Xtreme Bowling Game Design Document

Rio McDade, Sam Nuttall, and Jack Judge

Contents

[Xtreme Bowling Overview 4](#_Toc159408188)

[Game Concept 4](#_Toc159408189)

[Game Targets 4](#_Toc159408190)

[Game Flow Summary 5](#_Toc159408191)

[Look and Feel 5](#_Toc159408192)

[Gameplay 6](#_Toc159408193)

[Objectives 6](#_Toc159408194)

[Play Flow 6](#_Toc159408195)

[Mechanics 7](#_Toc159408196)

[Rules 7](#_Toc159408197)

[Explicit Rules 7](#_Toc159408198)

[Implicit Rules 7](#_Toc159408199)

[Physics 7](#_Toc159408200)

[Objects 8](#_Toc159408201)

[Bowling Ball 8](#_Toc159408202)

[Pin 8](#_Toc159408203)

[Actions 9](#_Toc159408204)

[Charging a throw 9](#_Toc159408205)

[Curving a ball 9](#_Toc159408206)

[Screen Flow 10](#_Toc159408207)

AI Implementation ……………………………………………………………….11

[Game World 13](#_Toc159408208)

[General Look and Feel of World 13](#_Toc159408209)

[Levels 13](#_Toc159408210)

[Appendix 14](#_Toc159408211)

# Xtreme Bowling Overview

Game Concept

This bowling experience takes place in a vaporwave-themed *world scene* accompanied by custom vaporwave music. In this game, it is expected that the following is adhered to:

* The game is played on a *lane* that is slightly *elevated* from the floor of the world scene,
* On one short side of the lane, there are 6 *pins* arranged in a triangular formation,
* In every round of the game, the player throws 2 *heavy balls*, with a pause between each throw, from the other short side of the lane so that the ball achieves the *objectives*:

1. *Roll* along the lane *without falling* off it on the floor.
2. Knock as many pins as possible from their *initial position*.
3. All pins *stop moving* after the throw.

* After the end of the round, all assets are reset to their initial position for the next round.
* The player receives as many points as the number of *knocked pins per round*, and bonus score for a *strike.*
* The game ends after 3 rounds.
* Realistic Physics: Spinning and Rolling Ball, Natural Physics, Application of Mechanics.
* Ending Conditions: The ball has stopped moving, 3 frames have been played.

Game Targets

* The player can win.
* Realistic gameplay.
* Gameplay that is more natural than simply realistic.
* Other ways of counting points to make the gameplay non-trivial.
* More challenging gameplay than the one in the game description that is not infeasible for a player to win.
* Assets that are not playable anymore.
* Ending conditions for a round.
* Define an AI agent that represent the player. The agent’s objective is to maximise its score by learning the optimal bowling strategy

Game Flow Summary

Upon loading the game, the user is fronted with the main menu. Music is playing and there is text prompting the user to “click anywhere to continue”. Upon clicking anywhere, the main menu scene transitions to the main game scene. The music switches to the main game music, and the agent is free to move. There exists an on-screen key guide, which translates to the capabilities of the agent.

The agent is then able to move freely (within the bounds of the movement range) and can play the game until it ends. The ending condition for the game is after *3 frames have been played*. This then segues the agent to the end game scene, where they can exit or play again.

Look and Feel

The basic look of the game is a space-inspired, vaporwave themed island, consisting only of the bowling that is taking place. Paired with the music, the game is enjoyable to play, with a various degree of complementary colours inviting the agents to play.

While implementing the movement, it was imperative to create a moving/aiming system that incorporated a challenging aspect to the simplicity of bowling (being that there was not entire control over the throwing capabilities). However, the playability of the game remains.

The models for the game (bowling ball, and pins) were made in **Blender 3.6**. See Figure 1: Simple bowling ball model in Blender and Figure 2: Simple pin model in Blender.

# Gameplay

Objectives

Ball rolls along the lane without falling off.

Knock as many pins as possible from their initial position.

Pins stop moving after the throw.

Implement reinforcement learning with an agent to perfect gameplay.

Play Flow

The game plays as follows:

The agent throws the ball. After each throw, for each pin that has been instantiated, the pin’s position and quaternion properties are analysed. See Figure 3: Checking for knocked pin and Figure 4: Checking each pin for score. For each pin that is knocked down, the game will deactivate them, so they do not interfere with the agent’s next throw. See Figure 3: Deactivating knocked pins.

Each time the ball is thrown, it will return to the player once *the ball has stopped moving*. See Figure 6: Retracting the ball. Upon returning to the player, the HandleBallStopped() method is called in the game manager. This method deals with the game logic of the throws and frames.

If the agent did not knock down all the pins on the first throw, they can then attempt to do so on the second throw. If the agent did knock all 6 pins down on the first throw, a strike bonus is granted to the agent. This creates an opportunity for the agent to reach significantly higher scores based on better performance. The pins will then reset to their initial position for the second throw, regardless of if it was a strike or not.

After the second throw, the current frame ends. This involves repeating the process for after the first throw, in addition to scoring on the *knocked pins per round,* incrementing the frame value, updating the world scene, and resetting the throw tally so the next frame can play smoothly. See Figure 7: Game logic after each throw.

Once *3 frames have been played*, the game comes to a halt, no more pins will reset, score can no longer be attained, and the end game screen pops up.

# Mechanics

Rules

Explicit Rules

The agent has control over moving, turning, and throwing. Each key required for these actions are displayed on the UI for the agent to see. See Figure 8: Basic key guide.

There is an indication to the agent whether they can throw the ball or not. See Figure 10: Throw indicator. When the ball is thrown, and until it is returned, the throw indicator explicitly tells the agent to wait, until the ball is returned. See Figure 9: Wait indication.

Implicit Rules

The agent is confined to 1 axis of translation, and 1 axis of rotation. This means that the agent can only move left and right and look left and right. As well the confinement of the movement ability, there are also restrictions to the range of movement and rotation. The agent can move entirely across the short edge of the lane and can angle from any corner on this edge to the opposite corner of the lane – at maximum range.

Each round (frame) consists of 2 throws, alike traditional bowling, giving the agent the opportunity to achieve spares, but also introducing the reward of hitting a strike on the first throw. The frame information is displayed on a screen for the agent to see.

The game is 3 frames long. This information is never stated to the agent; by playing the game (all 3 frames), the game will end, and the agent finishes with a score.

Physics

To achieve naturality, the intensity of gravity was manually edited in the project’s settings. This change was small (from -9.81 to -10), but it made a significant difference in the collisions between objects.

In addition to the gravity being altered, the objects in the game world also have default properties such as ‘mass’, ‘linear drag’, and ‘angular drag’. These values were fine-tuned for the result of a natural-looking interaction between the objects. This included creating a high mass, low drag bowling ball, and low mass, high drag bowling pins.

Objects

Bowling Ball

Being the only object that the agent interacts with directly, it should inherently be simple to use. This is attempted by displaying to the agent when they can throw, see Figure 10: Throw indicator, and creating a path for the ball to takes once it has stopped, that takes the balls end position and moves it from there towards its original position. Instead of deactivating the game object, this design choice was made to keep the agent drawn into the game, with such an interesting mechanic. With this too, it allows for extensibility of the game.

When implementing this mechanic, ray-casting played a crucial role. In this, a plane (the ray receiver) is used to receive rays drawn from the camera to where the mouse is pointing. Given that the mouse is fixed to the centre of the screen, the rays are drawn straight forward, firing the ball as intended every time. See Figure 11: Ray receiver and Figure 12: Ray casting.

When “rolling” the bowling ball, a velocity is applied to its rigid body, in the direction according to where the agent is facing, and with a power correlating to the charge that the agent gave it. See Figure 13: Launching the ball. This is a simple mechanic that will be explain further later.

Pin

The pins are crucial to the game of bowling, constituting for 2 of the 3 main objectives. The pins play an important role in many aspects of bowling: set up, knock down, give score, and deactivate.

The set up for the pins is a simple triangular formation identical to that of 6-pin bowling. See Figure 14: Pin set up. This is done programmatically using a simple loop that places each pin in a position based on the row and column. See Figure 15: Resetting pins.

When the pins are set up, a list is created that contains each pin, its position, and quaternion. This list is responsible for dealing with the knocked down pins, scoring on them and deactivating them when necessary.

Actions

Charging a throw

To throw a bowling ball, the agent must press the *spacebar*. However, upon pressing (and holding) the *spacebar*, a charge-up is initiated, and continues to charge until the agent has released the *spacebar*. This is represented to the agent as a slider, which slides from left (weakest) to right (strongest) when the agent presses space until they release it. See Figure 16: Throw charge slider.

In this mechanic, there is a minimum and maximum charge range created so that the physics of the game are not exploited. This prevents altered gameplay that would otherwise allow for broken mechanics (ball too fast) or faulty game flow (ball too slow). The values were tuned relevantly. See Figure 17: Handling charged throws.

Curving a ball

While this implementation does not currently work, all necessary components are in place. There is a slider that indicates the direction of curve the agent intends to apply, and the launch method (Figure 13: Launching the ball) can be modified to accept spin. This would then apply a force in the direction of spin, over time, creating a curving effect.

The handle on the slider originally starts in the centre of the slider. This indicates to the agent that there is no spin applied. The handle being more left or more right suggests the direction the ball should curve yet this implementation has not been finalised. See Figure 18: Accepting spin inputs.

The agent can interact with this curve slider with “Z” (left curve) and “C” (right curve). See Figure 19: Curve slider.

Screen Flow

Featured in this game are 2 separate canvases placed on “billboards”. These represent information relevant to the game that could be deemed as important to the agent for knowledge of performance or current game status. See Figure 20: Canvases on billboards. Included in this is: the score, a throw indicator, the current frame, and a video player that plays videos when the agent hits a strike on the frames first throw.

The information on the billboards is updated according to the following circumstances:

Score: The current frame has ended, or the agent hit a strike on the frames first throw.

Throw: The agent can throw the ball, or the agent cannot throw the ball.

Frame: The game is on the next frame.

Video Player: The agent hit a strike on the frames first throw.

The GameManager increments the score by the *knocked pins per round* accurately, with a strike bonus awarding the agent 2 extra points. See Figure 21: Strike bonus. On this occasion, the video player plays a short animation in response to their strike. See Figure 22: Strike animation. References to the UIController are made to deal with each of the UIs for the billboards. See Figure 7: Game logic after each throw.

AI Implementation

The implementation failed to be implemented due to challenges within the application of reinforcement learning . The following is a detailed outline of the implementation that would have taken place and the results expected to be seen. The choice of algorithm is Deep Q-Networks. This algorithm is an advanced form of the traditional Q-learning algorithm, enhanced with deep learning to handle the complex environments with high-dimensionality input spaces. Here’s why DQN is a particularly suitable choice for implementing AI in this bowling game:

Handling high-dimensionality state spaces: Traditional Q-learning struggles with high-dimensionality state spaces because it relies on a tabular approach, where each state-action pair needs to be stored and updated. DQNs, however, use a neural network to approximate the Q-value function, making it feasible to handle complex and high-dimensionality inputs efficiently.

Generalisation over states: DQNs excel in generalising learning across similar states. In the context of bowling, many situations recur with slight variations, such as different pin configurations after and initial throw. A DQN can learn to recognise these patterns and apply learned strategies effectively, improving its performance faster than traditional methods.

Robustness to varied gameplay dynamics: The DQN’s ability to continuously update its neural network allows for it to adapt to the changing dynamics of the game, optimising its strategy as it gathers more gameplay data. This is crucial for maintaining high performances as the game progresses and scenarios change.

Temporal difference learning: DQNs implement a version of TD learning, which allows the network to learn from incomplete sequences of gameplay, updating its estimates of Q-values based on the difference between predicted Q-values and the observed rewards. This feature is particularly useful in bowling, where the outcome of actions is quickly apparent, allowing rapid updates to the strategy.

Experience replay: DQN uses experience replay, where they store the agent’s experiences at each time-step in a memory-buffer. Random batches from this buffer are used to train the neural network. This approach helps in breaking the correlation between consecutive samples, leading to more stable and efficient learning. It is particularly beneficial when the agent can learn from a diverse set of situations by revisiting old experiences.

Target network: DQNs utilise a concept known as the target network to provide stable targets during training updates. This network is a clone of the value-estimating network but is updated less frequently to stabilise the learning updates. When the required outcomes do not changes, this stability is crucial for consistent learning and performance.

The solvability of the game through AI revolves around training an AI agent to develop optimal strategies for knocking down pins. The AI learns to solve the game by interacting with the environment, receiving feedback through rewards, and refining its strategies based on this feedback. See Figure 25: Creating the Agent.

The state space defines the inputs that the AI can observe. This includes the following:

Position and orientation of pins.

Position and velocity of bowling ball.

Number of pins knocked in the previous frames.

Current score and previous scores.

These variables provide the AI with comprehensive awareness, allowing it to assess the state of the game at any moment.

The action space define the actions that the AI can take. This includes the following:

Angle and force with which to roll the ball.

Adjustment for spin based on lane conditions.

These actions are compiled into a finite set of choices the AI can select from, allowing it to experiment with different throwing techniques.

The reward system trains the AI effectively. Rewards are given for knocking down pins, with higher rewards for strikes. The reward function is structured as follows:

A high reward for strikes.

A moderate reward for knocking 6 pins in 1 frame.

A small reward for knocking less than 6 pins in 1 frame.

Penalties for balls that fail to knock any pins down.

See Figure 26: Defining a Reward Function.

Using a reinforcement learning algorithm, the AI learns over multiple episodes of gameplay. Each episode involves:

Observing the current state of the environment.

Choosing and executing an action based on its current policy.

Observing the new state and receiving a reward.

Updating the policy is based on the reward and the transition from the old state to the new state. The learning process adjusts the AI’s policy so that it becomes increasingly effective at predicting which actions will yield the highest cumulative rewards.

The AI undergoes extensive training, where it plays thousands of bowling frames. During training, it explores various strategies:

Exploration: Trying random or less-tried actions to discover potentially superior strategies.

Exploitation: Leveraging known strategies that have previously resulted in high rewards.

This algorithm uses a reinforcement training model where the sensor to fire a ball using parameters firePower, fireAngle and firePosition based on the settings learned from previous episodes. It rewards the algorithm based on the number of pins that are knocked down, so the agent will learn to knock as many pins as possible to maximise this value per episode.

Exploitation:

Once the AI learns that certain angles and forces tend to knock down more pins, it will prefer those actions in similar future states. This includes repetitively aiming in a position, with the same force to continuously achieve strikes.

The AI also learns to adjust its strategy based on the specific arrangement of pins left after the first throw, choosing the optimal second throw that maximises pin knockdown.

Exploration:

Occasionally, the AI will choose actions at random instead of following the best-known strategy. This includes trying different angles or spins that it has not used frequently.

The AI can systematically vary its actions around those that are known to be effective to discover if slight modifications may lead to better outcomes. For example, if rolling the ball straight down the middle often works well, what happens if it’s slightly to the left or right?

Initially, when the AI has limited knowledge, it relies heavily on exploration to build its understanding of the game. This involves playing numerous frames where the AI randomly selects the angle and power of each throw. The AI uses an ε-greedy strategy (where ε denotes the probability of selecting a random action), and it gradually decreases, allowing more exploitation of the learned policy as the AI becomes more confident in its decision-making.

The training continues until the AI’s performance stabilises, indicating that it has converged on an effective strategy. The final policy allows the AI to consistently achieve high scores, demonstrating that it has “solved” the game by learning how to knock down pins effectively.

# Game World

General Look and Feel of World

The game world is a combination of simple shapes and simple colours. The colours are complementary, and the shapes are basic. This helps to maintain an inviting environment for the agent. See Figure 22: Look of the game. The world feels open, being that there is only a lane and the agent. The environment endorses and entices the agent to play. This is aided by the “scrolling” effect added to the lane that starts at the agent and ends at the pins. This design choice is purely to draw the agent’s attention towards the pins, and it does not provide any boost in velocity to the ball.

The “scrolling” effect implemented was a simple task of designing a tile (a design that can be placed iteratively seamlessly) and offsetting the texture with respect to time via script. See Figure 24: Implementing a scrolling effect.

The transitions between scenes are intended to be smooth, designing each scene to be a variation of the world the game is played in.

Levels

Currently in the game, there is only 1 level implemented. This level plays for 3 frames. For each frame, the agent has 2 throws. The agent’s goal is to score as many points as possible in the 3 frames. Upon finishing the game, the score is presented to the agent on the end screen.

# A blue and white marbled ball Description automatically generatedAppendix

A white and red bowling pin

Description automatically generated

Figure 1: Simple pin model in Blender

Figure 2: Simple bowling ball model in Blender

public bool IsKnockedDown**()**

**{**

**return** transform**.**position**.**y **<** startPosition**.**y **-** **0.1**f **||** Quaternion**.**Angle**(**transform**.**rotation**,** startRotation**)** **>** **45;**

**}**

Figure 3: Checking for knocked pin

private int CountKnockedDownPins**()**

**{**

**return** pinControllers**.**Count**(**pc **=>** !pc**.**gameObject**.**activeSelf **||** pc**.**IsKnockedDown**());**

**}**

Figure 4: Checking each pin for score

private void DeactivateKnockedDownPins**()**

**{**

foreach **(**var pinController **in** pinControllers**.**Where**(**pc **=>** pc**.**IsKnockedDown**()))**

**{**

pinController**.**gameObject**.**SetActive**(**false**);**

**}**

**}**

Figure 5: Deactivating knocked pins

private void Update**()**

**{**

**if** **(**isLaunched **&&** !isRetracting **&&** IsBallStopped**(**rb**))**

**{**

StartRetraction**();**

**}**

**}**

private bool IsBallStopped**(**Rigidbody ballRigidbody**)**

**{**

**return** ballRigidbody**.**velocity**.**magnitude **<** **8**f **&&** ballRigidbody**.**angularVelocity**.**magnitude **<** **8**f**;**

**}**

public void StartRetraction**()**

**{**

**if** **(**!isRetracting**)**

**{**

rb**.**isKinematic **=** true**;** **//** Disable physics **while** retracting

GetComponent**<**Collider**>().**enabled **=** false**;** **//** Disable the collider to prevent hitting pins on the way back

StartCoroutine**(**Coroutine\_RetractBall**());**

**}**

**}**

private IEnumerator Coroutine\_RetractBall**()**

**{**

isRetracting **=** true**;**

**while** **(**Vector3**.**Distance**(**transform**.**position**,** endPosition**)** **>** **0.01**f**)**

**{**

transform**.**position **=** Vector3**.**MoveTowards**(**transform**.**position**,** endPosition**,** retractionSpeed **\*** Time**.**deltaTime**);**

**yield** **return** null**;**

**}**

**yield** **return** new WaitForSeconds**(2);**

isRetracting **=** false**;**

isLaunched **=** false**;**

GameManager**.**Instance**.**HandleBallStopped**();**

Destroy**(**gameObject**);**

**}**

Figure 6: Retracting the ball

public void HandleBallStopped**()**

**{**

ballInPlay **=** false**;**

Destroy**(**currentBall**);**

currentBall **=** null**;**

uiController**.**UpdateCanThrowStatus**(**true**);**

throwCount**++;**

**if** **(**throwCount **>=** **2)** **//** This logic **is** only triggered after two throws **(1** frame**)**

**{**

**if** **(**currentFrame **==** **2)**

**{**

ScoreAndUpdate**();**

EndGame**();**

**}**

**else**

**{**

uiController**.**UpdateStrikeText**(**false**);**

ScoreAndUpdate**();**

currentFrame**++;**

uiController**.**UpdateFrameDisplay**(**currentFrame**);**

throwCount **=** **0;** **//** Reset throw count **for** the next frame

**}**

**}**

**else**

**{**

**if** **(**CountKnockedDownPins**()** **==** **6)**

**{**

ScoreAndUpdate**();**

scoreManager**.**AddScore**(2);**

uiController**.**UpdateStrikeText**(**true**);**

uiController**.**HideFrameText**(**true**);**

PlayStrikeVideo**();**

**}**

**else**

**{**

DeactivateKnockedDownPins**();**

uiController**.**UpdateStrikeText**(**false**);**

**}**

**}**

**}**

Figure 7: Game logic after each throw

Figure 8: Basic key guide

A pink sign with green and blue text

Description automatically generated

Figure 9: Throw indicator

A pink sign with black text

Description automatically generated

Figure 10: Wait indication

A grid with a grid and a pencil

Description automatically generated with medium confidence

Figure 11: Ray receiver

private Vector3 CalculateLaunchDirection**()**

**{**

Ray ray **=** Camera**.**main**.**ScreenPointToRay**(**Input**.**mousePosition**);**

**if** **(**Physics**.**Raycast**(**ray**,** out RaycastHit hit**,** Mathf**.**Infinity**,** **1** **<<** LayerMask**.**NameToLayer**(**"RayReceiver"**)))**

**{**

Vector3 launchDirection **=** **(**hit**.**point **-** spawnPoint**.**position**).**normalized**;**

**return** launchDirection**;**

**}**

**else**

**{**

Debug**.**Log**(**"Not aimed at a valid target."**);**

**return** Vector3**.**zero**;** **//** Return zero vector **if** no valid direction **is** found

**}**

**}**

Figure 12: Ray casting

public void Launch**(**Vector3 direction**,** float power**)**

**{**

GetComponent**<**Collider**>().**enabled **=** true**;** **//** Re**-**enable the collider

rb**.**isKinematic **=** false**;**

rb**.**constraints **=** RigidbodyConstraints**.**None**;** **//** Set constraints accordingly

**//** Apply the direction **and** scale it by the power **for** the linear velocity

rb**.**velocity **=** direction**.**normalized **\*** power**;**

isLaunched **=** true**;**

**}**

Figure 13: Launching the ball

A video game of bowling pins

Description automatically generated

Figure 14: Pin set up

public void ResetAllPins**()**

**{**

ClearCurrentPins**();** **//** Clear existing pins before setting up new ones

int pinCount **=** **0;**

**for** **(**int row **=** **0;** row **<** **3;** row**++)**

**{**

**for** **(**int col **=** **0;** col **<=** row**;** col**++)**

**{**

float posX **=** **(**col **-** row **\*** **0.5**f**)** **\*** **2**f**;**

float posZ **=** row **\*** **2**f**;**

Vector3 positionOffset **=** new Vector3**(**posX**,** **0,** posZ**);**

Vector3 pinPosition **=** pinStartPosition**.**position **+** positionOffset**;**

GameObject pinInstance **=** Instantiate**(**pinPrefab**,** pinPosition**,** Quaternion**.**identity**,** transform**);**

pinInstance**.**transform**.**localScale **=** pinScale**;**

currentPins**.**Add**(**pinInstance**);** **//** Track the instantiated pin

var pinRb **=** pinInstance**.**GetComponent**<**Rigidbody**>();**

**if** **(**pinRb **!=** null**)**

**{**

**//** Temporarily disable physics

pinRb**.**isKinematic **=** true**;**

**}**

pinCount**++;**

**if** **(**pinCount **>=** **6)** **break;** **//** Stop after **6** pins are instantiated

**}**

**if** **(**pinCount **>=** **6)** **break;** **//** Stop after **6** pins are instantiated

**}**

**//** Reset pin positions

foreach **(**GameObject pin **in** currentPins**)**

**{**

var pinRb **=** pin**.**GetComponent**<**Rigidbody**>();**

**if** **(**pinRb **!=** null**)**

**{**

**//** Re**-**enable physics

pinRb**.**isKinematic **=** false**;**

**}**

pin**.**transform**.**position **=** pinStartPosition**.**position**;**

pin**.**transform**.**rotation **=** Quaternion**.**identity**;**

pin**.**SetActive**(**true**);** **//** Make sure pins are active

**}**

**}**

Figure 15: Resetting pins

A screenshot of a video game

Description automatically generated

Figure 16: Throw charge slider

void Update**()**

**{**

**if** **(**!ballInPlay **&&** currentBall **==** null**)**

**{**

**if** **(**Input**.**GetKeyDown**(**KeyCode**.**Space**))**

**{**

isCharging **=** true**;**

chargeAmount **=** **0**f**;** **//** Start charging

uiController**.**ShowChargeSlider**(**true**);** **//** Show the slider

**}**

**else** **if** **(**isCharging **&&** Input**.**GetKey**(**KeyCode**.**Space**))**

**{**

chargeAmount **+=** Time**.**deltaTime **\*** **50;** **//** Increment charge amount over time

chargeAmount **=** Mathf**.**Min**(**chargeAmount**,** maxCharge**);** **//** Cap charge at maxCharge

uiController**.**UpdateChargeSlider**(**chargeAmount **/** maxCharge**);** **//** Update the slider value

**}**

**else** **if** **(**isCharging **&&** Input**.**GetKeyUp**(**KeyCode**.**Space**))**

**{**

isCharging **=** false**;**

LaunchBall**(**chargeAmount **/** maxCharge**);** **//** Launch the ball **with** normalised charge value

uiController**.**ShowChargeSlider**(**false**);** **//** Hide the slider

chargeAmount **=** **0**f**;** **//** Reset charge amount

**}**

**}**

**}**

Figure 17: Handling charged throws

private void HandleSpinInputs**()**

**{**

const float spinIncrement **=** **1**f**;**

const float maxSpin **=** **6**f**;**

**if** **(**Input**.**GetKeyDown**(**KeyCode**.**Z**))**

**{**

leftSpin **=** Mathf**.**Clamp**(**leftSpin **+** spinIncrement**,** **0**f**,** maxSpin**);**

rightSpin **=** Mathf**.**Clamp**(**rightSpin **-** spinIncrement**,** **0**f**,** maxSpin**);**

**}**

**else** **if** **(**Input**.**GetKeyDown**(**KeyCode**.**C**))**

**{**

rightSpin **=** Mathf**.**Clamp**(**rightSpin **+** spinIncrement**,** **0**f**,** maxSpin**);**

leftSpin **=** Mathf**.**Clamp**(**leftSpin **-** spinIncrement**,** **0**f**,** maxSpin**);**

**}**

**//** Update the UI slider **with** the normalised spin difference

float SpinDifference **=** **(**rightSpin **-** leftSpin**)** **/** maxSpin**;**

uiController**.**UpdateSpinSlider**(**SpinDifference**);**

**}**

Figure 18: Accepting spin inputs

A purple and white stripe with a white circle

Description automatically generated

Figure 19: Curve slider

A pink and black striped ball

Description automatically generated

Figure 20: Canvases on billboards

A pink sign with black text

Description automatically generated

Figure 21: Strike bonus

Figure 22: Look of the game



A drawing of a bowling pin

Description automatically generated

Figure 23: Strike animation

void Update**()**

**{**

**//** Calculate the new offset position based on time **and** speed

float newOffsetY **=** Mathf**.**Repeat**(**Time**.**time **\*** scrollSpeed**,** **1);**

**//** Set the new offset to the material to create a scrolling effect

Vector2 newOffset **=** new Vector2**(0,** newOffsetY**);**

runwayMaterial**.**SetTextureOffset**(**"\_MainTex"**,** newOffset**);**

**}**

Figure 24: Implementing a scrolling effect

Class Agent**{**

Int ballPower**;**

Float ballAngle

Vector3 agentPosition

GameObject**[]** Pins

Vector3 ballPosition

Void CollectObservations**(**sensor**){**

Sensor**.**AddObservation**(**ballPower**)**

Sensor**.**AddObservation**(**ballAngle**)**

Sensor**.**AddObservation**(**agentPosition**)**

**}**

Void OnActionRecieved**(**action**)**

Int firePower **=** actions**.**DiscreteActions**[0]**

Int fireAngle **=** actions**.**DiscreteActions**[1]**

Int firePosition **=** actions**.**DiscreteActions**[2]**

fireBall**(**firepower**,**fireAngle**,**firePosition**)**

**}**

Void OnPinsCollision**(){**

If **foreach** Pin**(**collision **==** **true){**

SetReward**(1f)**

**}**

**}**

Void OnBallEnd**(){**

If **(**ballPosition **==** endOfTheLane**){**

EndEpisode**()**

**}**

**}**

**}**

Figure 25: Creating the Agent

**public** **float** CalculateReward**(bool** strike**,** **bool** spare**,** **int** pinsKnockedDown**)**

**{**

**float** reward **=** **0f;**

**if** **(**strike**)** reward **+=** **10f;** *// High reward for strike*

**else** if **(**spare**)** reward **+=** **5f;** *// Moderate reward for spare*

reward **+=** pinsKnockedDown **\*** **0.1f;** *// Incremental reward for each pin knocked down*

**return** reward**;**

**}**

Figure 26: Defining a Reward Function