.
  - Target Position Limits: { 0m ~ 20m, 0m ~ 10m, 0m }

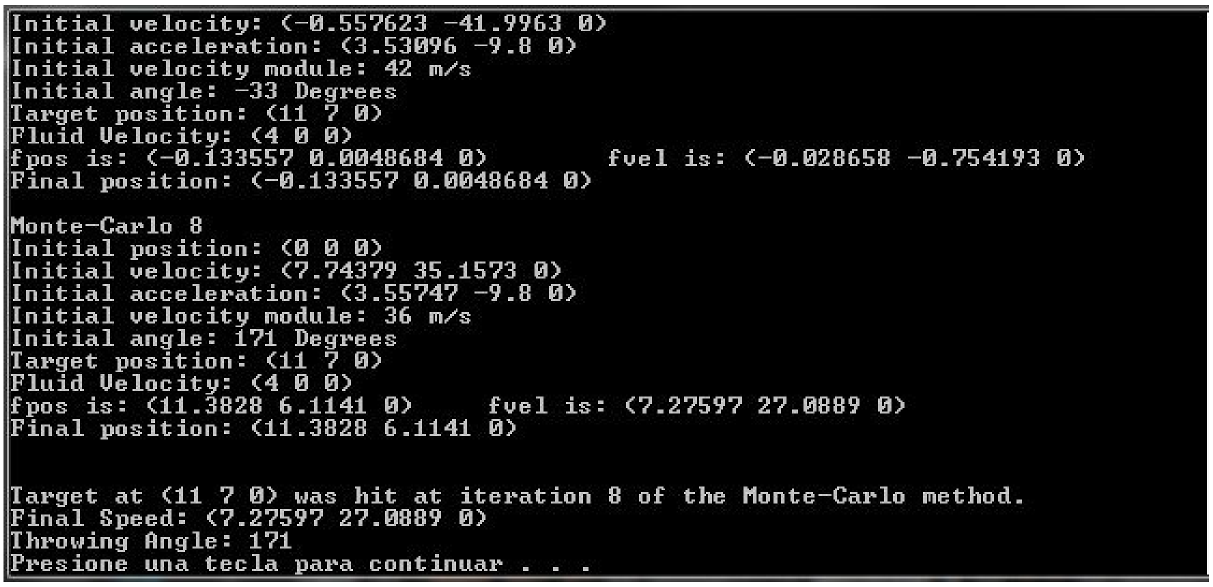
  It should be noted that, for this test, the wind's velocity vector and the target's position will also be randomized when the simulation world is initialized.

- The test will be run for 5 different target positions and wind velocity module values.

- To ensure that a there is at least one hit per target position within an acceptable range of iterations, the Monte-Carlo(...) Method (AimBot core) will be run for a 1000 iterations.

<u>Test Results</u>:

Case 1:

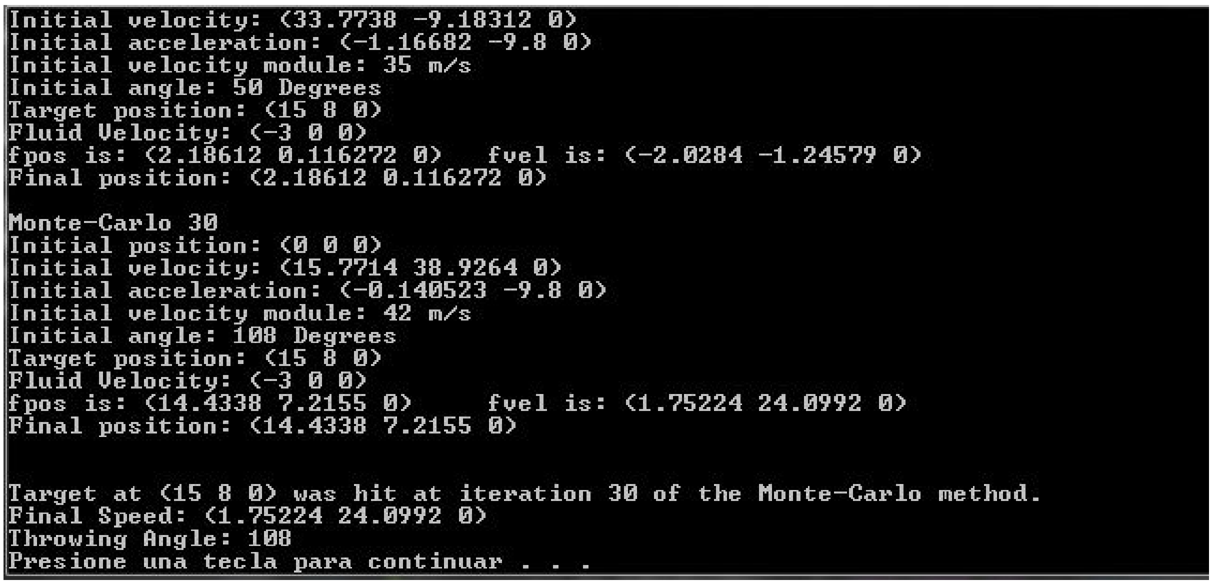


```
Initial velocity: (-0.557623 -41.9963 0)
Initial acceleration: (3.53096 -9.8 0)
Initial velocity module: 42 m/s
Initial angle: -33 Degrees
Target position: (11 7 0)
Fluid Velocity: (4 0 0)
fpos is: (-0.133557 0.0048684 0)        fvel is: (-0.028658 -0.754193 0)
Final position: (-0.133557 0.0048684 0)

Monte-Carlo 8
Initial position: (0 0 0)
Initial velocity: (7.74379 35.1573 0)
Initial acceleration: (3.55747 -9.8 0)
Initial velocity module: 36 m/s
Initial angle: 171 Degrees
Target position: (11 7 0)
Fluid Velocity: (4 0 0)
fpos is: (11.3828 6.1141 0)     fvel is: (7.27597 27.0889 0)
Final position: (11.3828 6.1141 0)


Target at (11 7 0) was hit at iteration 8 of the Monte-Carlo method.
Final Speed: (7.27597 27.0889 0)
Throwing Angle: 171
Presione una tecla para continuar . . .
```

- <u>Initial Conditions</u>:
  - Target Position: {11, 7, 0}
  - Wind Velocity Vector: {4, 0, 0}

- <u>On Target Hit</u>:
  - Monte-Carlo Iteration Index: 8
  - Initial Projectile Velocity Module: 36 m/s
  - Throwing Angle: 171º

Case 2:

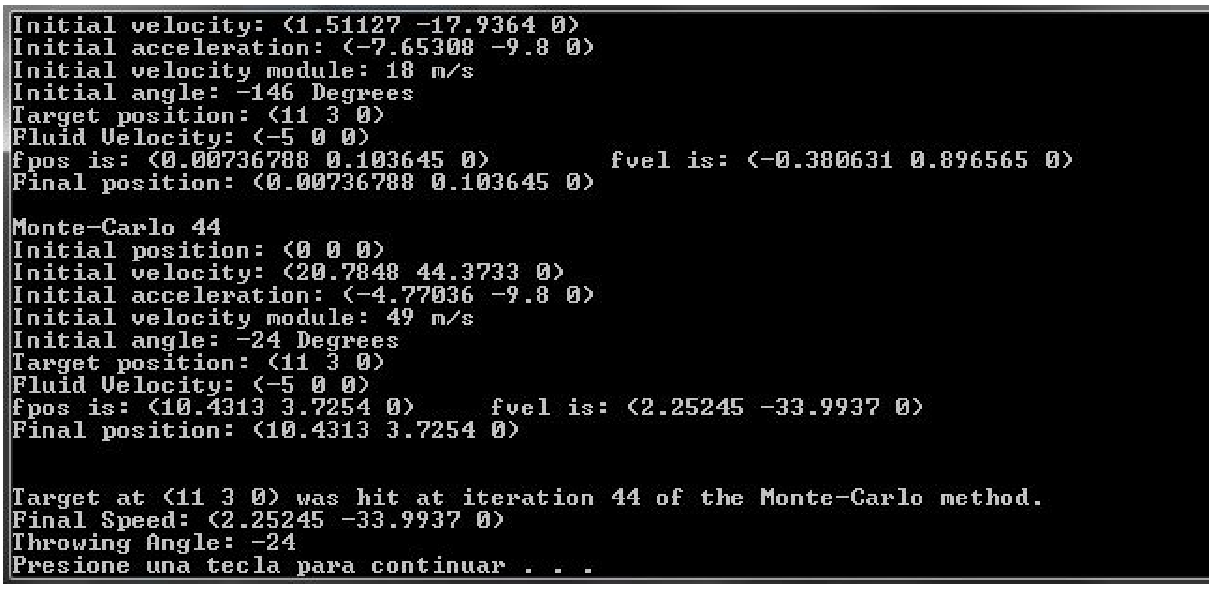


```
Initial velocity: (33.7738 -9.18312 0)
Initial acceleration: (-1.16682 -9.8 0)
Initial velocity module: 35 m/s
Initial angle: 50 Degrees
Target position: (15 8 0)
Fluid Velocity: (-3 0 0)
fpos is: (2.18612 0.116272 0)   fvel is: (-2.0284 -1.24579 0)
Final position: (2.18612 0.116272 0)

Monte-Carlo 30
Initial position: (0 0 0)
Initial velocity: (15.7714 38.9264 0)
Initial acceleration: (-0.140523 -9.8 0)
Initial velocity module: 42 m/s
Initial angle: 108 Degrees
Target position: (15 8 0)
Fluid Velocity: (-3 0 0)
fpos is: (14.4338 7.2155 0)     fvel is: (1.75224 24.0992 0)
Final position: (14.4338 7.2155 0)


Target at (15 8 0) was hit at iteration 30 of the Monte-Carlo method.
Final Speed: (1.75224 24.0992 0)
Throwing Angle: 108
Presione una tecla para continuar . . .
```

- Initial Conditions:
  - Target Position: {15, 8, 0}
  - Wind Velocity Vector: {-3, 0, 0}

- On Target Hit:
  - Monte-Carlo Iteration Index: 30
  - Initial Projectile Velocity Module: 42 m/s
  - Throwing Angle: 108º

Case 3:

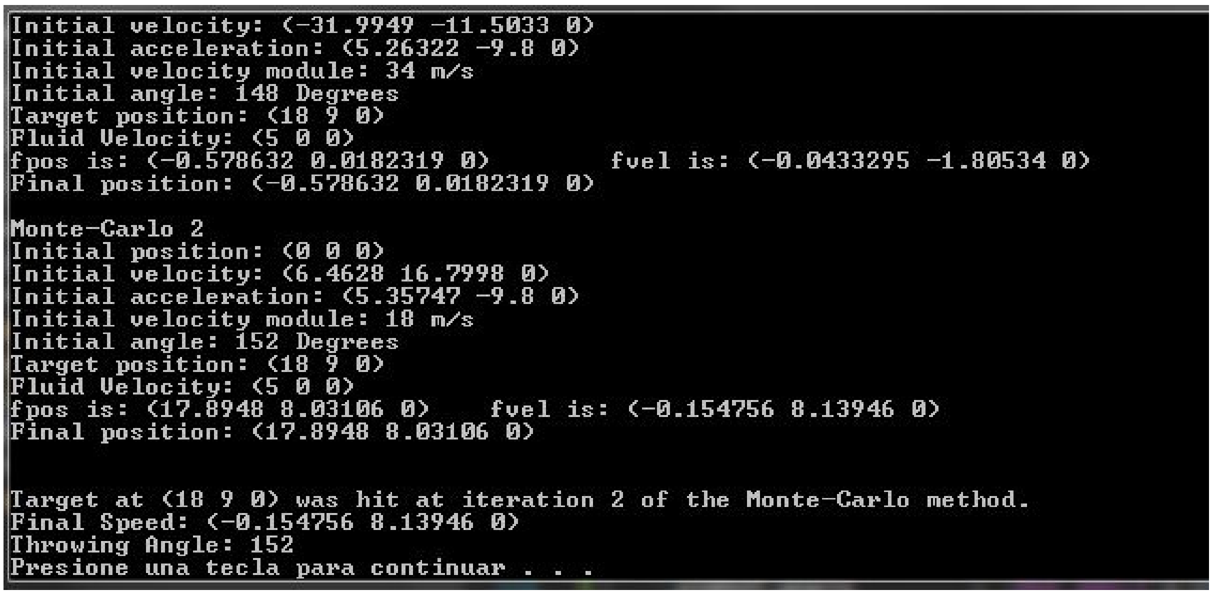


```
Initial velocity: (1.51127 -17.9364 0)
Initial acceleration: (-7.65308 -9.8 0)
Initial velocity module: 18 m/s
Initial angle: -146 Degrees
Target position: (11 3 0)
Fluid Velocity: (-5 0 0)
fpos is: (0.00736788 0.103645 0)        fvel is: (-0.380631 0.896565 0)
Final position: (0.00736788 0.103645 0)

Monte-Carlo 44
Initial position: (0 0 0)
Initial velocity: (20.7848 44.3733 0)
Initial acceleration: (-4.77036 -9.8 0)
Initial velocity module: 49 m/s
Initial angle: -24 Degrees
Target position: (11 3 0)
Fluid Velocity: (-5 0 0)
fpos is: (10.4313 3.7254 0)     fvel is: (2.25245 -33.9937 0)
Final position: (10.4313 3.7254 0)


Target at (11 3 0) was hit at iteration 44 of the Monte-Carlo method.
Final Speed: (2.25245 -33.9937 0)
Throwing Angle: -24
Presione una tecla para continuar . . .
```

- Initial Conditions:
  - Target Position: {11, 3, 0}
  - Wind Velocity Vector: {-5, 0, 0}

- On Target Hit:
  - Monte-Carlo Iteration Index: 44
  - Initial Projectile Velocity Module: 49 m/s
  - Throwing Angle: -24º

Case 4:

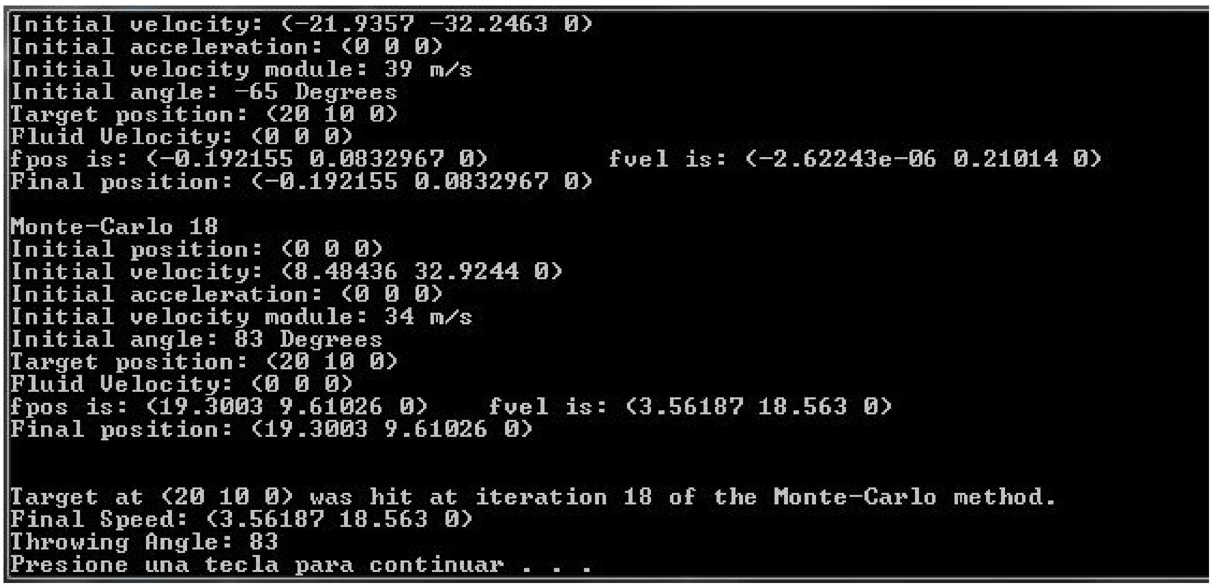


```
Initial velocity: (-31.9949 -11.5033 0)
Initial acceleration: (5.26322 -9.8 0)
Initial velocity module: 34 m/s
Initial angle: 148 Degrees
Target position: (18 9 0)
Fluid Velocity: (5 0 0)
fpos is: (-0.578632 0.0182319 0)        fvel is: (-0.0433295 -1.80534 0)
Final position: (-0.578632 0.0182319 0)

Monte-Carlo 2
Initial position: (0 0 0)
Initial velocity: (6.4628 16.7998 0)
Initial acceleration: (5.35747 -9.8 0)
Initial velocity module: 18 m/s
Initial angle: 152 Degrees
Target position: (18 9 0)
Fluid Velocity: (5 0 0)
fpos is: (17.8948 8.03106 0)    fvel is: (-0.154756 8.13946 0)
Final position: (17.8948 8.03106 0)


Target at (18 9 0) was hit at iteration 2 of the Monte-Carlo method.
Final Speed: (-0.154756 8.13946 0)
Throwing Angle: 152
Presione una tecla para continuar . . .
```

- Initial Conditions:
  - Target Position: {18, 9, 0}
  - Wind Velocity Vector: {5, 0, 0}

- On Target Hit:
  - Monte-Carlo Iteration Index: 2
  - Initial Projectile Velocity Module: 18 m/s
  - Throwing Angle: 152º

Case 5:



```
Initial velocity: (-21.9357 -32.2463 0)
Initial acceleration: (0 0 0)
Initial velocity module: 39 m/s
Initial angle: -65 Degrees
Target position: (20 10 0)
Fluid Velocity: (0 0 0)
fpos is: (-0.192155 0.0832967 0)        fvel is: (-2.62243e-06 0.21014 0)
Final position: (-0.192155 0.0832967 0)

Monte-Carlo 18
Initial position: (0 0 0)
Initial velocity: (8.48436 32.9244 0)
Initial acceleration: (0 0 0)
Initial velocity module: 34 m/s
Initial angle: 83 Degrees
Target position: (20 10 0)
Fluid Velocity: (0 0 0)
fpos is: (19.3003 9.61026 0)    fvel is: (3.56187 18.563 0)
Final position: (19.3003 9.61026 0)


Target at (20 10 0) was hit at iteration 18 of the Monte-Carlo method.
Final Speed: (3.56187 18.563 0)
Throwing Angle: 83
Presione una tecla para continuar . . .
```

- Initial Conditions:
    - Target Position: {20, 10, 0}
    - Wind Velocity Vector: {0, 0, 0}

- On Target Hit:
    - Monte-Carlo Iteration Index: 18
    - Initial Projectile Velocity Module: 34 m/s
    - Throwing Angle: 83º

Test Results:

As all the different targets were hit regardless of their position or wind speed (velocity vector), the test's hypothesis has been confirmed. As consequence of the aforementioned, the AimBot Core has been validated.

Conclusions

- A fully functional AimBot with an Euler Integrator as the base has been developed.

- Going along the lines of the above statement, the base AimBot core has been scoured for bugs and safety measures have been set in place to prevent any undesired outcomes.

- Although the development team originally set developing a graphical interface representing a projectile and a target as one of the main objectives, said objective has not been met before the project’s deadline.