Virtual Savannah:

AI for the simulation of an ecosystem

Master Thesis
Daniel Collado
Spring 2013
Aalborg University

AALBORG UNIVERSITET

Department of Architecture,
Design & Media Technology
Madialogy, 10th Semester

Title:

Virtual Savannah: AI for the simulation of an ecosystem

Semester Theme:

Master Thesis

Project Period:

MED10

Abstract:

The present report documents the implementation of an AI framework for wildlife simulation purposes. The Virtual Savannah project is used as a base to analyze the improvements that dynamic agent-based behaviour can add to a simulation compared to scripted behaviour, where events are predefined.

With the help of flocking AI and combined steering behaviours, a set of different animals have been given autonomous AI, and in some cases strategic thinking and decision making.

The results have been tested by running a series of simulations on different setups, together with a stress test. A quantitative and qualitative analysis has been performed, concluding that the emergent behaviour and dynamism has been succesfully achieved, although further improvements for robustness and added complexity are needed to complement some of the behaviours.

Members:

Daniel Alejandro Collado Locubiche

Supervisor:

Matthias Rehm

Circulation: 2

Number of pages: 59

Number of appendices and form:
2 (DVD and attached documents)

Delivered: 30th of May 2013

PREFACE

This report is the documentation of the work carried out as Master Thesis of the Master's programme at Medialogy, Aalborg University. The name of the project is "Virtual Savannah: AI for the simulation of an ecosystem."

An appendix DVD is enclosed. On this DVD the following material is present:

- Source code for the simulation (Unity project).
- An audiovisual production presenting the results of the work done.
- A digital version of this report.

Contents

Chapter 1

Introduction

Aalborg Zoo (Zoo 2013) has been working together with students from Aalborg University to create virtual environments where animal behaviour can be studied. The project "Virtual Savannah" (Eskildsen et al. 2013) is an example of how a team from Aalborg University created this environment, implementing different features that represent the life in the savannah, like the seasons, interaction between animals, animal life cycle, etc.

The goal of the Virtual Savannah was to serve as an application to study the behaviour of animals in their natural habitat that can be later seen in the zoo live. This application would give a lot of useful information, making use of parameters that could be tweaked, different events during different seasons and information about each animal amongst others.

However, the behaviour of the animals and the events in the simulation were scripted, which means that once the simulation started, the same events would happen unless the parameters were tweaked manually. Same conditions would give rise to the same behaviours and the same events, making the simulation useful to some extent, but static when it comes to show the dynamics of the savannah.

Scripting behaviours makes them predictable and unable to cover the vast space of possibilities that can happen in an environment where different species coexist. This is due to the inability of the animals to react to each situation, and thus, the scripting is needed in order to tell them what to do and when to do it.

This is why it is needed for the animals to think by themselves, becoming actors able to process the information that is the environment around them. When they gain autonomy, no pre-scripted actions need to be commanded, and they simply react to the situation.

The intelligence used by the actors gives the possibility of being able to react to any scenario. The fact that every animal acts on its own free will will make the simulation a dynamic experience, never giving rise to the same experience twice.

Initial Problem Statement

Based on these thoughts, the following problem statement has been phrased:

How is it possible to create a dynamic environment where animals act on their own will.

To research on this topic, an AI framework will be implemented. The game engine Unity (Technologies 2013) has been chosen as the development tool for the work presented in this report.

CHAPTER 2

PROBLEM ANALYSIS

To form the basis for the implementation of an AI framework, the following subjects will be treated in this chapter:

- Game AI.
- Decentralized AI.
- Flocking.
- Steering behaviours.

2.1 Game AI

When trying to create an AI for a real time application, it is important to follow a model that takes into account all the limitations of running in a frame basis. From the possible input, or environment scanning, to the execution where all the data is analyzed and computed (strategy, decision making, movement) and finally the outcome displayed on screen. This is well represented by the AI model presented in (Illington and Funge 2009, pp.32), as we can see in figure 2.1.

Depending on its purpose, an AI can be more or less complex. However, small changes in behaviour can give the illusion of greater intelligence, while adding advanced algorithms of big complexity doesn't always result in a better AI. This is known as the "Complexity Fallacy" (Illington and Funge 2009, pp.19). This is why finding the simple but significant mechanics in agent behaviour is key to balance resource consumption and performance, and achieve greater efficiency.

Strategy can be used to improve the coordination of a given group of agents. Although it slightly highjacks the idea of having non-supervised autonomous characters, a certain degree of information can be shared amongst peers in order to organize and achieve more complex strategies with little effort. "Half Life" (Corporation

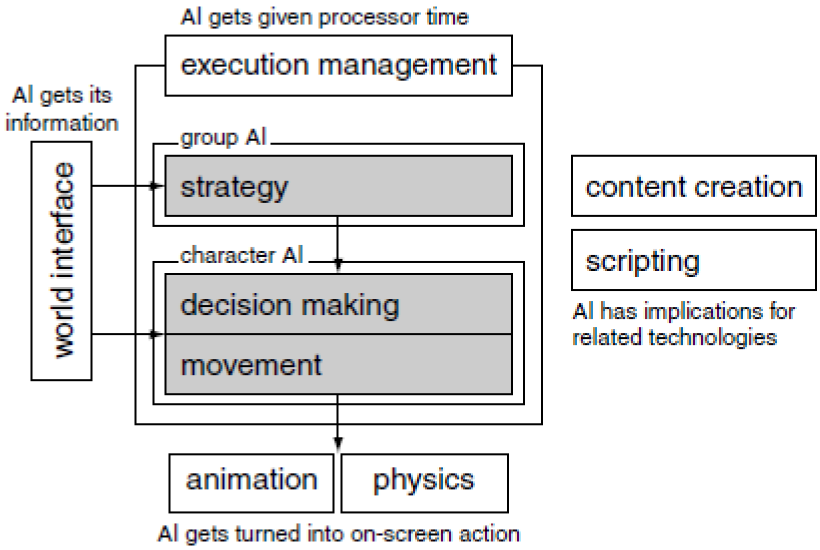

Figure 2.1: The game AI model.

1998) exploited this aspect with great results. In the game, the enemies would assume different roles when attacking, increasing the intelligence of the characters and introducing an added challenge for the player, who had to study the enemy before acting.

In order to build a proper AI, apart from the algorithms a proper infrastructure is needed. An important aspect of it is how the AI retrieves information from the game efficiently, in order to make decisions. This is known as "perception": defining how much each character knows about the world that surrounds them. Creating this interface with the world is a significant part of the effort when building an AI. In this work, the use of Unity's in built utilities like collision detection and physics layers (unity3d.com 2013b,a) will make this task easier.

Movement is a really important factor when trying to simulate intelligent behaviour, and a bigger one when it comes to collective movement. Different algorithms can turn decisions into motion, and the level of complexity of the motion can be improved by adding features like obstacle avoidance or fleeing from enemies. This aspect of the simulation is crucial for the work presented in this report, and will be covered extensively in section 4.3 of this report: Scripting Behaviours.

If the aim of the AI is to produce autonomous characters, it is recommended to have a bottom-up design, where the behaviour of each character is designed, and the AI required to implement it is created afterwards. The general outcome of the simulation will be determined by the combination of the interactions between the different agents' behaviours. Designing how the agent will react in each situation will determine the decision making process, chosing the most suitable course of action depending on the context. The next chapter covers the advantages and disadvantages of having such decentralized AI system.

2.2 Decentralized AI

In a big and complex environment such as a virtual savannah, controlling all the data, events and decisions of all the elements in the simulation would require a huge amount of time and resources. As discussed in section 1, simply scripting the events would undermine the dynamics and randomness of the simulation, reducing the realism, and therefore a distributed AI model is needed in order to break down the work into simpler parts.

A distributed AI consisting on a multi-agent system provides a way of coping with the bulk of decision making, however this requires the creation of an interface agent-world, previously defined as "perception". This infrastructure will provide a medium through which each character will analyze the environment, and make simple calculations to decide the next movement.

The most amazing aspect of multi-agent system is what is denominated as "Emergent Behaviour". Emergence is the way complex systems and patterns arise out of a multiplicity of relatively simple interactions. A very simple yet clear definition can be found in (Buckland 2005, pp.118):

Emergent behavior is behavior that looks complex and/or purposeful to the observer but is actually derived spontaneously from fairly simple rules. The lower-level entities following the rules have no idea of the bigger picture; they are only aware of themselves and maybe a few of their neighbors.

The fact that the workload has been broken down into simple small units, increases the efficiency of the computation. However the agent AI needs to be as simplified as possible, since now the computational cost will increase proportonially, even exponentially depending on the AI and how well coded it is.

There are disavantages to this approach, some of them being:

- Smaller control over the actors decisions and possible situations they may get into. Giving them the freedom to chose may lead to undesired situations.
- Unability to deal with all the possible situations effectively, giving rise to bugs or glitches. The AI needs to be robust to achieve uneventful simulations.
- The need for tuning and tweaking the AI in order to achieve an overall balanced behaviour interaction. Every time a new feature is added, the system must be tested to make sure it is balanced with the rest of behaviours.

- Unablity to monitor an overall state of the environment. The fact that no centralized system is keeping track of the whole simulation makes it difficult to debug on most cases.

For the purposes of this work, a distributed model is chosen to simulate the savannah environment designed within the scope of the project, and to serve as proof of concept for distributed AI frameworks. Nevertheless, a good programmer will combine different techniques at hand in order to create an overall robust and efficient AI, whether this techinques are centralized or distributed.

A common representation of a multi-agent system is often seen in animal simulations like flocks, herds, swarms or schools. All these are gathered under the category of “Boids” (Reynolds 1987). The “Boid” model for a artificial life simulation is explained in the following chapter.

2.3 Flocking

In 1986 Craig Reynolds developed an artificial life program called Boids. The pogram would simulate a flock of birds navigating together in a realistic manner. He published a paper on the topic in 1987 (Reynolds 1987), on the proceedings of the ACM SIGGRAPH (SIGGRAPH SIGGRAPH) conference. As many other life simulators, Boids would make use of the emergent behaviour by creating complex behaviour from simple autonomous agents. However this paper would become a reference to any life simulation by using simple rules like separation, cohesion and alignment that have a greater emergence outcome. More complex rules like obstacle avoidance and goal seeking could be added on top. Ever since, this AI model has been known as “flocking” in the academic AI world.

The importance of Reynold’s paper was due to the great improvent his approach supposed compared to previous traditional life simulation methods. Also, the adaptability of this approach to simulate different behaviours by adjusting the steering and other parameters, makes it easy to emulate different group behaviours. The common applications are on the flocking of birds, fish schools, herding animals and insect swarms. Ever since the publication of his work, many games and films have also used it to simulate animal behaviour. A good example for an animation would be Disney’s “Lion King” (Studios 1994) (figure 2.2) where a stampede is simulated using this technique. Games like Pikmin (EAD 2001) (figure 2.3) also have been using it to handle the movement of crowds.

Further improvements have been developed since, like the incorporation of fear effects (Delgado-Mata et al. 2007) or leadership behaviour (Hartman and Benes 2006). Also, multiple applications for this AI system have been found in different domains, like visualization (Moere 2004) and optimization(Cui and Shi 2009).

Figure 2.2: Stampede in Disney's Lion King.

Figure 2.3: Ingame screenshot of the game Pikmin.

The factor that this approach uses to create emergence is the movement that results of using a specified set of rules. This different rules create forces with different goals that, when blended together, create a final force which is considered the "decision" the agent has taken. This decision is normally complemented by an additional layer of general decision making and state control, as shown in chapter 4.3 where the Combined Steering Behaviours are explained in depth.

2.4 Steering Behaviours

Movement is key to simulate herding behaviour. The interactions between the animals need to be realistic, and the best way to produce this illusion of inteligence is trying to match the emergence seen in their natural habitat when they navigate together. This is what the flocking approach is trying to do, hence adequate movement algorithms that adapt to this AI model need to be discussed.

Movement algorithms take data about their own state and the state of the world, and come up with an output representing the desired movement to do next. This is depicted in figure 2.4.

The input taken varies depending on the algorithm. Some algorithms only require the character's position and orientation, while others need to process the world's geometry to perform different checks. The output can also come in different forms, from a simple direction to move towards, to a set of acceleration parameters. The first kind doesn't account for acceleration or slowing down, it is defined as "Kinematic"; the latter output type does, and these kind of algorithms are known as "Steering

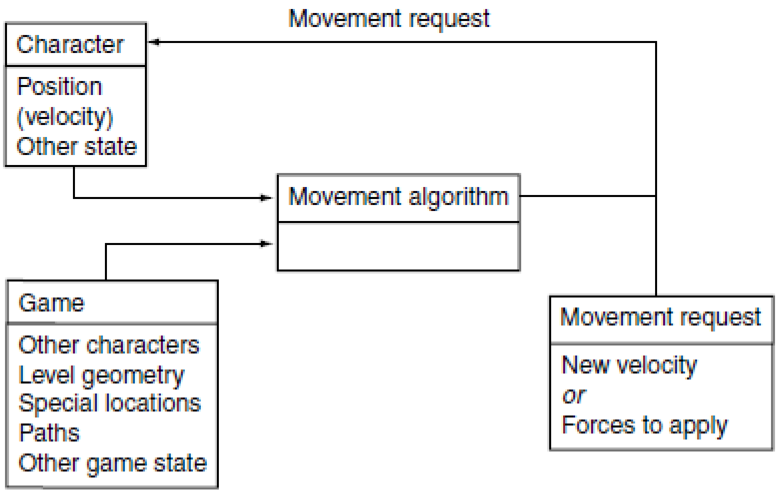

Figure 2.4: The movement alorithm structure from (Illington and Funge 2009, pp.41).

Behaviours".

On a side note, the direction the character is facing at a given moment will be defined as `orientation`, while the angular speed at which it is rotating will be referred to as `rotation`. This clarification is made since it is easy to mistake one for another, and the Unity engine defines as rotation what we define here as `orientation`. Also, `velocity` defines the motion vector, while `speed` defines `velocity`'s magnitude.

Kinematic behaviours are algorithms that create a movement output using the character's position and orientation. A target velocity is calculated, allowing the character's velocity to change instantly if the situation so requires. Many games need this kind of behaviours for many reasons: reaction time, robotic behaviour, game mechanics, and many other reasons. However this kind of movement can look very unrealistic, since it doesn't follow Newton's laws of motion, where velocities can't change instantly. In order to reduce the impact of this seemingly odd behaviour (compared to the real world), the current velocity can be changed over time, smoothing the motion when required.

"Steering behaviours" is the name given by Craig Reynolds to the movement algorithms in his paper about life simulation (Reynolds 1987). This kind of algorithms account for the current motion of the actor. Instead of just defining a target velocity, the algorithm also uses the current velocity and rotation (apart from the position and orientation) in order to calculate the steering output, creating more realistic movement patterns that obey the laws of physics more faithfully. The steering output consists on an angular acceleration and a linear acceleration, which are later used to update the character's velocity, rotation, position and orienteation.

Combined steering behaviours (like flocking or herding) consist on the blending of different steering methods. The integration of the different steering methods can be done by giving different weights to each one fo them, or by prioritizing one before another in some cases. Selecting a set of steering methods and assigning weights to each of them will define the resulting behaviour. Each of the steering methods used in this work are thoroughly explained in chapter 4.3, and the source code (exhaustively commented for better understanding) is listed in the annex.

2.5 Problem Analysis Summary

This section will provide a summary of the problem analysis chapter in this report. Throughout the problem analysis the following problem statement has been examined:

How is it possible to create a dynamic environment where animals act on their own will.

AI methods used in computer games to control characters have been examined to conclude that a decentralized model is needed in this case. A succesful example of decentralized animal behaviour can be found in the flocking AI, which combines steering methods to create the emergent behaviour of animal groups.

Once concluded that the flocking AI is the better suited to achieve the initial problem statement, it is needed to define the environment to which it will be applied. In section 4.1 the simulation scope and actors are defined, as well as their respective behaviours.

Chapter 3

Problem Statement

Based on the problem analysis the following problem statement has been phrased:

How is it possible to implement a flocking AI and adapt it to different animal behaviours to create a dynamic environment.

The creation and evaluation of a distributed AI framework applied to a virtual savannah simulation using in Unity will be described. Different setups will be created to evaluate a set of hypothesis and ultimately the problem statement will be revised. The purpose of the AI framework is to create a dynamic simulation with realistic animal behaviour and movement patterns.

CHAPTER 4

DESIGN AND IMPLEMENTATION

In this chapter the following topics will be covered:

- Animal Behaviour Design: An explanation of the design choices made for the animal behaviour in the simulation.
- Game Design in Unity: A description of how the Unity 3D Engine was used to set up the simulation.
- Behaviour Scripting: A presentation of the steering methods scripted to move the animals in the simulation.

4.1 Animal Behaviour Design

As explained in chapter 1, one of this simulation's objective is to represent the animals' movement patterns realistically. In order to achieve this, some design choices have been taken to try and describe each animal's behaviour in the wilderness. Different steering behaviours and small state machines have been used to define how they will react to certain events, and how the interaction will be between members of their own species and other species.

Since the purpose of this simulation is not being precise about the actual interactions between species, there might be inaccuracies between the way the actors behave in the simulation and the way they do in the wilderness. These choices were made for the sake of the steering behaviours and movement patterns adequate display and for simplicity.

4.1.1 Zebra

The zebra has been the herd animal of choice for the simulation. The zebras will behave differently depending on whether they are alone or in company of other members of the herd.

When alone, the zebras will wander aimlessly around the map, at a slow pace. If the zebras are gathered in a herd -a herd consists of two or more zebras- they will wander together while behaving in the following manner:

- They will keep some distance between each other, not to bump into neighbours.
- They will try to stick with the herd, not drifting away from it.
- They will try to keep up with the herd, not falling behind or speeding away from it.
- They will try to have a similar alignment, facing the same direction.
- They will show small differences in alignment and speed.

Whether the zebra is alone or in a herd, when encountering a lion, it will always try to flee from it, running in the opposite direction. However, if the zebra is in a herd, it will try to stay with the rest of its fellow zebras while fleeing as much as possible. Nevertheless the urge for survival and run away from the predator is stronger than the need for staying with the herd, and if required, the zebra will run away from the rest in an attempt to save its life. If more than one lion is approaching, the zebra will try and run away from them all towards the safest direction.

4.1.2 Lion

Te lion has been the predator of choice for this simulation. They will track the herd of zebras down and try to catch a prey. Depending on whether they are a pack or an individual, they will use different hunting strategies.

The lion's behaviour is broken down into different states:

- Idle
- Preparing for attack
- Attacking
- Eating
- Retreating

When Idle, the lion will wander at a slow pace until it finds a herd of zebras. If there are more than one lion, they will do the same, but navigating together as a pack, close to each other, but mantaining a certain separation between themselves. Once they identify a zebra or a herd, the lion or lions immediately switch to the "Preparing for attack" state.

The "Preparing for attack" state is the only one in which hunting alone or in a pack makes a difference. When alone, the lion will approach the herd slowly, getting close without being noticed. Once it is close enough it will switch into the "Attacking" state. On the other hand, if there are several lions, a surrounding attack will be performed. The alpha lion will approach from the front, close to the herd, but far enough not to be noticed, and the rest will go around the herd, to the opposite side, doing the same from the opposite direction. Once overy lion is in the designated position, the whole pack will switch into the "Attacking" state.

When a lion switches into the "Attacking" state, it will look for the nearest prey, and fully acceleraty towards it. Hunting in a pack should increase the chances of catching a prey, since the attack is being done from two opposite fronts, drawing the herd against the rest of the lions. The acceleration and the element of surprise are the strong points of the lions, since they can run just as fast as a zebras, however they have less stamina and if they don't get a catch fast, the zebras will just flee away since they have much more stamina and can keep a high running pace for an extended period of time.

Once the attacking has been performed, there are two possible outcomes. If any of the lions gets a catch on a prey, the rest of the pack will notice it, and the whole pack will switch into the "Eating" state. However if the predators don't manage to get any of the zebras, they will eventually get tired, with their stamina lowered. In this case, they will go into the "Retreating" state.

When in the "Eating" state, the lions will have hunted a prey, and they all will forget about the rest of the herd and gather around the dead zebra, eating. They will remain like this until the end of the simulation.

Once the lion is in the "Retreating" state, it will lower down its running pace to recover stamina, and gather with the rest of the pack if there are more lions, while wandering around the savannah. They will remain like this until the end of the simulation.

The fact that the "Eating" and "Retreating" states are final states in the simulation, will serve as a parameter for testing later on and determine whether or not the lions have been succesful on their hunting attempt. If the lions went back to attempt another hunt after retreating, they would keep trying until they were succesful. As discussed in chapter 1, the aim of the simulation is to create a coherent yet random outcome on each simulation.

4.1.3 Elephant

The elephant is the neutral actor in the simulation. The elephant will always wander alone, and will pay no attention to any of the animals on the savannah since none of them pose a threat to him. The lions and zebras will only try not to collide with him, and go around him if he is on their way. This same behaviour is displayed by the rest of the animals with other obstacles, like trees or big rocks.

4.2 Game Design in Unity

When creating a simulation using Unity, there are different aspects that need to be taken care of. In this chapter, the implementation of the simulation in the Unity3D engine will be described, explaining each of the elements used in the game scene and how the physics engine proved to be of great use. The scripting aspect of the simulation will be discussed in the following chapters.

The main assets for the simulation (terrain, models, textures and animations) were taken from the Savannah project as menctioned in chapter 1. However, none of the previous project scripts were used in this simulation, creating all used behaviours from scratch. The scene was stripped from all interactivity, leaving only the terrain and using the available assets to create the prefabs that will be our actors in the scene.

An important design choice was to remain in a two and a half dimension environment (Illington and Funge 2009, pp.43) to simplify the navigation and adapt it to the actual needs of the simulation. Movement in three dimensions is simple to implement, however orientation becomes a tricky problem in this case, and is best to avoid unnecessary complications. In two and a half dimensions, we deal with a three dimensional position, however leave the vertical axis to be dealt with by the force of gravity. In this project the two and a half dimension space consists in the X and Z axis movement through the plane that the savannah terrain creates, where the remaining half dimension is the orientation our actor is facing represented as a single value, which is confined to the [-180..180] space. Since the scene is essentially a plane, the vertical alignment will not be taken into account, and the movement will take place in the [X,Z] plane.

There is a prefab for each animal, which consists on the animal's mesh with animations and textures, together with a set of game objects with colliders tagged and layered for the proper detection of one another, and the correspondent scripts for each of them.

In th Zebra Prefab (figure 4.1) the root game object named "ZebraPrototype" and tagged as "Zebra" contains a detection trigger collider and the "HerdingBehaviour" script. This game object is in the "DefaultDetection" layer. The Zebra mesh is parented to the root game object. Another game object called "ZebraCollider" containing another trigger collider and a kinematic rigidbody is also parented to the root game object. It is also tagged as Zebra, however it is included in the "ZebraDetection" layer.

Figure 4.1: The zebra prefab in the Unity editor.

In the Lion Prefab (figure 4.2) the root game object, named "LionPrototype" and tagged as "Lion" contains a detection trigger collider and the "PackBehaviour" script. This game object is in the "DefaultDetection" layer. The Lion mesh is parented to the root game object. Another game object called "LionCollider" containing another trigger collider and a kinematic rigidbody is also parented to the root game object. It is also tagged as Lion, however it is included in the "LionDetection" layer.

Figure 4.2: The lion prefab in the Unity editor.

In the Elephant Prefab (figure 4.3) the root game object, named "ElephantPrototype" and tagged as "Elephant" contains the "LonerBehaviour" script. This game object is in the "DefaultDetection" layer. The Elephant mesh is parented to the root game object. Another game object called "ElephantCollider" containing a trigger collider is also parented to the root game object. It is also tagged as Elephant, however it is included in the "Obstacles" layer.

Figure 4.3: The elephant prefab in the Unity editor.

The setup of the prefabs in layers, different colliders, rigid bodies and tags has to do with the interactions handled by the physics engine. Unity requires at least one of the two game objects that are interacting through their colliders to have a rigid body attached, since the collisions are calculated through the physics engine. The rigid-bodies are only needed to trigger the events, that's why they are marked as kinematic, and the physics engine doesn't do any of the movement calculation.

The colliders in the root game object of the prefabs are used as an area of detection, to map the surroundings in search of other actors (that's why the elephant doesn't have one), while the colliders in the parented game objects serve as a way to be detected. Once another collider enters our detection zone, the tags they are labeled with serve as identifiers to determine what kind of animal they are. To solve the problem of non desired interactions between two colliders (such as a detection zone entering another detection zone) we make use of the layers in the physics engine. As you can see in figure 4.4, there is a matrix of booleans determining which layers can interact with each other. In this case we only need the "DefaultDetection" layer not to interact with other elements of the same layer.

The physics engine also offers utility functions such as ray casting. These have been useful when looking for objects to avoid, such as elephants, trees or rocks. In the following chapter the scripting handling all these interactions (from collisions and tags to ray casting) will be explain in depth.

4.3 Behaviour Scripting

In this chapter the scripting used to simulate animal behaviour is explained in depth. Different steering scripts have been implemented. When combined and blended properly they can achieve remarkable movement patterns that create the illusion of

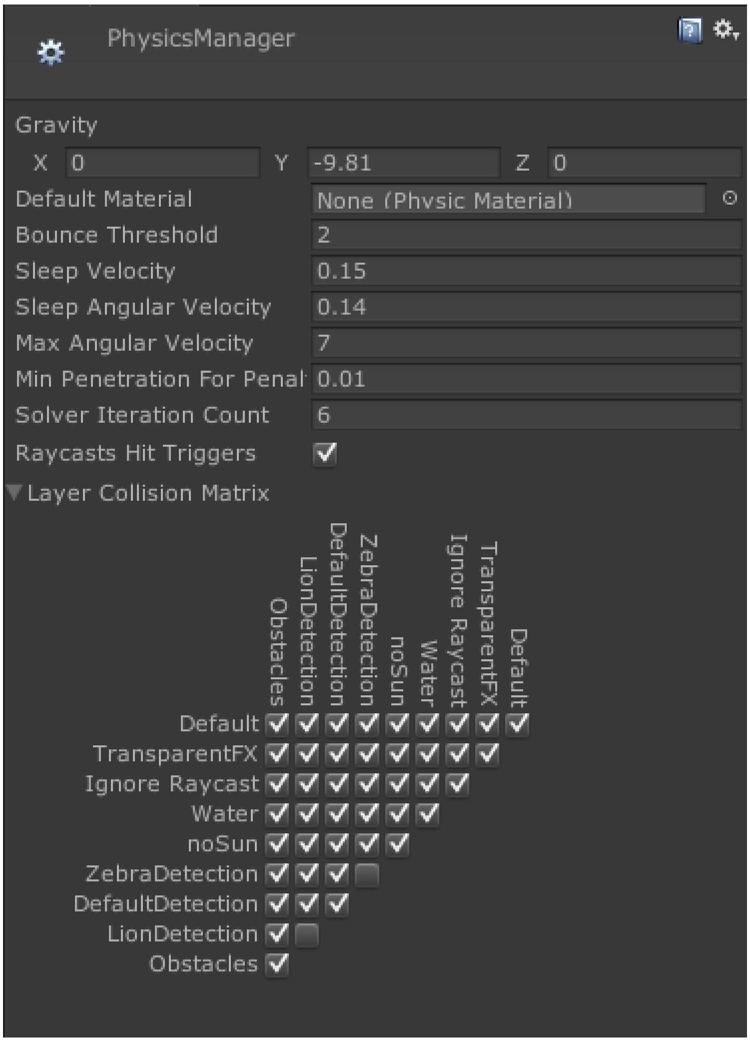

Figure 4.4: The layer interaction matrix from the physics engine.

intelligence and emergent behaviour as well as cooperation in the animal world.

As explained in section 2.4, the steering behaviours provide a steering output that can be blended in order to recreate certain movement patterns. The `Steering` class (listing 4.1) has been created in order to handle this output and also to handle some basic operations. The `Steering` functionalities are the following:

Listing 4.1: Extract from Steering Class C# script

```
public class Steering {
   //Steering
   public Vector3 linearAcceleration = Vector3.zero; //Linear
       acceleration
   public float   angularAcceleration = 0f;  //Angular acceleration

            [...]

            //Adds another steering to this one, given a specific
```

```
                weight
    public void Add(Steering newSteering, float weight)
    {
        linearAcceleration += newSteering.linearAcceleration * weight;
        angularAcceleration += newSteering.angularAcceleration * weight;
    }

    //Crops down to the specified maximums
    public void Crop(float maxLinearAcceleration, float
        maxAngularAcceleration)
    {
        //Linear
        if (linearAcceleration.magnitude > maxLinearAcceleration)
        {
            linearAcceleration.Normalize();
            linearAcceleration *= maxLinearAcceleration;
        }

        //Angular
        if (Mathf.Abs(angularAcceleration) > maxAngularAcceleration)
        {
            angularAcceleration /= Mathf.Abs(angularAcceleration);
            angularAcceleration *= maxAngularAcceleration;
        }
    }

[...]
```

- Storing linear ang angular acceleration for any given steering output.
- Resetting the steering to zero (useful since it is recalculated on every frame).
- Adding weighted stering. This makes it very easy to blend different steering outputs.
- Cropping down to maximums. Given a maximum linear and angular accelerations, if the currently stored ones exceed them, they are cropped down.
- Mapping to an angular range. This static method is used to translate the Unity rotation range [0..360] to the one used for the steering [-180..180].

The code shown in this section is exhaustively commented, and thus the descriptions focus on the functionalities and the calculation process, rather than on a line-by-line explanation. The complete code can be found at the end of this report, in the annex.

4.3.1 Simple Steering Scripts

The following scripts are classes designed to be called from other behaviours. They do not extend `MonoBehaviour` as usually Unity scripts do, since they are not supposed to be attached to any game objects or make use of any of the Unity callbacks. Instead they are dinamically created and used to be blended with other steering methods.

Different steering scripts will provide a certain type of steering output:

- Linear output: Only the linear acceleration is calculated.
- Angular output: Only the angular acceleration is calculated.
- Complete output: Calculates both linear and angular acceleration.

All the following classes have at least one `GetSteering()` method, which returns the steering output given a set of parameters. More than one `GetSteering()` method is implemented on some of the classes in order to adapt to the overloading needs that some of the subclasses have.

Note that some of the classes extend other steering methods. This is done for the purpose of code optimization, modularity and usability. They will delegate some part of the steering calculation to the subclass and avoid code repetition.

Align

The first steering class is `AlignBehaviour`. It returns the steering needed to rotate towards a certain orientation value. No linear movement is considered in this steering method, so the linear steering value returned will always be the zero vector.

The main GetSteering() method (listing 4.2) takes a list of GameObjects and calculates their average orientation in order to steer towards it. The second method is overloaded, and it does the exact same thing, except for the fact that the taken argument is not a list of GameObjects, but a Quaternion with a specified orientation. The method does exactly the same, skipping the part where it needs to calculate the average orientation, since the target orientation is already specified. The reason to create two different methods is to ensure code optimization using subclasses, as explained later. The rest of the arguments taken by these methods are the same: the object's transform, rotation speed, maximum rotation speed and maximum angular acceleration. The transform and rotation speed are used to calculate the steering outcome; the maximum rotation and maximum angular acceleration are used as control parameters.

Listing 4.2: GetSteering method from Align Class C# script

```
[...]
```

```
        //Returns the steering for align given a list of elements
           in range
public Steering GetSteering(List<GameObject> targets, Transform
   transform, float rotationSpeed, float maxRotation, float
   maxAngularAcceleration)
{
   alignSteering.Reset();

   if (targets.Count > 0)
   {
      //Calculate the average orientation of the neighbouring
         elements
      Quaternion newTarget = Quaternion.identity;
      Vector3 auxVector = Vector3.zero;
      //Loop through each target
      foreach (GameObject target in targets)
      {
         auxVector += target.transform.rotation.eulerAngles;
      }
      //Average
      auxVector /= targets.Count;
      newTarget = Quaternion.Euler(auxVector);

      //Naive direction to the target
      float rotationDelta =
         Steering.MapToRange(newTarget.eulerAngles.y) -
         Steering.MapToRange(transform.eulerAngles.y);
      float rotationSize = Mathf.Abs(rotationDelta);
      float targetRotation = 0f;

      //Fix for transitions between -180 to +180 and vice versa
      float openAngleFactor = 1f;
      if (rotationSize > 180)
         openAngleFactor = -1f;

      //If we are there, we do nothing
      if (rotationSize < targetRadius)
      {
         return alignSteering;
      }
      //If we are outside the slow radius, we turn at maximum
         rotation
      else if (rotationSize > slowRadius)
      {
         targetRotation = maxRotation;
      }
      //Otherwise we calculate a scaled rotation
      else
```

```
        {
            targetRotation = maxRotation * rotationSize / slowRadius;
        }

        //The final target rotation combines speed (already in the variable) and direction
        targetRotation *= (rotationDelta / rotationSize) * openAngleFactor;

        //Acceleration tries to get to the target rotation
        alignSteering.angularAcceleration = targetRotation - rotationSpeed;
        alignSteering.angularAcceleration /= timeToTarget;

        //Check if the acceleration is too great
        float absAngularAcceleration = Mathf.Abs(alignSteering.angularAcceleration);
        if (absAngularAcceleration > maxAngularAcceleration)
        {
            alignSteering.angularAcceleration /= absAngularAcceleration; //Get the sign of the acceleration
            alignSteering.angularAcceleration *= maxAngularAcceleration; //Set it to maximum permitted
        }
    }
    else
    {
        Debug.LogWarning("No neighbours found, Align aborted.");
    }

    return alignSteering;
  }
[...]
```

Whether it is a group of game objects with an average orientation or a specified Quaternion with an orientation, the aim of the class is to steer towards it. Before returning any value, the methods check for specified thresholds. The first threshold is the “target radius”, which specifies how close to the target orientation we have to be in order to consider we have achieved the orientation we wanted. The second one is the “slow radius”. This is a larger value, and specifies the threshold within which we have to start slowing down our rotaion speed in order to have a safe landing into the target radius, and don’t go past it. The steering calculation for each of these zones are:

- If our current orientation is within the specified target radius we don’t return any steering, since we consider we are already where we want to be.

- If our current orientation is out of a specified slow radius (a larger value than the target radius) we return the maximum rotation speed in the direction of the target orientation. Since we are too far away from the target rotation, we have no need to slow down.

- If our current orientation is not within the target radius, nor out of the slow radius, we are within the slowing zone. This zone is implemented so the rotation steering returned is smaller the closer we get to the target radius. This means we interpolate from the maximum rotation speed (when the orientation is at the limit of the slow radius) to zero rotation speed (when we reach the target radius).

Once we have calculated the rotation speed we want to achieve, we need to calculate the angular acceleration required, since the steering values are returned in terms of linear and angular acceleration. In order to calculate this angular acceleration, our current rotation speed is subtracted form the target rotation speed, and later divided by the time value that represents the time we want to take in order to achieve this angular speed (this time parameter is called timeToTarget). After this, we make sure our acceleration value doesn't surpass the specified maximum angular acceleration (and if it does, crop it down to the maximum), and then return it.

Face

The face behaviour's class `FaceBehaviour`, has also the goal of achieving a target orientation, however in this case the aim is to orient towards a target, instead of achieving the target's current orientation. Due to the similiarity with `AlignBehaviour`, `FaceBehaviour` is a subclass of it, and will delegate some of its functionalities.

There are two `GetSteering()` methods in this class. The first one is taking the current velocity of the object and rotating to face that direction. The second one (listing 4.3) takes a specific position in the world and rotates to face that location. Like in `AlignBehaviour`, the aim is to have different overloaded methods in order to adapt for subclasses that will delegate functions to this one.

Listing 4.3: GetSteering method from Face Class C# script

```
[...]
            //Returns the steering trying to face a specific position
   public Steering GetSteering(Vector3 position, Transform transform,
       float rotationSpeed, float maxRotation, float
       maxAngularAcceleration)
   {
      faceSteering.Reset();
      Vector3 direction = position - transform.position;
      //If zero, we make no changes
      if (direction.magnitude == 0f)
```

```
            return faceSteering;

        //Create the target rotation
        Quaternion target = Quaternion.Euler(new Vector3(0f,
            Steering.MapToRange(Mathf.Atan2(direction.x, direction.z) *
            Mathf.Rad2Deg), 0f));

        //Fetch it to align, and return the steering
        return base.GetSteering(target, transform, rotationSpeed,
            maxRotation, maxAngularAcceleration);
    }
[...]
```

The only difference between the two methods is that in the one where the current velocity of the object is taken, a world position is calculated by adding the velocity vector to the current position of the object, where in the second method the position is already given in the parameters. Once a position to be faced is calculated or given, the required target rotation is calculated by using the direction vector towards the specified location, and then it is delegated to the base class `AlignBehaviour`.

Wander

The wandering movement's class, `WanderBehaviour`, calculates the steering for an actor to move aimlessly about, but in a controlled manner. A random direction is calculated, but with control parameters that allow for direction, rotation and speed limits. Due to having in common some of its functionalities with `FaceBehaviour`, `WanderBehaviour` is a subclass of it, and delegates some of the workload to it.

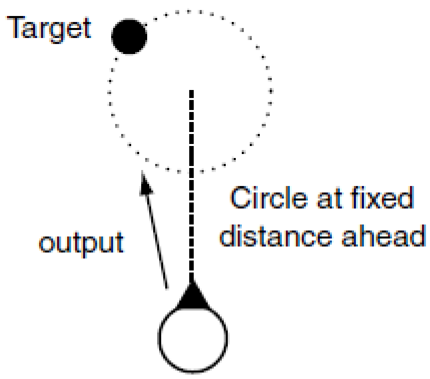

Figure 4.5: The wander behaviour.

There is only one `GetSteering()` method in the class (listing 4.4), which calculates a target location to delegate to `FaceBehaviour` from a set of given parameters. As depicted in figure 4.5, the direction is calculated by using three variables: the "wander offset", the "wander radius" and the "change rate". Since calculating a random direction from our current location would provide a highly variable outcome, a more contained orientation range is calculated by placing a circumfarence at a certain distance in front of our current location. The distance at which the circumference lays is the “wander offset”, while the size of the circumference is defined by the “wander radius”. The higher the radius value is, the bigger the range of our angular spectrum. The same happens if our offset decreases. By tweaking these two, the desired wander maneuverability is achieved.

Listing 4.4: GetSteering method from Wander Class C# script

```
[...]
        //Returns the steering for wander
    public Steering GetSteering(Transform transform, float
        maxRotation, float rotationSpeed, float maxLinearAcceleration,
        float maxAngularAcceleration)
    {
        //Calculate the target to delegate to Face
        //Update the wander target local orientation
        wanderOrientation = Steering.MapToRange(wanderOrientation +
            RandomBinomial() * wanderChangeRate);

        //Calculate the total combined target orientation
        targetOrientation = Steering.MapToRange(wanderOrientation +
            Steering.MapToRange(transform.eulerAngles.y));

        //Calculate the center of the wander circle
        circleCenter = transform.position + transform.forward *
            wanderOffset;

        //Calculate the target location
        wanderTarget = circleCenter +
            RotationToVector3(targetOrientation) * wanderRadius;

        //Delegate to Face to handle rotation steering
        wanderSteering = base.GetSteering(wanderTarget, transform,
            rotationSpeed, maxRotation, maxAngularAcceleration);

        //Set linear acceleration to maximum in the direction of the
            orientation
        wanderSteering.linearAcceleration = maxLinearAcceleration *
            RotationToVector3(Steering.MapToRange(transform.eulerAngles.y));

        return wanderSteering;
    }
[...]
```

In order to change direction at a certain rate within this range is by defining a “change rate”. A specific point in the circumference is selected by calculating a “wander orientation", which varies in one direction or another at a "change rate" speed. Notice how a random number close to zero is calculated in order to modify our current "wander orientation" by calling `RandomBinomial()`. This provides an easy way of obtaining small values close to zero by multiplying two random numbers between -1 and 1 and returning the result, which can be positive or negative, thus achieving variation in both directions. Once the new "wander orientation" is calculated, the specified position (depicted as "target") is given in absolute value to the `FaceBehaviour`, which calculates the angular steering.

Unlike the previus steering methods, `WanderBehaviour` also provides linear steering. In this case the linear steering is calculated simply by taking the normalized direction towards the "wander target" and multiplying it by the maximum linear acceleration recieved as a parameter. The rest of the parameters received are the ones required by the `FaceBehaviour` to calculate its steering.

Seek

The seek's steering class `SeekBehaviour` provides the steering needed to be drawn towards a specified location. The `GetSteering()` method (listing 4.5) receives a desired target location, the `Transform` of the actor and the maximum linear acceleration permitted. The function caluclates the vector from our current position to the target position, and normalizes it to get the direction of the desired acceleration.

Listing 4.5: GetSteering method from Seek Class C# script

```
[...]
            //Returns the steering for seek given a target position
                to reach
   public Steering GetSteering(Vector3 targetPosition, Transform
       transform, float maxLinearAcceleration)
   {
      seekSteering.Reset();

      //Calculate strength of the attraction
      Vector3 direction = targetPosition - transform.position;
      direction.y = 0f; //We make sure no vertical alignment is taken
         into account
      float distance = direction.magnitude;
      //If we have arrived, we don't need to steer
      if (distance < arriveThreshold)
         return seekSteering;
      //Otherwise, we calculate the strength of the attraction
      float strength = Mathf.Min(attractionCoefficient * distance *
         distance, maxLinearAcceleration);

      //Add acceleration
      direction.Normalize();
```

```
      seekSteering.linearAcceleration += strength * direction;

      return seekSteering;
   }
[...]
```

In order to calculate the magnitude of the acceleration, an attraction coefficient (which serves as a control variable) is multiplied by the distance to the location squared, so the closer the current location is to the target, the weaker the force is and vice versa. After the force is calculated, a check is made so it doesn't exceed the maximum linear acceleration. This steering class only provides linear steering. An "arrive threshold" variable is used to consider whether or not the location is already reached, and thus no steering is required.

Flee

Flee's steering class `FleeBehaviour` does exactly the opposite of `FleeBehaviour`. This class uses two different `GetSteering()` functions. Both methods use a target location the actor wants to flee from, the only difference is in the first one (listing 4.6) a list of elements is given, and their center of mass is used as target location, where in the second one the target location is already given as a parameter.

Listing 4.6: GetSteering method from Flee Class C# script

```
[...]
            //Returns the steering for flee given a set of targets to
               avoid
   public Steering GetSteering(List<GameObject> targets, Transform
      transform, float maxLinearAcceleration)
   {
      fleeSteering.Reset();

      if (targets.Count > 0)
      {
         Vector3 gravityCenter = Vector3.zero;
         //Loop through each target
         foreach (GameObject target in targets)
         {
            gravityCenter += target.transform.position;
         }

         //We've gone through all the targets, divide to get the
            average
         gravityCenter /= targets.Count;

         //Calculate strength of the repulsion
         Vector3 direction = transform.position - gravityCenter;
```

```
            direction.y = 0f; //We make sure no vertical alignment is
                taken into account
            float distance = direction.magnitude;
            float strength = Mathf.Min(repulsionCoefficient /
                (distance*distance), maxLinearAcceleration);

            //Add acceleration
            direction.Normalize();
            fleeSteering.linearAcceleration += strength * direction;
        }
        else
        {
            Debug.LogWarning("No targets found, aborted.");
        }

        return fleeSteering;
    }
[...]
```

The linear acceleration's direction is the vector to the target location normalized and inverted (this can be done simply by subtracting the locations in the inverse order, and then normalizing it). To calculate the strength of the repulsion a simillar formula to the one in `SeekBehaviour` is used, however this time the coefficient (used as a control variable) is a repulsion one, and the squared distance is inverted, so the closer the actor is to the target location, the stronger the repulsion. A check is done in case the resulting acceleration is bigger than the maximum allowed, and the linear steering is returned. No angular steering is returned.

Obstacle Avoidance

The `ObstacleAvoidenceBehaviour` script steers towards a safe location in order not to collide with obstacles that may be in the way. In this case, the only obstacles to avoid are the elephants and the trees. There is only one `GetSteering()` method (listing 4.7), which uses two variables to calculate the safe location: "look ahead" and "avoid distance". The first one determines the distance at which the scan for possible obstacles is done in front of the actor. The second one determines how far from it the actor needs to steer in order to avoid the obstacle.

Listing 4.7: GetSteering method from Obstacle Avoidance Class C# script

```
[...]
            //Returns the steering for obstacle avoidance
    public Steering GetSteering(Transform transform, Vector3 velocity,
        float maxLinearAcceleration)
    {
        obstacleAvoidanceSteering.Reset();

```

```
        Vector3 targetPosition = Vector3.zero;
        int raycastLayer = 12;//Obstacles layer
        RaycastHit hit;
        //Check in front
        if (Physics.Raycast(transform.position, transform.forward, out
            hit, lookAhead, 1 << raycastLayer))
        {
            targetPosition = hit.point + hit.normal * avoidDistance;
            obstacleAvoidanceSteering = base.GetSteering(targetPosition,
                transform, maxLinearAcceleration);
            return obstacleAvoidanceSteering;
        }
        else//Check the sides
        {
            //Left
            Vector3 leftRayDirection = (transform.forward -
                transform.right).normalized;
            if (Physics.Raycast(transform.position, leftRayDirection,
                out hit, lookAhead, 1 << raycastLayer))
            {
                targetPosition = hit.point + hit.normal * avoidDistance;
                obstacleAvoidanceSteering =
                    base.GetSteering(targetPosition, transform,
                    maxLinearAcceleration);
                return obstacleAvoidanceSteering;
            }
            else//Right
            {
                Vector3 rightRayDirection = (transform.forward +
                    transform.right).normalized;
                if (Physics.Raycast(transform.position,
                    rightRayDirection, out hit, lookAhead, 1 <<
                    raycastLayer))
                {
                    targetPosition = hit.point + hit.normal *
                        avoidDistance;
                    obstacleAvoidanceSteering =
                        base.GetSteering(targetPosition, transform,
                        maxLinearAcceleration);
                    return obstacleAvoidanceSteering;
                }
                else
                {
                    return obstacleAvoidanceSteering;
                }
            }
        }
    }
```

```
[...]
```

The scan for obstacles in front of the actor is done by raycasting forward. In order not to interact with undesired elements (for instance, another animal that is not an elephant), a specific layer is set for all the elements in the scene considered obstacles (figure 4.4). The raycast only collides with this specified layer, thus ignoring the rest.

The first step when raycasting is scanning our immediate forward position. To specify the casted ray, the current position of the actor is used, the "forward" vector is retrieved (built-in utility in Unity) to determine the direction, and the above menctioned "look ahead" distance to determine the length of it. If an obstacle is detected, the target location we want to steer towards is calculated by retrieving the location of the collision and adding the normal of the geometry in the hit point as depicted in figure 4.6 multiplied by the "avoid distance".

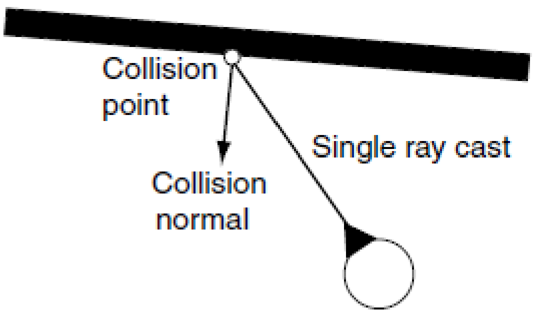

Figure 4.6: The obstacle avoidance behaviour.

Since `ObstacleAvoidenceBehaviour` is a subclass of `SeekBehaviour`, after finding the target location the actor needs to steer towards, it is delegated to the base class to calculate the linear steering. No angular steering is calculated in either class.

Only raycasting ahead of the actor makes it possible for obstacles to be approached sideways and not being noticed. To prevent this, at the end of the first raycast, if no obstacle is found, another raycast is done in a 45 degree angle to the right and left sides. In order to avoid unnecessary checks, the secondary raycasts will only be done if the previous ones have been negative. If no obstacle is found at the end of all the ray casting, no steering is returned.

Separation

`SeparationBehaviour` uses a single `GetSteering()` method (listing 4.8). It receives a list of elements from which the actor needs to keep some distance in order not to bump into them. A threshold is set to check which of the elements of the list is too close and therefore the actor needs to be separated from. A "decay coefficient" is used as a control parameter to calculate the strength of the repulsion.

Listing 4.8: GetSteering method from Separation Class C# script

```
[...]
            public Steering GetSteering(List<GameObject> targets,
               Transform transform, float maxAcceleration)
   {
      separationSteering.Reset();

      if (targets.Count > 0)
      {
         //Loop through each target
         foreach (GameObject target in targets)
         {
            //Check if the target is close
            Vector3 direction = target.transform.position -
               transform.position;
            float distance = direction.magnitude;
            if (distance < threshold)
            {
               //Calculate strength of the repulsion
               float strength = Mathf.Min(decayCoefficient /
                  (distance * distance), maxAcceleration);

               //Add acceleration
               direction.Normalize();
               separationSteering.linearAcceleration += -strength *
                  direction;
            }
         }
      }
      else
      {
         Debug.LogWarning("No neighbours found, Separation aborted.");
      }

      //We've gone through all the targets, return the result
      separationSteering.linearAcceleration.y = 0f;
      return separationSteering;
   }
[...]
```

The script loops through the list of elements, checking if they are too close. If so, the strength of the repulsion is calculated in the same manner as in the `FleeBehaviour` but using the "decay coefficient" instead, while the direction is the vector from the actor's position to the corresponding element, normalized and inverted. The resulting linear steering is added to the total steering (after checking it doesn't exceed the maximum linear acceleration), alltogether with the rest of the resulting steerings for each of the elements in the list. In the end it is averaged by the total number of elements in the list. No angular steering is calculated in this script.

Cohesion

The `CohesionBehaviour` makes sure that the actor stays close to his neighbours by steering towards their center of mass. The class has only one `GetSteering()` method (listing 4.9) which receives the list of neighbours, the actor's transform and the maximum linear acceleration.

Listing 4.9: GetSteering method from Cohesion Class C# script

```
[...]
            //Returns the steering for cohesion
   public Steering GetSteering(List<GameObject> targets, Transform
      transform, float maxAcceleration)
   {
      cohesionSteering.Reset();

      if (targets.Count > 0)
      {
         Vector3 gravityCenter = Vector3.zero;
         //Loop through each target
         foreach (GameObject target in targets)
         {
            gravityCenter += target.transform.position;
         }

         //We've gone through all the targets, divide to get the
            average
         gravityCenter /= targets.Count;

         //Calculate strength of the attraction
         Vector3 direction = gravityCenter - transform.position;
         float distance = direction.magnitude;
         float strength = Mathf.Min(attractionCoefficient * distance,
            maxAcceleration);

         //Add acceleration
         direction.Normalize();
         cohesionSteering.linearAcceleration += strength * direction;
      }
      else
      {
         Debug.LogWarning("No neighbours found, Cohesion aborted.");
      }

```

```
      cohesionSteering.linearAcceleration.y = 0f;
      return cohesionSteering;
   }
[...]
```

In order to calculate the location to steer towards (the center of mass of the group), the method loops through all the elements, adding up their position and averaging by the total number of elements. Once the center of mass is calculated, the distance from the actor to the neighbour's center of mass is multiplied by an "attraction coefficient" to calculate the force of the attraction (reducing it if it exceeds the maximum linear acceleration), while taking the direction to the target location and normalizing it in order to get the direction of the linear steering. Finally, the direction is multiplied by the strength to get the final linear steering. No angular steering is calculated in this class.

Velocity Matching

The `CohesionBehaviour` tries to steer in order to match the velocity of the actor's neighbours. The only `GetSteering()` method (listing 4.10) receives the list of neighbours, the actor's current velocity, and the maximum linear acceleration.

Listing 4.10: GetSteering method from Velocity Matching Class C# script

```
[...]
            //Returns the steering for velocity matching
   public Steering GetSteering(List<GameObject> targets, Vector3
       velocity, float maxAcceleration)
   {
      velocityMatchingSteering.Reset();

      if (targets.Count > 0)
      {
         Vector3 averageVelocity = Vector3.zero;
         //Loop through each target
         foreach (GameObject target in targets)
         {
            if(target.tag=="Zebra")
               averageVelocity +=
                   target.GetComponent<HerdingBehaviour>().GetVelocity();
            if(target.tag == "Lion")
               averageVelocity +=
                   target.GetComponent<PackBehaviour>().GetVelocity();
         }

         //We've gone through all the targets, divide to get the
             average
         averageVelocity /= targets.Count;
```

```
        //Acceleration tries to get to target velocity
        velocityMatchingSteering.linearAcceleration =
            averageVelocity - velocity;
        //If the vector is too small, we ignore it.
        //This is made so they dont alway have the exact same
            orientation + velocity (more realistic)
        if (velocityMatchingSteering.linearAcceleration.magnitude <
            1f)
        {
            velocityMatchingSteering.linearAcceleration =
                Vector3.zero;
        }
        //Time to target
        velocityMatchingSteering.linearAcceleration /= timeToTarget;

        //Check if the acceleration is too fast.
        if (velocityMatchingSteering.linearAcceleration.magnitude >
            maxAcceleration)
        {
            velocityMatchingSteering.linearAcceleration.Normalize();
            velocityMatchingSteering.linearAcceleration *=
                maxAcceleration;
        }
    }
    else
    {
        Debug.LogWarning("No neighbours found, Velocity Matching
            aborted.");
    }

    velocityMatchingSteering.linearAcceleration.y = 0f;
    return velocityMatchingSteering;
}
[...]
```

The function loops through the elements in the list in order to retrieve their linear velocity, adding them up and averaging it by the number of elements in the list in order to calculate the average velocity. Once the target velocity is calculated, the linear steering is calculated by subtracting the actor's velocity from it (resulting in a "velocity difference" vector). If this vector is too small, it is considered that the velocities are simillar enough, and no steering is returned. If the vector is big enough, it is divided by the "time to target" variable in order to calculate the final linear acceleration to be returned. This "time to target" variable represents the time that it takes to reach the target velocity.

4.3.2 Combined Steering Behaviours

Once the basic steering behaviours have been implemented, a script must be created so they can be blended and used in the simulation by attching them to a `GameObject`.

The following combined behaviours extend the `MonoBehaviour` class in order to recieve event callbacks and interact with other components. Some of the callbacks commonly use are:

- `Awake()`: Is called when the script instance is eing loaded. Used to initialize variables.
- `Update()`: Is called once per frame. It is used here for calling debugging functions that need to be persistenly drawn on sreen on every frame.
- `FixedUpdate()`: Is called every fixed framerate frame, which is every step of the physics engine. This is used to recalculate the steering and define the new position and orientation by calling `UpdatePositionAndRotation()`.
- `OnTriggerEnter()`: It is called when another collider enters the trigger attached to this `GameObject`.
- `OnTriggerExit()`: It is called when another collider exits the trigger attached to this `GameObject`.

The `OnTriggerEnter/Exit` events are used to track the other animals moving in our surroundings. The triggers attached to them will serve as detection zones, and when another animal enters them, they are added to the list of "herd" or "pack" depending on their tags. These different collider settings are explained in chapter 4.2.

The blending of the different steering behaviours will result in a final `Steering` output. Once this information is calculated it is used to determine the new position and orientation for the actor, and this is done by calling the above menctioned `UpdatePositionAndRotation()` function that every one of these combined steering behaviours have. Before explaining what the function does, some common variables need to be defined. Unity provides the data structures containing the postion and orientation of any given `GameObject`, however it is needed to create variables when trying to track the linear and rotation speed and creating a non-kinematic movement:

- Properties

  Velocity: Stores the current linear speed of the actor.

  Rotation speed: Stores the current angular speed of the actor.

- Control variables

  Maximum linear speed: Limit on the linear speed.

  Maximum rotation speed: Limit on the angular speed.

Maximum linear acceleration: Limit on the linear acceleration.

Maximum angular acceleration: Limit on the angular acceleration.

These variables are initialized to certain values, however on every `FixedUpdate()` linear and angular speed need to be recalculated using the steering output. This is done by using the simplified Newton-Euler-1 integration update (listing 4.11) (Illington and Funge 2009, pp.47).

Listing 4.11: UpdatePositionAndRotation method from the combined steering behaviours.

```
[...]
            //Does the calculations for the position and rotation
               update
   private void UpdatePositionAndRotation(Steering steering)
   {
      //Using Newton-Euler-1 integration
      transform.position += velocity * Time.deltaTime;
      Vector3 auxVector = new
         Vector3(Steering.MapToRange(transform.eulerAngles.x),
         Steering.MapToRange(transform.eulerAngles.y) +
         (rotationSpeed * Time.deltaTime),
         Steering.MapToRange(transform.eulerAngles.z));
      transform.rotation = Quaternion.Euler(auxVector);

      //Update velocity and rotation
      velocity += steering.linearAcceleration * Time.deltaTime;
      if (velocity.magnitude > currentMaxSpeed) //Max Speed control
      {
         velocity = velocity.normalized * currentMaxSpeed;
            //Normalize and set to max
      }

      rotationSpeed += steering.angularAcceleration * Time.deltaTime;
         //Max rotation control
      if (rotationSpeed > maxRotation)
      {
         rotationSpeed /= Mathf.Abs(rotationSpeed); //Get sign
         rotationSpeed *= maxRotation;    //Set to max rotation
      }
   }
[...]
```

The source code for the combined steering behaviours can be found in the annex at the end of the report. Unlike the previous scripts, this ones are extensive and for practical reasons have been decided not to be included in the chapter.

Herd

The script implementing the herd behaviour, `HerdingBehaviour` contains two lists: one storing all the neigbouring herd members, and another one storing all the neighbouring predators. Other variables and constants in this class are used for different purposes, covering the different functionalities the script implements.

An importnat aspect of the `HerdingBehaviour` are the possible states it can be in. There are only two possible `HerdState`: `Idle` or `Fleeing`. This state definition will help in the calculation of some parameters as explained below.

The initializations are done in the `InitializeVariables()` method, which is called in the `Awake()` event. A random initial steering is also called in this event. These initializations consist on the creation of the steering behaviours, the random creation of an age for the actor and the stamina maxed out.

The callback used for updating the state of the actor is the `FixedUpdate()` function. This callback is called for each step of the physics engine. The function calls a method which updates the currents stamina, another one that updates the current maximum speed, and the last one which updates the position and orientation of the actor by retrieving the steering output from the `GetSteering()` function.

The reason the maximum speed needs to be updated is because depending on his state, a herd member will be running at a certain speed relative to their current maximum. It is important to understand the difference between the `ABS_MAX_SPEED` constant, which determines the maximum speed at which a herd member can go at any time, and the `currentMaxSpeed` variable, which determines our current maximum speed according to the herd member's state, age, stamina and absolute maximum speed. These calculations give raise to different speeds on different states, but also different speed on different levels of stamina and different age ranges within the herd. If the herd member is too old or too young, it won't be able to run as fast as a regular adult; if it is too tired because his stamina is low, it will speed down gradually.

The age is calculated by using the `MAX_AGE` constant, which determines the maximum age any herd member can be. A random number is calculated between 1 and `MAX_AGE` and set up as the actor's age. Afterwards, the `SetAgeModifier()` method is called to calculate the speed modifier for the age `ageModifier`. Another constant `MAX_AGE_MOD` is used to set a maximum to this speed modifier, and depending on whether the herd member it is too old or two young, this modifier will increase or decrease. Notice it is interpolated, so the older or younger the actor is, the bigger the modifier. In this case members older than 20 years old will be considered old, and younger than 5 years old will be considered young. The rest will not have any age speed modifier.

Stamina is set to the maximum defined by `MAX_STAMINA` at the initialization. Every `FixedUpdate()` the `UpdateStamina()` function calculates the gain or loss of stamina by comparing the current speed to the absolute maximum speed. If the speed is too close to the maximum, the actor starts to lose stamina gradually, while if the speed is relatively low, the actor will start recovering stamina. If the current stamina goes below a "exhausted threshold" (which is simply a percentage of the maximum stamina), the actor is considered to be exhausted, and a speed modifier is calculated in the same manner as the age modifier. The use of another constant named `MAX_STAMINA_MOD` is used to calculate the modifier, and interpolation is also used in this case, so the lower the stamina, the bigger the sped modifier.

As discussed before, the per-frame update of position and orientation requires new information about the steering of the actor. In order to calculate it, the `GetSteering()` method handles all the blending required and returns it. Since the steering is recalculated at every physic's engine step, the `Steering` variable is reseted in the beginning of the method to avoid any previously calculated steering to be added on top.

After making sure the steering is resetted, the first thing to do is a check on whether there is any other herd members near the actor or not, since a different set of steering behaviours will be used for each case. If there are one or more neighbours, the steering behaviours blended are the following:

- Linear steering behaviours:

  Velocity Matching: Makes sure the herd will try to head the same way.

  Separation: Prevents the herd members from bumping into each other.

  Cohesion: Ensures the unity of the herd, making members to stay near they neighbours.

- Angular steering behaviours:

  Align: Tries to orient the actor the same way his neighbours are aligned.

  Face: Tries to orient the actor towards the current velocity vector.

- Complete steering behaviours:

  Wander: Used to add a bit of randomness to the movement and don't give the feeling of perfect, robot-like coordination.

In the other hand, if there are no neighbours nearby, only the wander behaviour is used, letting it handle both angular and linear steering.

After the check for neighbours is done, a check for predators determines the state of the herd. If there are one or more predators, the state of the actor is set to `Fleeing`, and a blend of the `FleeBehaviour` is added on top of the previously calculated steering. If there are no predators around, the state of the actor is set as `Idle`. On top of that, the `ObstacleAvoidenceBehaviour` is added, since it is always required for the herd members to avoid obstacles no matter the situation.

Finally, the resulting steering is cropped down to the defined maximum angular and linear accelerations, and it is returned for the `UpdatePositionAndRotation()` to calculate our new movement.

A function named `Killed()` is called whenever a predator manages to kill a herd member. This function notifies the rest of the herd, calling their `NotifyDeath()` function, which removes the deceased member from their neighbours list, if it was in the list.

The function `DrawDebug()`, called every frame using the `Update()` callback, although irrelevant for steering purposes, was of great help when debugging the force vectors that conformed the different steering behaviours blended. A set of boolean checkboxes were set in the editor for an easy debug of the whole herd and individual herd members (figure 4.7). This specially helped when balancing forces and setting up control parameters and constants.

Figure 4.7: Herd debugging on the Unity editor.

Pack

The script implementing the pack behaviour, `PackBehaviour` also contains two lists: one storing all the neigbouring pack members, and another one storing all the neighbouring preys (herd members). Other variables and constants in this class are used for different purposes, covering the different functionalities the script implements.

`PackBehaviour` has a set of states it can be driven into. The states and their definitions are the following:

- Wandering: Wander aimlessly around.
- PreparingForAttack: Move towards the attack positions.
- Attacking: Run towards the closest prey to try and catch it.
- Eating: Go towards the spot where a prey was caught.
- Retreating: Wander at a slow pace, regaining stamina.

The initializations follow the same structure as in `HerdingBehaviour()`, where the `InitializeVariables()` and `RandomInitialSteering()` functions are called in the `Awake()` callback. The difference is, even though the same stamina system is used, the age in the predators is not used, and instead a system of power hierarchy is implemented.

The predator's power level is a random number between 0 and 100. This is used to determine the alpha male of the pack while hunting. Basically the pack member with the highest power level is considered the alpha male, and the rest of the pack is notified and assume their roles as non-alpha. This has only consequences when hunting.

The `FixedUpdate()` callback is exactly the same as in `HerdingBehaviour()`: it updates the current stamina, the maximum speed and the position and orientation given a steering output.

The way the `GetSteering()` method is implemented in this class is different from the one implemented in `HerdingBehaviour`. The predators behave differently whether or not they are alone or in a pack, and whether or not there are preys around or not. Besides this, they have many different states that also use different steering behaviours and blends. In order to make it easy to move from one state to another, a state machine is implmented, and functions implementing the behaviour blending are called. The functions are the following:

- `SteerForWanderInPack()` blends:

  Linear: Velocity Matching, Spearation, Cohesion.

  Angular: Align, Face.

- `SteerForWanderAlone()` blends:

  Complete: Wander.

- `SteerForPreparationAlone()` blends:

  Linear: Seek, Flee.

  Angular: Face.

- `SteerForAttackingAlone()` blends:

  Linear: Seek.

  Angular: Face.

- `SteerForEating()` blends:

  Linear: Seek.

  Angular: Face.

- `SteerForPreparation()` blends:

  Linear: Seek, Flee, Separation.

  Angular: Face.

- `SteerForAttacking()` blends:

  Linear: Seek.

  Angular: Face.

Some of the functions have the exact same implementation, however they are kept in different calls for two reasons: making the code more understandable and leaving an option for the different steering calls to be modified independently if they need to be changed in the future.

The `GetSteering()` method starts by resetting the `Steering` to start a new behaviour blend. A check for preys is done afterwards. If there are no preys the predator will wander, and depending whether or not he is alone or with other predators, the function `SteerForWanderAlone()` or the function `SteerForWanderInPack()` will be called. If in the other hand preys are on sight and the actor is in the `Wandering` state, it will automatically change into the `PreparingForAttack` state. Next the pack list is checked to determine whether or not there are more predators in the pack or the actor is alone hunting.

If the predator is not alone and therefore hunting in a pack, the current state is checked:

- PreparingForAttack:

  If the alpha male hasn't been set, the `SetUpHierarchy()` function is called in order to crown one. Tis function looks for the pack member with the most power, and notifies the rest. If all the pack members are in position for attacking, the state is changed to `Attacking`, and `SteerForAttacking()` is called. If not, `SteerForPreparation()` is called.

- Attacking:

  If a prey hasn't been caught by another pack member previously, a check is run to see if the actor has caught one in this moment (if otherwise there is a kill, `SteerForEating()` is called). If the prey is killed this very frame, the rest of the pack is notified, and `SteerForEating()` is called. The state of the predator is changed to `Eating`. If no prey has been caught a check is run to see if the predator is tired. If so, the state is changed to `Retreating()`, otherwise `SteerForAttacking()` is called.

- Eating:

  This is an end-state in the simulation. The actor will be calling `SteerForEating` until the end of the simulation.

- Retreating:

  This is also an end-state in the simulation. The actor will be calling `SteerForWanderInPa` until the end of the simulation.

If the predator is hunting alone, the state is also checked, with different consequences:

- PreparingForAttack:

  If the predator is not in position for attack, `SteerForPreparationAlone()` is called, otherwise `SteerForAttackingAlone()` is, and the state is set to `Attacking`.

- Attacking:

  A check is done to see if the predator has gotten close enough to catch a prey. If so, the prey is killed, `SteerForEating()` is called, and the state is set to `Eating`. If no prey is yet caught, the predator's stamina is checked. If it's tired, the state is changed to `Retreating`. Otherwise, it continues to attack by calling `SteerForAttackingAlone()`.

- Eating:

  This is an end-state in the simulation. The actor will be calling `SteerForEating` until the end of the simulation.

- Retreating:

  This is also an end-state in the simulation. The actor will be calling `SteerForWanderInPa` until the end of the simulation.

On top of the already blended steering behaviours,the `ObstacleAvoidenceBehaviour` is added. It is always required from the predators to avoid obstacles no matter the situation or state.

Finally, the resulting steering is cropped down to the defined maximum angular and linear accelerations, and it is returned for the `UpdatePositionAndRotation()` to calculate the new movement.

In order to do the different checks and calculate attacking positions, several utility functions have been implemented. The full code can be seen in the annex, but here are some relevant methods briefly explained:

- `CalculateAttackPosition()`: Calculates the attack position according to whether or not the actor is the alpha male or one of the rest. The herd's center of mass and perimeter are used as parameters to define the final position.
- `InPositionForAttack()`: Checks whether or not the predator is in the attack position.
- `AllInPositionForAttack()`: Checks whether or not all predators in the pack are in the attack position.
- `GetClosestTarget()`: Selects the closest of the preys in sight.
- `CatchedPrey()`: Checks whether or not the predator is close enough to kill the prey.

In this class, the function `DrawDebug()`, was also called every frame using the `Update()` callback. Although irrelevant for steering purposes, proved extremely useful when debugging the vectors that conformed the different behaviours, and also the calculations for attacking. A set of boolean checkboxes were set in the editor for an easy debugging process, just like with the zebras. This specially helped when balancing forces and also setting up control parameters and constants.

Loner

The `LonerBehaviour` is extremely simple compared to the two previous combined steering behaviours, since the only aim of the script is for an actor to wander without any regard towards other actors, except for other obstacles.

The `Awake()` event calls the initialization of variables and a random intial steering, like seen on the two previous scripts. In this case there are no extra properties, so only the behaviours are initialized.

The `FixedUpdate()` function only calls for `UpdatePositionAndRotation()`, retrieving the steering through the `GetSteering()` function.

The `GetSteering()` function resets the steering, and always does the same blending:

- Linear: Obstacle avoidance.
- Angular: Face.

- Complete: Wander.

The steering is then cropped down to the maximum linear and angular accelerations.

CHAPTER 5

SIMULATION TESTING

This chapter describes the testing process of the simulation. The following testing hypothesis work as a manifestation of the purpose of the experiment:

The predators' success rate significantly varies depending on their numbers and their strategy.

The predators' success rate is not significantly affected by the number of zebras in the herd.

During the final stages of implementation, the different parameters defining the behaviours were tweaked to achieve the desired emergent behaviour. The aim of this tweaking process was to achieve a robust simulation, able to react properly to any possible situation. Different aspects of the simulation are expected to adapt to their designed weight into the general outcome. In this case, the number of predators is assumed to increase the success rate, while the number of zebras should not affect the outcome. The elephants and other obstacles don't have a specified use for the outcome, and are supposed to be a random factor in the simulation since they affect both preys and predators.

Although not specified as a goal, code optimization is an important factor to any simulation, and even more when dealing with a big number of animals, as this project is supposed to do. Also a framework like this is supposed to work along other resource-consuming processes, like visual effects and possibly other game mechanics. These are the reasons why a stress test has also been performed in order to fathom the limits of the simulation.

In the following sections the setup used for the experiment is described, and the results showcased.

5.1 Setup

In this section the setup for the experiment is described.

The experiment was conducted on the unity editor using a computer with the following specifications:

- Model: ASUS Notebook N61Jq Series
- Processor: Intel(R) Core(TM) i7 CPU Q720 @ 1.60 GHz 1.60 GHz
- Ram: 4 GB
- Graphic card: ATI Mobility Radeon HD 5730

The total number of simulations were 300, divided into six different setups:

- 1st: 3 lions, big herd: 50 simulations.
- 2nd: 2 lions, big herd: 50 simulations.
- 3rd: 1 lion, big herd: 50 simulations.
- 4th: 3 lions, small herd: 50 simulations.
- 5th: 2 lions, small herd: 50 simulations.
- 6th: 1 lion, small herd: 50 simulations.

A big herd consisted on 30 zebras; a small one consisted on 10 zebras. The reason for the variation in the number of predators and the number of zebras is to put to test the hypothesis above mentioned. The starting positions of each animal were exactly the same in all cases, and if a certain animal had to be removed from the simulation, only its `GameObject` was disabled, to ensure the exact same position once it was activated again.

The placement consisted on the herd of zebras in the middle of the map. Nearby, an elephant was placed north-west of the herd, facing the zebras' direction. The lions were placed further to the west of the herd. The reason for this setup is ensuring that the maximum number of different situations is mapped through the testing. Having an elephant facing the herd will make sure there will be an interaction with it in most cases, alltogether with some other static obstacles like the trees on the scene.

For the stress test, an increasing number of zebras has been placed for each iteration. These six iterations have gone from 10, 20, 30, 40, 50 to 60 zebras. The zebras have been placed alone, without other animals, however some static obstacles like trees or rocks still remain in the scene.

The different simulations were run, reocrding the outcome for each of the scenarios.

5.2 Results

Table 5.1 shows the success rates for each of the setups. A successful simulation is the one where the lions manage to hunt a prey. A failed simulation is one in which the lions end up retreating without having caught any zebra.

Setup	Success	Fail	Simulations	Rate
1: 3 Predators (Big Herd)	21	29	50	0.42
2: 2 Predators (Big Herd)	12	38	50	0.24
3: 1 Predator (Big Herd)	4	46	50	0.08
4: 3 Predators (Small Herd)	28	22	50	0.56
5: 2 Preadtors (Small Herd)	29	21	50	0.58
6: 1 Predator (Small Herd)	0	50	50	0
TOTAL	94	206	300	0.31

Table 5.1: Number of successful and failed simulations on each setup, together with the success rate.

Tables 5.2 and 5.3 show the success rates according to number of predators hunting. When hunting in a pack (more than one lion), the predators use a more advanced strategy.

Setup	Success	Fail	Simulations	Rate
3 Predators	49	51	100	0.49
2 Predators	41	59	100	0.41
1 Predator	4	96	100	0.04

Table 5.2: Number of successful and failed simulations according to the number of predators, together with the success rate.

Setup	Success	Fail	Simulations	Rate
Pack (>1 predators)	90	110	200	0.45
Alone (1 predator)	4	96	100	0.04

Table 5.3: Number of successful and failed simulations according to the strategy used, together with the success rate.

Table 5.4 shows the success rates according to the size of the herd.

Table 5.5 shows the frame rates according to the size of the herd. The framerate fluctuates during the simulation, and so an averaged value has been recorded.

Setup	Success	Fail	Simulations	Rate
Big Herd	37	113	150	0.25
Small Herd	57	93	150	0.38

Table 5.4: Number of successful and failed simulations according to the herd's size, together with the success rate.

Setup	Frame Rate
10 zebras	185 fps
20 zebras	142 fps
30 zebras	106 fps
40 zebras	71 fps
50 zebras	42 fps
60 zebras	17 fps

Table 5.5: Frame rate according to the herd's size.

CHAPTER 6

DISCUSSION

This chapter consists on an analysis of the results from both quantitative and qualitative approaches. The quantitative analysis will be discussed from the point of view of the previously phrased testing hypothesis, while the qualitative analysis will be discussed according to the goals set in chapter 1, resolving on whether or not the AI framework achieves them. Furthermore, future lines of work will be discussed, analyzing feedback from Aalborg Zoo's staff members, and definig possible improvements.

6.1 Conclusions

In order to draw conclusions from the work presented, the problem statement phrased in chapter 3 must be evaluated:

How is it possible to implement a flocking AI and adapt it to different animal behaviours to create a dynamic environment.

To solve this problem, a distributed AI framework has been implemented using steering methods and combined steering behaviours. The framework has been later on tested and evelauated.

6.1.1 Quantitative Analysis

From a quantitative point of view, the results need to be analyzed from different angles. Although the ideal balance between successful and failed simulations would be 50%, the overall outcome shows a satisfying balance: 31%, which means that one of every three simulations will result in the lions successfully hunting a prey.

Most of the weight on the failure side is due to the low success percentage on test cases 3 and 6, where a single lion was attemptiong to hunt. This shows a vast difference between hunting alone or with other lions. The average success rate when hunting alone is 4%, while the average when hunting in a pack is 45%, as showcased in table 5.3.

As seen in 5.2 the average success rate increases in an 8% when adding a third lion rather than having just 2. This cannot be directly attributed to the increase in the number of lions, since when analyzing the 5.1 table, it can be appreciated that in the cases where the herd is small, adding a third lion decreases the success rate. Probably further testing with an increasing number of lions would be required in order to establish a correlation.

Analyzing table 5.4, the average success rate when using a big herd is 25%, while being 38% when using a small one. Although there's no clear factor determining why, some observations while testing indicate that it may be caused by a stronger and therefore faster reaction to predators when more herd members are nearby. However when looking at the cases where only one lion is hunting, the big herd case is the only one where the lion manages to get any prey. During the testing it was observed that the more zebras in the herd, the more likely it is for one to get caught between obstacles, or awkward situations, providing the lion an easy catch.

Given these results it can be concluded that:

- The hypothesis ***The predators' success rate significantly varies depending on their numbers and their strategy.*** is proven true for the strategy part, however the data concerning the increasing number of predators is inconclusive.
- The hypothesis ***The predators' success rate is not significantly affected by the number of zebras in the herd.*** is proven false. The results show an increase of 13% success rate for a small herd.

Taking a look at the stress test table 5.5, an almost linear decrease in frame rate can be appreciated as the number of zebras increases. During the testing, small frame rate drops happened when a certain number of zebras were interacting with obstacles (around 5 to 15 fps, depending on the number of zebras). It is important to notice how real herds of zebras can get as big as 200.000 members. Although it is very unlikely to currently achieve such numbers, it is important to optimize as much as possible in order to increase the maximum number of zebras that can be handled in the same scene. The current maximum for the computed used for testing is around 60, but at that stage the framerate is already low (around 15 to 20 fps).

6.1.2 Qualitative Analysis

One of the fears when using obstacles in the simulation was they would become a decisive factor on the success rate. During the testing it could be appreciated how obstacles played a positive or negative role for each animal depending on the situation. In some cases they would keep the predators away from the zebras, and in some others the zebras would be cornered by them. Sometimes the obstacles would drive the zebras towards the lions, and sometimes away from them. This is how the obstacles were intended to be, an extra aspect of interaction between animals, but random enough not to affect the overall hunting outcome.

Another succesful aspect is how the predator's tactics work effectively. It can be appreciated how the tactics give the leverage to hunt more efficiently, and how the number of lions executing the maneuver is not of such importance.

By simply running some iterations it can be easily seen that there are many things that can be improved. To begin with, whenever the herd spreads too much, the herd's perimeter grows, and the distance from which the lions attack is proportional to this perimeter. If this happens, the lions attack from too far away, and this leads to them being tired too fast, lowering their chances of hunting a zebra.

In some of the iterations, the herd has been divided into two or even three, because of the combination of static and moving obstacles. In some cases the two groups manage to get back together, however this does not always happen. Even if only 1 zebra wanders away from the rest, this still supposes a problem, since the lions take all the zebras into account when calculating the perimeter of the herd.

An undesired situation takes place when, after the lions becoming tired and retreating, a zebra trying to avoid an obstcle or another lion bumps into one of the retreating predators. However since the predator is retreating, the zebra will not be killed. This situation was not contemplated while implementing the behaviours, and is something that should definately be fixed.

While testing the setups where there is only one lion, it could be easily appreciated that the few succesful hunts were due to eventful circumstances, like an elephant cornering a zebra between himself and a tree. This situations would happen more often with a higher number of zebras on scene.

Another aspect that wasn't completely satisfying was the targetting. At first, targetting the closest prey seemed like a good way of maximizing success, however this gave rise to situations where the lions would hesitate, retargetting from one zebra to another. Also, targetting zebras with a certain speed would make the lion drift away when the direction of the zebra's speed was different from the predator's speed direction. This could be minimized by predicting trajectories. This aspect was one of the most concerning ones when analyzing the outcome of the simulation because it diminishes the realism and feel of the animal behaviour. In the real world, predators use predictions of where their preys are heading towards, and react accordingly.

Although obstacle avoidance successfully worked as intended, some situations where the elephant approached other animals from the side were not handled as well as the ones where the obstacles would be coming from the front. Also, in some occasions the herd would cluster too much when trying to avoid obstacles (getting too close and bumping onto each other). Even though it was intended for the separation to become less important when trying to avoid obstacles and flee from predators, sometimes it looks too unrealistic.

The heavy conditions under which the testing was conducted were trying to cover all possibilities in animal interaction, and because of this, all the previous analysis was possible. However in further iterations where the framework is used for specific scenarios or new purposes, an ad hoc testing should be performed to make sure the animal behaviour adapts to the desired outcome.

From a qualitative point of view, the simulation was a success. An emergent behaviour is appreciated, and different situations involving predators, preys and obstcles are handled with minor fixable errors. The designed behaviours for each animal are executed as designed, and every iteration of the simulation gives rise to new situations, given the same starting conditions. Even though many aspects can be improved, the general outcome satisfies the problem statement and accomplishes the defined goals.

6.2 Aalborg Zoo Meeting

As explained in chapter 1, this project was done in collaboration with Aalborg Zoo. On the 14th of May 2013, a presentation of the work here implemented was done at the Zoo, receiveing feedback from the Zoo's staff members Rikke Kruse Nielsen (Education Officer) and Susanne Solskov (Marketing Manager).

It was made clear that the final purpose of this AI framework is to eventually be included into an application the Zoo can use to educate children on how animals behave in their natural habitat. Most of the feedback recieved was concerning the potential of the framework and how it could be improved on future iterations in order to fit the application's goals.

The overall reaction was satifactory, the animal behaviour was good. They were concerned on how this framework could adapt to either a simple simulation (only defining the initial conditions) or an interactive one. It was explained to them that it could go either way, since additional animals could be introduced during the simulation, and differents parameters could be tweaked too. Since the primary target audience would be children, there was a special interest on a possible interface and how this interface would work in order to change parameters and add or remove animals in real time. As a proof of concept, the simulation doesn't have any interface indicators of the state of the animals, and this was an important factor to them, since the user would need to be aware of the events on the simulation.

A faithful representation of the predator's behaviour in their environment should be implemented in order to become a proper educative application. The strategy used to hunt in pack was just for showcasing purposes, and it should be changed to adapt to reality (for instance, hunting against the wind and hiding in the bushes, instead of surrounding). Other complex behaviours were discussed to be introduced in the future, like fight between zebras for dominance, male zebras trying to keep their herd together and protected, and newborns sticking with their mothers.

Other biological indicators would help the implementation of new behaviours, like definition of gender, thirst level, huger level, etc. Using these, more complex behaviours could make the herd search for food, water, protection, etc.

6.3 Future lines of work

In this section, the possible improvements in future iterations are defined, reflecting on the analysis of the results and the feedback from Aalborg Zoo's staff members.
There are several techincal aspects that can be improved. The first one would be the targetting system for the lions. The current system only selects the closest zebra and moves towards it. A better solution would be identifying the closest zebras, and predicting their next movement and speed. That would be useful for avoiding drifting and indecision.

The obstacle avoidance scirpt could also be improved in order to account for the obstacles approaching from the sides and from behind. Also a better blend with the separation beaviour would avoid massive clustering of zebras in some situations when trying to avoid obstacles.

A system should be implemented in order to avoid permanent division in the herd. After one or more herd members separate from the herd for whatever reason, they should automatically steer back with the bulk of the zebras. An easy solution would be increasing the detection zone for the zebras when they are too far away from the herd's center of gravity, and when they manage to get back with the rest, set it back to normal size.

One of the improvements that need to be done in order to avoid lions getting exhausted too soon, is to change their attack postions from being directly proportional to the herd's perimeter (which can vary if they spread too much, or they are divided) into being at a fixed distance away from the perimeter. Right now they separate from the herd 8 times the herd's radius, so as soon as the radius increases, the distance escalates proportionally. Instead, a fixed distance from the perimeter should be defined, so the change in the herd's distribution over the map doesn't affect their attack position so drastically, and leads them to get tired before they reach the zebras. Also, the fact that the lions don't kill a zebra once they're exhausted even if the zebra bumps into them doesn't make sense. They should kill the prey if they have the chance to.

In order to adapt the framework for an educational application, research on the real behaviour of lions hunting should be done and implemented on the pack behaviour. Also, an interface displaying the state of the animals and the events happening during the simulation would be needed.

Using new biological indicators like thirst, hunger and gender, complex behaviours could be implemented in order to showcase other aspects of each species' behaviour and their interactions with other animals, whether they are from their own species or another:

- Zebras looking for water sources when thirsty.
- Lions trying to hunt more dangerous preys when they are starving.
- Male zebras fighting for dominance.
- Newborns in the herd, sticking close to their mother.
- Male zebras trying to keep their herd together and looking out for predators.

Whatever the application this framewrok is used for, it would be necessary to run some new testing with the new features in order to tweak and adapt the parameters to the new setup.

BIBLIOGRAPHY

(2013). Aalborg zoo website. `http://www.aalborgzoo.dk/`.

Buckland, M. (2005). *Programming Game AI by Example.* Wordware Publishing.

Corporation, V. (1998). Half life. Sierra Entertainment.

Cui, Z. and Z. Shi (2009). Boid particle swarm optimisation. *International Journal of Innovative Computing and Applications 2*(2), 77–85.

Delgado-Mata, C., J. I. Martinez, S. Bee, R. Ruiz-Rodarte, and R. Aylett (2007). On the use of virtual animals with artificial fear in virtual environments. *New Generation Computing 25*(2), 145–169.

EAD, N. (2001). Pikmin. Nintendo GameCube Game.

Eskildsen, S., K. Rodil, and M. Rehm (2013). *Identification and Analysis of Primary School Children's Knowledge Acquisition.* IGI global.

Hartman, C. and B. Benes (2006). Autonomous boids. *Computer Animation and Virtual Worlds 17*(3-4), 199–206.

Illington, I. and J. Funge (2009). *Artificial Intelligence for Games* (2nd ed.). Morgan Kaufmann Publishers.

Moere, A. (2004). Time-varying data visualization using information flocking boids. *IEEE Symposium on Information Visualization*, 97–104.

Reynolds, C. (1987). Flocks, herds, and schools: A distributed behavioural model. *Computer Graphics 21*(4), 25–34. (SIGGRAPH'87 Conference Proceedings).

SIGGRAPH, A. Acm siggraph website.

Studios, W. D. (1994). The lion king. Motion Picture.

Technologies, U. (2013). Unity3d. `http://unity3d.com/`.

unity3d.com (2013a). Layer-based collision detection. `http://unity3d.com/support/documentation/Components/Layer%20Based%20Collision%20detection.html`.

unity3d.com (2013b). Physics. `http://unity3d.com/support/documentation/Manual/Physics.html`.

APPENDIX A

AI FRAMEWORK SOURCE CODE

Listing A.1: Steering Class C# script

```
using UnityEngine;
using System.Collections;

public class Steering {
   //Steering
   public Vector3 linearAcceleration = Vector3.zero; //Linear acceleration
   public float   angularAcceleration = 0f;  //Angular acceleration

   public Steering()
   {
      linearAcceleration = Vector3.zero;
      angularAcceleration = 0f;
   }

   public void Reset()
   {
      linearAcceleration = Vector3.zero;
      angularAcceleration = 0f;
   }

   //Adds another steering to this one, given a specific weight
   public void Add(Steering newSteering, float weight)
   {
      linearAcceleration += newSteering.linearAcceleration * weight;
      angularAcceleration += newSteering.angularAcceleration * weight;
   }

   //Crops down to the specified maximums
   public void Crop(float maxLinearAcceleration, float maxAngularAcceleration)
```

```
    {
        //Linear
        if (linearAcceleration.magnitude > maxLinearAcceleration)
        {
            linearAcceleration.Normalize();
            linearAcceleration *= maxLinearAcceleration;
        }

        //Angular
        if (Mathf.Abs(angularAcceleration) > maxAngularAcceleration)
        {
            angularAcceleration /= Mathf.Abs(angularAcceleration);
            angularAcceleration *= maxAngularAcceleration;
        }
    }

    //Static method for mapping angles into the -180/+180 space
    public static float MapToRange(float rotation)
    {
        //First, map it down to 360º
        rotation = rotation % 360f;
        //Now, map it to -180º + 180º
        if (rotation > 180)
            rotation = rotation - 360;
        return rotation;
    }

}
```

Listing A.2: Align Behaviour C# script

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
/*
 * Align tries to achieve the average orientation of the neighbouring
    elements
 * @Author: Daniel Collado
 */
public class AlignBehaviour {

    //Align
    private Steering alignSteering; //Data structure containing
        steering information
    private Transform targetRotation; //Target rotation
    private float targetRadius = 1f; //Threshold for arrival
    private float slowRadius = 20f; //Threshold for landing
    private float timeToTarget = 0.1f; //Time to achieve target speed

```

```
    //Constructor
    public AlignBehaviour()
    {
       alignSteering = new Steering();
    }

    //Returns the steering for align given a list of elements in range
    public Steering GetSteering(List<GameObject> targets, Transform transform, float rotationSpeed, float maxRotation, float maxAngularAcceleration)
    {
       alignSteering.Reset();

       if (targets.Count > 0)
       {
          //Calculate the average orientation of the neighbouring elements
          Quaternion newTarget = Quaternion.identity;
          Vector3 auxVector = Vector3.zero;
          //Loop through each target
          foreach (GameObject target in targets)
          {
             auxVector += target.transform.rotation.eulerAngles;
          }
          //Average
          auxVector /= targets.Count;
          newTarget = Quaternion.Euler(auxVector);

          //Naive direction to the target
          float rotationDelta = Steering.MapToRange(newTarget.eulerAngles.y) - Steering.MapToRange(transform.eulerAngles.y);
          float rotationSize = Mathf.Abs(rotationDelta);
          float targetRotation = 0f;

          //Fix for transitions between -180 to +180 and vice versa
          float openAngleFactor = 1f;
          if (rotationSize > 180)
             openAngleFactor = -1f;

          //If we are there, we do nothing
          if (rotationSize < targetRadius)
          {
             return alignSteering;
          }
          //If we are outside the slow radius, we turn at maximum rotation
          else if (rotationSize > slowRadius)
```

```
            {
                targetRotation = maxRotation;
            }
            //Otherwise we calculate a scaled rotation
            else
            {
                targetRotation = maxRotation * rotationSize / slowRadius;
            }

            //The final target rotation combines speed (already in the variable) and direction
            targetRotation *= (rotationDelta / rotationSize) * openAngleFactor;

            //Acceleration tries to get to the target rotation
            alignSteering.angularAcceleration = targetRotation - rotationSpeed;
            alignSteering.angularAcceleration /= timeToTarget;

            //Check if the acceleration is too great
            float absAngularAcceleration = Mathf.Abs(alignSteering.angularAcceleration);
            if (absAngularAcceleration > maxAngularAcceleration)
            {
                alignSteering.angularAcceleration /= absAngularAcceleration; //Get the sign of the acceleration
                alignSteering.angularAcceleration *= maxAngularAcceleration; //Set it to maximum permitted
            }
        }
        else
        {
            Debug.LogWarning("No neighbours found, Align aborted.");
        }

        return alignSteering;
    }

    //Returns the steering for align given a target orientation
    public Steering GetSteering(Quaternion newTarget, Transform transform, float rotationSpeed, float maxRotation, float maxAngularAcceleration)
    {
        alignSteering.Reset();

        //Naive direction to the target
        float rotationDelta =
```

```
        Steering.MapToRange(newTarget.eulerAngles.y) - Steering.MapToRange(transform.eulerAngles.y);
    float rotationSize = Mathf.Abs(rotationDelta);
    float targetRotation = 0f;

    //Fix for transitions between -180 to +180 and vice versa
    float openAngleFactor = 1f;
    if (rotationSize > 180)
        openAngleFactor = -1f;

    //If we are there, we do nothing
    if (rotationSize < targetRadius)
    {
        return alignSteering;
    }
    //If we are outside the slow radius, we turn at maximum rotation
    else if (rotationSize > slowRadius)
    {
        targetRotation = maxRotation;
    }
    //Otherwise we calculate a scaled rotation
    else
    {
        targetRotation = maxRotation * rotationSize / slowRadius;
    }

    //The final target rotation combines speed (already in the variable) and direction
    targetRotation *= (rotationDelta / rotationSize) * openAngleFactor;

    //Acceleration tries to get to the target rotation
    alignSteering.angularAcceleration = targetRotation - rotationSpeed;
    alignSteering.angularAcceleration /= timeToTarget;

    //Check if the acceleration is too great
    float absAngularAcceleration = Mathf.Abs(alignSteering.angularAcceleration);
    if (absAngularAcceleration > maxAngularAcceleration)
    {
        alignSteering.angularAcceleration /= absAngularAcceleration; //Get the sign of the acceleration
        alignSteering.angularAcceleration *= maxAngularAcceleration; //Set it to maximum permitted
    }

    return alignSteering;
```

```
    }

}
```

Listing A.3: Face Behaviour C# script

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
/*
 * Face tries to rotate towards the current velocity vector
 * @Author: Daniel Collado
 */
public class FaceBehaviour : AlignBehaviour {

    //Face
    private Steering faceSteering; //Data structure containing
        steering information

    //Constructor
    public FaceBehaviour()
    {
        faceSteering = new Steering();
    }

    //Returns the steering trying to face our velocity
    public Steering GetSteering(Transform transform, Vector3
        currentVelocity, float rotationSpeed, float maxRotation, float
        maxAngularAcceleration)
    {
        faceSteering.Reset();

        //We want to face our current velocity
        //If zero, we make no changes
        if (currentVelocity.magnitude == 0f)
            return faceSteering;

        //Create the target rotation
        Quaternion target = Quaternion.Euler(new Vector3(0f,
            Steering.MapToRange(Mathf.Atan2(currentVelocity.x,
            currentVelocity.z) * Mathf.Rad2Deg), 0f));

        //Fetch it to align, and return the steering
        return base.GetSteering(target, transform, rotationSpeed,
            maxRotation, maxAngularAcceleration);
    }

    //Returns the steering trying to face a specific position
    public Steering GetSteering(Vector3 position, Transform transform,
```

```
        float rotationSpeed, float maxRotation, float
        maxAngularAcceleration)
    {
        faceSteering.Reset();
        Vector3 direction = position - transform.position;
        //We want to face our current velocity
        //If zero, we make no changes
        if (direction.magnitude == 0f)
            return faceSteering;

        //Create the target rotation
        Quaternion target = Quaternion.Euler(new Vector3(0f,
            Steering.MapToRange(Mathf.Atan2(direction.x, direction.z) *
            Mathf.Rad2Deg), 0f));

        //Fetch it to align, and return the steering
        return base.GetSteering(target, transform, rotationSpeed,
            maxRotation, maxAngularAcceleration);
    }

}
```

Listing A.4: Wander Behaviour C# script

```
using UnityEngine;
using System.Collections;
/*
 * The wander behaviour return the steering of a character moving
     aimlessly about.
 * Author: Daniel Collado
 */
public class WanderBehaviour : FaceBehaviour {

    //Wander
    private Steering wanderSteering;    //Data structure containing
        steering information
    private Vector3 wanderTarget = Vector3.zero; //New target for
        wander behaviour
    private float wanderOffset = 60f;   //Offset of the wander circle
    private float wanderRadius = 7.5f;  //Radius of the wander circle
    private float wanderChangeRate = 1f; //Maximum rate at which the
        wander orientation can change
    private float wanderOrientation = 0f; //Holds the current
        orientation (local) of the wander target
    private float targetOrientation = 0f; //Holds the current
        orientation (world) of the wander target
    private Vector3 circleCenter = Vector3.zero; //Holds the position
        of the center of the wandering circle
    //Constructor
```

```
public WanderBehaviour()
{
   wanderSteering = new Steering();
}

 //Returns the steering for face
public Steering GetSteering(Transform transform, float
    maxRotation, float rotationSpeed, float maxLinearAcceleration,
    float maxAngularAcceleration)
{
   //Calculate the target to delegate to Face
   //Update the wander target local orientation
   wanderOrientation = Steering.MapToRange(wanderOrientation +
      RandomBinomial() * wanderChangeRate);

   //Calculate the total combined target orientation
   targetOrientation = Steering.MapToRange(wanderOrientation +
      Steering.MapToRange(transform.eulerAngles.y));

   //Calculate the center of the wander circle
   circleCenter = transform.position + transform.forward *
      wanderOffset;

   //Calculate the target location
   wanderTarget = circleCenter +
      RotationToVector3(targetOrientation) * wanderRadius;

   //Delgeate to Face to handle rotation steering
   wanderSteering = base.GetSteering(wanderTarget, transform,
      rotationSpeed, maxRotation, maxAngularAcceleration);

   //Set linear acceleration to maximum in the direction of the
      orientation
   wanderSteering.linearAcceleration = maxLinearAcceleration *
      RotationToVector3(Steering.MapToRange(transform.eulerAngles.y));

   return wanderSteering;
}

//Returns a length 1 vector with the specified orientation
private Vector3 RotationToVector3(float orientation)
{
   float orientationRads = orientation * Mathf.Deg2Rad;
   return new Vector3(Mathf.Sin(orientationRads), 0f,
      Mathf.Cos(orientationRads));
}

//Returns a random number between -1 and 1 where values around 0
```

```
        are more likely.
    private float RandomBinomial()
    {
        return Random.value - Random.value;
    }
}
```

Listing A.5: Seek Behaviour C# script

```
using UnityEngine;
using System.Collections;
/*
 * Seek steers towards a specified target
 * @Author: Daniel Collado
 */
public class SeekBehaviour {

    //Seek
    private Steering seekSteering;      //Data structure containing
        steering information
    private float attractionCoefficient = 100f; //Holds the constant
        coefficient to calculate reoulsion
    private float arriveThreshold = 10f; //Distance at which we
        consider we've arrived to our destination, and no more
        steering is applied.

    //Constructor
    public SeekBehaviour()
    {
        seekSteering = new Steering();
    }

    //Returns the steering for seek given a target position to reach
    public Steering GetSteering(Vector3 targetPosition, Transform
        transform, float maxLinearAcceleration)
    {
        seekSteering.Reset();

        //Calculate strength of the attraction
        Vector3 direction = targetPosition - transform.position;
        direction.y = 0f; //We make sure no vertical alignment is taken
            into account
        float distance = direction.magnitude;
        //If we have arrived, we don't need to steer
        if (distance < arriveThreshold)
            return seekSteering;
        //Otherwise, we calculate the strength of the attraction
        float strength = Mathf.Min(attractionCoefficient * distance *
            distance, maxLinearAcceleration);
```

```

        //Add acceleration
        direction.Normalize();
        seekSteering.linearAcceleration += strength * direction;

        return seekSteering;
    }
}
```

Listing A.6: Flee Behaviour C# script

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
/*
 * Flee steers away from the element that is being avoided.
 * @Author: Daniel Collado
 */
public class FleeBehaviour {

    //Flee
    private Steering fleeSteering;      //Data structure containing
        steering information
    private float repulsionCoefficient = 1000f; //Holds the constant
        coefficient to calculate repulsion

    //Constructor
    public FleeBehaviour()
    {
        fleeSteering = new Steering();
    }

    //Returns the steering for flee given a set of targets to avoid
    public Steering GetSteering(List<GameObject> targets, Transform
        transform, float maxLinearAcceleration)
    {
        fleeSteering.Reset();

        if (targets.Count > 0)
        {
            Vector3 gravityCenter = Vector3.zero;
            //Loop through each target
            foreach (GameObject target in targets)
            {
                gravityCenter += target.transform.position;
            }

            //We've gone through all the targets, divide to get the
                average
```

```
            gravityCenter /= targets.Count;

            //Calculate strength of the repulsion
            Vector3 direction = transform.position - gravityCenter;
            direction.y = 0f; //We make sure no vertical alignment is taken into account
            float distance = direction.magnitude;
            float strength = Mathf.Min(repulsionCoefficient / (distance*distance), maxLinearAcceleration);

            //Add acceleration
            direction.Normalize();
            fleeSteering.linearAcceleration += strength * direction;
        }
        else
        {
            Debug.LogWarning("No targets found, Cohesion aborted.");
        }

        return fleeSteering;
    }

    //Returns the steering for flee given a single target to avoid
    public Steering GetSteering(Vector3 target, Transform transform, float maxLinearAcceleration)
    {
        fleeSteering.Reset();

        //Calculate strength of the repulsion
        Vector3 direction = transform.position - target;
        direction.y = 0f; //We make sure no vertical alignment is taken into account
        float distance = direction.magnitude;
        float strength = Mathf.Min(repulsionCoefficient / (distance * distance), maxLinearAcceleration);

        //Add acceleration
        direction.Normalize();
        fleeSteering.linearAcceleration += strength * direction;

        return fleeSteering;
    }

}
```

Listing A.7: Obstacle Avoidance Behaviour C# script

```
using UnityEngine;
using System.Collections;
```

```

public class ObstacleAvoidanceBehaviour: SeekBehaviour {

   //Obstacle Avoidance
   private Steering obstacleAvoidanceSteering; //Data structure
      containing steering information

   //Raycast
   private float lookAhead = 30f;         //Distance at which we
      check for obstacles in front of us
   private float avoidDistance = 20f;     //Distance that we want to
      separate from the obstacle

   //Constructor
   public ObstacleAvoidanceBehaviour()
   {
      obstacleAvoidanceSteering = new Steering();
   }

   //Returns the steering for seek given a target position to reach
   public Steering GetSteering(Transform transform, Vector3 velocity,
      float maxLinearAcceleration)
   {
      obstacleAvoidanceSteering.Reset();

      Vector3 targetPosition = Vector3.zero;
      int raycastLayer = 12;//Obstacles layer
      RaycastHit hit;
      //Check in front
      if (Physics.Raycast(transform.position, transform.forward, out
         hit, lookAhead, 1 << raycastLayer))
      {
         targetPosition = hit.point + hit.normal * avoidDistance;
         obstacleAvoidanceSteering = base.GetSteering(targetPosition,
            transform, maxLinearAcceleration);
         return obstacleAvoidanceSteering;
      }
      else//Check the sides
      {
         //Left
         Vector3 leftRayDirection = (transform.forward -
            transform.right).normalized;
         if (Physics.Raycast(transform.position, leftRayDirection,
            out hit, lookAhead, 1 << raycastLayer))
         {
            targetPosition = hit.point + hit.normal * avoidDistance;
            obstacleAvoidanceSteering =
               base.GetSteering(targetPosition, transform,
```

```
                maxLinearAcceleration);
            return obstacleAvoidanceSteering;
        }
        else//Right
        {
            Vector3 rightRayDirection = (transform.forward +
                transform.right).normalized;
            if (Physics.Raycast(transform.position,
                rightRayDirection, out hit, lookAhead, 1 <<
                raycastLayer))
            {
                targetPosition = hit.point + hit.normal *
                    avoidDistance;
                obstacleAvoidanceSteering =
                    base.GetSteering(targetPosition, transform,
                    maxLinearAcceleration);
                return obstacleAvoidanceSteering;
            }
            else//Right
            {
                return obstacleAvoidanceSteering;
            }
        }
    }
}

}
```

Listing A.8: Separation Behaviour C# script

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
/*
 * Separation tries to avoid getting too close to neighbouring
    elements
 * @Author: Daniel Collado
 */
public class SeparationBehaviour {

    //Separation
    private Steering separationSteering; //Data structure containing
        steering information
    private float threshold = 25f; //Holds the threshold to take action
    private float decayCoefficient = 20f; //Holds the constant
        coefficient of decay for the inverse square law force

    //Constructor
    public SeparationBehaviour()
```

```
    {
        separationSteering = new Steering();
    }

    public Steering GetSteering(List<GameObject> targets, Transform transform, float maxAcceleration)
    {
        separationSteering.Reset();

        if (targets.Count > 0)
        {
            //Loop through each target
            foreach (GameObject target in targets)
            {
                //Check if the target is close
                Vector3 direction = target.transform.position - transform.position;
                float distance = direction.magnitude;
                if (distance < threshold)
                {
                    //Calculate strength of the repulsion
                    float strength = Mathf.Min(decayCoefficient / (distance * distance), maxAcceleration);

                    //Add acceleration
                    direction.Normalize();
                    separationSteering.linearAcceleration += -strength * direction;
                }
            }
        }
        else
        {
            Debug.LogWarning("No neighbours found, Separation aborted.");
        }

        //We've gone through all the targets, return the result
        separationSteering.linearAcceleration.y = 0f;
        return separationSteering;
    }
}
```

Listing A.9: Cohesion Behaviour C# script

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
/*
 * Cohesion tries to move towards the center of gracity of
```

```
   neighbouring elements
 * @Author: Daniel Collado
 */
public class CohesionBehaviour {

   //Cohesion
   private Steering cohesionSteering;  //Data structure containing
       steering information
   private float attractionCoefficient = 0.1f; //Holds the constant
       coefficient to calculate attraction

   //Constructor
   public CohesionBehaviour()
   {
      cohesionSteering = new Steering();
   }

   //Returns the steering for cohesion
   public Steering GetSteering(List<GameObject> targets, Transform
       transform, float maxAcceleration)
   {
      cohesionSteering.Reset();

      if (targets.Count > 0)
      {
         Vector3 gravityCenter = Vector3.zero;
         //Loop through each target
         foreach (GameObject target in targets)
         {
            gravityCenter += target.transform.position;
         }

         //We've gone through all the targets, divide to get the
             average
         gravityCenter /= targets.Count;

         //Calculate strength of the attraction
         Vector3 direction = gravityCenter - transform.position;
         float distance = direction.magnitude;
         float strength = Mathf.Min(attractionCoefficient * distance,
             maxAcceleration);

         //Add acceleration
         direction.Normalize();
         cohesionSteering.linearAcceleration += strength * direction;
      }
      else
      {
```

```
            Debug.LogWarning("No neighbours found, Cohesion aborted.");
        }

        cohesionSteering.linearAcceleration.y = 0f;
        return cohesionSteering;
    }
}
```

Listing A.10: Velocity Matching Behaviour C# script

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
/*
 * Velocity Matching tries to achieve the average velocity of the
     neighbouring elements
 * @Author: Daniel Collado
 */
public class VelocityMatchingBehaviour {

    //VelocityMatching
    private Steering velocityMatchingSteering; //Data structure
        containing steering information
    private float timeToTarget = 0.1f; //Holds the time over which to
        achieve target speed

    //Constructor
    public VelocityMatchingBehaviour()
    {
        velocityMatchingSteering = new Steering();
    }

    //Returns the steering for velocity matching
    public Steering GetSteering(List<GameObject> targets, Vector3
        velocity, float maxAcceleration)
    {
        velocityMatchingSteering.Reset();

        if (targets.Count > 0)
        {
            Vector3 averageVelocity = Vector3.zero;
            //Loop through each target
            foreach (GameObject target in targets)
            {
                if(target.tag=="Zebra")
                    averageVelocity +=
                        target.GetComponent<HerdingBehaviour>().GetVelocity();
                if(target.tag == "Lion")
                    averageVelocity +=
```

```
                    target.GetComponent<PackBehaviour>().GetVelocity();
          }

          //We've gone through all the targets, divide to get the average
          averageVelocity /= targets.Count;

          //Acceleration tries to get to target velocity
          velocityMatchingSteering.linearAcceleration = averageVelocity - velocity;
          //If the vector is too small, we ignore it.
          //This is made so they dont alway have the exact same orientation + velocity (more realistic)
          if (velocityMatchingSteering.linearAcceleration.magnitude < 1f)
          {
             velocityMatchingSteering.linearAcceleration = Vector3.zero;
          }
          //Time to target
          velocityMatchingSteering.linearAcceleration /= timeToTarget;

          //Check if the acceleration is too fast.
          if (velocityMatchingSteering.linearAcceleration.magnitude > maxAcceleration)
          {
             velocityMatchingSteering.linearAcceleration.Normalize();
             velocityMatchingSteering.linearAcceleration *= maxAcceleration;
          }
       }
       else
       {
          Debug.LogWarning("No neighbours found, Velocity Matching aborted.");
       }

       velocityMatchingSteering.linearAcceleration.y = 0f;
       return velocityMatchingSteering;
    }
}
```

Listing A.11: Herding Behaviour C# script

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
/*
 * HerdingBehaviour combines different steering behaviours to
```

```
    simulate herd movement
 * Each steering method has its own weight into the mixture
 * @Author: Daniel Collado
 */
[RequireComponent(typeof(Collider))]
public class HerdingBehaviour : MonoBehaviour {

   //Neighborhood
   private List<GameObject> herd;         //List with all the neighbours within our range

   //Predators
   private List<GameObject> predators;    //List with all the predators within our range

   //States in which the herd can be
   public enum HerdState
   {
      Idle,
      Fleeing
   }

   private HerdState herdState = HerdState.Idle; //State of the herd

      //Steering
   private Steering herdSteering;         //Data structure containing steering information
   private Vector3 velocity = Vector3.zero; //Linear speed
   private float  rotationSpeed = 0f;     //Rotation speed
   private float  currentMaxSpeed = 3f;   //Current maximum linear speed
   private float  maxRotation = 45f;      //Maximum rotation speed
   private float  maxLinearAcceleration = 5f; //Maximum linear acceleration
   private float  maxAngularAcceleration = 45f; //Maximum angular acceleration

   //Individual traits
   private float  age = 0f;               //Age of the animal. Too old or too young will decrease its speed
   private float  ageModifier = 0f;       //Malus that modifies our maximum speed depending on our age
   private float  currentStamina = 0f;    //Stamina that we currently have left.
   private float  effortFactor = 0f;      //Variable [0..1] indicating how much effort are we using according to our speed
   private float  restThreshold = 0.3f;   //Point below which our effort lets us gain stamina
```

```
    private float  effortThreshold = 0.6f; //Point above which our effort makes us use stamina
    private float  staminaModifier = 0f;   //Malus that modifies maximum speed depending on our tiredness
    private float  exhaustedThreshold = 0.25f; //Percentage of our stamina at which we become tired and gain a malus in speed

    //Constants
    private const float ABS_MAX_SPEED = 7f; //Absolute maximum linear speed for any herd member
    private const float MAX_AGE_MOD = 1f;  //Maximum value for the speed modifier according to age.
    private const float MAX_STAMINA = 80f; //Time at which we can hold maximum speed, then we tire down.
    private const float MAX_STAMINA_MOD = 1f; //Maximum value for the speed modifier according to stamina
    private const float MAX_AGE = 25f;     //Zebras can live up to around 25 years in the wilderness (40 in captivity)

    //Behaviours
    //Linear
    private VelocityMatchingBehaviour velocityMatchingBehaviour; //Behaviour for velocity matching
    private float velocityMatchingWeight = 0.2f;    //Weight for velocity matching

    private SeparationBehaviour separationBehaviour; //Behaviour for separation
    private float separationWeight = 0.7f;          //Weight for separation

    private CohesionBehaviour cohesionBehaviour;    //Behaviour for cohesion
    private float cohesionWeight = 0.1f;            //Weight for cohesion

    private FleeBehaviour fleeBehaviour;            //Behaviour for flee
    private float fleeWeight = 1.0f;                //Weight for flee

    private ObstacleAvoidanceBehaviour obstacleAvoidanceBehaviour; //Behaviour for obstacle avoidance
    private float obstacleAvoidanceWeight = 2.0f;   //Weight for obstacle avoidance

    //Angular
    private AlignBehaviour alignBehaviour;          //Behaviour for align
```

```
    private float alignWeight = 0.25f;              //Weight for align

    private FaceBehaviour faceBehaviour;            //Behaviour for face
    private float faceWeight = 0.75f;               //Weight for face

    //Combined
    private WanderBehaviour wanderBehaviour;        //Behaviour for wander
    private float wanderWeight = 1.0f;              //Weight for wander

    //Debug
    public bool debugHerd = false;                  //Flag for general script debugging
    public bool debugStates = false;                //Flag for state debugging
    public bool debugVelocity = false;              //Flag for velocity debugging
    public bool debugSeparation = false;            //Flag for separation debugging
    private Vector3 separationVector = Vector3.zero; //Vector for separation debugging
    public bool debugCohesion = false;              //Flag for cohesion debugging
    private Vector3 cohesionVector = Vector3.zero;  //Vector for cohesion debugging
    public bool debugVelocityMatching = false;      //Flag for velocity matching debuggin
    private Vector3 velocityMatchingVector = Vector3.zero; //Vector for velocity matching debugging
    public bool debugFlee = false;                  //Flag for avoid debugging
    private Vector3 fleeVector = Vector3.zero;      //Vector for avoid debugging
    private float vectorDebugFactor = 10f;          //Factor to scale debuggin vectors

    //Initialization
    void Awake()
    {
        InitializeVariables();
        RandomInitialSteering();
    }

    //Setting up varibales
    private void InitializeVariables()
    {
        herd = new List<GameObject>();
```

```
        predators = new List<GameObject>();
        herdSteering = new Steering();

        //Behaviours
        velocityMatchingBehaviour = new VelocityMatchingBehaviour();
        alignBehaviour = new AlignBehaviour();
        separationBehaviour = new SeparationBehaviour();
        cohesionBehaviour = new CohesionBehaviour();
        faceBehaviour = new FaceBehaviour();
        wanderBehaviour = new WanderBehaviour();
        fleeBehaviour = new FleeBehaviour();
        obstacleAvoidanceBehaviour = new ObstacleAvoidanceBehaviour();

        //Age
        age = Random.Range(1f, MAX_AGE);
        SetAgeModifier();

        //Stamina
        currentStamina = MAX_STAMINA;
    }

    //Give a random initial acceleration
    private void RandomInitialSteering()
    {
        velocity = new Vector3(Random.value, 0f, Random.value);
        velocity.Normalize();
        velocity *= Random.Range(0.1f, currentMaxSpeed / 4);

        herdSteering.linearAcceleration = new Vector3(Random.value, 0f, Random.value);
        herdSteering.linearAcceleration.Normalize();
        herdSteering.linearAcceleration *= Random.Range(0.1f, maxLinearAcceleration);
    }

    //Per-frame update
    void Update()
    {
        DrawDebug();
    }

    //Method to handle all the visual debugging
    private void DrawDebug()
    {
        if (debugVelocity)
        {
            Debug.DrawLine(transform.position, transform.position + velocity * vectorDebugFactor, Color.white);
```

```
        }
        if (debugCohesion)
        {
            Debug.DrawLine(transform.position, transform.position +
                cohesionVector * vectorDebugFactor, Color.red);
        }
        if (debugSeparation)
        {
            Debug.DrawLine(transform.position, transform.position +
                separationVector * vectorDebugFactor, Color.green);
        }
        if (debugVelocityMatching)
        {
            Debug.DrawLine(transform.position, transform.position +
                velocityMatchingVector * vectorDebugFactor, Color.blue);
        }
        if (debugFlee)
        {
            Debug.DrawLine(transform.position, transform.position +
                fleeVector * vectorDebugFactor, Color.magenta);
        }
    }

    //Physics update
    void FixedUpdate()
    {
        UpdateStamina();
        UpdateMaxSpeed();
        UpdatePositionAndRotation(GetSteering());
    }

    //Steering blending method
    public Steering GetSteering()
    {
        herdSteering.Reset();

        Steering st;

        if (herd.Count > 0) //If we have neighbours, we behave as a herd
        {
            //Settings
            maxRotation = 45f;

            //Linear
            //Velocity Matching
            st = velocityMatchingBehaviour.GetSteering(herd, velocity,
                maxLinearAcceleration);
            velocityMatchingVector = st.linearAcceleration;
```

```
            herdSteering.Add(st, velocityMatchingWeight);
            //Separation
            st = separationBehaviour.GetSteering(herd, transform,
                maxLinearAcceleration);
            separationVector = st.linearAcceleration;
            herdSteering.Add(st, separationWeight);
            //Cohesion
            st = cohesionBehaviour.GetSteering(herd, transform,
                maxLinearAcceleration);
            cohesionVector = st.linearAcceleration;
            herdSteering.Add(st, cohesionWeight);
            //Wander (Linear + Angular)
            herdSteering.Add(wanderBehaviour.GetSteering(transform,
                maxRotation, rotationSpeed, maxLinearAcceleration,
                maxAngularAcceleration), 0.1f);

            //Angular
            //Align
            herdSteering.Add(alignBehaviour.GetSteering(herd, transform,
                rotationSpeed, maxRotation, maxAngularAcceleration),
                alignWeight);
            //Face
            herdSteering.Add(faceBehaviour.GetSteering(transform,
                velocity, rotationSpeed, maxRotation,
                maxAngularAcceleration), faceWeight);
        }
        else//We behave as a wandering individual
        {
            //Settings
            maxRotation = 5f;
            //Wander (Linear + Angular)
            herdSteering.Add(wanderBehaviour.GetSteering(transform,
                maxRotation, rotationSpeed, maxLinearAcceleration,
                maxAngularAcceleration), wanderWeight);
        }

        //Check fo predators
        if (predators.Count > 0)
        {
            herdState = HerdState.Fleeing;
            //We add the avoid steering
            st = fleeBehaviour.GetSteering(predators, transform,
                maxLinearAcceleration);
            fleeVector = st.linearAcceleration;
            herdSteering.Add(st, fleeWeight);
        }
        else
        {
```

```
            herdState = HerdState.Idle;
        }

        //Obstacle avoidance
        herdSteering.Add(obstacleAvoidanceBehaviour.GetSteering(transform,
            velocity, maxLinearAcceleration), obstacleAvoidanceWeight);

        //Crop down to the maximums
        herdSteering.Crop(maxLinearAcceleration,
            maxAngularAcceleration);

        return herdSteering;
    }

    //Does the calculations for the position and rotation update
    private void UpdatePositionAndRotation(Steering steering)
    {
        //Using Newton-Euler-1 integration
        transform.position += velocity * Time.deltaTime;
        Vector3 auxVector = new
            Vector3(Steering.MapToRange(transform.eulerAngles.x),
            Steering.MapToRange(transform.eulerAngles.y) +
            (rotationSpeed * Time.deltaTime),
            Steering.MapToRange(transform.eulerAngles.z));
        transform.rotation = Quaternion.Euler(auxVector);

        //Update velocity and rotation
        velocity += steering.linearAcceleration * Time.deltaTime;
        if (velocity.magnitude > currentMaxSpeed) //Max Speed control
        {
            velocity = velocity.normalized * currentMaxSpeed;
                //Normalize and set to max
        }

        rotationSpeed += steering.angularAcceleration * Time.deltaTime;
            //Max rotation control
        if (rotationSpeed > maxRotation)
        {
            rotationSpeed /= Mathf.Abs(rotationSpeed); //Get sign
            rotationSpeed *= maxRotation;    //Set to max rotation
        }
    }

    //Events
    void OnTriggerEnter(Collider other)
    {
        if (other.transform.parent.gameObject !=
            this.gameObject)//Check it's not our own collider
```

```
        {
            if (other.tag == "Zebra")
            {
                herd.Add(other.transform.parent.gameObject);
                if (debugHerd)
                {
                    Debug.Log("New entry: " + other.transform.parent.name
                        + ", number of elements: " + herd.Count);
                }
            }
            else if (other.tag == "Lion")
            {
                predators.Add(other.transform.parent.gameObject);
                if (debugHerd)
                {
                    Debug.Log("New predator: " +
                        other.transform.parent.name + ", number of
                        predators: " + predators.Count);
                }
            }
        }
    }
    void OnTriggerExit(Collider other)
    {
        if (other.tag == "Zebra")
        {
            herd.Remove(other.transform.parent.gameObject);
            if (debugHerd)
            {
                Debug.Log("Element exitted, number of elements: " +
                    herd.Count);
            }
        }
        else if (other.tag == "Lion")
        {
            predators.Remove(other.transform.parent.gameObject);
            if (debugHerd)
            {
                Debug.Log("Predator exitted, number of predators: " +
                    predators.Count);
            }
        }
    }

    //Getters
    public Vector3 GetVelocity()
    {
        return velocity;
```

```
    }

    //Utilities
    //Calculates the speed modifier according to age
    private void SetAgeModifier()
    {
        if (age < 5f)//1 to 5 years, young.
        {
            ageModifier = ((5f - age) / 5f) * MAX_AGE_MOD;
        }
        else if (age > 20f)//20 o 25 years, old.
        {
            ageModifier = ((age - 20f) / 5f) * MAX_AGE_MOD;
        }

        //It has to be negative
        ageModifier *= -1f;
    }
    //Calculates the current max speed, taking into account age
        modifier and stamina
    private void UpdateMaxSpeed()
    {
        switch (herdState)
        {
            case HerdState.Idle:
                currentMaxSpeed = Mathf.Min(ABS_MAX_SPEED / 5f,
                    ABS_MAX_SPEED + ageModifier + staminaModifier);
                break;
            case HerdState.Fleeing:
                currentMaxSpeed = ABS_MAX_SPEED + ageModifier +
                    staminaModifier;
                break;
        }
    }
    //Updates our current stamina
    private void UpdateStamina()
    {
        //Depending at which speed we're running, we may be losing or
            gaining stamina.
        //We take (ABS_MAX_SPEED + ageModifier) as an individual
            asbolute max speed for reference
        //We lose stamina at a 1/s rate, and gain it at the same.
        effortFactor = velocity.magnitude/(ABS_MAX_SPEED + ageModifier);
        if (effortFactor <= restThreshold)//We're not using any effort,
            we gain stamina
        {
            currentStamina += Time.deltaTime;
        }
```

```
        else if (effortFactor >= effortThreshold) //We're putting
            effort, we lose stamina
        {
            currentStamina -= Time.deltaTime;
        }

        //Control stamina doesn't go above max or below min
        if (currentStamina > MAX_STAMINA)
            currentStamina = MAX_STAMINA;
        if (currentStamina < 0f)
            currentStamina = 0f;

        //Calculate stamina modifier
        if (currentStamina < exhaustedThreshold * MAX_STAMINA)//We are
            getting tired (below exhaustedThreshold of our total
            stamina)
        {
            staminaModifier = MAX_STAMINA_MOD * (currentStamina /
                (MAX_STAMINA*exhaustedThreshold));
            staminaModifier *= -1f;//It's a malus, need to make it
                negative
        }
        else//We still have stamina left
        {
            staminaModifier = 0f;
        }
    }
    //This animal dies, abandoning the herd.
    public void Killed()
    {
        foreach (GameObject go in herd)
        {
            go.GetComponent<HerdingBehaviour>().NotifyDeath(this.gameObject);
        }
        this.gameObject.SetActive(false);
    }
    //Removes the dead member from the herd list
    public void NotifyDeath(GameObject deadMember)
    {
        herd.Remove(deadMember);
    }
}
```

Listing A.12: Pack Behaviour C# script

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
/*
```

```
 * PackBehaviour combines different steering behaviours to simulate a pack mentality
 * @Author: Daniel Collado
 */
[RequireComponent(typeof(Collider))]
public class PackBehaviour : MonoBehaviour {

    //Pack
    private List<GameObject> pack;      //List with all the pack members within our range
    //Herd
    private List<GameObject> herd;      //List with all the preys within our range

    //States in which the predator can be
    public enum PackState
    {
        Wandering,
        PreparingForAttack,
        Attacking,
        Eating,
        Retreating
    }

    private PackState packState = PackState.Wandering; //State of the pack

    //Steering
    private Steering packSteering;      //Data structure containing steering information
    private Vector3 velocity = Vector3.zero; //Linear speed
    private float rotationSpeed = 0f;   //Rotation speed
    private float currentMaxSpeed = 7f; //Maximum linear speed
    private float maxRotation = 45f;    //Maximum rotation speed
    private float maxLinearAcceleration = 5f; //Maximum linear acceleration
    private float maxAngularAcceleration = 45f; //Maximum angular acceleration

    //Constants
    private const float ABS_MAX_SPEED = 7f; //Absolute maximum linear speed
    private const float MAX_STAMINA = 40f; //Time at which we can hold maximum speed, then we tire down.
    private const float MAX_STAMINA_MOD = 3f; //Maximum value for the speed modifier according to stamina

    //Pack behaviour
```

```
private Vector3 preparationPosition = Vector3.zero; //Position we
    need to be fore the attack starts
private GameObject targetPrey = null;     //Prey we are currently
    chasing
private float catchThreshold = 5f;        //Distance at which we
    can kill a prey
private float powerLevel = 0f;            //Variable that defines
    the overall power of this predator
private int rank = 0;                     //Variable that defines
    the rank of this predator in the pack hierarchy
private GameObject alphaMale = null;      //Variable containing
    the alpha male of the pack
private bool hierarchySet = false;        //Variable determining
    whether or not a hierarchy has been set yet
private Vector3 herdCenter = Vector3.zero; //Variable containing
    the center of gravity of the herd
private bool preyCatched = false;         //Variable determining
    if any of the pack members has gotten a kill

//Individual traits
private float currentStamina = 0f; //Stamina that we currently
    have left.
private float effortFactor = 0f;  //Variable [0..1] indicating
    how much effort are we using according to our speed
private float restThreshold = 0.3f; //Point below which our effort
    lets us gain stamina
private float effortThreshold = 0.6f; //Point above which our
    effort makes us use stamina
private float staminaModifier = 0f; //Malus that modifies maximum
    speed depending on our tiredness
private float exhaustedThreshold = 0.25f; //Percentage of our
    stamina at which we become tired and gain a malus in speed

//Behaviours
//Linear
private VelocityMatchingBehaviour velocityMatchingBehaviour;
    //Behaviour for velocity matching
private float velocityMatchingWeight = 0.2f;    //Weight for
    velocity matching

private SeparationBehaviour separationBehaviour; //Behaviour for
    separation
private float separationWeight = 0.7f;          //Weight for
    separation

private CohesionBehaviour cohesionBehaviour;    //Behaviour for
    cohesion
private float cohesionWeight = 0.1f;            //Weight for
```

```
cohesion

private SeekBehaviour seekBehaviour;            //Behaviour for seek
private float seekWeight = 1.0f;                //Weight for seek

private FleeBehaviour fleeBehaviour;            //Behaviour for seek
private float fleeWeight = 1.0f;                //Weight for seek

private ObstacleAvoidanceBehaviour obstacleAvoidanceBehaviour; //Behaviour for obstacle avoidance
private float obstacleAvoidanceWeight = 2.0f;   //Weight for obstacle avoidance

//Angular
private AlignBehaviour alignBehaviour;          //Behaviour for align
private float alignWeight = 0.25f;              //Weight for align

private FaceBehaviour faceBehaviour;            //Behaviour for face
private float faceWeight = 0.75f;               //Weight for face

//Combined
private WanderBehaviour wanderBehaviour;        //Behaviour for wander
private float wanderWeight = 1.0f;              //Weight for wander

//Debug
public bool debugPack = false;      //Flag for pack debugging
public bool debugStates = false;    //Flag for pack state debugging
public bool debugPreparation = false; //Flag for preparation debugging
public bool debugAttack = false;    //Flag for attack debugging
private Vector3 seekVector = Vector3.zero; //Debug vector for seek
private Vector3 fleeVector = Vector3.zero; //Debug vector for flee
private float vectorDebugFactor = 10f; //Factor for scaling debug vectors

//Initializations
void Awake()
{
   InitializeVariables();
   RandomInitialSteering();
}

//Setting up varibales
```

```
    private void InitializeVariables()
    {
        packSteering = new Steering();
        pack = new List<GameObject>();
        herd = new List<GameObject>();

        //Behaviours
        wanderBehaviour = new WanderBehaviour();
        velocityMatchingBehaviour = new VelocityMatchingBehaviour();
        alignBehaviour = new AlignBehaviour();
        separationBehaviour = new SeparationBehaviour();
        cohesionBehaviour = new CohesionBehaviour();
        faceBehaviour = new FaceBehaviour();
        wanderBehaviour = new WanderBehaviour();
        seekBehaviour = new SeekBehaviour();
        fleeBehaviour = new FleeBehaviour();
        obstacleAvoidanceBehaviour = new ObstacleAvoidanceBehaviour();

        //Give a random power level
        powerLevel = Random.Range(0f, 100f);
        //Stamina
        currentStamina = MAX_STAMINA;

    }

    //Give a random initial acceleration
    private void RandomInitialSteering()
    {
        //Give a random initial acceleration
        velocity = new Vector3(Random.value, 0f, Random.value);
        velocity.Normalize();
        velocity *= Random.Range(0.1f, currentMaxSpeed / 4);

        packSteering.linearAcceleration = new Vector3(Random.value, 0f,
            Random.value);
        packSteering.linearAcceleration.Normalize();
        packSteering.linearAcceleration *= Random.Range(0.1f,
            maxLinearAcceleration);
    }

    //Update
    void Update()
    {
        DrawDebug();
    }

    //Debug function
    private void DrawDebug()
```

```
{
   if (debugPreparation && (packState==
       PackState.PreparingForAttack))
   {
      Debug.DrawLine(transform.position, preparationPosition,
          Color.white);
      Debug.DrawLine(transform.position, transform.position +
          seekVector * vectorDebugFactor, Color.green);
      Debug.DrawLine(transform.position, transform.position +
          fleeVector * vectorDebugFactor, Color.red);
   }
   if (debugAttack && (packState == PackState.Attacking))
   {
      Debug.DrawLine(transform.position, transform.position +
          seekVector * vectorDebugFactor, Color.green);
      Debug.DrawLine(transform.position,
          targetPrey.transform.position, Color.white);
   }
}

//Physics update
void FixedUpdate()
{
   UpdateStamina();
   UpdateMaxSpeed();
   UpdatePositionAndRotation(GetSteering());
}

//Steering method
public Steering GetSteering()
{
   packSteering.Reset();

   if (herd.Count == 0)//No preys
   {
      packState = PackState.Wandering;
      if (pack.Count > 0)//Wander with pack
      {
         SteerForWanderInPack();
      }
      else//Wander alone
      {
         SteerForWanderAlone();
      }
   }
   else//Preys on sight
   {
      if (packState == PackState.Wandering)
```

```
            packState = PackState.PreparingForAttack;
         if (pack.Count > 0)//Hunt in pack
         {
            switch (packState)
            {
               case PackState.PreparingForAttack:
                  if (!hierarchySet)
                     SetupHierarchy();

                  if (!AllInPositionForAttack())
                  {
                     SteerForPreparation();
                  }
                  else
                  {
                     SteerForAttacking();
                     packState = PackState.Attacking;
                  }
                  break;
               case PackState.Attacking:
                  if (!preyCatched)
                  {
                     if (CatchedPrey())
                     {
                        KillPrey();
                        NotifyCatch();
                        SteerForEating();
                        packState = PackState.Eating;
                     }
                     else
                     {
                        if (Tired())
                        {
                           packState = PackState.Retreating;
                        }
                        else
                        {
                           SteerForAttacking();
                        }

                     }
                  }
                  else
                  {
                     SteerForEating();
                  }
                  break;
               case PackState.Eating:
```

```
                SteerForEating();
                break;

            case PackState.Retreating:
                SteerForWanderInPack();
                break;
        }
    }
    else//Hunt alone
    {
        switch (packState)
        {
            case PackState.PreparingForAttack:
                if (!InPositionForAttack())
                {
                    SteerForPreparationAlone();
                }
                else
                {
                    SteerForAttackingAlone();
                    packState = PackState.Attacking;
                }
                break;
            case PackState.Attacking:
                if (CatchedPrey())
                {
                    KillPrey();
                    SteerForEating();
                    packState = PackState.Eating;
                }
                else
                {
                    if (Tired())
                    {
                        packState = PackState.Retreating;
                    }
                    else
                    {
                        SteerForAttackingAlone();
                    }
                }
                break;
            case PackState.Eating:
                SteerForEating();
                break;

            case PackState.Retreating:
                SteerForWanderAlone();
```

```
                    break;
                }
            }
        }

        //No matter in which state, we always want to have the obstacle avoidance behaviour
        //Obstacle avoidance
        packSteering.Add(obstacleAvoidanceBehaviour.GetSteering(transform, velocity, maxLinearAcceleration), obstacleAvoidanceWeight);

        //Crop down to the maximums
        packSteering.Crop(maxLinearAcceleration, maxAngularAcceleration);

        return packSteering;
    }

    //Steering calls
    private void SteerForWanderInPack()
    {
        //Linear
        //Velocity Matching
        packSteering.Add(velocityMatchingBehaviour.GetSteering(pack, velocity, maxLinearAcceleration), velocityMatchingWeight);
        //Separation
        packSteering.Add(separationBehaviour.GetSteering(pack, transform, maxLinearAcceleration), separationWeight);
        //Cohesion
        packSteering.Add(cohesionBehaviour.GetSteering(pack, transform, maxLinearAcceleration), cohesionWeight);

        //Angular
        //Align
        packSteering.Add(alignBehaviour.GetSteering(pack, transform, rotationSpeed, maxRotation, maxAngularAcceleration), alignWeight);
        //Face
        packSteering.Add(faceBehaviour.GetSteering(transform, velocity, rotationSpeed, maxRotation, maxAngularAcceleration), faceWeight);
    }
    private void SteerForWanderAlone()
    {
        //Wander (Linear + Angular)
        packSteering.Add(wanderBehaviour.GetSteering(transform, maxRotation, rotationSpeed, maxLinearAcceleration, maxAngularAcceleration), wanderWeight);
```

```
    }
    private void SteerForPreparationAlone()
    {
        Steering st;
        //Linear
        //Seek (preparation position)
        st = seekBehaviour.GetSteering(preparationPosition, transform,
            maxLinearAcceleration);
        seekVector = st.linearAcceleration;
        packSteering.Add(st, seekWeight);
        //Flee (herd center)
        st = fleeBehaviour.GetSteering(herdCenter, transform,
            maxLinearAcceleration);
        fleeVector = st.linearAcceleration;
        packSteering.Add(st, fleeWeight);

        //Angular
        packSteering.Add(faceBehaviour.GetSteering(transform.position +
            velocity, transform, rotationSpeed, maxRotation,
            maxAngularAcceleration), 1.0f);
    }
    private void SteerForAttackingAlone()
    {
        //Linear
        targetPrey = GetClosestTarget();
        //Seek
        Steering st =
            seekBehaviour.GetSteering(targetPrey.transform.position,
            transform, maxLinearAcceleration);
        seekVector = st.linearAcceleration;
        packSteering.Add(st, seekWeight);
        //Angular
        packSteering.Add(faceBehaviour.GetSteering(targetPrey.transform.position,
            transform, rotationSpeed, maxRotation,
            maxAngularAcceleration), 1.0f);
    }
    private void SteerForEating()
    {
        //Linear
        //Seek
        Steering st =
            seekBehaviour.GetSteering(targetPrey.transform.position,
            transform, maxLinearAcceleration);
        seekVector = st.linearAcceleration;
        packSteering.Add(st, seekWeight);
        //Angular
        packSteering.Add(faceBehaviour.GetSteering(targetPrey.transform.position,
            transform, rotationSpeed, maxRotation,
```

```
        maxAngularAcceleration), 1.0f);
    }
    private void SteerForPreparation()
    {
        Steering st;
        //Linear
        //Seek (preparation position)
        st = seekBehaviour.GetSteering(preparationPosition, transform, maxLinearAcceleration);
        seekVector = st.linearAcceleration;
        packSteering.Add(st, seekWeight);
        //Flee (herd center)
        st = fleeBehaviour.GetSteering(herdCenter, transform, maxLinearAcceleration);
        fleeVector = st.linearAcceleration;
        packSteering.Add(st, fleeWeight + 40.0f);
        //Separation
        packSteering.Add(separationBehaviour.GetSteering(pack, transform, maxLinearAcceleration), separationWeight);

        //Angular
        packSteering.Add(faceBehaviour.GetSteering(transform.position + velocity, transform, rotationSpeed, maxRotation, maxAngularAcceleration), 1.0f);
    }
    private void SteerForAttacking()
    {
        //Linear
        targetPrey = GetClosestTarget();
        //Seek
        Steering st = seekBehaviour.GetSteering(targetPrey.transform.position, transform, maxLinearAcceleration);
        seekVector = st.linearAcceleration;
        packSteering.Add(st, seekWeight);
        //Angular
        packSteering.Add(faceBehaviour.GetSteering(targetPrey.transform.position, transform, rotationSpeed, maxRotation, maxAngularAcceleration), 1.0f);
    }

    //Utilities
    //Returns true if we are close enough to the preparation point, false otherwise
    //Also updates preparation position
    public bool InPositionForAttack()
    {
        float positionThreshold = 20f;
```

```
        preparationPosition = CalculateAttackPosition();
        if ((preparationPosition - transform.position).magnitude <
            positionThreshold)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    //Calculates the preparation position
    private Vector3 CalculateAttackPosition()
    {
        //First, we calculate the herd center
        herdCenter = Vector3.zero;
        foreach (GameObject go in herd)
        {
            herdCenter += go.transform.position;
        }
        herdCenter /= herd.Count;

        //Then we calculate the perimeter (radius) by calculating the
            average distance of the zebras to the center of the herd,
            and multiplying by two
        float radius = 0f;
        foreach (GameObject go in herd)
        {
            radius += (go.transform.position - herdCenter).magnitude;
        }
        radius /= herd.Count;

        Vector3 direction = Vector3.zero;
        float distanceFactor = 8f;
        //Now, depending on our rank, we will calculate our attack
            position
        if (rank == 0)//Alpha male, attack from the front to move the
            preys towards the rest of the pack
        {
            //We proceed to point the closest position that is x times
                as far as the perimeter from the center of the herd.
            direction = transform.position - herdCenter;
            direction.Normalize();
            direction *= radius * distanceFactor;
        }
        else//Lower ranked predators, they will be opposite of the
            alpha male, receiving the scared preys
        {
```

```
            if (alphaMale != null) //wait until an alpha male is crowned
            {
               //We calculate the opposite position of the alpha male
               direction = herdCenter - alphaMale.transform.position;
               direction.Normalize();
               direction *= radius * distanceFactor;
            }
         }

         //We will attack from there, return the value
         return herdCenter + direction;
      }
      //Returns the closest prey
      private GameObject GetClosestTarget()
      {
         float auxF = Mathf.Infinity;
         GameObject auxGO = null;
         foreach (GameObject go in herd)
         {
            float distance = (go.transform.position -
                transform.position).magnitude;
            if ( distance < auxF)
            {
               auxGO = go;
               auxF = distance;
            }
         }
         return auxGO;
      }
      //Returns true if we catched our prey, false otherwise
      private bool CatchedPrey()
      {
         if ((targetPrey.transform.position -
             transform.position).magnitude < catchThreshold)
         {
            preyCatched = true;
            Debug.Log("CATCH");
            return true;
         }
         else
         {
            return false;
         }
      }
      //Disables the prey's game object, it died.
      private void KillPrey()
      {
         targetPrey.GetComponent<HerdingBehaviour>().Killed();
```

```
}
//Ranks the members of the pack from strongest to weakest,
   assigning roles
private void SetupHierarchy()
{
   //NOTE: The "alpha male" will be rank 0, and the bigger the
      rank number, the lower this predator is in the pack
      hierarchy
   rank = 0;
   //See what is our rank according to the power level of each
      member of the pack
   foreach (GameObject go in pack)
   {
      if (go.GetComponent<PackBehaviour>().GetPower() > powerLevel)
      {
         rank++;
      }
   }

   //If we are the alpha male, we notify the others (he will serve
      as a reference for the positioning)
   if (rank == 0)
   {
      foreach (GameObject go in pack)
      {
         alphaMale = this.gameObject;
         go.GetComponent<PackBehaviour>().SetAlphaMale(this.gameObject);
      }
   }

   hierarchySet = true;
}
//Checks if all the members of the pack are in position for
   attacking, returning false otherwise
private bool AllInPositionForAttack()
{
   bool ready = InPositionForAttack();
   foreach (GameObject go in pack)
   {
      ready = ready &&
          go.GetComponent<PackBehaviour>().InPositionForAttack();
   }
   return ready;
}
//Notify the rest of the pack that a prey has been catched and
   it's time to eat
private void NotifyCatch()
{
```

```
      foreach (GameObject go in pack)
      {
         PackBehaviour ps = go.GetComponent<PackBehaviour>();
         ps.ChangeState(PackState.Eating);
         ps.PreyCatched(targetPrey);
      }
   }
   //Recieves the notification for catched preys
   public void PreyCatched(GameObject catchedPrey)
   {
      preyCatched = true;
      targetPrey = catchedPrey;
   }
   //Returns wether or not the lion is tired
   private bool Tired()
   {
      return (currentStamina < exhaustedThreshold * MAX_STAMINA);
   }

   //Does the calculations for the position and rotation update
   private void UpdatePositionAndRotation(Steering steering)
   {
      //Using Newton-Euler-1 integration
      transform.position += velocity * Time.deltaTime;
      Vector3 auxVector = new
          Vector3(Steering.MapToRange(transform.eulerAngles.x),
          Steering.MapToRange(transform.eulerAngles.y) +
          (rotationSpeed * Time.deltaTime),
          Steering.MapToRange(transform.eulerAngles.z));
      transform.rotation = Quaternion.Euler(auxVector);

      //Update velocity and rotation
      velocity += steering.linearAcceleration * Time.deltaTime;
      if (velocity.magnitude > currentMaxSpeed) //Max Speed control
      {
         velocity = velocity.normalized * currentMaxSpeed;
             //Normalize and set to max
      }

      rotationSpeed += steering.angularAcceleration * Time.deltaTime;
          //Max rotation control
      if (rotationSpeed > maxRotation)
      {
         rotationSpeed /= Mathf.Abs(rotationSpeed); //Get sign
         rotationSpeed *= maxRotation;    //Set to max rotation
      }
   }
```

```
    //Events
    void OnTriggerEnter(Collider other)
    {
       if (other.transform.parent.gameObject !=
          this.gameObject)//Check it's not our own collider
       {
          if (other.tag == "Zebra")
          {
             herd.Add(other.transform.parent.gameObject);
             if (debugPack)
             {
                Debug.Log("New herd entry: " +
                   other.transform.parent.name + ", herd size: " +
                   herd.Count);
             }
          }
          else if (other.tag == "Lion")
          {
             pack.Add(other.transform.parent.gameObject);
             if (debugPack)
             {
                Debug.Log("New predator: " +
                   other.transform.parent.name + ", number of
                   predators: " + pack.Count);
             }
          }
       }
    }
    void OnTriggerExit(Collider other)
    {
       if (other.tag == "Zebra")
       {
          herd.Remove(other.transform.parent.gameObject);
          if (debugPack)
          {
             Debug.Log("Herd member exitted, herd size: " +
                herd.Count);
          }
       }
       else if (other.tag == "Lion")
       {
          pack.Remove(other.transform.parent.gameObject);
          if (debugPack)
          {
             Debug.Log("Predator exitted, number of predators: " +
                pack.Count);
          }
       }
```

```
    }

    //Getters
    public Vector3 GetVelocity()
    {
        return velocity;
    }
    public float GetPower()
    {
        return powerLevel;
    }

    //Setters
    public void SetAlphaMale(GameObject am)
    {
        alphaMale = am;
    }

    //Method that changes states
    public void ChangeState(PackState newState)
    {
        packState = newState;
    }

    //Updates our current stamina
    private void UpdateStamina()
    {
        //When attacking, depending at which speed we're running, we
            may be losing or gaining stamina.
        //We lose stamina at a 1/s rate, and gain it at the same.
        effortFactor = velocity.magnitude / ABS_MAX_SPEED;
        if (effortFactor <= restThreshold)//We're not using any effort,
            we gain stamina
        {
            currentStamina += Time.deltaTime;
        }
        else if (effortFactor >= effortThreshold) //We're putting
            effort, we lose stamina
        {
            if(packState == PackState.Attacking)//Only if attacking
                currentStamina -= Time.deltaTime;
        }

        //Control stamina doesn't go above max or below min
        if (currentStamina > MAX_STAMINA)
            currentStamina = MAX_STAMINA;
        if (currentStamina < 0f)
            currentStamina = 0f;
```

```

        //Calculate stamina modifier
        if (currentStamina < exhaustedThreshold * MAX_STAMINA)//We are getting tired (below exhaustedThreshold of our total stamina)
        {
            staminaModifier = MAX_STAMINA_MOD * (currentStamina / (MAX_STAMINA * exhaustedThreshold));
            staminaModifier *= -1f;//It's a malus, need to make it negative
        }
        else//We still have stamina left
        {
            staminaModifier = 0f;
        }
    }

    //Calculates the current max speed, taking into account stamina modifier
    private void UpdateMaxSpeed()
    {
        switch (packState)
        {
            case PackState.Wandering:
                SetupWandering();
                break;
            case PackState.PreparingForAttack:
                SetupPreparing();
                break;
            case PackState.Attacking:
                SetupAttacking();
                break;
            case PackState.Eating:
                SetupEating();
                break;
        }
    }

    //Settings for each state
    private void SetupWandering()
    {
        currentMaxSpeed = Mathf.Min(ABS_MAX_SPEED / 3f, ABS_MAX_SPEED + staminaModifier);
        rotationSpeed = 5f;
    }
    private void SetupPreparing()
    {
        //Set up variables for preparing to attack
```

```
        if (rank == 0)
        {
            currentMaxSpeed = Mathf.Min(ABS_MAX_SPEED / 2f,
                ABS_MAX_SPEED + staminaModifier);
        }
        else
        {
            currentMaxSpeed = ABS_MAX_SPEED + staminaModifier;
        }

        maxRotation = 45f;
    }
    private void SetupAttacking()
    {
        //Set up variables for attacking
        currentMaxSpeed = ABS_MAX_SPEED + staminaModifier;
        maxRotation = 45f;
    }
    private void SetupEating()
    {
        //Set up variables for eating
        currentMaxSpeed = Mathf.Min(ABS_MAX_SPEED / 4f, ABS_MAX_SPEED +
            staminaModifier);
        maxRotation = 5f;
    }

}
```

Listing A.13: Loner Behaviour C# script

```
using UnityEngine;
using System.Collections;
/*
 * LonerBehaviour consists on a wander behaviour simulating a lone
     animal wandering about
 * @Author: Daniel Collado
 */
public class LonerBehaviour : MonoBehaviour {

    //Steering
    private Steering lonerSteering;     //Data structure containing
        steering information
    private Vector3 velocity = Vector3.zero; //Linear speed
    private float rotationSpeed = 0f;   //Rotation speed
    private float maxSpeed = 0.5f;      //Maximum linear speed
    private float maxRotation = 10f;    //Maximum rotation speed
    private float maxLinearAcceleration = 5f; //Maximum linear
        acceleration
    private float maxAngularAcceleration = 45f; //Maximum angular
```

```
acceleration

    //Behaviours
    private WanderBehaviour wanderBehaviour;        //Behaviour for wander
    private float wanderWeight = 1.0f;              //Weight for wander

    private FaceBehaviour faceBehaviour;            //Behaviour for face
    private float faceWeight = 1.0f;               //Weight for face

    private ObstacleAvoidanceBehaviour obstacleAvoidanceBehaviour;    //Behaviour for obstacle avoidance
    private float obstacleAvoidanceWeight = 2.0f;   //Weight for obstacle avoidance

    //Initialization
    void Awake()
    {
        InitializeVariables();
        RandomInitialSteering();
    }

    //Setting up varibales
    private void InitializeVariables()
    {
        lonerSteering = new Steering();

        wanderBehaviour = new WanderBehaviour();
        faceBehaviour = new FaceBehaviour();
        obstacleAvoidanceBehaviour = new ObstacleAvoidanceBehaviour();
    }

    //Give a random initial acceleration
    private void RandomInitialSteering()

    {
        velocity = new Vector3(Random.value, 0f, Random.value);
        velocity.Normalize();
        velocity *= Random.Range(0.1f, maxSpeed / 4);

        lonerSteering.linearAcceleration = new Vector3(Random.value, 0f, Random.value);
        lonerSteering.linearAcceleration.Normalize();
        lonerSteering.linearAcceleration *= Random.Range(0.1f, maxLinearAcceleration);
    }
    //Physics update
```

```
void FixedUpdate()
{
   UpdatePositionAndRotation(GetSteering());
}

//Steering method blending
public Steering GetSteering()
{
   lonerSteering.Reset();

   //Wander (Linear + Angular)
   lonerSteering.Add(wanderBehaviour.GetSteering(transform,
      maxRotation, rotationSpeed, maxLinearAcceleration,
      maxAngularAcceleration), wanderWeight);
   //Face (angular)
   lonerSteering.Add(faceBehaviour.GetSteering(transform,
      velocity, rotationSpeed, maxRotation,
      maxAngularAcceleration), faceWeight);
   //Obstacle avoidance
   lonerSteering.Add(obstacleAvoidanceBehaviour.GetSteering(transform,
      velocity, maxLinearAcceleration), obstacleAvoidanceWeight);

   //Crop down to the maximums
   lonerSteering.Crop(maxLinearAcceleration,
      maxAngularAcceleration);

   return lonerSteering;
}

//Does the calculations for the position and rotation update
private void UpdatePositionAndRotation(Steering steering)
{
   //Using Newton-Euler-1 integration
   transform.position += velocity * Time.deltaTime;
   Vector3 auxVector = new
      Vector3(Steering.MapToRange(transform.eulerAngles.x),
      Steering.MapToRange(transform.eulerAngles.y) +
      (rotationSpeed * Time.deltaTime),
      Steering.MapToRange(transform.eulerAngles.z));
   transform.rotation = Quaternion.Euler(auxVector);

   //Update velocity and rotation
   velocity += steering.linearAcceleration * Time.deltaTime;
   if (velocity.magnitude > maxSpeed) //Max Speed control
   {
      velocity = velocity.normalized * maxSpeed; //Normalize and
         set to max
   }
```

```

      rotationSpeed += steering.angularAcceleration * Time.deltaTime; //Max rotation control
      if (rotationSpeed > maxRotation)
      {
         rotationSpeed /= Mathf.Abs(rotationSpeed); //Get sign
         rotationSpeed *= maxRotation;    //Set to max rotation
      }
   }
}
```

Listing A.14: Camera Control C# script

```
using UnityEngine;
using System.Collections;

public class CameraControl : MonoBehaviour {

   private float cameraSpeed = 5f;
   private float zoomStep = 20f;
   private float minZoomDistance = 1f;
   private float maxZoomDistance = 300f;


      // Update is called once per frame
      void Update () {
      ProcessInput();
      }

   private void ProcessInput()
   {
      float hInput = Input.GetAxis("Horizontal");
      float vInput = Input.GetAxis("Vertical");
      float wInput = Input.GetAxis("Mouse ScrollWheel");
      //Translation
      Vector3 translationVector = new Vector3(hInput, 0 , vInput);
      translationVector *= cameraSpeed;
      transform.Translate(translationVector);
      //Zoom
      if (wInput > 0f)
      {
         if(transform.position.y >= minZoomDistance)
            transform.position -= transform.up * zoomStep;
      }
      else if (wInput < 0f)
      {
         if(transform.position.y <= maxZoomDistance)
            transform.position += transform.up * zoomStep;
      }
```

```
    }
}
```