*University of Derby*

*College of Engineering and Technology School of Electronics, Computing and Mathematics*

# 6CC529 - Game Behaviour Report

*Report for Game Behaviour - 6CC529*

100416392

January 2020

# Contents

# Chapter 1 Introduction

Videogames have come a long way, from simple 2D games, made with simplistic mechanics, yet astonishing considering the limitations that developers faced at the time, to breath taking games with challenging Artificial Intelligent (**AI**) and realistic physics, or at least defying the laws of physics in an accurate and believable way.

Videogames like Armed Assault III (ArmA III) [1] represent the power of modern physics engines, powered by NVIDIA's PhysX [2] engine, ArmA is capable of simulating realistic projectiles physics with lower performance cost. PhysX is implemented within popular game engines, like Unreal Engine or Unity 3D Engine, and have been the physics engine of choice for highly successful games such as The Witcher 3 or Batman Arkham Knight.

For the past four months I have been developing a prototype in Unity, a relevant game engine widely used in the game industry, with its graphics and physics capabilities I have been able to simulate realistic projectiles trajectories, taking advantage of its collision system I extended it to create an accurate locomotion system for the player and enemies. And with their in-build AI API I have created challenging zombies capable of adapting to the environment, react to emergent behaviour and coordinate with others to bring down the player.

This report is structured in two chapters, one for Physics and one for AI, Physics including sub-sections where Projectiles, Forces, and other physics subjects are explored in depth, and topics like Pathfinding, Behaviour trees, Adaptive behaviour and others are thoroughly reviewed in the AI section. The last chapter includes a detailed conclusion.

# Chapter 2 Physics

In this chapter an extended discussion about Physics implemented in my prototype. A brief look into the different collisions types and why I have chosen so for my prototype, evaluating the performance and fidelity of them. The ballistics systems implemented in the prototype, a breakdown of the formulas used and how they work in the engine, based in the ArmA III ballistics motion. Not explored in detail but assuming a Locomotion system exists, unlike Unreal Engine, Unity does not come with an in-build third person character controller, hence I had to consider my approach to it. Constraint and forces, discussing how the prototypes uses the Unity's in-build API, and how it works under the hood. In addition to other physics subjects briefly being discussed in this chapter.

## 2.1 Newton's Laws of Motion

First is first, we cannot talk about game physics not knowing the laws of Motions defined by one of the brilliants minds of all time, Sir Isaac Newton (1642-1727), an English mathematician and physicist that in his famous book *Philosophiæ Naturalis Principia Mathematica* [3] developed the three laws of motions that laid the foundations of classic mechanics [4] [5] [6]:

- Law I, Inertia: A resting body or moving in a straight line remains in a constant velocity, unless an external force acts on it.
- Law II, Force, Mass and Acceleration: When an external force acts upon a body, it proportionally accelerates in accordance to the force applied to the body and it accelerates in the same direction of the force. This is mathematically represented as **F = ma** where **F** is the resultant force in Newtons (*N*) equal to 1 kg-m/s$^2$, **m** is the mass of the body in kg and **a** is the acceleration in m/s$^2$.

- Law III, Equal and Opposite Forces: For every force exerted on a body (action) there is an equal force in the opposite direction (reaction).

These laws of motion are the principles for the physics that we are going to describe in this report and key to understand some of the behaviour of objects that are later explored in this paper.

## 2.2 Collisions

Without collisions there would not be games, collisions are one of the most important components of physics engines, it allows the player to interact with the environment, it allows objects to do so as well. Without collisions the ballistic systems that I implemented in my prototype would make no sense as bullets need to hit something and interact with such hit object. As programmers we need to think what happens when two objects collide, do they bounce, go through the object or suddenly stop. Two distinctions [5] need to be made when talking about collisions, detection and response. **Collision detection** determines whether two or more objects have geometrically collided or not. **Collision response** is the subsequent kinetic motion after two or more objects have collided.

There are multiple ways to do Collision detection, these also depend on whether the object is convex or concave. Firstly, this paper discusses the different algorithms to detect convex objects.

**Bounding volumes** used to improve efficiency [5], it represents the space occupied by an object in world-space. Instead of using the original mesh of the object, a more simplistic shape is formed around the object, boosting performance when calculating

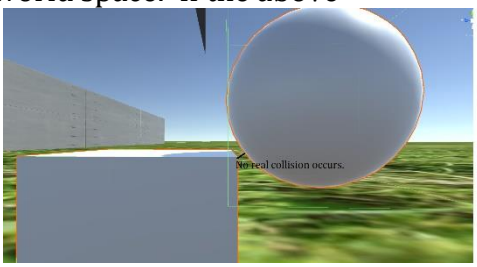
the object intersecting with others. Although performance can be jeopardized if non-convex objects are used, thus why we later explore better algorithms to represent these objects.

Bounding volumes types:

- Axis Aligned Bounding Box (**AABB**) [7] [8]: it is one of the best approaches for bounding volumes, especially good for boxing objects that do not require a change in orientation, as doing so increases the performance cost as the bounding box needs to be recalculated based on the new orientation. After using a bounding box around objects, collisions can be determinate as follows - in a 3D world space [9]:

  $$P(x,y,z) >= B_{min}(x,y,z) \;\&\&\; P(x,y,z) <= B_{max}(x,y,z).$$

  Being P the point in world space, and B the bounding box, composed by the bounding box minimum point and maximum point in world space. If the above condition returns true then the bounding box has collided with such point, none collision occurs otherwise. The same condition can be modified to detect whether two Bounding boxes have intersected by substituting P with $B_2$.

  
  
  *Figure 1. AABB representing a Sphere*

  However Bounding boxes can sometimes give us none wanted effects, like a ball colliding with a box even though they are not, this is due to ball's spherical nature and bounding boxes replacing their collision shape to a box.

  This problem is solved by adding a sphere to enclose the object instead of using a box, this sphere can still interact with AABB objects. To do so we need to

calculate the closest points from the Sphere centre to the AABB, and if the distance it is less than the sphere radius then it has collided.

- Oriented Bounding Box (**OBB**) [8] [10] [11]: An OBB is like an AABB, but it can be rotated, without recalculating the bounding volume. There are many ways to represent an OBB but the most preferred is having a centre point ($C$) plus an orientation matrix and three half-edge lengths, this method is the cheapest to calculate intersection between two OBBs, nonetheless checking if two OBB are overlapping is quite complex. It uses the **separated axis test**, indicating that two or more OBBs are separated when the projected radiuses are less than the distance from centres, otherwise the OBBs are overlapping. Represented in a mathematical way as:

$$| T * L | > r_a + r_b \quad (1)$$

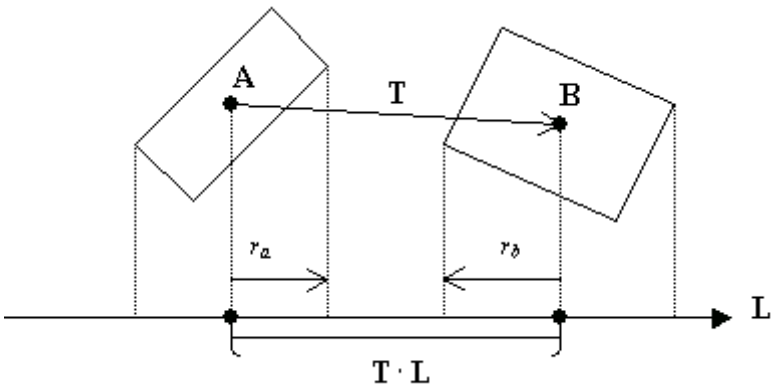



*Figure 2. OBB representation*

Where T is the distance from the centres, L is the axis defined, $r_a$ is the projection radius of A onto L and $r_b$ is the projection radius of B onto L. To detect that OBBs are correctly intersecting, fifteen separated axes tests are needed to check that three axes from A, another three from B and nine perpendicular axes are overlapping.

- **Bounding Sphere** [8] [10]: is simpler than all of the above but not ideal to use with objects that are not spherical, and the calculations could become quite expensive if not used properly. From Bounding Spheres, we can enclose **Capsules**, known as well as sphere-swept lines (**SSL**), these are more attractive for defining the collision mesh of characters or enemies, this is one of the approaches taken in my prototype as it gives better results than bounding boxes, especially with animated characters. To check that two spheres are colliding the distance between them is compared against the sum of radiuses.
- Discrete Orientation Polytopes (**k-DOP**) [8] [10] [12]: Unfortunately for us (programmers) most objects' shapes in games are not simplistic boxes or spheres, but polyhedral and that it is why DOPs help us to enclose objects without completely losing its form. The K is the number of axis-aligned planes, basically a 6-DOP is like an AABB as it is aligned in 6 directions, or an 8-DOP aligned in 8 directions. To check whether two k-DOP intersect we need to test in intervals of k/2, here is the pseudo code for 8-DOP:

```
//Define structure for 8-DOP
struct 8-DOP
{
	int min[4];
	int max[4];
};

//Function that would return true or false if intersecting
bool intersects(...args)
{
	for(int i = 0; i < k/2; i++)
	{
		if(dop1.min[i] > dop2[i].max ||
		   dop1.max[i] < dop2[i].min)
			return false;
		return true;
	}
}
```

This method is similar to testing if AABBs are intersecting, and as AABBs, k-DOPs needs to be recalculated when rotating as axes need to realign.

- **Convex-hull** [8] [10]: is a set of points that after merging all of them it forms the minimal convex polygon around the convex shape. To test that two convex hulls, intersect all vertices need to be within the faces of the other hull. Performance is boosted by ordering the vertices. Nonetheless performance costs are considerably high with convex hulls, thus other bounding volumes are preferred if possible.

However not all the objects in a game are convex, some others are concave, making it harder to calculate collisions for them. In a concave shape, two points connected by a line segment do not fall completely within the shape [13]. Ideally to reduce performance costs we could try to enclose the shape with a convex-hull, if doing so does not give us weird-looking collisions. Inevitable the object would lose is concavity properties and give us invisible collisions [14], therefore what we could do is splitting the concave shape into smaller convex shapes, then we can use one of the bounding volumes, explained in detail above, to calculate collisions for the split shape solving the problem about invisible collisions.

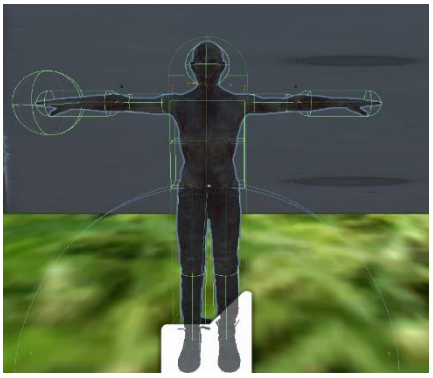


*Figure 3. Enemy split in AABB*

## 2.3 Constraints and Forces

In physics, constraint happens when the motion of an object is limited, forces apply to the object restricting its movement to the motional constraint [15]. Unity has in-build support for Joints [16], two connected objects with constrained movement [17], these has allowed me to create a variety of gameplay mechanics used in my game, one of those being Ragdolls by combining character joints and the other hinge joints, to create wreck-balls to crush zombies. To understand these joints, we need to understand Springs and Damping forces, springs exert equal and opposite forces in two connected objects, according to Hooke's Law the spring returns to its original form after the force has been removed [18], spring forces can be represented mathematically as [5]:

$$F_s = -k_s\,(L - r) \quad (2)$$

$F_s$ is the spring force, $k_s$ is the spring constant, the negative sign indicates that the force is in the opposite direction, L is the length of the string it can be either compressed or stretched, r is the rest of the spring's length [5]. Dampers is the force that, eventually, stops the motion of the spring if no other forces are applied (like swinging) to it [19], it is the drag that acts against the velocity, mathematically represented as [5]:

$$F_d = -k_d(v_1 - v_2) \quad (3)$$

Where $F_d$ is the damping force, proportional to the relative velocity of connected objects ($v_1$ and $v_2$) related to $k_d$, the damping constant. We can then say that joints are a combination of a spring-damper formula. [5]

Now getting into more details about hinge-joints, Unity allows to connect two objects together or an object to a fixed point in space, constraining its movement but no its rotation, thus allowing us to create chains by connecting multiple objects together to a fixed point in space in one end and to another object at the other end, using this technique I was able to create a wrecking-ball to smash enemies. [20]

Ragdolls are a combination of joints, instead of just using classic animations I used a combination of both animations and ragdoll physics. By combining Unity's character joints that acts like a ball and a socket joint [16], I created a ragdoll body, so when an enemy's joint receives a force that exceeds *X* Newtons (variable configurable in the inspector) the ragdoll is triggered (all joints are switch to enable), giving us a more different and natural result, this is specially used to substitute death, giving the player a more satisfactory reward, and crushing zombies.

## 2.4 Ballistics

There are many ways to shoot bullets in games, most of them take the simplistic approach of ray-casting bullets and others simple instantiate a bullet and apply a force to it. Both approaches work great for games, it is satisfactory for the player and does not require to learn the physics behinds bullets, in addition, to give it a bit more of a challenge, games designer add recoil and a bit of bullet drop depending on the distance travelled by the projectile. However, for the sake of this modules I created a realistic ballistic system similar to the physics that bullets face in real world, this realistic model is only implemented in simulation games or military-like games, like ArmA III, where I have got the inspiration for my system, or Squad, in these games

ballistics is a core mechanic, it has a steep learning curve, especially for fire-arms like snipers, anti-tanks or handguns, and has to be dominated in order to achieve the end goal.

Ballistics are broken down in four different stages known as [21]:

- **Internal Ballistics**: the build up motion that happens inside the tube, all the effects that occur within determines the velocity (**M**uzzle **V**elocity) of the bullet when exiting the muzzle.
- **Transitional Ballistics**: a not so exact science that studies what happens after the bullet exits the muzzle and transitions into the exterior, a boost to the MV happens due to escaping gasses, but it immediately decreases due to drag.
- **External Ballistics** [22]: all the external factors that affect the bullet's motion, if it was not for: gravity, angular speed of earth, Coriolis and Eötvös effect, wind, centripetal force, the bullet's spin (magnus effect), air density, etc, the bullet would carry on a straight line until hitting something.
- **Terminal Ballistics**: studies what happens to a projectile after hitting an object and ending its trajectory flight, calculating the total force applied to the object depending on the projectile's mass and velocity on impact, and if it penetrates or fragments it.

In my prototype I focused in external ballistics, for internal ballistics I used empirical data from bullets and a pre-determinate muzzle velocity, and then for terminal ballistics I made sure that bullets would bounce in the right direction if the object could not be penetrated, and if so that it would lose some of its trajectory velocity and force properties.

The main force acting on a projectile during flight is gravity applying a downwards force to the projectile's motion [23]. Followed by drag [24] an opposite force to the

direction of the projectile generated by the projectile's contact with fluids (like air). However, calculating drag every interaction is really expensive, that is why a Coefficient of Drag [25] [26] (CD or ballistic coefficient) using empirical data is used to boost performance. The Coefficient of Drag relates the air drag to the air density, tested by shooting a bullet multiple times to generate a curve of drag, from there what is known as the G model was created, CDs for different bullet shapes, in this project I have used the G1 model and G7 model for calculating the CD of a bullet [27].

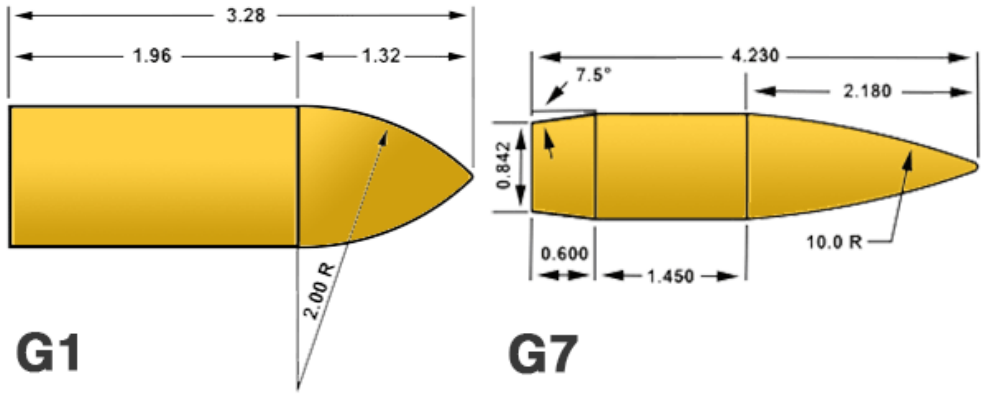



*Figure 4. G1 and G7 models representation*

Both represent the standard shape for NATO bullets, in this prototype the bullet used for the rifles is the standard 5.56×45mm NATO, G1 model.

To model of drag I have used is Dr. Arthur J. Pejsa drag model, described in his book "Modern Practical Ballistics", an analytical way of using the G-model and adding Pejsa's retardation coefficient, with that I found an empirical table which allowed me calculate the projectile's retardation for different velocities [22].

The Coriolis effect [22] [28] [29] is the drift of an object in a rotating sphere, in this case Earth. The Coriolis effect is different depending on your latitude, maximum effect

occurs at the poles and zero at the equator, also the effect varies depending on the direction of the shot. However, the Coriolis effect does not have a great impact for small fire-arms with short trajectories but rather long-flight times trajectories, nonetheless it should be considered for shots that required accuracy, that is why I have implemented it in my prototype. The deflection is based on the angular speed of earth, 7.2921159 × 10−5 rad/s [30] the velocity of the projectile and as indicated before the latitude. The effect takes place on the horizontal axis.

The Eötvös effect [31] [32] is similar to the Coriolis effect but only in the vertical axis, is the change in the perceived gravitational acceleration, due to the centrifugal force generated by Earth. An object moving eastbound experiences and increase in its angular velocity, contrary to an object moving westbound, this is due to eastbound objects moving in the same direction as Earth rotation, thus increasing its speed, objects moving westbound cancel their effective velocity as going against Earth's rotation. In ballistics the centrifugal force generated by the Eötvös effect pushes westbound projectiles to the ground, thus projectiles fired to the east hitting a bit higher than west projectiles hitting lower.

Gyroscopic drift, and Spin drift, [22] [33] [34] [35]for a projectile to keep pointing forward in flight it needs to spin fast enough to maintain itself stable. A bullet always experiences a sideways spin even in good weather conditions, this spin causes the projectile to deflect slightly sideways. A projectile needs to keep itself stable at a gyroscopic stability above 1.0, but without over spinning as the faster the projectile spins the less accurate the shot is. The gyroscopic stability is calculated as [33] [34] [35]:

$$S_g = \frac{8\pi}{p_{air} t^2 d^5 C_{M\alpha}} \frac{A^2}{B} \quad (4)$$

Where $p_{air}$ is the air density, $t^2$ is the barrel twist, $d^5$ is the bullet calibre and $C_{ma}$ the projectiles overturning coefficient which is equal to 0.57 x l, and where $\frac{A^2}{B}$ is equal to $\frac{md^2}{4.83(1+l^2)}$ , where m is the bullet's mass and l is the bullet length.

Last but not least, Wind, it is implemented in my prototype as a simple vector, that can be input by the game designer or later calculated in an environment manager script.

The last part of this Ballistics chapter is Terminal Ballistics [36], what happens to the bullet when it hits an object, depending on the material that it hits, it either bounces, breaks, or penetrates the object.

At low-speed [36], depending on the material strength the projectile pression exerted on the material determines what happens to it, if the pressure is below the material's elastic strength the material goes back into its original shape after removing the pressure, if is greater that the material's elastic strength deformation occurs, if the pressure surpasses the material's yield point it weakens ultimately leading to rupture, to go through an object there needs to be enough room for the projectile, as it goes through.

At high-speed [36] as the projectile goes through the material it needs to push the fallen bits out of its way as doing so it is passing on its kinematic energy to the material, at really high speeds the material is considered a fluid without strength and we now encounter a hydrodynamic problem.

However, the physics in my prototype assume that bullets go at an intermediate speed, combining both low-speed and high-speed cases we get the following formula [36]:

$$X_M = X_C \log(1 + \frac{v}{v_{thr}}^2) \quad (5)$$

Where $X_C$ is the penetration distance being equal to $\frac{m}{A}\frac{1}{C_d\, p_{mat}}$ where m is the bullet mass, A the bullet's area, $C_d$ the drag coefficient and $p_{mat}$ the material density. And $v_{thr}$ is the threshold velocity being equal to $\sqrt{\frac{2Y}{C_d\, p_{mat}}}$, where Y is the yield point. And v is the bullet's velocity. After going through an object, the bullet loses kinetic energy, becoming less lethal.

# Chapter 3 Artificial Intelligence

Since the dawn of games there has been Artificial Intelligence (**AI**), Alan Turing the father of computer science created the, arguably, first ever video-game to prove that a machine could beat a human at chess [37]. Since then AI has continuously evolve to a point where sometimes it cannot be beat or hard to, games like the Souls-saga impose a challenge to the player in its most difficult mode, where strategy must be carefully laid out before attacking an enemy, and where skill and critical thinking are rewarded. In my prototype I have developed a Zombie AI, although a zombie on its own may be an opponent easy to beat, its zombies' herds what can present a challenge to the player, like in Souls-like game blindly attacking herds may result in a Game Over. My Zombie AI can adapt to the environment, communicate with others, hear, smell, and hunt down a player that is not moving carefully.

## 3.1 Pathfinding

We just simply cannot tell the AI to move to the player, this would result in an unrealistic behaviour as they would get stuck in any blockades along the path, instead we need

to be accountant for static and dynamic obstacles. There are many techniques to calculate paths for AI, solutions for calculating short-paths like Dijkstra's algorithm, breadth first search algorithm, etc, works great for smalls games but as they scale a faster solution for longer and complex paths is needed, a solution like A* algorithm, which is what Unity uses internally to calculate paths within a navigation mesh (more on this later) [38] [39].

A* [40] [41], pronounced "A star", is an extension of Dijkstra's algorithm, optimized for a single destination, the idea of this algorithm is having a starting point, an end point, and splitting the game area in a grid, on this a ring with all the allowed directions is kept, if there is a non-traversable wall it gets removed from the possible directions. Once the list is completed, an iteration in all directions, until finding the end destination occurs, each node from the grid gets a score depending on its distance from the starting point and the estimated movement cost to the end point. The ideal path is chosen based on the lowest movement cost from the starting point to the end point.

A nav mesh [39] is a type of data structure containing a collection of convex polygons defining the traversable areas, in combination with the A* algorithm, it gives fantastic results for AI to move around whilst avoiding obstacles. A nav mesh agent is Unity's build-in component that contains the logic for entities to navigate on a nav mesh and avoid obstacles, a destination is set to the agent (A* end point), this destination can be updated every frame, so we can keep track of moving objects, like Players, its steering behaviour allows the AI entities to turn and move at a desirable speed and steer around obstacles without colliding with them. To track moving obstacles, like doors, nav mesh cannot simply be re-baked, technically nav mesh are always pre-baked before running the game, the data is stored in the scenes data files, nonetheless there is techniques to re-bake the nav mesh on runtime, however this tends to be expensive wasting resources and jeopardizing performance,

recomputing the nav mesh for a moving door would sky-rocket fps, that is why Unity has built-in components that flags objects as obstacles, if these are completely obstructing the path they get carved onto the nav mesh, meaning that the closest polygons around it are modified, creating a hole around it, this method changes the nav mesh so it is, if often used, a computationally expensive operation, but it is better than recomputing all of it. A cheaper alternative is that if object are not stationary or not blocking the path, like a barrel, instead of carving it onto the nav mesh, the obstacle is detected by the Agent's local avoidance, and steers around it if possible. That is how we get dynamic pathfinding in Unity, making the Zombies in my prototype capable of hunting down the player regardless of stationary or moving obstacles ahead.

## 3.2 Complex fixed behaviours

Creating good AI is not as simple as making enemies follow the player and avoid obstacles, players need to engage with them and feel like a challenge is presented to them, yet this is not always the norm, games like DOOM demonstrate that players are more than happy to crush demons rather than demons crushing them. Nonetheless in my prototype I choose the approach of creating smart AI capable of bringing down the player if not played carefully, but without punishing the player, I believe that game designers should not confuse challenging with unbearable.

To achieve a realistic Zombie behaviour, I decided to use a Hierarchical State Machine, rather than Behaviour Trees, unfortunately unlike Unreal Engine 4, which has a complex Behaviour Trees built-in, Unity has not built-in AI behaviour components. That meant that I had to code a complex behaviour system from scratch, in addition

to designing it. Due to time constraints I could not made a complex behaviour tree like Unreal Engine 4, thus me taking the decision to go for a Hierarchical State Machine (HSM).

Behaviour trees are like HSM [42] [43], but the hierarchy nodes are modular and unlinked. Tasks are organised in tree-like structures, the tree is traversed every iteration, each node inform their parents depending on their statuses, Success, Failure and Running, here is where a behaviour tree shows its true power, as different statuses propagate to different leaves executing the desired behaviour. These defines the flow of a Behaviour Tree, where nodes can adopt different types defined as:

- Composite, used for handling one or more children nodes, these can be executed in an orderly manner (first to last or vice versa) or in a random way, depending on how they are structure, but structuring them in order allow game designers to create sequences.
- Decorator, like composite nodes but can only hold one child, commonly used as inverters, as the child fails it return success to the parent, useful for sequences or conditional tests.
- Leaf, a node incapable of having children, however these are what make behaviour trees useful as they contain the action to execute.
- Succeeder, always returns Success despite of the child return status, specially used when the child node is expected to return failure, but it is desired to keep running the sequence. Using inverters turn this in a Failureer.
- Repeater, a node that re-runs its child node every time the child returns a status.
- Repeater Until Fail, like the repeater but stops after the child returns Failure.

Behaviour trees are easier to visualise than HSM and easier to organise. Nonetheless they present some drawbacks, as Behaviour trees grow large iterating through them can have a negative impact on performance, even though that the idea is that sub trees are created to iterate through them rather than the whole tree it makes the system more complex and harder to debug.

Hierarchical State Machine [44] [45] is my approach for this module, based on Finite State machines but with more complex transitions and sub-state trees. Zombies have a main child node, Idle, after a few seconds if the player is not detected the zombie is in a "relaxed" tree, here it has statuses like wandering, in this state random points within a sphere are selected for the enemy to navigate to, if its sensors detect something it searches for it, if a player is detected then it jumps into the aggressive tree, if there is a path available it starts following the player, if not it screams to alert other zombies, when a Zombie is alerted the information passed by others is assessed and if worth searching for it fetches the player if not it goes back to the main node, if a player is found it goes into follow mode, if player is within attack distance it then executes the attack state if not it keeps following it, if the zombie loses the player it then goes into a search state repeating the cycle. A detailed view of the zombies' behaviour is found in the figure below.

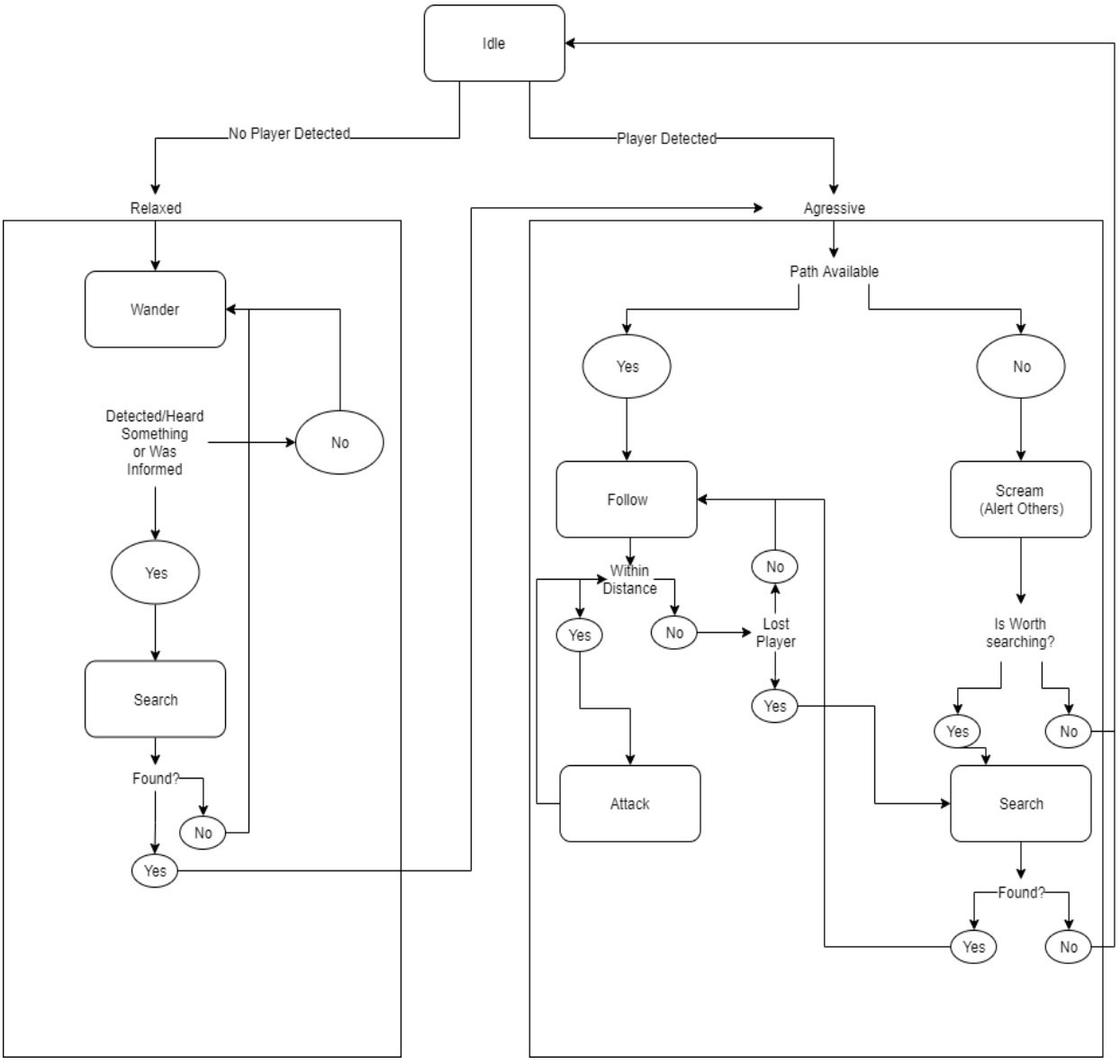



*Figure 5. Diagram showing enemy HSM logic*

Not included in this figure but at any time the zombie can go into the death state, regardless of its current state.

## 3.3 Advanced Topics

After covering the last two sections it could be said that there the prototype has a functional AI, however that is not the case, for it to work it needs a couple of sub-systems that are core to create the illusion that Zombies are critically thinking and planning its moves.

One of these sub-systems are sensors, the zombies have a wide-range of sensors that pass them information about the environment and what is going on around them, processing the information and taking actions based on them, these sensors are:

- **Smell**: basically, a capsule around the zombie with a predefined radius, it detects players that are really close to the zombie and none physicals dense objects, like a wall, are in the way.
- **Sight**: a raycast projected on the local forward axis of the enemy, at the end of it a capsule surrounds it (capsule raycast). It has a maximum range but never reaches it if it stumbles upon other objects that obstructs its vision, like a wall, or detects a player.
- **Hearing**: a capsule around the enemy, that captures any sound within distance emitted from objects or the player.
- **Herding**: a capsule around the enemy, that captures information of actions taken from other enemies around it.

Games like Days Gone, prove that zombies on their own are a beatable target, however when in a horde like in a multi-agent system the player faces a bigger challenge, here strategy and, specially in my game, accuracy, takes a big role to keep progressing through the level. Agents interact with each other through its herding sensor, in a decentralised manner spreading information to those around them and those doing

so as well, after the information is assessed they take decisions individually. These leads to interesting behaviours like ambushes.

Finally, my agents can adapt to the environment by learning what happens to them or other agents. There are a couple of components in my game that are available for agents to learn, like bullets when the player starts shooting to them, they feel threaten if not killed soon the Zombie gradually becomes more aggressive towards the player, if the player is lost and the zombie survives the information is passed to others. Agents can adapt to fire, when a Zombie observes that others are burning and subsequentially dying or their themselves are burning they become more sensitive to fire and try to avoid it if seen, informing to others its dangers.

# Chapter 4 Conclusion

Knowing the theory underneath physics and AI components in Game Engines helps programmers to optimize them and make good use of them. Today the amount it takes to make believable physics or intelligent agents is taken by granted as Engine programmers create tools to speed up the workflow saving time from coding core systems and allowing coders to invest these in making engaging gameplay mechanics.

In this paper we have explored the maths behind collisions, in my prototype I have decided to use a combination of both convex and concave polygons, but optimizing when possible to simplistic shapes, like the player by using a bounding capsule instead of having a collider faithful to its shape, also optimization was done even when more accurate collisions were required, like with enemies, every part of the body its surrounded by a bounding box, speeding up calculations. Constraint and Forces were

analysed, by learning the formulas behind the Unity's Joints - a wrecking ball and ragdolls were created, and finally a truthful ballistic system by using a wide-range of information and formulas found in research papers and hobbyist magazines mimicking the physics of games like ArmA III.

Finally, this paper explored the different ways to make AI challenging and believable, fetching the player or navigating through the environment by using Unity's Nav Mesh and its build-in A* algorithm, allowing the enemy not to get stuck with dynamic obstacles and updating the path when the player or a dynamic obstacle moved. A powerful hierarchical state machine that defines the brain of the AI, with nice transitions in function of what it is being perceived by their sensors, and extendable. Complex sensors that allows the AI to scan the environment and take actions if necessary, with a hording system that makes the AI hard to beat when combine with others and finally a learning/adapting behaviour that makes the player feel that the AI is alive and changes depending on their actions.

This prototype has taken three months to make, making the individual components and then merging together to create a gameplay experience that can resemble to games like ArmA III and Days Gone, source of inspiration for this paper.

# Bibliography


[1] “ArmA III,” Bohemia Interactive, Czech Republic, 2013.

[2] “PhysX,” NVIDIA Corporation, United States, 2001.

[3] I. Newton, Philosophiæ Naturalis Principia Mathematica, London, England, 1687.

[4] G. Palmer, Physics for Game Programmers, Apress, 2005.

[5] D. M. Bourg and B. Bywalec, Physics for Game Developers, O'Reilly Media, 2013.

[6] J. Lucas, “Newton's Laws of Motion,” LiveScience, 27 September 2017. [Online]. Available: https://www.livescience.com/46558-laws-of-motion.html. [Accessed 28 December 2019].

[7] J. Lander, “When Two Hearts Collide: Axis-Aligned Bounding Boxes,” Gamasutra, 3 February 2000. [Online]. Available: https://www.gamasutra.com/view/feature/131833/when_two_hearts_collide_.php. [Accessed 28 December 2019].

[8] “Collision Detection (Advanced Methods in Computer Graphics),” what-when-how, [Online]. Available: http://what-when-how.com/advanced-methods-in-computer-graphics/collision-detection-advanced-methods-in-computer-graphics-part-1/. [Accessed 28 December 2019].

[9] “3D collision detection,” MDN Web Docs, [Online]. Available: https://developer.mozilla.org/en-US/docs/Games/Techniques/3D_collision_detection. [Accessed 28 December 2019].

[10] C. Ericson, Real-Time Collision Detection, 2004.

[11] M. Gomez, “Simple Intersection Tests For Games,” Gamasutra, 18 October 1999. [Online]. Available: https://www.gamasutra.com/view/feature/131790/simple_intersection_tests_for_games.php. [Accessed 22 December 2019].

[12] “Add a K-DOP collision hull to a Static Mesh,” Unreal Engine Documentation, [Online]. Available: https://docs.unrealengine.com/en-

US/Engine/Physics/Collision/HowTo/AddDOP/index.html. [Accessed 29 December 2019].

[13] "SAT (Separating Axis Theorem)," dyn4j, 1 January 2010. [Online]. Available: http://www.dyn4j.org/2010/01/sat/. [Accessed 30 December 2019].

[14] N. Souto, "Video Game Physics Tutorial - Part II: Collision Detection for Solid Objects," Toptal, [Online]. Available: https://www.toptal.com/game/video-game-physics-part-ii-collision-detection-for-solid-objects. [Accessed 30 December 2019].

[15] "Module 3 -- Constrained Motion and Constraint Forces," MIT, 2011. [Online]. Available: https://scripts.mit.edu/~srayyan/PERwiki/index.php?title=Module_3_--_Constrained_Motion_and_Constraint_Forces. [Accessed 30 December 2019].

[16] "Joints," Unity Documentation, [Online]. Available: https://docs.unity3d.com/Manual/Joints.html. [Accessed 30 December 2019].

[17] A. "Physics in construct 2: forces, impulses, torque and joints," Construct, 29 September 2011. [Online]. Available: https://www.construct.net/en/tutorials/physics-in-construct-2-forces-impulses-torque-and-joints-70. [Accessed 30 December 2019].

[18] "What is Hooke's Law?," Khan Academy, [Online]. Available: https://www.khanacademy.org/science/physics/work-and-energy/hookes-law/a/what-is-hookes-law. [Accessed 30 December 2019].

[19] "Damping," Britannica, [Online]. Available: https://www.britannica.com/science/damping. [Accessed 30 December 2019].

[20] "Hinge Joint," Unity Documentation, [Online]. Available: https://docs.unity3d.com/2017.3/Documentation/Manual/class-HingeJoint.html. [Accessed 30 December 2019].

[21] Tactics, Techniques, and Procedures for FIELD ARTILLERY MANUAL CANNON GUNNERY, Washington, United States: Marine Corps, Warfighting Publication, 1996.

[22] “External Ballistics Summary,” Close Focus Research, 3 May 2009. [Online]. Available: http://closefocusresearch.com/external-ballistics-summary. [Accessed 31 December 2019].

[23] R. Cleckner, “How To: The Effect Of Gravity On A Bullet’s Path,” GunDigest, 27 September 2017. [Online]. Available: https://gundigest.com/article/understanding-gravity-effects-bullets. [Accessed 31 December 2019].

[24] “What is drag?,” NASA, [Online]. Available: https://www.grc.nasa.gov/www/k-12/airplane/drag1.html. [Accessed 31 December 2019].

[25] W. T. McDonald and T. C. Almgren, “The ballistic coefficient,” 2008.

[26] B. Litz, “Aerodynamic Drag Modeling for Ballistics,” Applied Ballistics.

[27] “G1 & G7 Ballistic Coefficients... What's the Difference?,” KestrelMeters, [Online]. Available: https://kestrelmeters.com/pages/g1-g7-ballistic-coefficients-what-s-the-difference. [Accessed 31 December 2019].

[28] B. Litz, “Gyroscopic (spin) Drift and Coriolis Effect,” Applied Ballistics.

[29] “Coriolis force,” Britannica, [Online]. Available: https://www.britannica.com/science/Coriolis-force. [Accessed 2 January 2020].

[30] J. Atkins, “Angular Speed of the Earth,” The Physics Factbook, 2002. [Online]. Available: https://hypertextbook.com/facts/2002/JasonAtkins.shtml. [Accessed 2 January 2020].

[31] A. Baldi, “Long Range Shooting: External Ballistics – The Coriolis Effect,” Loadout Room, 20 July 2017. [Online]. Available: https://loadoutroom.com/thearmsguide/external-ballistics-the-coriolis-effect-6-theory-section/. [Accessed 2 January 2020].

[32] “Eötvös Effect: Evidence of Spherical, Rotating Earth,” Flat Earth, [Online]. Available: https://flatearth.ws/eotvos. [Accessed 2 January 2020].

[33] D. Miller, “A New Rule for Estimating Rifling Twist,” Precision Shooting, 2005.

[34] A. Baldi, “Long Range Shooting: External Ballistics – Spin Drift,” Loadout Room, 5 August 2017. [Online]. Available:

https://loadoutroom.com/thearmsguide/long-range-shooting-external-ballistics-spin-drift-13-theory-section/. [Accessed 2 January 2020].

[35] "Gyroscopic Stability Calculator," Bison Ballistics, [Online]. Available: https://bisonballistics.com/calculators/stability. [Accessed 2 December 2020].

[36] "Under the hood. The physics of projectile ballistics," Panoptesv, [Online]. Available: http://panoptesv.com/RPGs/Equipment/Weapons/Projectile_physics.php. [Accessed 3 January 2020].

[37] M. Stezano, "In 1950, Alan Turing Created a Chess Computer Program That Prefigured A.I," History, 29 August 2017. [Online]. Available: https://www.history.com/news/in-1950-alan-turing-created-a-chess-computer-program-that-prefigured-a-i. [Accessed 03 January 2020].

[38] X. Cui and H. Shi, "A*-based Pathfinding in Modern Computer Games," 2010.

[39] "Inner Workings of the Navigation System," Unity Documentation, [Online]. Available: https://docs.unity3d.com/Manual/nav-InnerWorkings.html. [Accessed 3 January 2020].

[40] "Introduction to the A* Algorithm," Red Blob games, 26 May 2014. [Online]. Available: https://www.redblobgames.com/pathfinding/a-star/introduction.html. [Accessed 3 January 2020].

[41] M. Rhino, "A* Pathfinding for Beginners," Gamedev, 9 October 2003. [Online]. Available: https://www.gamedev.net/articles/programming/artificial-intelligence/a-pathfinding-for-beginners-r2003/. [Accessed 3 January 2020].

[42] J. Rasmussen, "Are Behavior Trees a Thing of the Past?," Gamasutra, 27 April 2016. [Online]. Available: https://www.gamasutra.com/blogs/JakobRasmussen/20160427/271188/Are_Behavior_Trees_a_Thing_of_the_Past.php. [Accessed 5 January 2020].

[43] C. Simpson, "Behavior trees for AI: How they work," Gamasutra, 7 July 2014. [Online]. Available: https://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php. [Accessed 5 January 2020].

[44] “Hierarchical State Machine,” EventHelix, [Online]. Available: https://www.eventhelix.com/RealtimeMantra/HierarchicalStateMachine.htm. [Accessed 5 January 2020].

[45] M. Samek, “Introduction to Hierarchical State Machines (HSMs),” BARR Group, 4 May 2017. [Online]. Available: https://barrgroup.com/embedded-systems/how-to/introduction-hierarchical-state-machines.