MINISTRY OF EDUCATION OF REPUBLIC OF MOLDOVA

Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Department of Software Engineering and Automatics

Report

on Artificial Intelligence Fundamentals

Laboratory Work nr. 2

Performed by: **Vasile Boaghi, Ion Dodon**, FAF-172

Verified by: **Mihail Gavrilița**, asist. univ.

Chișinău, 2021

Contents

Flocking Behaviour

In this paper the object that moves using a flocking behaviour is called as a boid. Boids is an artificial life program, developed by Craig Reynolds in 1986, which simulates the flocking behaviour of birds. His paper on this topic was published in 1987 in the proceedings of the ACM SIGGRAPH conference. The name "boid" corresponds to a shortened version of "bird-oid object", which refers to a bird-like object. Incidentally, "boid" is also a New York Metropolitan dialect pronunciation for "bird."

As with most artificial life simulations, Boids is an example of emergent behavior; that is, the complexity of Boids arises from the interaction of individual agents (the boids, in this case) adhering to a set of simple rules. The rules applied in the simplest Boids world are as follows:

- **separation**: steer to avoid crowding local flockmates
- **alignment**: steer towards the average heading of local flockmates
- **cohesion**: steer to move towards the average position (center of mass) of local flockmates

More complex rules can be added, such as obstacle avoidance and goal seeking.

The basic model has been extended in several different ways since Reynolds proposed it. For instance, Delgado-Mata et al. extended the basic model to incorporate the effects of fear. Olfaction was used to transmit emotion between animals, through pheromones modelled as particles in a free expansion gas. Hartman and Benes introduced a complementary force to the alignment that they call the change of leadership. This steer defines the chance of the boid to become a leader and try to escape.

The movement of Boids can be characterized as either chaotic (splitting groups and wild behaviour) or orderly. Unexpected behaviours, such as splitting flocks and reuniting after avoiding obstacles, can be considered emergent.

The boids framework is often used in computer graphics, providing realistic-looking representations of flocks of birds and other creatures, such as schools of fish or herds of animals. It was for instance used in the 1998 video game Half-Life for the flying bird-like creatures seen at the end of the game on Xen, named "boid" in the game files.

The Boids model can be used for direct control and stabilization of teams of simple Unmanned Ground Vehicles (UGV) or Micro Aerial Vehicles (MAV) in swarm robotics. For stabilization of heterogeneous UAV-UGV teams, the model was adapted for using onboard relative localization by Saska et al.

At the time of proposal, Reynolds' approach represented a giant step forward compared to the traditional techniques used in computer animation for motion pictures. The first animation created with the model was Stanley and Stella in: Breaking the Ice (1987), followed by a feature film debut in Tim Burton's film Batman Returns (1992) with computer generated bat swarms and armies of penguins marching through the streets of Gotham City.

The boids model has been used for other interesting applications. It has been applied to automatically program Internet multi-channel radio stations. It has also been used for visualizing information[10] and for optimization tasks.

The infomration from above was taken from `https://en.wikipedia.org/wiki/Boids` [2]

The Task

You are given a tool developed by a guy called Ijon, which should provide a good basis for the simulation, so you can expand on that. You learn that Solanum tuberosum (the name given to these creatures) exhibit flocking behaviour when undisturbed. S. tuberosum also can exhibit evading behavior, thus trying to evade any unknown objects, like starships and their missiles, while still flocking (much like a school of fish attacked by a predator). When attacking, they thrust their asteroid towards a starship, aiming to collide and destroy them, while still evading collision with other asteroids (similar to a bunch of people trying to fit in a bus). What triggers them to switch between behaviors remains unknown, so it's up to you how your simulation will handle that.

It is needed to write a program that simulates a flocking behaviour in the small game that is given. The mandatory task is to simulate their calm behavior which is their most encountered behavior. When finished with that, continue with their evading and attacking behavior, thus yielding a more complete simulation of S. tuberosum's behavior patterns. Play around with different parameters (e.g. asteroid number and speed, or boid coherence, separation and alignment) to create a visually pleasant (if not truthful) simulation.

Solution Description

First of all we changed the behaviour of the asteroid from their original behaviour to a flocking one. To implement a flocking behaviour we calculated three vectors based on the neighbour in a perception radius. These three vectors we named flocking vectors and they are being calculated by the *flock_vectors* in the behaviour implementation code. In order to have flocking behaviour we need six main components:alignment vector, cohesion vector, separation vector, perception radius, a max speed value and max force value that are being used to normalize the flocking vectors.

The max speed and max force values are used to take under control the behaviour of the rocks in order to have a clear pleasant behaviour. The perception radius represents under what radius other rocks/boids are considered neighbours.

The alignment vector is calculated as the arithmetic sum of the velocities of all the neighbours and in other words we can say that this vector will mostly influence the movement direction of a flock.

The The cohesion vector has the role to keep the flock members together. cohesion vector is calculated as the arithmetic sum of the position of neighbours. Then the resulting vector is subtracted with the position of the current boid and then the velocity of the boid is also subtracted.

The separation vector is calculated as the arithmetic sum of the vectors that turn in the opposite direction of the center of mass of the neighbours. Fro the resulting vector is subtracted the vector of velocity of the current boid. The separation vector is kinda opposite to the collection vector and is used to keep the flock members at a decent distance between them.

Finally, having the three vectors, the sum them and this sum will be the acceleration of the boid. This acceleration is used to steer the boid in the next position. Each boid has a position and velocity. In order to move the void With add to the position vector the velocity vector. In order to change the direction we add to the velocity vector the acceleration vector. And as explained above, the acceleration vector is calculated with each frame as the sum of the flocking vectors: alignment vector, cohesion vector and separation vector.

All that is explained above is only to have a flocking behaviour. To have a more interesting/-complex behaviour considered that the default behaviour is the CALM behaviour for the boids and we added an ATTACKing behaviour that means that while still flocking the boids/rocks will go towards the missiles and the ship. And also we added the DEFENSE behaviour that means that while still flocking the boids/rocks will try to avoid collision with the ship and its missiles.

That triggers the switch between these behaviours is the number of missiles launched by the ship. If there are no missile in space then the boids will be calm, if there are between 0 and 3 missiles then the boids will attach and if there are more than 3 missile then the boids will defend.

Code and Mentions

The following function is used to calculate the acceleration based on the flocking vectors and also depending of the current behaviour of the boids that in turn depends on the number of missiles.

Listing 1: Base function to update boid's position

```
def update_rock_position(current, boids, ship, missiles):
    acceleration = [0, 0]
    alignment, cohesion, separation = flock_vectors(current, boids)

    total = np.add(acceleration, alignment)
    total = np.add(total, cohesion)
    total = np.add(total, separation)
    total = native_array(total)
    if total != [0, 0]:
        acceleration = native_array(total)

    position = np.array(current.pos)
    velocity = np.array(current.vel)
    acceleration = np.array(acceleration)

    position = np.add(position, velocity)
    velocity = np.add(velocity, acceleration)

    # by default CALM behaviour is used
    no_missiles = len(missiles)
    # if there are less than 3 missiles the attack behaviour is used
    if 0 < no_missiles <= 3:
        position, velocity = attack(position, velocity, acceleration, current,
    ship, missiles)
    # if there are mote than 3 missiles then defense behaviour is used
    elif no_missiles > 3:
        position, velocity = defense(position, velocity, acceleration, current,
    ship, missiles)

    current.pos = native_array(position)
    current.vel = native_array(velocity)

    edge(current)
```

The next function is the one that calculates the flocking vectors: alignment vector, cohesion vector and the separation vector.

Listing 2: The function that calculates the flocking vectors

```
    def flock_vectors(current, boids):
    alignment = np.array([0, 0])
    cohesion = np.array([0, 0])
    separation = np.array([0, 0])
    total = 0
    for other in boids:
        diff = np.array(other.pos) - np.array(current.pos)
        dist = np.linalg.norm(diff)
        if other is not current and dist < PERCEPTION_RADIUS:
            alignment = np.add(alignment, np.array(other.vel))

            cohesion = np.add(cohesion, np.array(other.pos))

            diff_vec = np.subtract(np.array(current.pos), np.array(other.pos))
            diff_vec = np.divide(diff_vec, dist) if dist != 0 else diff_vec
            separation = np.add(separation, np.array(diff_vec))

            total += 1
    if total > 0:
        alignment = np.divide(alignment, total)
        alignment = (alignment / np.linalg.norm(alignment)) * MAX_SPEED
        alignment = np.subtract(alignment, np.array(current.vel))

        cohesion = np.divide(cohesion, total)
        cohesion = np.subtract(cohesion, np.array(current.pos))
        cohesion = max_speed(cohesion)
        cohesion = np.subtract(cohesion, np.array(current.vel))
        cohesion = max_force(cohesion)

        separation = np.divide(separation, total)
        separation = max_speed(separation)
        separation = np.subtract(separation, np.array(current.vel))
        separation = max_force(separation)

    return alignment, cohesion, separation
```

And, finally, the following two functions are used to calculate the vector for the direction towards which to attack or the vector in which direction to defense. To calculate either one of these vectors the same principle of cohesion was used. In the case when there are no missiles then the rocks will go towards the ship. For example to determine in which direction to attach is used the first missile and the rocks will go towards it. In the case of defense the rocks will try to avoid the first missile.

Listing 3: Functions for attacking and defense behaviour

```

def attack(rock_position, rock_velocity, rock_acceleration, current, ship,
    missiles):
    missiles_list = list(missiles)
    attack_direction = np.subtract(ship.pos, current.pos)
    if len(missiles_list) > 0:
        attack_direction = np.subtract(missiles_list[0].pos, current.pos)

    attack_direction = max_speed(attack_direction)
    attack_direction = np.subtract(attack_direction, np.array(current.vel))
    attack_direction = max_force(attack_direction)

    attack_direction = native_array(attack_direction)
    if attack_direction != [0, 0]:
        rock_acceleration = native_array(attack_direction)

    rock_position = np.add(rock_position, rock_velocity)
    rock_velocity = np.add(rock_velocity, rock_acceleration)

    return rock_position, rock_velocity


def defense(rock_position, rock_velocity, rock_acceleration, current, ship,
    missiles):
    missiles_list = list(missiles)
    defense_direction = np.subtract(current.pos, ship.pos)
    if len(missiles_list) > 0:
        defense_direction = np.subtract(current.pos, missiles_list[0].pos)

    defense_direction = max_speed(defense_direction)
    defense_direction = np.subtract(defense_direction, np.array(current.vel))
    defense_direction = max_force(defense_direction)

    defense_direction = native_array(defense_direction)
    if defense_direction != [0, 0]:
        rock_acceleration = native_array(defense_direction)

    rock_position = np.add(rock_position, rock_velocity)
    rock_velocity = np.add(rock_velocity, rock_acceleration)

    return rock_position, rock_velocity
```

Conclusions

Working on this laboratory work we have understood how to implement a flocking behaviour and even how to create a more complex behaviour depending on some conditions. The flocking behaviour of the boids could make someone who has no idea about programming that the computer is intelligent because it knows how to control the boids in a manner that looks natural and even know how to protect the bids from obstacles or how to make the boids attack something. But, again, there is not intelligence, but only rules to be followed. And even in real live, many thing can be explained in rules. And we can use those rules to make simulations. We have also gained some knowledge how to work with vectors and what is the role for each vector in the movement of an object.

References

[1] Vasile Boaghi, Ion Dodon. Source code for the laboratory work. Accessed February 18, 2021. https://github.com/iondodon/FIA/tree/main/Lab2.

[2] Boids https://en.wikipedia.org/wiki/Boids

[3] Boids - Background and Update - by Craig Reynolds https://www.red3d.com/cwr/boids/