Objects in Unity

How does the game loop works in Unity

What are frames

In gaming, frame rate is the rate in which the image on the screen is refreshed. For example, movies are made up of moving images and most are projected at 24 frames per second, there are 24 still images shown, one after the other, in one second. This creates a very natural feel – most people would not be able to tell the difference between 24fps and 48fps. This value has been carefully chosen for the best experience and space requirements since a 48fps movie file would be twice the size. In games actual objects are updated every frame and everything has to be rendered by the graphics card again and again, so the frame rate does not affect the size of the game but is itself affected by the hardware. A high end graphics card would be able to render more frames in one second than a low end one. Most computer games will be run on a variety of hardware which will render the game in different frame rates, so it is important to for example, make sure that the physics of the game are not dependent on the frame rate. <https://www.lifewire.com/optimizing-video-game-frame-rates-811784>

Physics

Game physics engine is a software that introduces and simulates laws of physics into a game. These laws can either be based on real-world physics or can be entirely new. It simulates the collision detection, rigid or soft body dynamics and fluid dynamics. Unity provides a built-in physics engines for 3D and 2D games that handle the physical simulation. The main components it provides are the Rigidbody2D and the Collider. These are explained in … but in essence, Rigidbody2D allows the game objects to be affected by various forces like the gravity. Collider allows various objects to collide with each other and if paired with Rigidbody, allows objects to affect each other.

<https://docs.unity3d.com/Manual/PhysicsSection.html>

MonoBehaviour

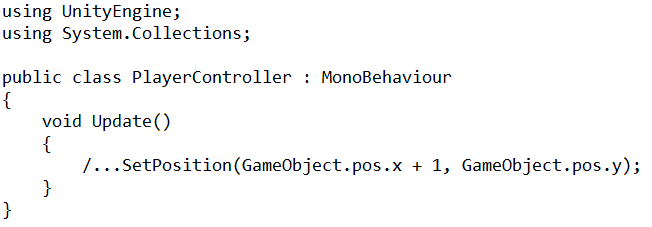
When using C# in Unity it is important to derive every class from MonoBehaviour because it is the base class from which every other class derives. MonoBehaviour has all the important functions that are used by the Unity in order to work properly. The methods are Start, Update, FixedUpdate, LateUpdate, OnGUI, OnDisable, OnEnable.

The last three were not used in this project so there is no need to explain them in detail. OnDisable is called when the object is disabled or destroyed and is usually used for cleanup code. OnEnable is the opposite, it is called when the object is enabled.

OnGUI as the name suggests is used for rendering Graphic User Interface Events like clicking a button on a screen, using GamePad to navigate the GUI etc. Therefore it may be called for each event – several times per frame. This was also not used in this project because the Graphic User Interface was implemented by using the built in User Interface objects, this is explained in ….

The most important functions are Start and the three Update methods. Start method is called in the same frame that the script is enabled and before its Update methods. Start is great to initialize the game objects by enabling the developer to set up the object before the object is updated. It is ran only once in the lifetime of its script. Start is used similarly to a constructor and should be used instead of it when deriving from MonoBehaviour to avoid issues or unexpected results as explained in. <http://ilkinulas.github.io/development/unity/2016/05/30/monobehaviour-constructor.html>

The Update methods are very similar to each other with slight variations in how they are called. The Update method is called every frame and is used, as the name suggests, to update the game object it is attached to. The update is most commonly used method along with the Start, because it allows adding a behavior to an object. For example, if an object has to move from one side to the other, its x or y position could be changed by adding a number to it in the update method. The code in figure … would make the player move by one pixel to the right every frame. The update methods are called only if the MonoBehabviour is enabled.



LateUpdate is just like the Update method – it is called every frame but it is called only once all Update methods, inside the object and inside other objects, have been called. An example where this would be used is when following the player with a camera, updating the camera position should be done only once other objects’ position has been updated to avoid issues and unexpected behavior.

The last method, FixedUpdate, is called every fixed time not dependent on the frame rate. It is mostly used to calculate physics and to avoid issues various issues with time dependent methods. For example if the player has to move right at a certain speed, its position should be updated using the FixedUpdate method. In figure … the speed in which the object would move would be dependent on the frame rate, if the game was running at 25 frames per second, the value would be updated 25 times in one second but if the game was running at 100fps, it would be updated 100 times. Therefore the higher the frame rate, the faster the object would move. FixedUpdate gets rid of this issue by updating the object in a fixed time. This time value can be accessed by using the Time.fixedDeltaTime which shows the in game time in seconds.

<https://docs.unity3d.com/ScriptReference/MonoBehaviour.html>

In Unity scenes contain environments, objects and the menu for the game. Each scene is like a level with its own objects, to easily separate the game into small pieces. For example, almost every object that will be used in the main game level is not needed in a menu therefore there is no need to have it in the game scene.

<https://docs.unity3d.com/Manual/CreatingScenes.html>

In Unity game object is a base class which all objects like the characters, props and scenery in the scene implement. It provides all the necessary functionality that allows the object to be affected and act like containers for other components. For example, attaching controls, Rigidbody and a Collider to an object makes it a playable character. All object come with a Transform component which describes it position in a game scene.

<https://docs.unity3d.com/ScriptReference/GameObject.html>

<https://docs.unity3d.com/560/Documentation/Manual/class-GameObject.html>

How User Interface is implemented in Unity

The user interface in Unity is easily implemented by creating a Canvas. All user interface elements should be inside the Canvas area. As explained in … Canvas is a basic Game Object with a Canvas component attached to it. Adding a UI element to Unity automatically create a Canvas and sets it as a parent to that element. EventSystem object has to be added to use as a messaging system otherwise it would not be possible to interact with the objects.

Event system is used to send event messages to various objects by using a mouse, keyboard or other input systems. Event system has a very basic functionality exposed only because it is designed as a manager and coordinator of messages between the modules. Its main objectives are to manage the selected game objects, manage and update the input modules and other like managing Raycasting.

Unity sets up everything that is needed to make the user interface work properly automatically. Default settings were used for this project therefore there was no need to set everything up manually.

In game UI

Game Setup

Designing the UI - HUD

The User Interface is should be easy to use, intuitive and convey lots of information in a very easy to understand way. Modern games range from excessive information on the screen to as little as possible. An example of a game where there is an excessive amount of information on the screen is the Horizon Zero Dawn action role-playing game developed by Guerrilla Games and released in 2017. Figure… shows the UI and as it can be seen, the player is given all the important information on the screen like health points, stamina, weapons, map and quick access items. The game has RPG elements to therefore giving the player all these information is very important. Role-Playing Games have a much more focus on the game mechanics and the game is driven by the player. That is, the player makes all of the decisions, chooses what to do, the player character skills and usually makes in game decisions which steer the story in a unique direction.

On the other end there are games that are more like the movies, which means that in a way they play themselves. Uncharted 4, an action adventure game developed by Naughty Dog and released in 2017, is an example of a game where the player does not have too much impact on the game world and almost every event is scripted. This allows the game designers to have as little information on the screen at one time as possible. In fact, most of the time the HUD is hidden except the times when the player is using weapons.

The game in this project will be in the middle, there will be enough information for the player to understand what is going on but not too much where the player is distracted. This is especially important because the game is very fast paced and competitive so the player will only be able to take a brief look at the HUD every some time. It takes an inspiration from older arcade games like Donkey Kong or Space Invaders. It will also be kept as basic as possible to avoid spending too much time on implementation since only few days will be given for that. The plan is to have a bar for health, stamina and shield in one of the top corners of the game and the weapon and ammo in one of the bottom corners.

The Menu will be broken down into four scenes; Title Screen, Game Setup, Main and End Game.

Title screen will be the first thing the players see when they open the game. It will be kept very basic, with a name and a logo of the game, creator name and start button. It is also planned to have a theme song playing in the background just like in the games mentioned above since this makes the game much more memorable and keeps with the theme of arcade games.

Start button in the Title screen would send the user to the game setup screen; here the user will be able to setup the settings of the game to his likings. User will be able to change settings like the maximum game score, time of the game, starting stats (health, stamina etc.), type of opponent (another player or CPU) and other, more trivial settings like shield size or ball size. Since the user will see this screen after every match, the total game score for each team will also be shown here.

Game setup has been taken from more modern games like Call Of Duty where players are given an option to create custom games that fit their play style. In arcade games there were usually fixed settings of the game settings will be preset to default settings which will be tested and which the game will be designed around. It is also planned to give user an option to save the settings of the game and the game score to a file although this will not be a priority since time will be very limited.

The End Game screen will be kept very simple; it will have the name of the team that won the game or “Draw!” if there is a draw. It will also have the score of the current game and the Continue button which will send the user back to the Game Setup. This will be like a transition screen between the game and setup screen so there is no need to add too much information.

Additional

The user will be able to transition between the game scenes by using the available buttons. Figure shows in detail which scenes the user will be access from each scene. One thing worth pointing out is that the user should not be able to access the End Game screen from other scene than the Main game. End Game screen shows the end of the game scores which are unknown until the end of the game.

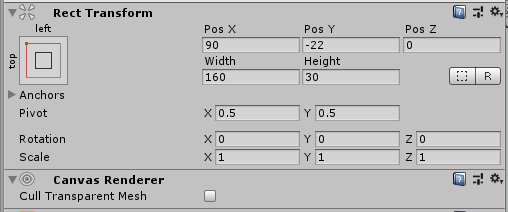
<https://guides.gamepressure.com/horizon_zero_dawn/guide.asp?ID=38878>

Implementing the UI

The first User Interface that has been implemented is the Head-Up Display (HUD). The information that was meant to be shown to the user as mentioned in the design has been implemented but had to be simplified. Instead of a rectangle or a bar, a numerical value is shown for the health, stamina, shield and ammo. Unfortunately there was not enough time to have two separate HUDs for player vs. CPU and player vs. player, game modes. So, instead one of the sides is for the first player and the other, right, side is for the second player/CPU. It was useful to do it that way when implementing the game because it allowed seeing what the AI’s status is.

However, this way is very limited because if there would be more than two players on each team, it would be hard to find space to show each player’s status. The best solution would be to have a separate HUD for each of the game modes; one HUD would have the AI’s opponent health only displayed above its character and hide everything else the other HUD would be designed for a player vs. player and have the HUD as it is but show the weapon and ammo in the bottom corners. FIGURE

The HUD has been implemented in a very basic way – a Canvas and Event System object has been created and default Text objects have been added as a child of the Canvas object. The Text objects allow positioning the text with the Rect Transform component and to add text by using the Text script component. The settings have been kept default with only the font and font size changed.



To show the score a new game object has been created with a Score UI Controller component inside. It:

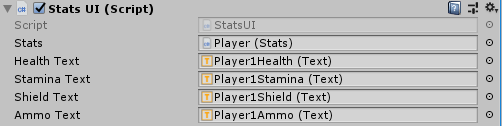
* takes two TeamController components
* extracts the scores

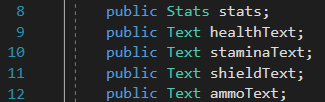


* creates a string out of them
* passes them to a Text component

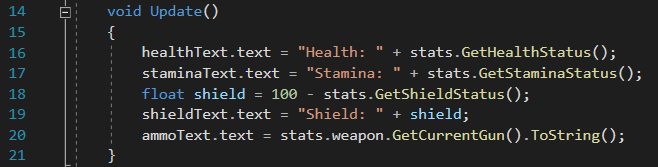


Stats have been done similarly, a StatsHUD object with two StatsUI components for each player have been created. StatsUI is very similar to Score UI Controller but it takes StatsUI object, which gives all of the player’s statistics like health or stamina, and turns them into a string and passes to Text objects accordingly. Figure … shows a Stats UI object with everything set up. As it can be seen it is very general and can be easily duplicated in case the teams are expanded.





Figure… shows the update loop which is very basic since it only has one task – to convert the status into a string.

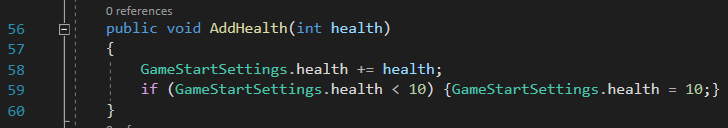


The Game Scene

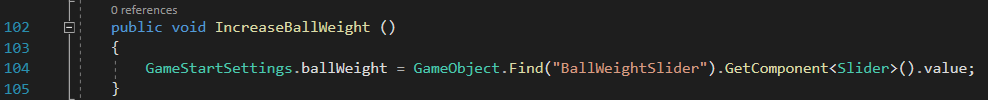
The game settings have been set up very similarly to above, update method creates a string from the status info that it takes from the GameStartSettings component. Each button is made up of few different objects and components. As seen in fig… a button that changes the amount of health each player starts with has four game objects under it; two text objects and two buttons. First Text object is used to show the name of the variable that is being changed, in this case the health, and the second Text is used to show the current value of that variable.

Each text is done through the Start Settings component inside the StartSettings game object. This component takes care of building strings out of the values inside the GameStartSettings game object and it takes care of all of the methods used by the buttons.

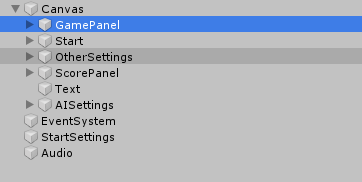
The methods are very basic because their only role is to either add or subtract from the value inside GameStartSettings. Fig… shows an example of that methods, as it can be seen, a value is passed to the method which is then added to the given variable. Fig… shows how that method is used inside the button, Unity has a very straightforward way to pass a value to the method it is calling. AddHealth method is called with 1 passed to it, in the Add button and the same method is called to subtract from that variable but -1 is passed to it. Fig. shows the AddHealth method that is called when the Add button is pressed.

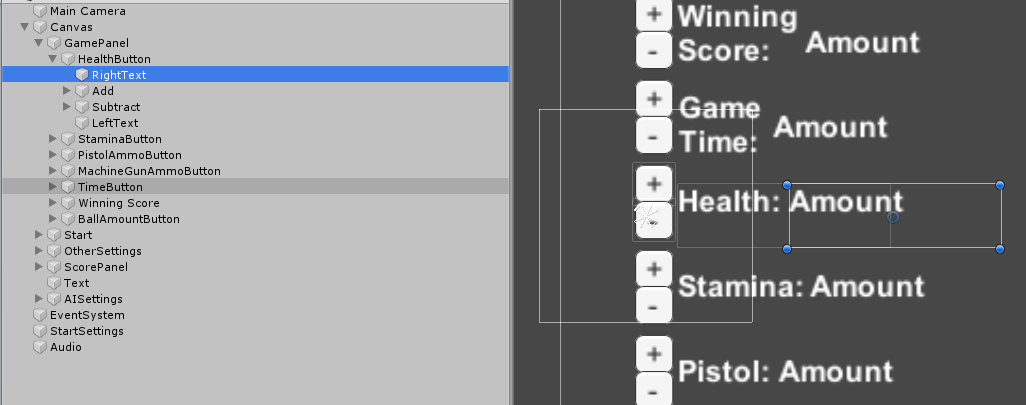


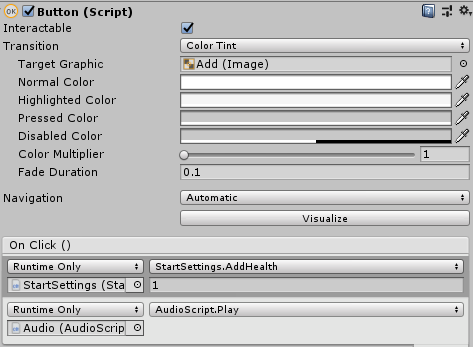
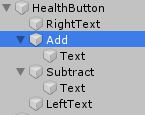
Some objects have a slider instead of two buttons to fit the variable much more. Slider allows users to select a numeric value by pressing on a button and dragging it left or right. The slider component allows calling a method when the value changes, change the minimum and maximum value or set the starting value. Fig. shows the method that is run when the slider is changed.



<https://docs.unity3d.com/Manual/script-Slider.html>







GameStartSettings component is a static class that holds the information of the game as static variables. This was implemented that way because information will have to be passed between the game scenes and while Unity gives an option to pass information between the scenes, this was the fastest and easiest way to implement it. It plays a role of a Configure file.

GAME MODES

The game modes are the way the user can play a game. Most games come with various games modes to find the one that will be most popular with users on which then the developers focus on the most. An example is Fortnite developed by Epic Games and released in 2017, it started with numerous game modes but the Battle Royale has been the most successful so now the developers are mainly focusing on updating it.

This game will be designed around two games modes to avoid making it overcomplicated, especially because any extra thing added to a game will impact the creation of the Artificial Intelligence and it is important to keep the scope of the game on a reasonable level. The two modes will be human player against another human player, also known as couch co-op, and the other will be human player against the Artificial Intelligence.

The first mode should be relatively easy to implement once one player character is working since the player game object would just have to be duplicated. Then the control buttons will have to be changed to fit another gamepad/keyboard, then it will have to be assigned to a different team. Important aspect that will have to be considered is to make sure that each user is able to see its character’s status like health or stamina points. This is explained further in the User Interface chapter.

The second mode is almost the same like the first but here the player will play against the Artificial Intelligence. This is the main mode of the game and the project focuses on this aspect in a big part. Here, the user will not have to see its opponents’ status so there will be much more space to spread out the HUD. The HUD is also explained in the User Interface chapter.

The game modes will be implemented in a way that they will be able to be expanded to bigger teams but for now, to make sure the project will be possible to finish in the given time, the teams will be limited to one player only on each team. However, if possible, this can be expanded to more players if there will be enough time to expand the Artificial Intelligence’s behavior to work in a team.

TEAM

To make the game independent of the amount of player teams had to be made. Since there are only two goals, there can only be two teams – Red and Blue. Each player will be given it Team component which will tell what who are his teammates, opponents, which goal he should focus on, starting position of each player and the score of the game.

This will allow to avoid having a score inside the player game object so adding new players will be much easier. The Team component could have a method like add score, so when a players health reaches 0, opponents abstract Team component will have to be called with the add score method. The team component could either be Blue Team’s or Red Team’s, this allows to have objects independent of each other.

Implementation

Teams have been implemented just like planned – Team game object has been created with two game objects, one for each team, that hold the TeamController component. TeamController takes care of the the players inside the team, the score, opponents and the goals. It has some additional methods like ResetPlayers which changes all of its players to their default positions which are held inside the GlobalGameSettings. It also holds information about the State Boundaries which are the places where the AI changes its state.

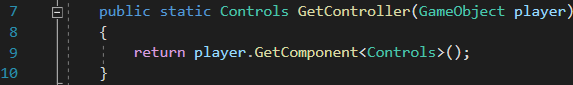
Artificial Intelligence Implementation

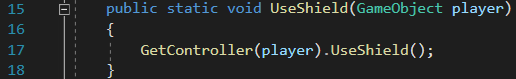
Artificial Intelligence has been implemented in the same way as planned – there are a number of behaviours, the play area is divided into four parts and the precise behavior is based on the position of the players. The behaviours have been broken down into various groups, and each behavior is implemented as a static method inside a Static Class. Implementing behaviours as static method seemed like a best way because it allowed to separate the part that takes care of the logic with the part that does an actual method. This part will first explained the behaviours that the AI can choose from and then will go on to explain the logic behind it – that is, how it chooses the behavior.

The behaviours are separated as follows; Basic Behaviour, Direction Behaviour, Global Behaviour, Movement Behaviour and Team Behaviour. There are also helper methods that are not actual behavior, i.e. they do not do an actual thing but they are closely correlated with the methods inside the behaviours, these are; Calculate, General and Check.

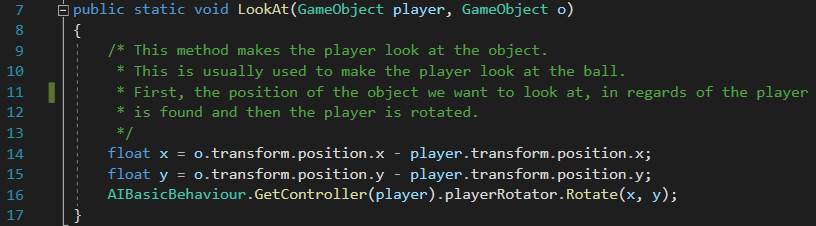
Basic Behaviour Class has a set of methods that are no more than two lines. Their tasks are just the basic controls like using a sword or a shield, swapping guns or sprinting. Figure… shows the first method, GetController which takes out the Controls component from the player GameObject. Figure … shows one of the methods inside the Basic Behaviour class. While this is very basic, it makes the game code easier to read since these methods will be used often. It changes player.controls.UseShield() into AIBasicBehaviour.UseShield(player)

Appendix.. shows all of the methods.

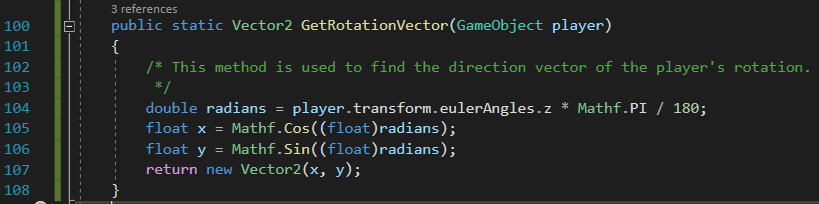


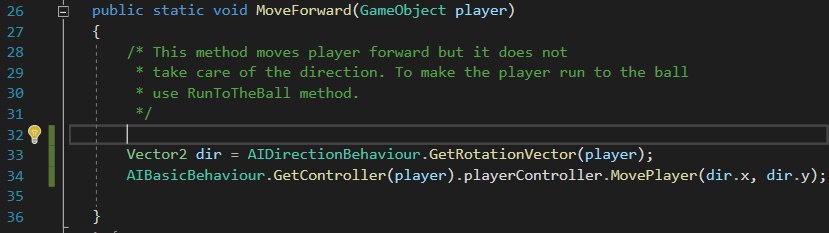


Movement Behavior Class takes care of all things that have to do with player movement and player rotation. These methods are slightly more complex than in the Basic Behaviour Class but their tasks are simple – rotate to ‘look’ in a given direction or at an object and move towards an object, a given direction, or up or down. Most of the methods are already written inside the player objects, like a method to rotate, which is inside the PlayerRotator component in a player game object or a method to move like a method to move the player towards x and y direction, inside the Player Controller component. Fig.. shows the Look At method which finds the x and y coordinates of the object in relation to the player, and rotates player towards it.



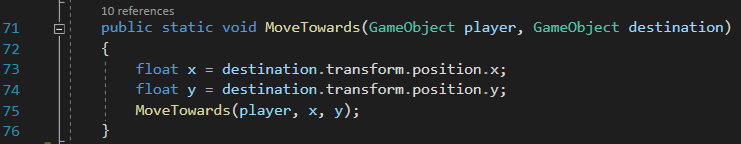
The move methods are very simple, the player can move forward in the same direction it is facing or direction vectors can be given to move in a wanted direction. Fig. and fig. shows the first Move Forward method.



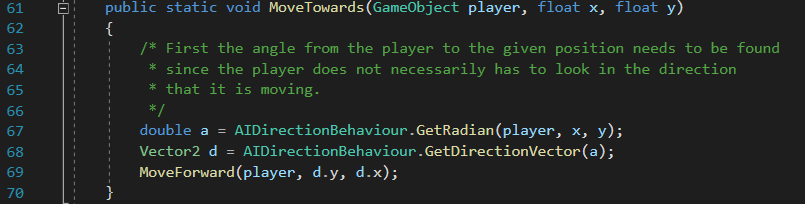


A more complex method inside this class is the MoveTowards method, which is used to move the player towards raw x and y coordinates (that can be found by using GameObject.transform.position) or towards an object.

The second method, shown in fig… finds the x and y position of the object and call the first method.



The first method however, is much more complicated because the x and y coordinated have to be converted to a direction vector. Fig. shows the Move Towards method, as it can be seen there are three stages, first the radian towards the given x, y position is found, then that radian is converted to Direction Vector and lastly, the player object is moved forward by using the Move Forward method mentioned above.



This introduces a new set of behaviors which is the Direction Behaviour Class. This class takes care of finding direction and rotation vectors, radians and the position of a direction vector. There are four main methods inside this class, GetRadian, GetDirectionVector, GetRotationVector and FindPositionOf. These are used to guide the player towards a ball and to make sure it positions itself in a way that it can shoot at the goal easily.

Fig. shows the GetRadian method, along with its explanation.

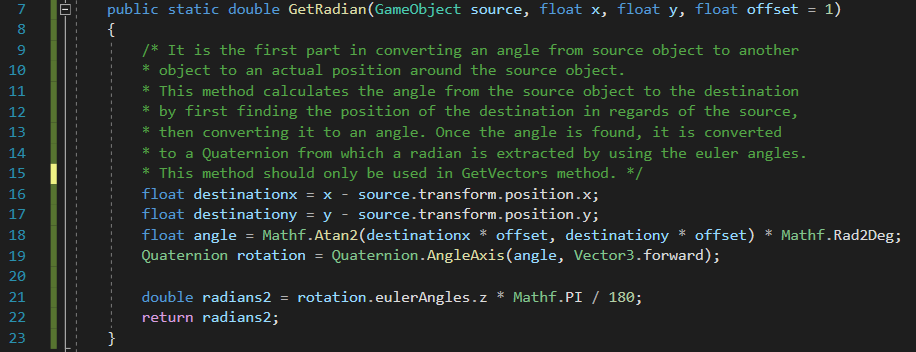
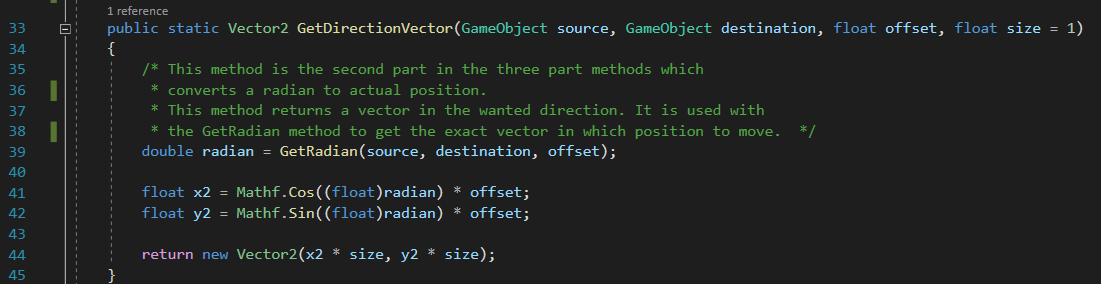
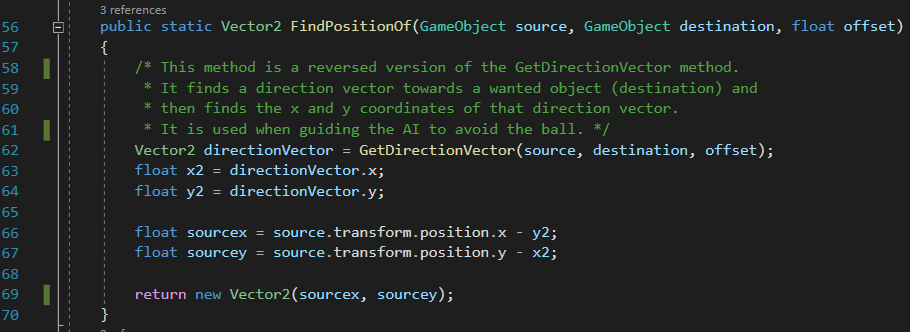


Figure. Shows the GetDirectionVector method which converts a radian to a directional vector. A more basic version of this method is used inside the MoveTowards method, since a radian is first found by using the GetRadian method, then the GetDirectionVector is found separately. Fig. shows that method.



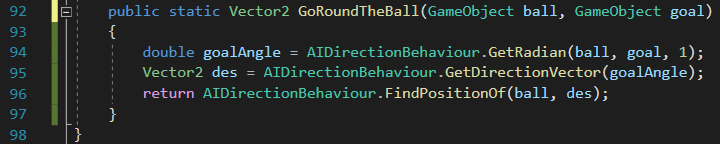
The next method is the FindPositionOf which is the reversed version of the GetDirectionVector method. Fig. shows the method and its explanation.



The first step as explained in the Design chapter, is to create the player movement, this is the most important step because without the AI movement, the player will not be able to score goals. As explained, the movement has been broken down into a number of steps. The first step is to move towards a ball, this is achieved by using the MoveTowards method which takes a player game object and a ball game object. Next, a new point has to be found where the player should position so that it faces an opponent’s goal. Fig. shows how that position should look like and where the player should move towards – the green dot. The arrow should point towards the goal at all times.

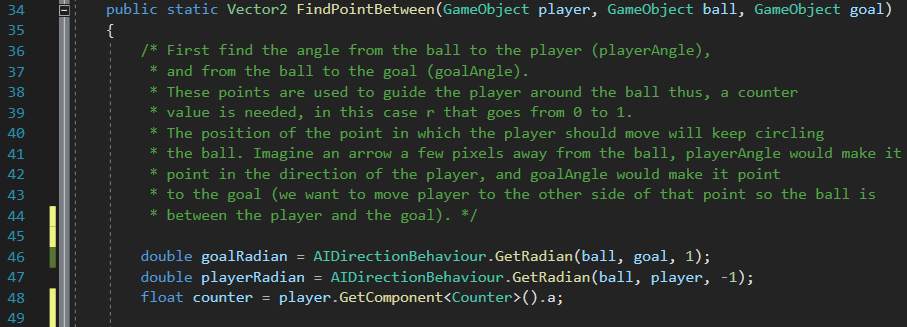
Point towards which the player should move

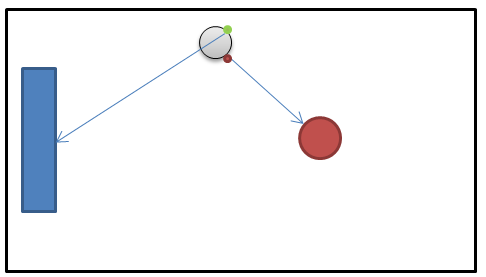
This position is found by using the three methods explained above; fig. shows the first version of this code which returns it as a Vector2.



However, there is one more issue, if the ball is between the player and the goal, the player game object can move towards it but if the player is between the ball and the goal, it would keep pushing the ball away from itself. The next step would be to find a way to go round the ball; this can be achieved by adding few more steps into the code.

First, an additional point has to be found – point that faces the player. This can be achieved in the exact same way as in fig above but the radian has to be inversed since now it does not have to be between the player and the ball. Fig. shows two points, green and red, green shows the point acing the goal and the red point shows the point facing the player. Fig. shows the first part of the code.





Next step would be to guide the player to the green point. This can be achieved by creating a new point, which would go around the ball. To create that point, green and red points have to be combined, and multiplied by a counter method that would go from 0.1 to 1 – at 0.1 the new point would be in the same position as the red point, at 0.5 it would be between red and green and at 1 it would be at green point which is the player destination. The equation is:



Since only one counter is needed and the class is static, it was easier to have the counter be a component of the player game object, as can be seen in fig. The class is very simple as its only job is to cycle from 0 to 1 in a steady step. Fig. shows that class. The value added in line 14 determines the speed, this has been carefully chosen to match the player speed.

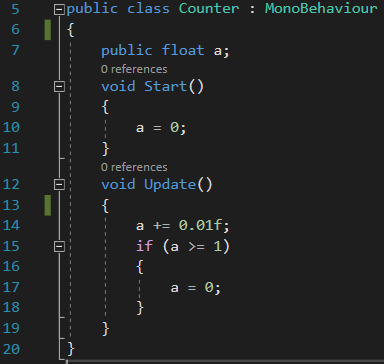
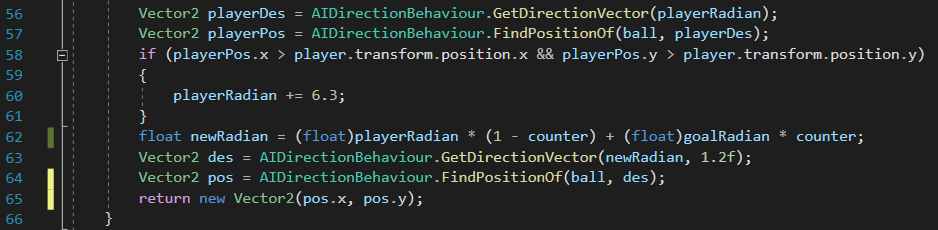
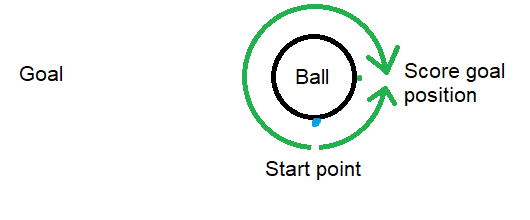
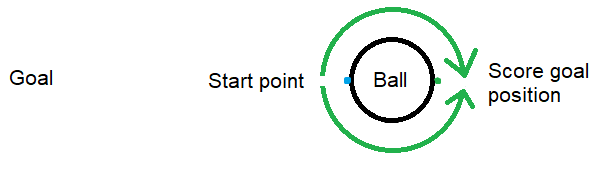


Fig. shows the last part of the method.

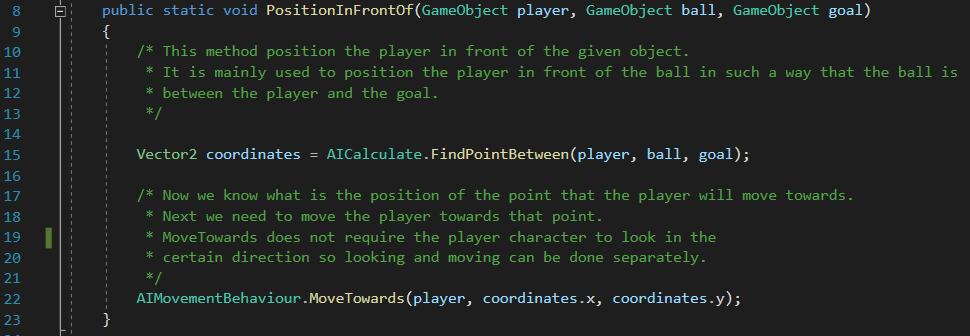


After testing and visualizing the new point, it was found that the point would circle the ball in a wrong way because the start and end point of the ball was at the bottom. Fig. shows the old start point, if it was kept like that, the player character would have to circle almost whole ball to position itself. Fig. shows its new path.

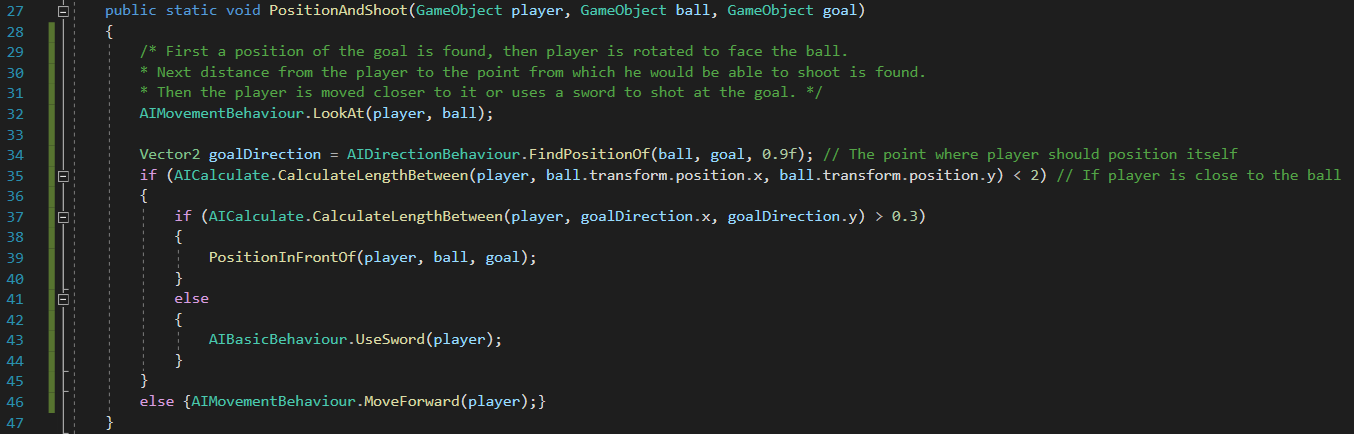




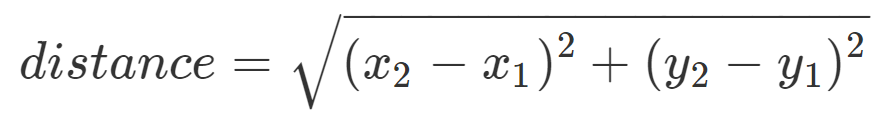
This method returns a Vector2 with a position of the point towards which the player should move towards. So, the next step would be to guide the player towards that vector. This is done inside the Global Behaviour Class, this class takes care of the main movement logic like positioning in front of the ball; positioning and shooting at the goal; and shooting the ball in the middle of the pitch. Fig. shows the first method – this method shows how the method above has been implemented.

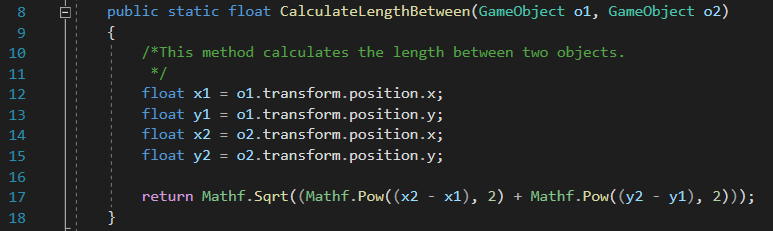


The next method is PositionAndShoot which is a slightly improved version of the above method, as it also makes the player look at the ball and use a sword to shoot towards the goal. Fig. shows the full method, as it can be noted there is no need to calculate the points mentioned in fig. so when the player is farther than 2 (around 20 pixels), it only just moves towards the ball. The third method is very similar to the one shown in fig. but there is an additional method call before the UseSword, which rotates the player to look at the middle of the pitch.



The last method that needs explaining is the CalculateLengthBetween used in the above methods. This method uses the distance formula which is show in eq… Fig. shows how it has been implemented.





The logic

How it chooses the behaviours

AI movement design

The player AI movement should be broken down into a number of steps. The first step is to either rotate the player character towards the wanted object or to find an angle or a direction vector towards the wanted object. The next step would be to move towards that object. However, this only works if the object is a pickup or another player game object, if it is a ball, more steps are needed.

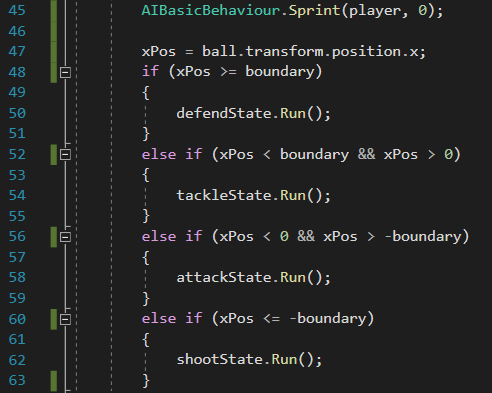
The last class, CheckBehaviour, has helper methods – CheckIfCloseToPickup, GetClosestPickup, CheckIfFarFromBall, CheckIfCloseToOpponent, CheckOpponentsHealth and CheckIfBallApproaching.

|  |  |
| --- | --- |
| Name | Task |
| CheckIfCloseToPickup | It goes through each pickup inside a passed list, calculates the distance between it and returns a bool value based on the results. |
| GetClosestPickup | This method checks if there are any active pickups in the game, checks which one is the closest and returns its position as a vector. |
| CheckIfFarFromBall | This method calculates the length between the player and the ball, and if the distance from ball is larger than allowable distance, from Global Settings and returns a Boolean value based on the results. |
| CheckIfCloseToOpponent | Check if the distance between first game object and second game object is 2 and returns a bool. |
| CheckOpponentsHealth | Check if the opponents health is lower than 20. |
| CheckIfBallApproaching | Check if the ball is close to the player by checking that the distance is between 2 and 3. |

The logic

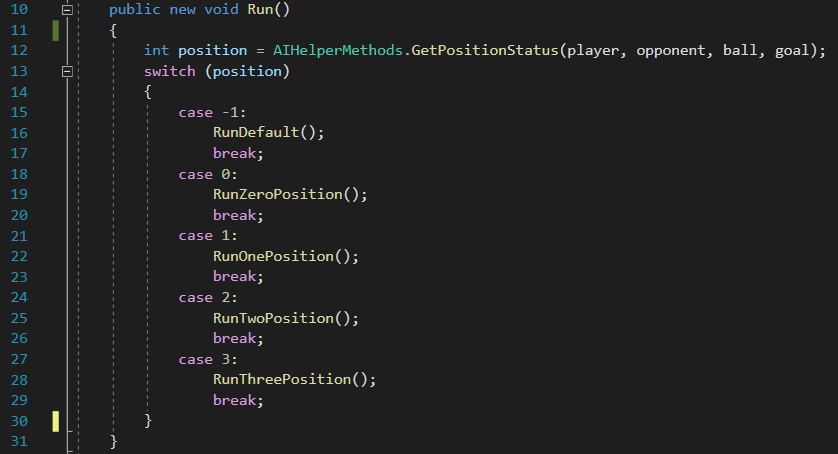
The last part to explain is the logic, how the player chooses its behavior. As it can be seen, each of the above behaviors creates a basic tree - each behavior builds upon other behavior to create more complex behavior, fig. shows how PositionAndShoot connects with other behaviours. Similar approach will be used for the logic as explained in Design chapter. Since the time was limited there was not enough time to refactor the code to take most of the advantages of behavior trees, a more basic approach has been taken. This part explains how the basic set of behavior is chosen by splitting the play field into four areas, and later goes into more detail how it then chooses a smaller set of behaviors based on the position of the ball until it finally chooses a concrete behavior.

As explained in the design chapter (fig. ), there are four states – (opponents goal) Shoot, Attack, Tackle and Defend (player’s goal). These states are of equal size, so naturally the play area is broken down first into two parts in the middle of the pitch and then, each is broken down again at around quarter of the pitch – the exact value is stored inside the TeamController component as an int value, “State Boundary”. The logic, shown in fig., that chooses the behavior is stored inside the AIController component which main task is to run a correct state inside a FixedUpdate method. As it can be noted it is a very basic ‘if’ statement that chooses the state based on the x position of the ball. Full Class can be found in appendix…



Each of the states inherits a AIState abstract class which has all necessary methods that should be implemented and references to objects that are used inside each state. The full class can be found in Appendix.. but the most important methods are Run (seen in fig. line 50, 54, 58 and 62), Run[position number from zero to three]Position and RunDefault. RunDefault is used as a safety measure in case everything else fails, it runs a PositionAndShoot method from the GlobalBehaviour.

The more important methods are the Run method and RunPosition methods. Run method chooses the behavior that should be ran based on the player, opponent and the ball position as explained in the Design chapter. It uses a switch based on the position given by the GetPositionStatus method. Fig. shows the full method.



The GetPositionStatus returns a number from 0 to 3, and -1 if the position has not been found. It uses logic equations to determine which number to return. For example, the zero position is when the ball is between the opponents goal and the player. So, the propositions are:

G = Goal is on the left side of the ball,

LO = Opponent is on the left side of the ball,

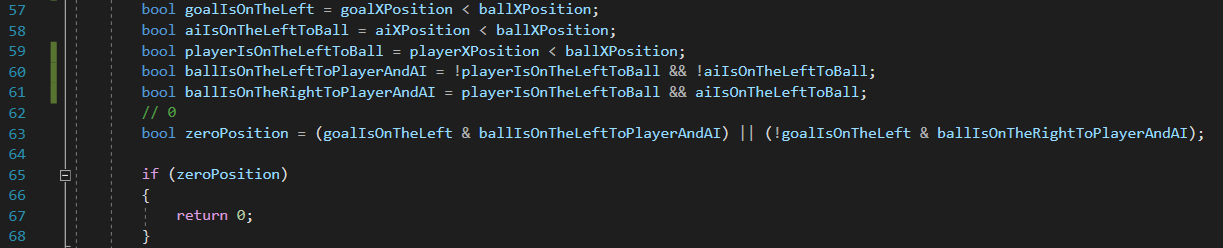
LP = Player is on the left side of the ball,

LB = Ball is on the left side of the opponent and the player,

Zero Position = Goal is on the left side of the ball, Ball is on the left to player and the opponent =>

G ^ (¬LO^¬LP ) => G ^ LB

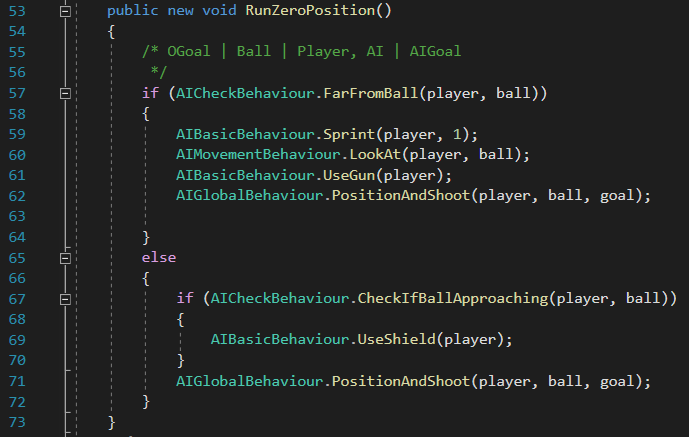
Fig. shows how Zero Position has been implemented. Appendix.. shows the full method but the same principle is used throughout.



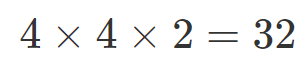
The other four RunPosition methods that are inside each state run the behaviours explained in previous part. As was explained, there are four different positions, creatively named Zero, One, Two and Three. If the time was not limited, these behaviours would be held inside a separate class or in an actual tree since the class clashes with the SOLID principles, like Single Reponsibility Principle which says that every module should only have one responsibility. Unfortunately this was a faster and safer way to implement that.

<https://itnext.io/solid-principles-explanation-and-examples-715b975dcad4>

The first and last thing that has to be checked inside each method is the distance from the ball. This is done by using the CheckIfFarFromBall method inside the Check Behaviour. This follows the same principles as the states and run methods, this time the player should behave differently depending on the distance it is from the ball. For example, if the player is in a defend state and far from ball, it could use a gun to kick the ball away from its goal to make it harder for the opponent to score a goal. If it is close to the ball it could use a shield to block the opponent from scoring a goal. Fig. shows a RunZeroPosition method inside the Defend State, as it can be noted, the behavior is basic but since the position of the ball changes very fast, it creates a dynamic behavior.



There are four states, each state has four positions and each position has another two positions (not counting the behaviours inside each distance), there is a total of:



AI DESIGN ADD APPENDIX WHERE EACH OF THESE METHODS EXPLAINS WHAT IT DOES