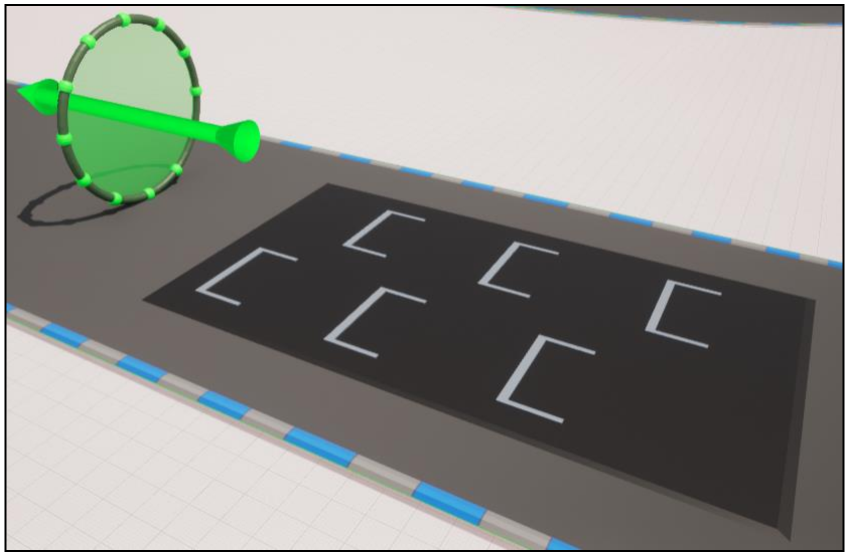
Game Engine Architecture: Race Report

Oscar Wilkinson
Department of Computer Science and Creative Technologies
University of the West of England
Coldharbour Lane
Bristol, UK
Oscar2.Wilkinson@live.uwe.ac.uk
21007294



Abstract

Every racing game, with no impact of the game engine, has the same race mechanic and is considered a requirement. The mechanic frequently has the exact same components for each game's implementation, the only difference is how each component is managed. This report will go through two separate engine versions of this concept and compare their similarities and differences and highlight how each engine has its own strengths and weaknesses.

Author Keywords

Unity; Unreal Engine 5; Game Manager; Custom Editor; UI

Introduction

This implementation is to construct a suite of objects that can be placed into the world and create the game logic of a race in two game engines, which are Unity and Unreal Engine 5. The race must include checkpoints to function as a route and respawn anchor, starting/finishing lines to function as the logic for the beginning and end of the race and a lap count that controls how long every contender must go around the entire race. These components can be easily implemented into any racing game to either compete player vs player, player vs AI or player vs time and can be further tweaked to suit other bespoke mechanics.

Background & Research

To be able to construct this package of components, first other implementations and games need to be evaluated and broken down to understand how the most advanced companies approach this scenario.

The main example this report will follow is the races in the game Banjo Kazooie: Nuts & Bolts, so it will be addressed first.

Banjo Kazooie: Nuts & Bolts

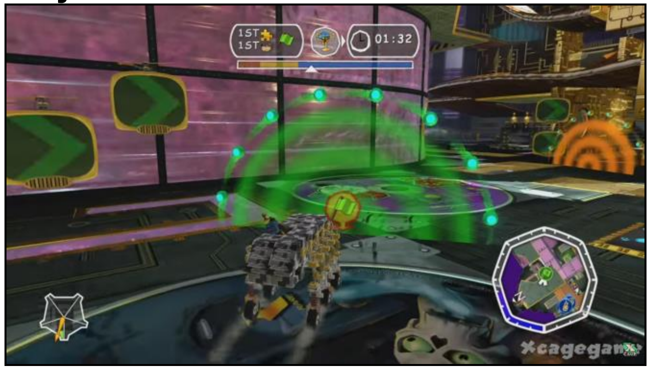


Figure 1: Race in Banjo Kazooie N&B

It can be seen from gameplay and screenshots that the game consists of a few components. The dynamic of the race is that the contenders go through visual checkpoints to progress the race, and the race ends for the contender once they reach the finishing ring with all their laps completed. There is a manager which maintains connections with other components to piece together the race. There are individual actors which provide little functionality but communicate with the manager to provide information about the race, one actor example being the rings shown in Figure 1 which, when the player collides with, activates the next ring. Then there are external components that are the primary influencers of the race which consist of the players and AI's.

Mario Kart 8 Deluxe:

Figure 2: Race in Mario Kart 8 Deluxe

Every Mario Kart game follow very similar rules and mechanics and add slight differentiation between each game. But the general system implemented is that the players go through invisible gates that act as checkpoints that all add up to construct the layout of the race. There are multiple routes but all lead to the same finishing point and ends for the player if they have passed all the laps.

Both Mario Kart 8 Deluxe and Banjo Kazooie: Nuts & Bolts follow the same concepts so it can be concluded that an efficient method to construct this is to have multiple individual actors (e.g. rings) that are controlled by one single manager that is affected by external actors (e.g. cars).

Implementations

The version of Unity used is 2022.1.16f released September 2022, allowing for a more accurate demonstration of modern day game engines.

The Unity Editor is an object-oriented-programming form of game developing software that has a hierarchy of game objects given scripts and classes to act out functions. The scripts are designed in C# type script that can be developed in any C# based integrated-development-environment, the IDE used for this implementation was Visual Studio 2019.

First Iteration

Upon starting a new project with Unity, the user is provided multiple pre-built templates to choose from. This ranges between 2D, 3D, virtual reality, augmented reality, platformer and so on. There is a template for karting, but this version is started with the 3D template as I wanted to discover the feasability of implementing a player from the beginning.

The first steps towards the race mechanic was to create a prefab of a ring, what will be considered the checkpoints of the race and dictate the shape of the map.

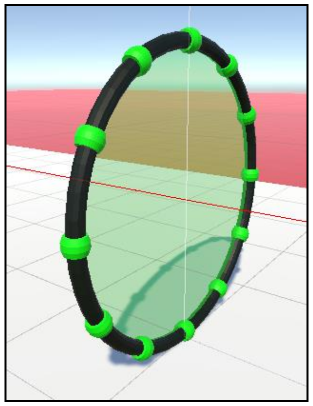


Figure 3: Ring gameobject

The ring needed to contains variables to dictate what type of ring it is, what rings are after it, what rings were before it and a toggle to state what movement the player is transformed to upon colliding with it.

When a contendor collides with a ring, the ring needs to then process multiple scenarios. These include: needing to change what the contendors next ring is to progress in the race and display it to that contendor, what movement type the contendors now has access to, and if the ring is an end ring deciding whether they contine on to the next lap, or finish the race. Other small functions are changing it’s respawn position and rotation and changing the visual state of the current ring.

The final feature implemented with the rings only applies whilst the game scene is being editted and not played. Each ring draws a debug line to their next ring(s) which allows the developer to show the complete path of the race and discover any breaks.

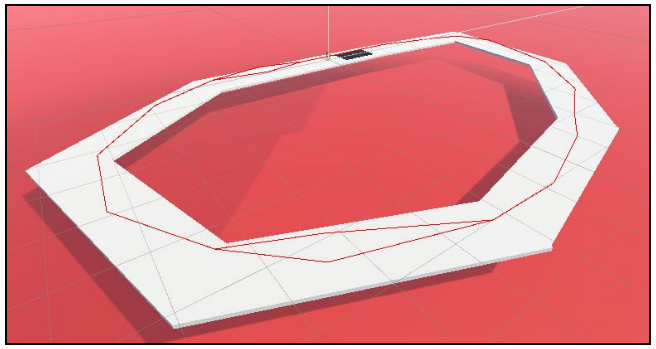
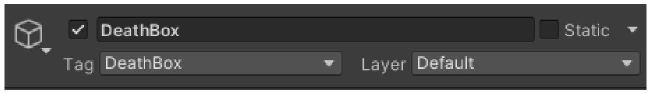


Figure 4: Full race shown with custom editor lines

The next prefab that was started on is the movement for the player, allowing testing of the races mechanics. It contained simple movement of forward and backwards, and rotation of left and right, the only other function was to respawn upon colliding with any gameobject that has the tag "DeathBox", reseting their position and rotation to the last checkpoint reached.



DeathBox Static
Tag DeathBox Layer Default

Figure 5: Game object with the tag "DeathBox"

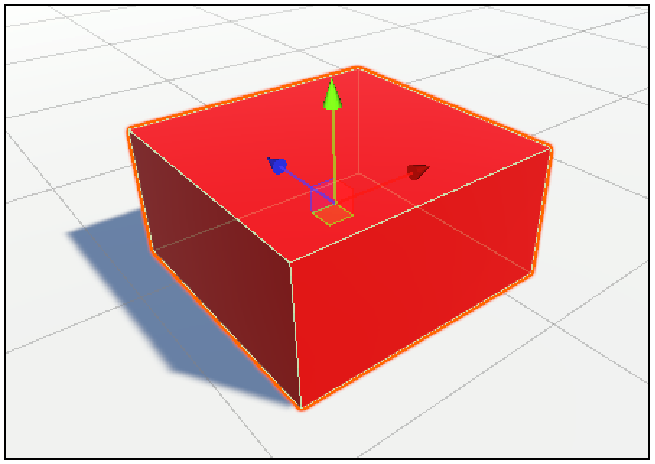


Figure 6: Player character

Then was generating the player into the race, which was decided by the prefab of the start grid.

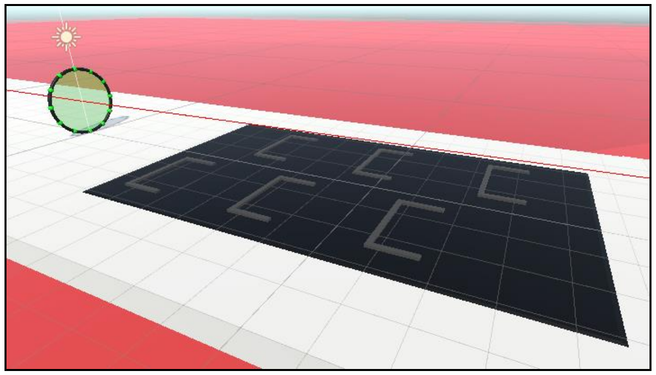
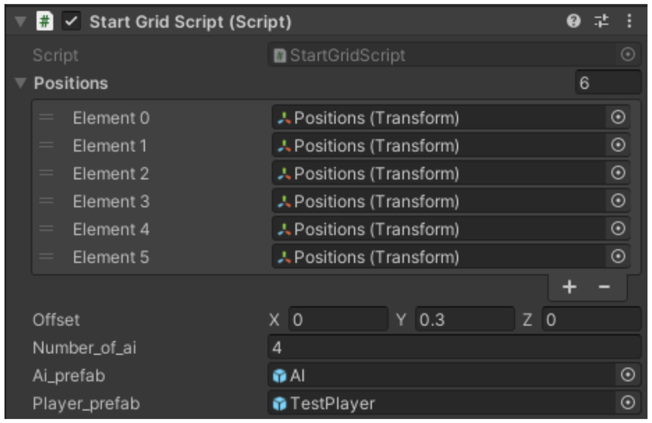


Figure 7: Start grid with starting ring

It is collection of empty gameobjects that state the possible starting positions of the race and randomly generate the desired number of players from front to back.


Start Grid Script (Script)

Script StartGridScript

Positions 6

Element 0 Positions (Transform)
Element 1 Positions (Transform)
Element 2 Positions (Transform)
Element 3 Positions (Transform)
Element 4 Positions (Transform)
Element 5 Positions (Transform)

Offset X 0 Y 0.3 Z 0

Number_of_ai 4

Ai_prefab AI

Player_prefab TestPlayer

Figure 8: Inspector showing public variables of the start grid

To finalise this iteration of the whole race mechanic and start testing, the race manager was created. This manager will piece each component together to act as the bridge for those components and cause the race to function as a whole.

The first bridge that the manager makes is between the rings and players. It will update the rings active state based on the players next ring and check the lap count when finishing the lap.

Second Iteration

Next the players needed competition, so adding AI competitors solved this problem. They have the same movement capabilities as the player but the only difference is they need to be controlled without any input. Their movement follows the logic that, if the next ring is to their left by a certain angle, it will rotate left and move forward at the same time, if it is to their right by a certain angle, it will rotate right and move forward, but if the angle is between the two, they will only move forward.

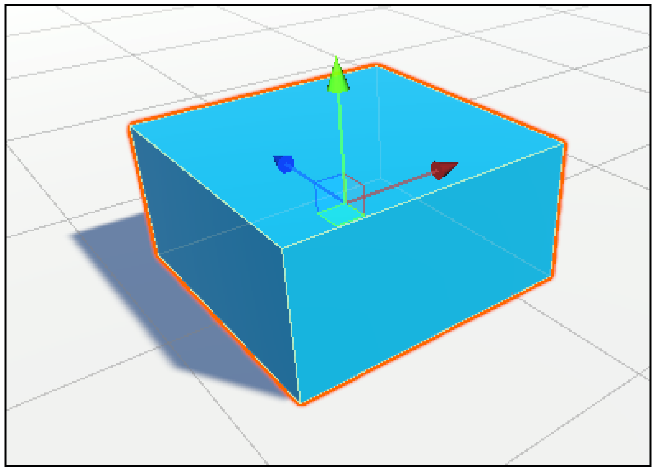


Figure 9: AI character

The only difficult change between AI and player is that the AI has their next rings managed within their own script, whereas the player has it controlled by the race manager. The AI is allocated a random possible next ring upon reaching their desired ring to simulate random behaviour, otherwise every AI contender would choose either the same route each time or the most optimal route each time.

Another small change with this iteration was the toggle of being able to move upon completing the race, which will be improved to show a leaderboard to the UI.

Third Iteration

To make the system feel like a race, the next step was to generate the UI for the player to show how well they are compeating.

The UI needs to consist of: a text box to indicate the players position, a text box to track how many laps have been passed, an image to follow where the next checkpoint is and a mini map to display what the track looks like ahead of the player. Resulting in the figure below.

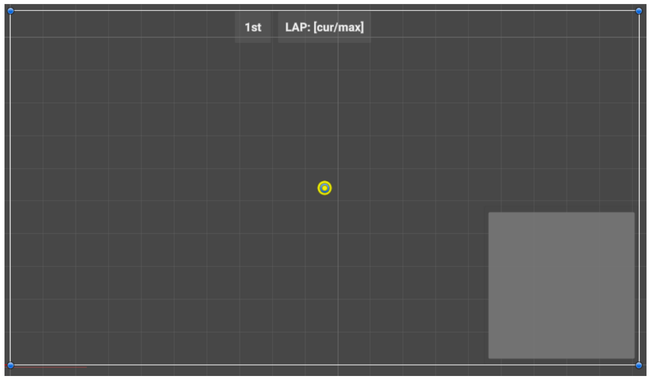


Figure 10: User interface of the race

For the UI to know the position of every player, the manager first finds each players distance to the end. This is achieved by cycling through every possible path that contender can go through to get to the next lap, tracking the distance float whilst doing so, then returning the lowest possible distance.

This distance, along with their lap count, is compared with ever player to create a list of positions, that finally gets displayed onto the UI's text box.

To show the mini map in the bottom corner of the UI a second camera with an orthographic view is parented to the player to follow them around and show a top down view of the race.

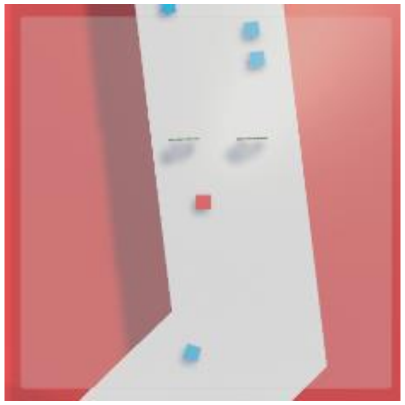


Figure 11: Minimap screenshot of the race

The final feature of this iteration was the leaderboard shown upon completing the entire race, this was controlled within another script that linked with the race manager. The information fed into the script was a list of finished players, which gets updated each time a new player finishes the race, also tracking their time of completion in a parallel list.

These two lists are then displayed in preset UI elements that present each finished contendors position, name and time.

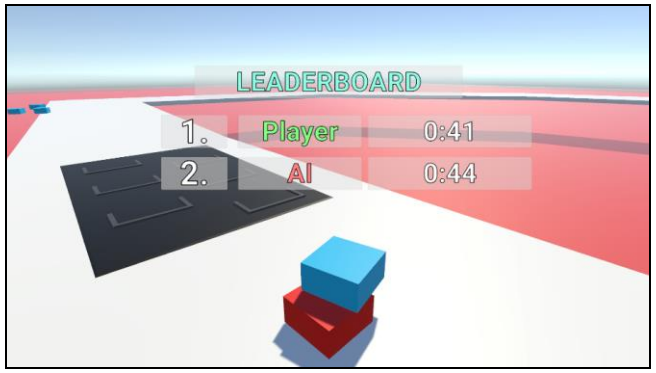


Figure 12: Leaderboard with two finished contendors

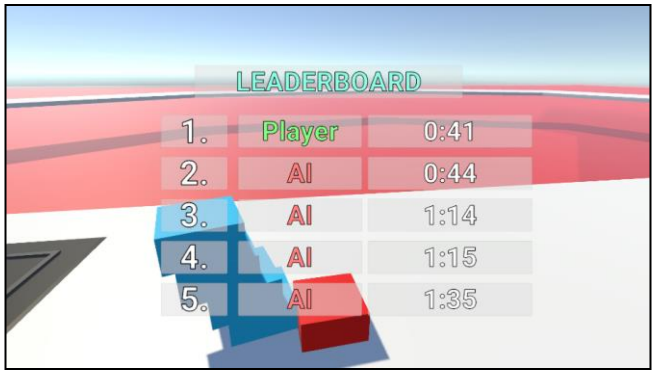


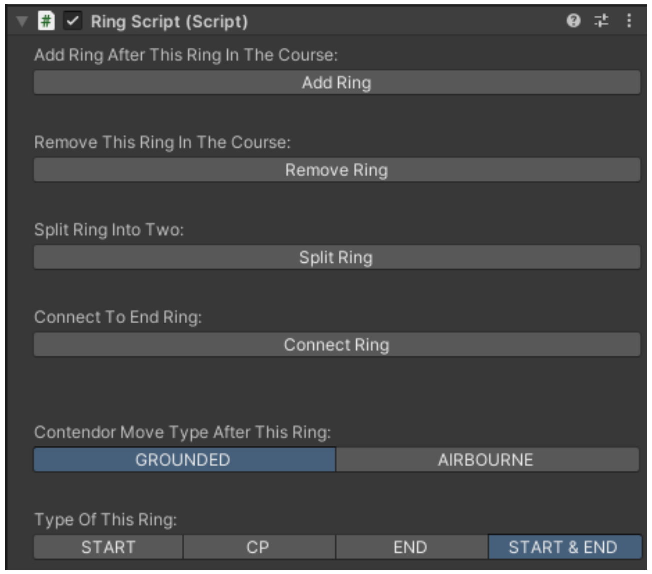
Figure 13: Same leaderboard a few seconds later with 5 finished contendors.

Final Iteration

The final iteration adds a few changes to pollish or improve other scripts.

First customising the inspector for each ring gameobject by utilising Unity's freedom of customisation with *[CustomEditor(typeof(RingScript))]*.

This allows for buttons to be introduced into the side-inspector to act out a function within the script of the rings. These functions allowed for easy development for creating the races as it is as simple as 'Add Ring' and moving it forward to the required position, repeating this, and then connecting the ring to the end, with the possibility of splitting it into multiple paths.



Ring Script (Script)

Add Ring After This Ring In The Course:
Add Ring

Remove This Ring In The Course:
Remove Ring

Split Ring Into Two:
Split Ring

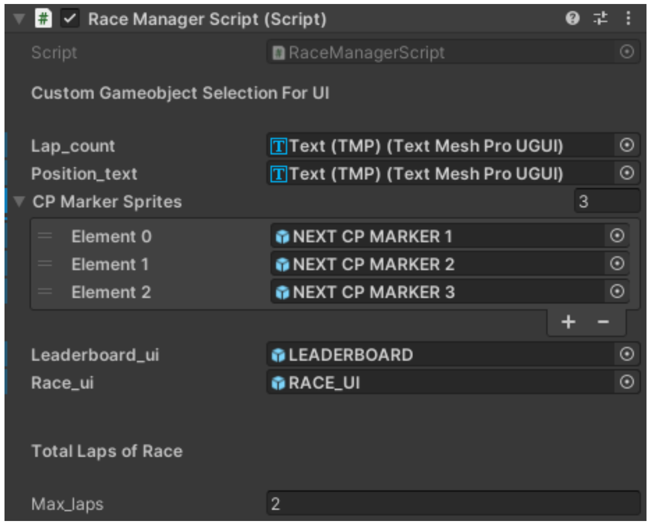
Connect To End Ring:
Connect Ring

Contendor Move Type After This Ring:
[✓ GROUNDED] AIRBOURNE

Type Of This Ring:
START CP END [✓ START & END]

Figure 14: Custom inspector for ring prefab

Without the use of the custom inspector, the inspector can show titles/headers above variables to organise them for the developers.



Race Manager Script (Script)

Script	RaceManagerScript

Custom Gameobject Selection For UI

Lap_count	Text (TMP) (Text Mesh Pro UGUI)
Position_text	Text (TMP) (Text Mesh Pro UGUI)
CP Marker Sprites	3
Element 0	NEXT CP MARKER 1
Element 1	NEXT CP MARKER 2
Element 2	NEXT CP MARKER 3
Leaderboard_ui	LEADERBOARD
Race_ui	RACE_UI

Total Laps of Race

Max_laps	2

Figure 15: Inspector editting without full customisation

Another addition created a race startup animation combining use of the camera and UI, giving the player time to prepare for the race to start.

The final package end up with the visuals of figure 14, and the low number of game objects in the scene as shown in figure 15.

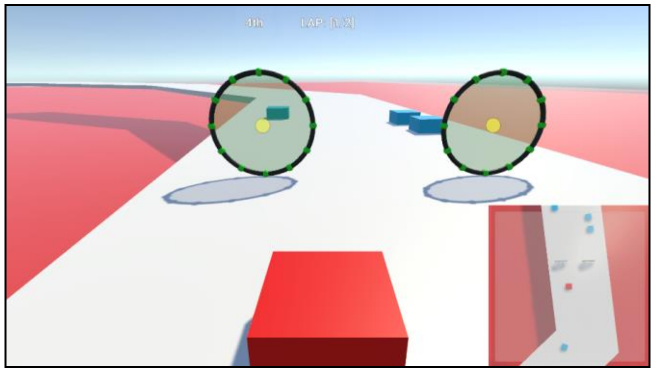
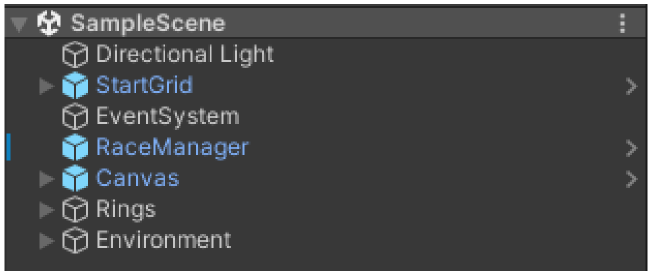


Figure 16: Screenshot of race gameplay



SampleScene
- Directional Light
- StartGrid
- EventSystem
- RaceManager
- Canvas
- Rings
- Environment

Figure 17: Game scene heirarchy

After producing this system, it would have been more time-efficient and practical to have started with the karting template instead of the 3D template as there was no necessity to create a low-level player character, however this did allow for a more customisable implementation of another AI prefab object.

The version of Unreal Engine used is 5.0.3 released on July of 2022, also showing an accurate and up to date game engine for comparison.

Unreal Engine 5 is another object-oriented-programming form of game developing software that has a hierarchy of game objects given scripts and classes to act out functions. Instead of most game developing tools, a form of visual programming is used for Unreal called blueprints, it utilizes C# functions and overall programming practice, but instead in a visual environment built into the Unreal Engine editor.

First Iteration

Unreal engine 5 provides base templates to start a project with, which varies for 1st person, 3rd person, vehicle, and virtual reality. For this project, the vehicle template was chosen which gives access to their template car blueprint, removing the process of creating a player, creating a quicker workflow.

Figure 18: Car blueprint template given with first person and third person camera.

The rings are also implemented in a similar fashion where it has: an active state Boolean, an Enum for ring type and movement type, and an array of rings after it.

The ring acts as a trigger collision system, which sends an event to any actor that collides with it under the *ActorBeginOverlap* event. The only function the rings direction undergo is every frame it checks its active state and sets its visibility accordingly. Every ring does not function as an involved manager, instead it acts as a collision box that holds data.

To make the players and rings interact the template version of the cars needed to be modified. The first modification being the *ActorBeginOverlap* event, when this is called in the player blueprint it checks the tag of the collider called, if said tag is of type 'Ring' then it progresses with the 'Hit Ring' macro (a macro being a quick function that can be called).

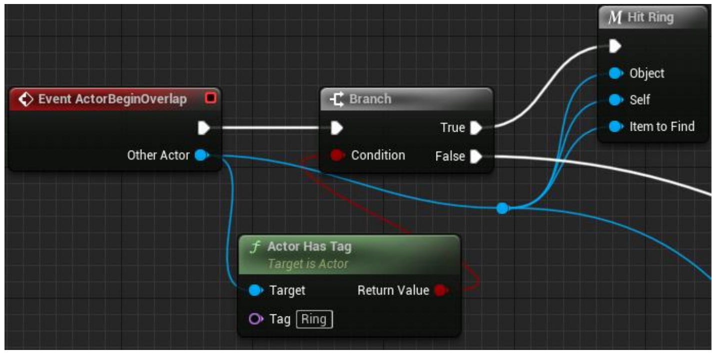


Figure 19: Blueprint of *ActorBeginOverlap* event

Within the macro multiple processes are done: the respawn position and rotation is set to the new ring, the new movement type is allocated, and then splits the function depending on the ring type. If it is a “Start” ring or “Start & End” ring, then the current lap count and max lap count are compared, ending the race if they’re equal, or continuing to the next lap if not. If the race continues, the next ring is swapped to the ones pointed to by the ring collided with.

Another feature given to the template is the collision with any actor containing the tag ‘Respawn’, which would reset their position and rotation to the last checkpoint reached, setting the velocity and acceleration to zero simultaneously.

The final blueprint required to complete this iteration is the race manager. Upon starting the race, the manager generates all required players randomly onto a preset start grid, holding 6 possible positions to start at, going from front to back, whilst also gathering all rings and contenders into an array for future use.

The other feature that the manager has is checking if a new contender has finished the race or not, appending their actor reference and completion time into an array.

Second Iteration

Next the AIs were introduced to provide a competing driver into the race. The AI class is a child class of the template car class, having the exact same functionality as the player, now with the capability of adding self-controlled driving.

When the race starts, the AI is given a random seed (integer) to dictate how random its movement will be, changing its desired path of the possible rings shown. Every tick of the game, the AI chooses their desired ring from their seed, accelerates forward if they are not at a certain velocity and turns according to the angle that their destination is compared to them.

This iteration of AI worked to an extent but would depend on the course of the race, as if it were a difficult race with multiple drops or obstacles, the AI does not have any measures to adapt to them. In addition, if at any point the car was upside down, they would not be able to move under any circumstance. To solve this, the AI would reset their rotation if they had a ’Roll’ rotation less than -120 degrees.

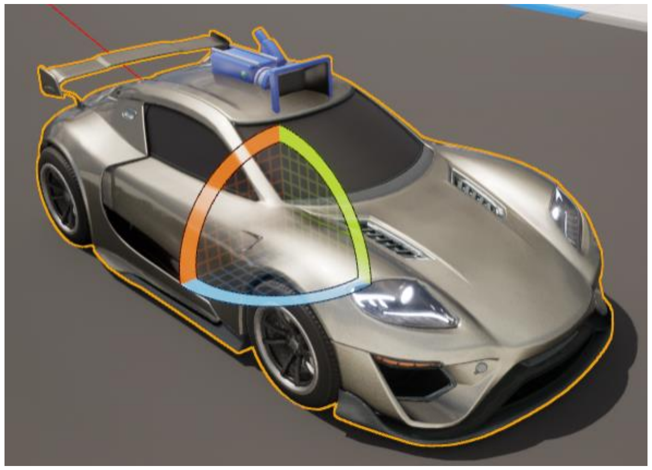


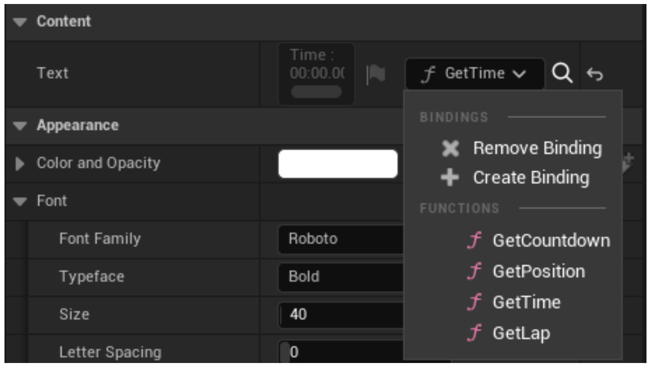
Figure 20: The rotation axis of the car object, red axis being the ‘Roll’ rotation.

Third Iteration

The next system to introduce to make the race feel more accurate to previously implemented examples is to introduce the user interface.

This is created by starting with a widget blueprint which Unreal Engine consider their UI creator and editor. The widget allows for premade classes to be dragged and dropped into the flat canvas, such as text and images.

Each text is controlled through a 'binding', essentially pointing to a function that returns a text variable.


Content

Text: Time : 00:00.0(— GetTime

Appearance

Color and Opacity

Font

Font Family: Roboto

Typeface: Bold

Size: 40

Letter Spacing: 0

Figure 21: UI Text variable binding

The text objects being controlled by these bindings are to get the time spent on the race so far, the position of said player and the lap count they are on. The only function of these that needed external assistance was the position bind. This required a change on the template car blueprint, to add a float for their distance to the end of the lap, which involved going through every possible path of rings and adding their distance up to return the lowest possible distance. With this new float, every current participant can be compared with each other in the manager blueprint to produce a list of players in order of position.

To help the player with seeing what the race looks like, a mini map is shown in the bottom right of the screen. This is an image element put on the UI canvas with a material called 'TopMaterial' which receives the RGB values from a top camera rendered in orthographic view.

This RGB sample is then compared with a circle mask which sets the opacity of anything outside the circle to 0 (transparent) and anything inside the circle to 1 (opaque) with a slight gradient at the edge of it.

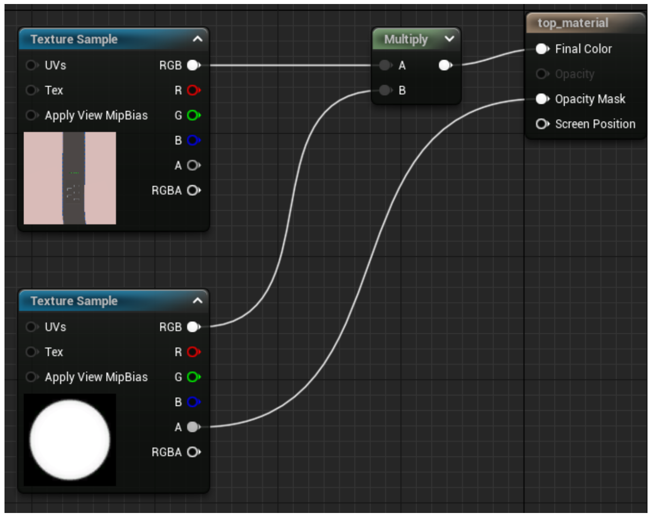


Figure 22: Material editor in Unreal Engine

To allow the players to process what the race looks like at the start, there is a countdown shown on the UI, preventing all movement of any participant, which then gets relinquished upon reaching 0.

The final UI element is the leaderboard which is another widget blueprint with only one text box controlled by a single binding. The binding gathers every player that has finished the race and displays them along with their time of completion, then gets the rest of the players playin and displays them with the current timer of the race.

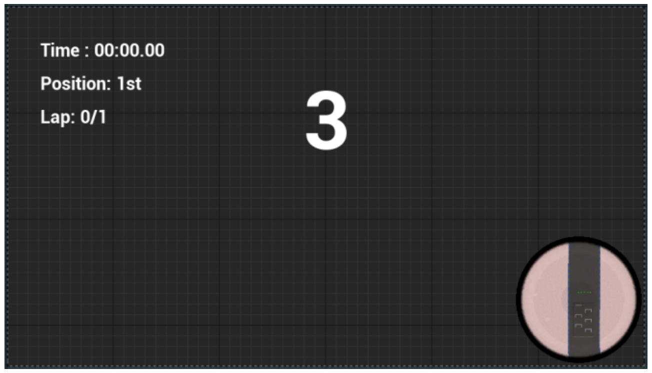
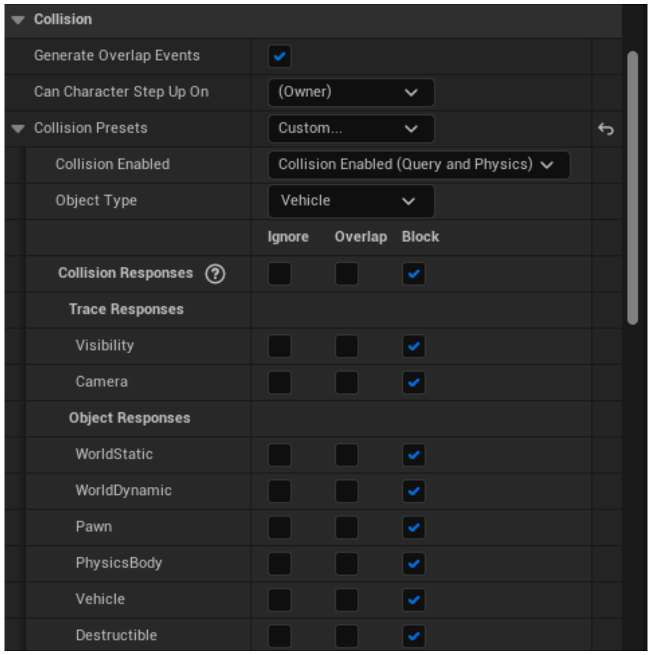


Figure 23: Final user interface (showing countdown of the race in the center)

Final Iteration

The final iteration consisted of slight bug fixes and tweaks to improve the overall functionality. The main bug to address was that when any contender would respawn at the same time as another player both their collider boxes would overlap and greatly accelerate both objects, shooting them in almost random directions. To combat this problem, the car blueprint would toggle the collision checks between this car and other cars from 'Block' to 'Overlap' which allowed them to temporarily pass-through other cars and create a timer. During this timer, any collisions with other cars would be checked to track how many cars are currently colliding with them. By the end of the timer, if there are no cars in current contact, the collision preset is then swapped back to 'Block' which gives physical contact, but if there are still cars in contact, it waits until no cars are in contact and then swaps the collision response.



Collision

Generate Overlap Events	✓		
Can Character Step Up On	(Owner)		
Collision Presets	Custom...		
Collision Enabled	Collision Enabled (Query and Physics)		
Object Type	Vehicle		
	Ignore	Overlap	Block
Collision Responses			✓
Trace Responses			
Visibility			✓
Camera			✓
Object Responses			
WorldStatic			✓
WorldDynamic			✓
Pawn			✓
PhysicsBody			✓
Vehicle			✓
Destructible			✓

Figure 24: Collision inspector for car blueprint

Evaluation

Unity had many strengths for game development and workflow. The freedom in customization is extremely broad, allowing for editing even the inspector itself, make debugging and creation of levels/scenes very efficient. With the availability of template project samples, it was faster than implementing this system with other engines as all the rendering and system programming is already completed. The readily available classes and objects made development quicker as well as there was no issue with having to develop collision systems and geometry rendering.

The system implemented in unity met all required criteria of having multiple individual actors be controlled by a single overarching manager to construct the flow of a race with lap counts and splitting paths.

However, this implementation can be developed further to suit multiple genres of races. The first features that can be developed or improved upon can be: the AI to develop more random behavior or adapt to their environment, or different movement types for every contender, such as flying or gliding.

Unreal Engine 5 also had many benefits for production and application. Every feature or aspect can be efficiently modified or developed with the wide range of functions and tools available. The engine provides every form of editor that is required, for example, it has a visual blueprint editor, a material editor and 3D model viewer built in. The templates given to the developers provided much quicker development for the system, and each system provided with the engine performed greatly, with the rendering having multiple scalability versions for low-end devices and high-end devices, producing accurate and customizable lighting.

All requirements were met with this implementation as there were separate actors controlled by a single manager to maintain connections between functions and data to produce the structure of a race.

The way that this package can be improved upon can be to expand upon the existing blueprints to direct towards a certain required race. The primary change that would assist in the most types of races would be to improve the AI so it can understand the layout of the race more to function as a more competing racer.

Both Unity and Unreal had very efficient workflows with instant access to templates to bypass the production time of generating an entire games system, however, Unreal provided more in-depth systems for each template. However, Unreal lacked in the customizability for the in-editor features, Unity allowed for freedom with functions being able to process while only in the editor, and for custom inspectors.

Unreal was slightly faster in implementing small features than in Unity, as within the drag-and-drop environment of blueprints large functions were as simple as a search, compared to Unity which requires finding the correct plugins or import to get the required function.

Unity required the user to have their own integrated-development-environment whereas Unreal did not require any external software, making Unreal easier to use and distribute with other workers.

Both systems can have their prefabs/blueprints expanded more to provide genre specific requirements. For example, they can function as collectables or equipment like Mario Kart or can go into the territory of consumers producing their own levels/race layouts for multiplayer use.

The optimal scenario in which a new object-oriented-programming based game engine is produced with the best of both Unity and Unreal would provide an engine that: has complete freedom in customization with editor functions and inspectors, instant access to a wide range of in-depth templates, quick access to all external systems to be imported or used, and every editor is provided by the engine.

Acknowledgements

Thanks to all the lecturers and staff of the University of The West of England involved in the feedback and help for the creation of this report.

References

Technologies, U. (n.d.). Unity - Manual: Prefabs. [online] docs.unity3d.com. Available at: https://docs.unity3d.com/Manual/Prefabs.html [Accessed 10 Jan. 2023].

www.youtube.com. (n.d.). Banjo Kazooie Nuts & Bolts Full Gameplay Walkthrough (Longplay). [online] Available at: https://www.youtube.com/watch?v=UjwZqdESuII [Accessed 10 Jan. 2023].

www.youtube.com. (n.d.). *Longplay of Mario Kart 8 Deluxe*. [online] Available at: https://www.youtube.com/watch?v=byjxUYfdXrY [Accessed 10 Jan. 2023].

docs.unrealengine.com. (n.d.). Blueprints Visual Scripting. [online] Available at: https://docs.unrealengine.com/5.1/en-US/blueprints-visual-scripting-in-unreal-engine/ [Accessed 10 Jan. 2023].

Technologies, U. (n.d.). Unity - Manual: Project Templates. [online] docs.unity3d.com. Available at: https://docs.unity3d.com/2021.1/Documentation/Manual/ProjectTemplates.html [Accessed 14 Jan. 2023].

docs.unrealengine.com. (n.d.). Templates Reference. [online] Available at: https://docs.unrealengine.com/5.0/en-US/unreal-engine-templates-reference/

[Accessed 14 Jan. 2023].

Unity Technologies (2019). Unity - Scripting API: ExecuteInEditMode. [online] Unity3d.com. Available at: https://docs.unity3d.com/ScriptReference/ExecuteInEditMode.html

[Accessed 14 Jan. 2023].

Technologies, U. (n.d.). Unity - Scripting API: Collider.OnTriggerEnter(Collider). [online] docs.unity3d.com. Available at: https://docs.unity3d.com/ScriptReference/Collider.OnTriggerEnter.html

[Accessed 14 Jan. 2023].

Technologies, U. (n.d.). Unity - Manual: Tags. [online] docs.unity3d.com. Available at: https://docs.unity3d.com/Manual/Tags.html

[Accessed 14 Jan. 2023].

Technologies, U. (n.d.). Unity - Manual: UI Toolkit. [online] docs.unity3d.com. Available at: https://docs.unity3d.com/Manual/UIElements.html

[Accessed 14 Jan. 2023].

Technologies, U. (n.d.). Unity - Manual: Custom Editors. [online] docs.unity3d.com. Available at: https://docs.unity3d.com/Manual/editor-CustomEditors.html

[Accessed 14 Jan. 2023].

docs.unrealengine.com. (n.d.). On Actor Begin Overlap. [online] Available at: https://docs.unrealengine.com/5.0/en-US/BlueprintAPI/Collision/OnActorBeginOverlap/

[Accessed 14 Jan. 2023].

docs.unrealengine.com. (n.d.). Macros. [online] Available at: https://docs.unrealengine.com/5.0/en-US/macros-in-unreal-engine/

[Accessed 14 Jan. 2023].

docs.unrealengine.com. (n.d.). Property Binding. [online] Available at: https://docs.unrealengine.com/5.0/en-US/property-binding-for-umg-in-unreal-engine/

[Accessed 14 Jan. 2023].

docs.unrealengine.com. (n.d.). Material Editor UI. [online] Available at: https://docs.unrealengine.com/5.0/en-US/unreal-engine-material-editor-ui/

[Accessed 14 Jan. 2023].

docs.unrealengine.com. (n.d.). Creating Widgets. [online] Available at: https://docs.unrealengine.com/5.0/en-US/creating-widgets-in-unreal-engine/

[Accessed 14 Jan. 2023].

docs.unrealengine.com. (n.d.). Collision Response Reference. [online] Available at: https://docs.unrealengine.com/5.0/en-US/collision-response-reference-in-unreal-engine/

[Accessed 14 Jan. 2023].