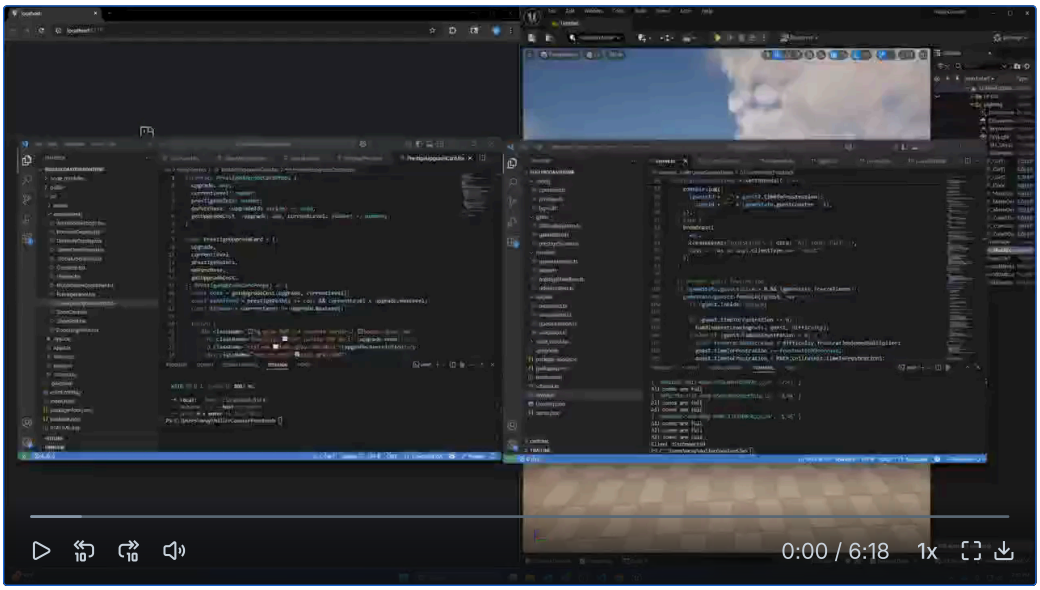
Typescript/Unreal Engine Rollercoaster Simulation

Typescript WebSocket Server



Simulation Showcase

WebSocket Theme Park Server Documentation

Table of Contents

Overview

The WebSocket Theme Park Server is a real-time single player game server that manages a theme park simulation with guests, rides, zones, and a prestige system. The server handles multiple client connections (React frontend and Unreal Engine game client) and provides real-time updates for all game events.

Key Features

- **Real-time Guest Management**: Spawn and track guests across multiple zones
- **Ride Operations**: Queue management, ride capacity, and operations
- **Prestige System**: Multi-layered progression with upgrades and milestones
- **Dynamic Difficulty**: Scaling challenges based on prestige level
- **Zone Upgrades**: Improve ride capacity, speed, and queue sizes
- **Global Upgrades**: Park-wide improvements and temporary boosts

Project Structure

```
├── config/                 # Configuration and constants
│   ├── constants.ts        # Game constants and base costs
│   ├── prestige.ts         # Prestige upgrades and milestones
│   └── types.ts            # TypeScript interfaces and types
├── handlers/               # Message event handlers
│   ├── index.ts            # Handler exports
│   ├── gameHandlers.ts     # Game mechanics handlers
│   ├── prestigeHandlers.ts # Prestige system handlers
│   └── rideHandlers.ts     # Ride operation handlers
├── game/                   # Core game logic
│   ├── gameState.ts        # Centralized game state management
│   ├── prestigeSystem.ts   # Prestige calculations and logic
│   └── difficultySystem.ts # Dynamic difficulty scaling
├── helpers/                # Utility functions
│   ├── broadcast.ts        # WebSocket broadcasting utilities
│   ├── calculations.ts     # Mathematical calculations
│   └── validation.ts       # Input validation helpers
├── utils/                  # External utilities
│   └── utils.ts            # Guest management utilities
├── schema.ts               # Data schemas (Guest, Zone, Message)
├── zones.json              # Zone configuration data
└── server.ts               # Main server entry point
```

Core Systems

1. Game State Management

The `GameState` class centralizes all game data:

```typescript
class GameState {
  public balance: number          // Player's money
  public guestCounter: number     // Total guests spawned
  public prestige: number         // Current prestige level
  public prestigePoints: number   // Available prestige points

  public readonly guests: Map<string, Guest>           // Active guests
  public readonly zones: Map<string, Zone>             // Park zones
  public readonly prestigeUpgrades: Map<string, number> // Upgrade levels
  public readonly unlockedFeatures: Set<string>       // Unlocked features
}
```

2. Prestige System

A multi-layered progression system with:

Prestige Upgrades

- **Money Boost**: Increases ride revenue (15% per level)
- **Guest Patience**: Counteracts difficulty increases (3s per level)
- **Starting Capital**: Begin with more money (75 per level)
- **Upgrade Efficiency**: Reduces upgrade costs (8% per level)
- **Damage Control**: Reduces penalties (10% per level)

Milestone Rewards

- **Bronze Tier** (Prestige 3): Automation features
- **Silver Tier** (Prestige 6): New zone unlock
- **Gold Tier** (Prestige 9): Advanced management tools

3. Difficulty Scaling

Dynamic difficulty that increases with prestige:

```
interface DifficultyScaling {
  frustrationSpeedMultiplier: number;  // Guest patience reduction
  leavingPenaltyMultiplier: number;    // Money lost multiplier
  upgradeCostMultiplier: number;       // Cost increase multiplier
}
```

4. Zone Management

Each zone has configurable properties:

- **Queue Capacity**: Maximum guests in line
- **Ride Capacity**: Guests per ride cycle
- **Ride Time**: Duration of each ride
- **Ticket Price**: Revenue per guest

API Reference

WebSocket Message Format

All messages follow this structure:

```
{
  EventType: string,
  EventTimestamp: string,
  Source: { ClientType: string },
  Data: any
}
```

Client → Server Events

Game Operations

```
// Add guests to ride
{
  EventType: "addToRide",
  Data: { guests: string } // JSON array of guest IDs
}

// Start a ride
{
  EventType: "startRide",
  Data: { zoneId: string, guests: string[] }
}

// Ride completed
{
  EventType: "rideEnded",
  Data: string // zoneId
}
```

Upgrades

```
// Zone upgrade
{
  EventType: "zoneUpgrade",
  Data: {
    zoneId: string,
    upgradeType: "reduceRideTime" | "increaseQueueCapacity" | "increaseRideCapacity"
  }
}

// Global upgrade
{
  EventType: "globalUpgrade",
  Data: {
    upgradeType: "moneyMultiplier" | "extendGuestTimersTemporary" | "freezeGuestTimers"
  }
}
```

Prestige System

```
// Purchase prestige upgrade
{
  EventType: "purchasePrestigeUpgrade",
  Data: { upgradeId: string }
}

// Reset for prestige
{
  EventType: "prestigeReset",
  Data: {}
}
```

Server → Client Events

Game State Updates

```
// Regular park data broadcast
{
  EventType: "parkData",
  Data: {
    zones: Record<string, Zone>,
    guests: Record<string, Guest>,
    balance: number,
    prestige: number,
    prestigePoints: number,
    bonuses: GameBonuses,
    difficulty: DifficultyScaling,
    unlockedFeatures: string[],
    prestigeUpgrades: PrestigeUpgrade[],
    currentUpgrades: Record<string, number>,
    prestigeRequirement: number
  }
}

// Guest spawned
{
  EventType: "guestData",
  Data: { [guestId]: string } // "zoneId,position"
}

// Guest left frustrated
{
  EventType: "guestLeft",
  Data: {
    guestId: string,
    penalty: number,
    message: string
  }
}
```

System Events

```
// Game over (negative balance)
{
  EventType: "gameOver",
  Data: {}
}

// Prestige milestone unlocked
{
  EventType: "milestoneUnlocked",
  Data: {
    milestone: MilestoneReward,
    newFeatures: string[]
  }
}
```

Configuration

Constants (`config/constants.ts`)

```
export const PORT = 8080;
export const GUEST_SPAWN_INTERVAL = 1000;
```

```
export const BASE_COSTS = {
  REDUCE_RIDE_TIME: 150,
  INCREASE_QUEUE_CAPACITY: 300,
  INCREASE_RIDE_CAPACITY: 350,
  MONEY_MULTIPLIER: 500,
  EXTEND_GUEST_TIMERS: 250,
  FREEZE_GUEST_TIMERS: 100,
};
```

Prestige Configuration (`config/prestige.ts`)

Modify upgrade costs, effects, and milestone rewards here.

Zone Configuration (`zones.json`)

```
[
  {
    "zoneId": "1",
    "zoneName": "Roller Coaster",
    "queueCapacity": 5,
    "rideCapacity": 3,
    "rideTime": 10,
    "ticketPrice": 25,
    "queueCount": 0,
    "rideCount": 0,
    "isRunning": false
  }
]
```

Installation & Setup

Prerequisites

- Node.js 16+
- TypeScript
- WebSocket-compatible clients

Installation

```
# Clone repository
git clone [<https://github.com/Falcons-Creative-Group/RollerCoasterWSS.git]>
cd RollerCoasterWSS

# Install dependencies
npm install

# Start server
npx ts-node server.ts
```

Usage Examples

Basic Client Connection

```javascript
const ws = new WebSocket('ws://localhost:8080');

// Identify client type
ws.send(JSON.stringify({
  EventType: "identify",
  Data: { client: "react" }
}));

// Listen for park updates
ws.on('message', (data) => {
  const message = JSON.parse(data.toString());
  if (message.EventType === 'parkData') {
    console.log('Park state:', message.Data);
  }
});
```

Upgrading a Zone

```javascript
// Reduce ride time for zone 1
ws.send(JSON.stringify({
  EventType: "zoneUpgrade",
  EventTimestamp: new Date().toISOString(),
  Source: { ClientType: "react" },
  Data: {
    zoneId: "1",
    upgradeType: "reduceRideTime"
  }
}));
```

Prestige Reset

```javascript
// Reset for prestige (requires sufficient balance)
ws.send(JSON.stringify({
  EventType: "prestigeReset",
  EventTimestamp: new Date().toISOString(),
  Source: { ClientType: "react" },
  Data: {}
}));
```

Development Guide

Adding New Features

1. New Event Handler

```javascript
// 1. Add to handlers/gameHandlers.ts
export function handleNewFeature(message: any, wss: any) {
  // Implementation
}

// 2. Add to handlers/index.ts
export { handleNewFeature } from './gameHandlers';

// 3. Add to server.ts message switch
case "newFeature":
  handleNewFeature(message, wss);
```

```
  break;
```

2. New Prestige Upgrade

```
// Add to config/prestige.ts
{
  id: "newUpgrade",
  name: "New Upgrade",
  description: "Does something cool",
  cost: 3,
  maxLevel: 10,
  effect: (level) => 0.1 * level
}
```

3. New Validation

```
// Add to helpers/validation.ts
export function validateNewFeature(data: any): boolean {
  // Validation logic
  return true;
}
```

Performance Considerations

Memory Management

- Guest cleanup on ride completion
- Zone state resets on disconnection
- Map-based storage for O(1) lookups

Broadcasting Optimization

- Client type filtering
- Selective event broadcasting
- Batch updates where possible

State Synchronization

- Atomic operations for balance changes
- Consistent state updates
- Race condition prevention

Troubleshooting

Common Issues

1. WebSocket Connection Failed

```
Error: connect ECONNREFUSED 127.0.0.1:8080
```

Solution: Ensure server is running and port 8080 is available.

2. Guest Spawn Issues

```
Warning: All zones full
```

Solution: Check zone queue capacity and guest cleanup logic.

3. Prestige Calculation Errors

```
Error: Cannot read property 'effect' of undefined
```

Solution: Verify upgrade IDs match configuration in `config/prestige.ts`.

4. Balance Going Negative

```
Event: gameOver
```

Solution: Check penalty calculations and upgrade costs.

React Typescript Frontend

Theme Park Game - Full Stack Documentation

Table of Contents

Overview

The Theme Park Game is a real-time multiplayer simulation where players manage a theme park with guests, rides, zones, and a prestige system. The application consists of:

- **Backend**: Node.js WebSocket server handling game logic, state management, and real-time updates
- **Frontend**: React application providing an interactive UI for park management
- **Real-time Communication**: WebSocket connection enabling instant updates between server and clients

Key Features

- **Real-time Guest Management**: Dynamic guest spawning and tracking across zones
- **Interactive Ride Operations**: Queue management, capacity planning, and ride operations
- **Prestige System**: Multi-layered progression with upgrades and milestone rewards
- **Dynamic Difficulty Scaling**: Increasing challenges based on prestige level
- **Zone & Global Upgrades**: Comprehensive improvement systems
- **Smart Automation**: Auto-ride and smart queue management features

Backend Server

Project Structure

```
server/
├── config/                 # Configuration and constants
│   ├── constants.ts        # Game constants and base costs
│   ├── prestige.ts         # Prestige upgrades and milestones
│   └── types.ts            # TypeScript interfaces
├── handlers/               # WebSocket message handlers
│   ├── gameHandlers.ts     # Core game mechanics
│   ├── prestigeHandlers.ts # Prestige system logic
│   └── rideHandlers.ts     # Ride operations
├── game/                   # Core game systems
│   ├── gameState.ts        # Centralized state management
│   ├── prestigeSystem.ts   # Prestige calculations
│   └── difficultySystem.ts # Dynamic difficulty
├── helpers/                # Utility functions
│   ├── broadcast.ts        # WebSocket broadcasting
│   ├── calculations.ts     # Game calculations
│   └── validation.ts       # Input validation
├── schema.ts               # Data schemas
├── zones.json              # Zone configuration
└── server.ts               # Main server entry point
```

Core Game State

```
class GameState {
  public balance: number              // Player's money
  public guestCounter: number         // Total guests spawned
  public prestige: number             // Current prestige level
  public prestigePoints: number       // Available prestige points

  public readonly guests: Map<string, Guest>           // Active guests
  public readonly zones: Map<string, Zone>             // Park zones
  public readonly prestigeUpgrades: Map<string, number> // Upgrade levels
  public readonly unlockedFeatures: Set<string>       // Unlocked features
}
```

Prestige System Features

Prestige Upgrades

Upgrade	Effect	Max Level	Cost Formula
Money Boost	+15% ride revenue per level	10	Base cost + (level ÷ 3)
Guest Patience	+3s patience per level	10	Base cost + (level ÷ 3)
Starting Capital	+75 starting money per level	5	Base cost + (level ÷ 3)
Upgrade Efficiency	-8% upgrade costs per level	8	Base cost + (level ÷ 3)
Damage Control	-10% penalties per level	10	Base cost + (level ÷ 3)

Milestone Rewards

- **Prestige 3**: Auto-ride feature unlocked
- **Prestige 6**: Smart queue management
- **Prestige 9**: Third zone "Water Park" unlocked

Frontend React Application

Project Structure

```
src/
├── components/             # React components
│   ├── Header.tsx          # Top navigation and stats
│   ├── NotificationContainer.tsx # Toast notifications
│   ├── BonusesDisplay.tsx  # Active bonuses display
│   ├── DifficultyDisplay.tsx # Difficulty indicators
│   ├── GlobalUpgrades.tsx  # Park-wide upgrades
│   ├── AutoRideSettings.tsx # Automation settings
│   ├── ZoneGrid.tsx        # Zone container
│   ├── ZoneCard.tsx        # Individual zone display
│   ├── ZoneUpgrades.tsx    # Zone-specific upgrades
│   ├── GuestList.tsx       # Guest management
│   ├── PrestigePanel.tsx   # Prestige system UI
│   ├── PrestigeUpgradeCard.tsx # Individual upgrades
│   └── GameOverScreen.tsx  # Game over state
├── App.tsx                 # Main application component
├── schema.ts               # Shared type definitions
└── App.css                 # Styling
```

State Management

The main App component manages all application state:

```
// Core game state
const [zones, setZones] = useState<Map<string, Zone>>(new Map());
const [guests, setGuests] = useState<Map<string, Guest>>(new Map());
const [balance, setBalance] = useState<string>();
const [gameOver, setGameOver] = useState(false);

// Prestige system state
const [prestige, setPrestige] = useState(0);
const [prestigePoints, setPrestigePoints] = useState(0);
const [prestigeUpgrades, setPrestigeUpgrades] = useState<any[]>([]);
const [currentUpgrades, setCurrentUpgrades] = useState<any>({});

// UI state
const [selectedZone, setSelectedZone] = useState<string>("");
const [selectedGuests, setSelectedGuests] = useState<string[]>([]);
const [activeTab, setActiveTab] = useState<"park" | "prestige">("park");
```

Component Architecture

Header Component

Purpose: Display current game stats and navigation

Props:

```
interface HeaderProps {
  balance?: string;
  prestige: number;
  prestigePoints: number;
  activeTab: "park" | "prestige";
  onTabChange: (tab: "park" | "prestige") => void;
}
```

ZoneCard Component

Purpose: Display individual zone information and controls

Key Features:

- Zone statistics (queue/ride capacity, timing)
- Zone-specific upgrades
- Ride timer display
- Lock state for premium zones

GuestList Component

Purpose: Manage guests in queues and on rides

Key Features:

- Visual patience indicators (color-coded)

- Guest selection for ride assignment
- Critical guest warnings (⚠️ for <1s patience)
- Ride control buttons

PrestigePanel Component

Purpose: Handle prestige system interactions

Key Features:

- Progress tracking toward next prestige
- Upgrade purchase interface
- Prestige reset functionality
- Upgrade level visualization

WebSocket Communication

Connection Setup

```
// Client connection initialization
const ws = useRef<WebSocket | null>(null);

useEffect(() => {
  ws.current = new WebSocket("ws://localhost:8080");

  ws.current.onopen = () => {
    ws.current?.send(JSON.stringify({
      EventType: "identify",
      EventTimestamp: Date.now(),
      Source: { ClientType: "REACT_UI" },
      Data: { client: "react" }
    }));
  };

  ws.current.onmessage = (event) => {
    const parsed = JSON.parse(event.data);
    handleServerMessage(parsed);
  };
}, []);
```

Message Flow

Client → Server Events

```
// Adding guests to ride
{
  EventType: "addToRide",
  EventTimestamp: number,
  Source: { ClientType: "REACT_UI" },
  Data: { guests: string } // JSON array of guest IDs
}

// Starting a ride
```

```
{
  EventType: "startRide",
  Data: { zoneId: string, guests: string[] }
}

// Purchasing upgrades
{
  EventType: "zoneUpgrade",
  Data: { zoneId: string, upgradeType: string }
}

// Prestige actions
{
  EventType: "purchasePrestigeUpgrade",
  Data: { upgradeId: string }
}
```

Server → Client Events

```
// Park state updates (broadcast every second)
{
  EventType: "parkData",
  Data: {
    zones: Record<string, Zone>,
    guests: Record<string, Guest>,
    balance: number,
    prestige: number,
    prestigePoints: number,
    bonuses: GameBonuses,
    difficulty: DifficultyScaling,
    unlockedFeatures: string[],
    prestigeUpgrades: PrestigeUpgrade[],
    currentUpgrades: Record<string, number>,
    prestigeRequirement: number
  }
}

// Real-time notifications
{
  EventType: "guestLeft",
  Data: { guestId: string, penalty: number, message: string }
}

{
  EventType: "milestoneUnlocked",
  Data: { milestone: MilestoneReward, newFeatures: string[] }
}
```

API Reference

Zone Management

Zone Upgrades

Upgrade Type	Cost	Effect

`reduceRideTime`	$150	Reduces ride duration by 2 seconds
`increaseQueueCapacity`	$300	Adds +1 queue slot
`increaseRideCapacity`	$350	Adds +1 ride slot

Global Upgrades

Upgrade Type	Cost	Effect
`moneyMultiplier`	$500	+0.2x money multiplier (permanent)
`extendGuestTimersTemporary`	$250	+10s patience for all guests
`freezeGuestTimers`	$100	Freeze all guest timers for 5s

Guest Management

Guest Types

```typescript
interface Guest {
  id: string;
  zoneId: string;
  type: "BASIC" | "PREMIUM" | "VIP";
  ticketPrice: number;
  timeToFrustration: number;
  inRide: boolean;
}
```

Guest Patience System

- **Green**: >3 seconds remaining
- **Orange**: 1-3 seconds remaining
- **Red (pulsing)**: <1 second remaining (⚠️)
- **Penalty**: -$5 per frustrated guest (scaled by difficulty)

Prestige System

Prestige Requirements

```
function getPrestigeRequirement(): number {
  return 10 * Math.pow(1.5, gameState.prestige);
}
```

Difficulty Scaling

```
interface DifficultyScaling {
  frustrationSpeedMultiplier: number;  // 1.0 + (prestige * 0.15)
  leavingPenaltyMultiplier: number;    // 1.0 + (prestige * 0.1)
  upgradeCostMultiplier: number;       // 1.0 + (prestige * 0.2)
}
```

Setup & Installation

Prerequisites

- Node.js 16+
- npm or yarn
- Modern web browser with WebSocket support

Backend Setup

```
# Clone and setup server
git clone [repository-url]
cd theme-park-server
npm install

# Start development server
npx ts-node server.ts
# Server runs on ws://localhost:8080
```

Frontend Setup

```
# Setup React application
cd theme-park-client
npm install

# Start development server
npm start
# Application runs on <http://localhost:3000>
```

Configuration Files

zones.json

```
[
  {
    "zoneId": "1",
    "zoneName": "Roller Coaster",
    "queueCapacity": 5,
    "rideCapacity": 3,
    "rideTime": 10,
```

```
    "ticketPrice": 25,
    "queueCount": 0,
    "rideCount": 0,
    "isRunning": false
  }
]
```

constants.ts

```ts
export const PORT = 8080;
export const GUEST_SPAWN_INTERVAL = 1000;
export const BASE_COSTS = {
  REDUCE_RIDE_TIME: 150,
  INCREASE_QUEUE_CAPACITY: 300,
  INCREASE_RIDE_CAPACITY: 350,
  // ... more costs
};
```

Development Guide

Adding New Components

1. Create Component File

```tsx
// components/NewComponent.tsx
interface NewComponentProps {
  data: any;
  onAction: (value: string) => void;
}

const NewComponent = ({ data, onAction }: NewComponentProps) => {
  return (
    <div className="new-component">
      {/* Component JSX */}
    </div>
  );
};

export default NewComponent;
```

2. Update Parent Component

```tsx
// App.tsx
import NewComponent from './components/NewComponent';

// Add to render method
<NewComponent
  data={someData}
  onAction={handleAction}
/>
```

Adding New WebSocket Events

1. Backend Handler

```ts
// handlers/gameHandlers.ts
export function handleNewEvent(message: any, wss: any) {
  // Validate input
```

```
  if (!validateNewEventData(message.Data)) {
    return;
  }

  // Process event
  const result = processNewEvent(message.Data);

  // Broadcast update
  broadcastToClients(wss, "newEventResult", result);
}
```

2. Frontend Handler

```
// App.tsx WebSocket onmessage handler
else if (parsed.EventType === "newEventResult") {
  // Handle new event result
  setGameState(parsed.Data);
  addNotification("New event completed!");
}
```

Troubleshooting

Common Frontend Issues

1. WebSocket Connection Failed

Symptoms: App shows no data, console errors about WebSocket

Solutions:

- Verify backend server is running on port 8080
- Check firewall settings
- Ensure WebSocket URL is correct in code

2. Guest Selection Not Working

Symptoms: Clicking guests doesn't select them

Solutions:

- Check if guest list is properly filtered
- Verify onClick handlers are attached
- Check for CSS pointer-events: none

3. Components Not Re-rendering

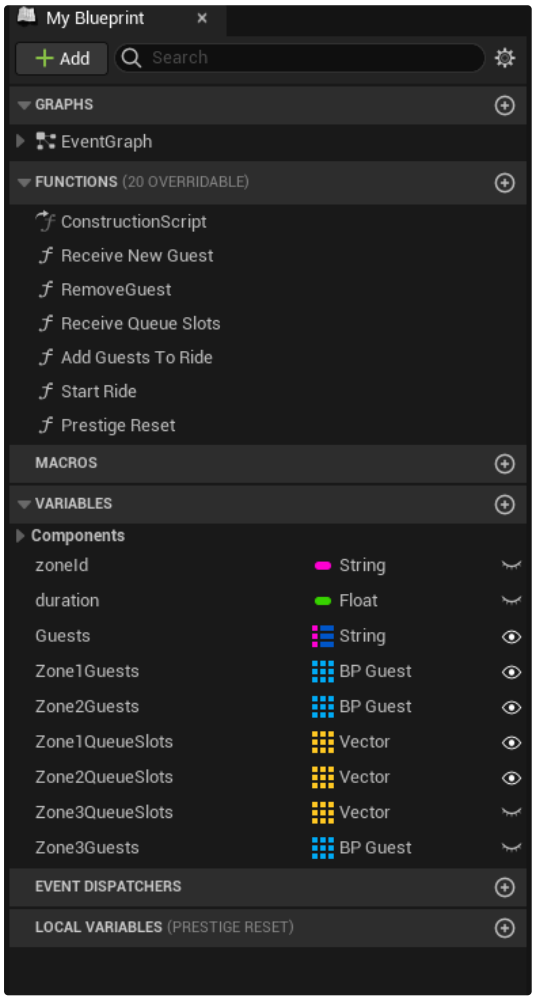
Symptoms: UI doesn't update when state changes

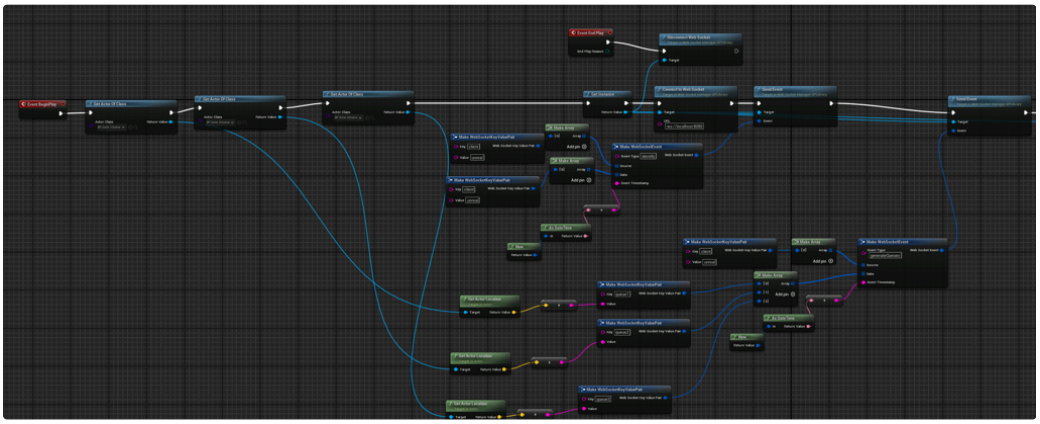
Solutions:

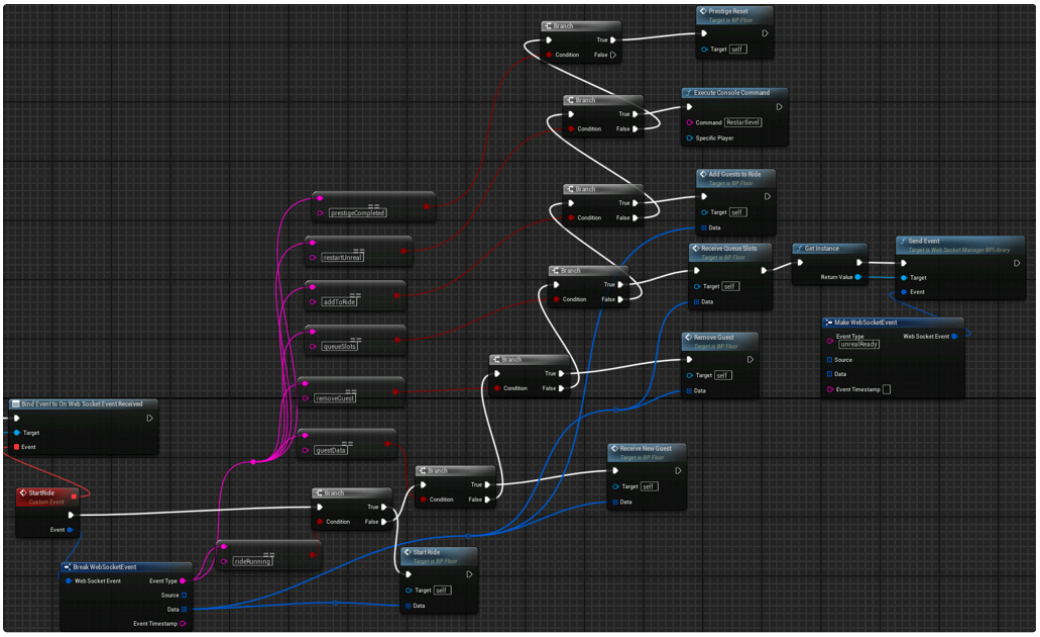
- Ensure state updates are immutable
- Check useEffect dependencies
- Verify props are being passed correctly

Unreal Engine 5.3 Simulation

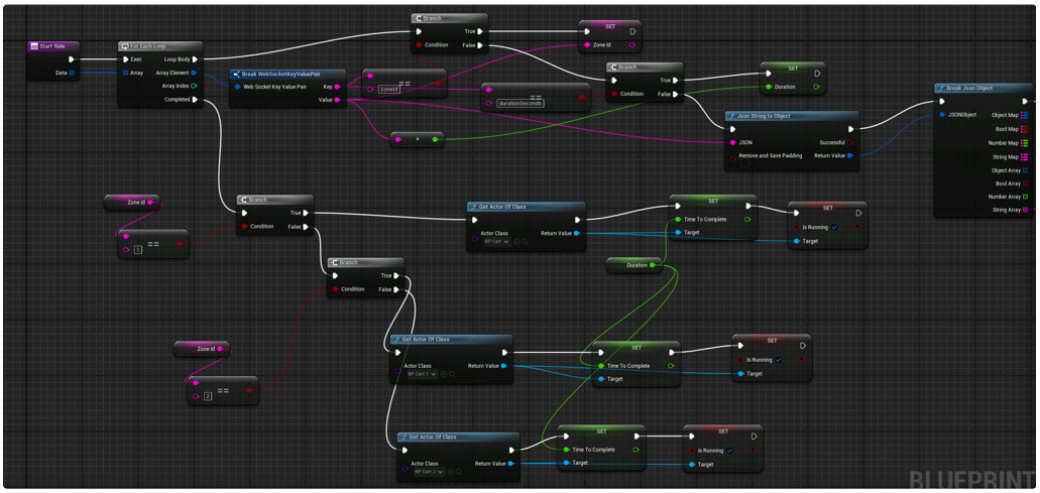
- Create Blank Unreal Engine Project with starter content enabled
- Create a Blueprints folder and make your BP_Floor blueprint by right-clicking in the content browser and selecting Blueprint Class
- In your event graph create the necessary functions and variables shown below

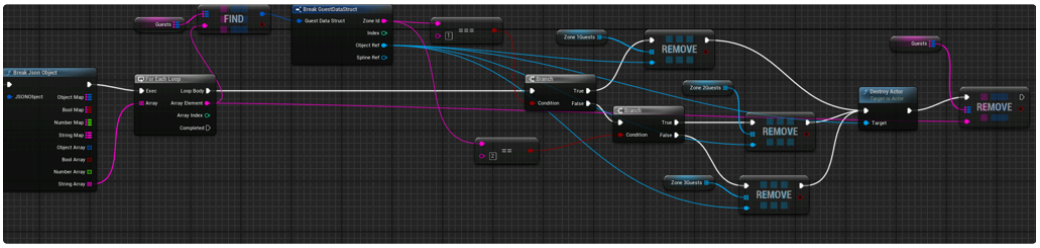




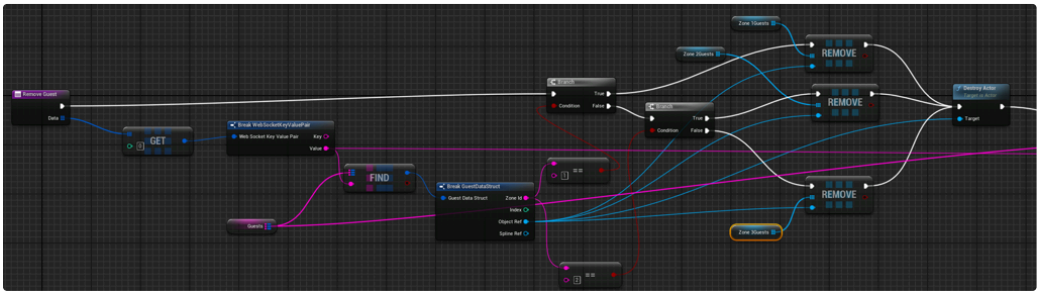


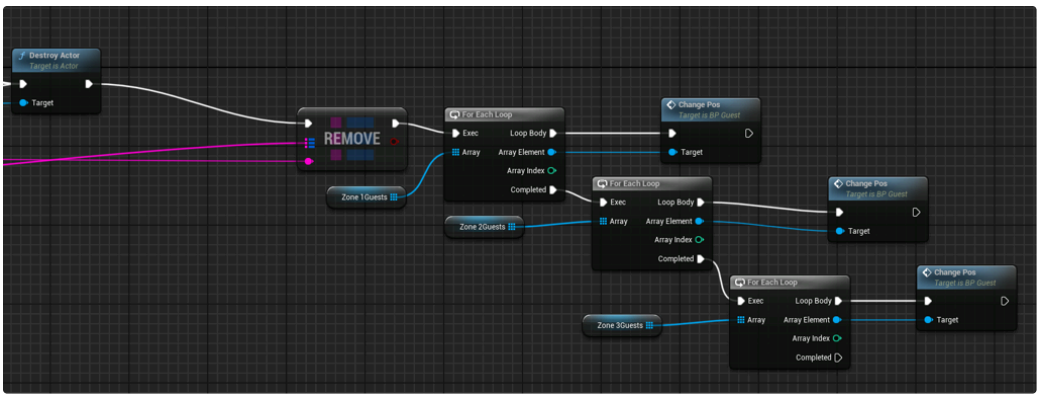
Start Ride Function



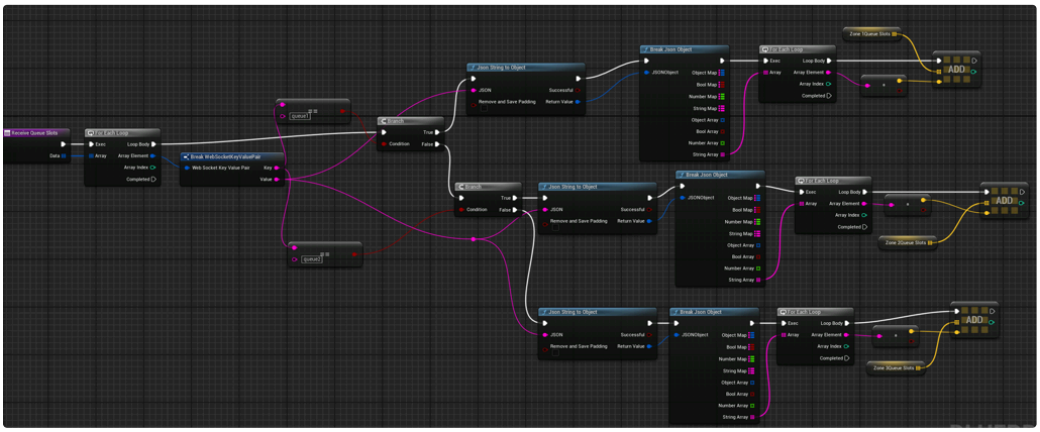


Remove Guest Function

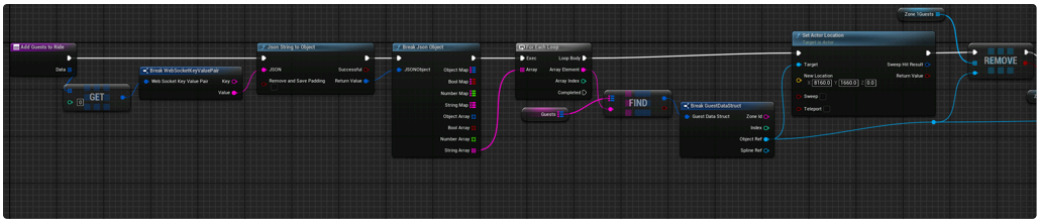


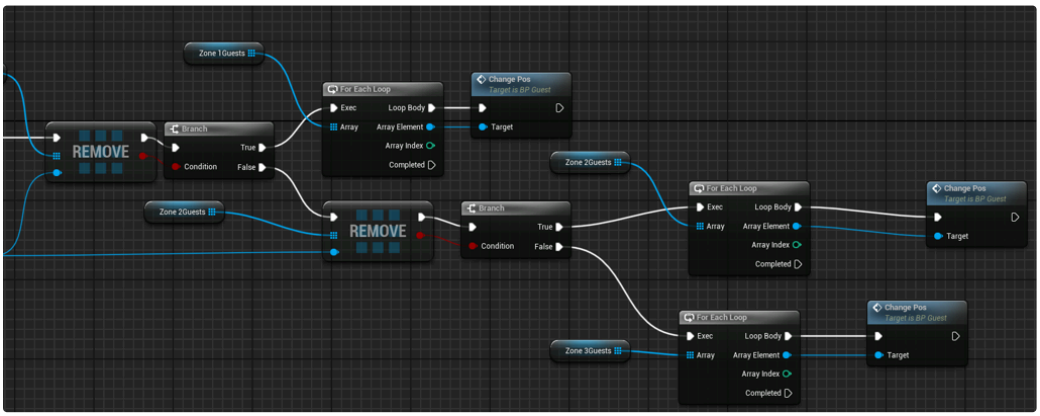


Receive Queue Slots Function

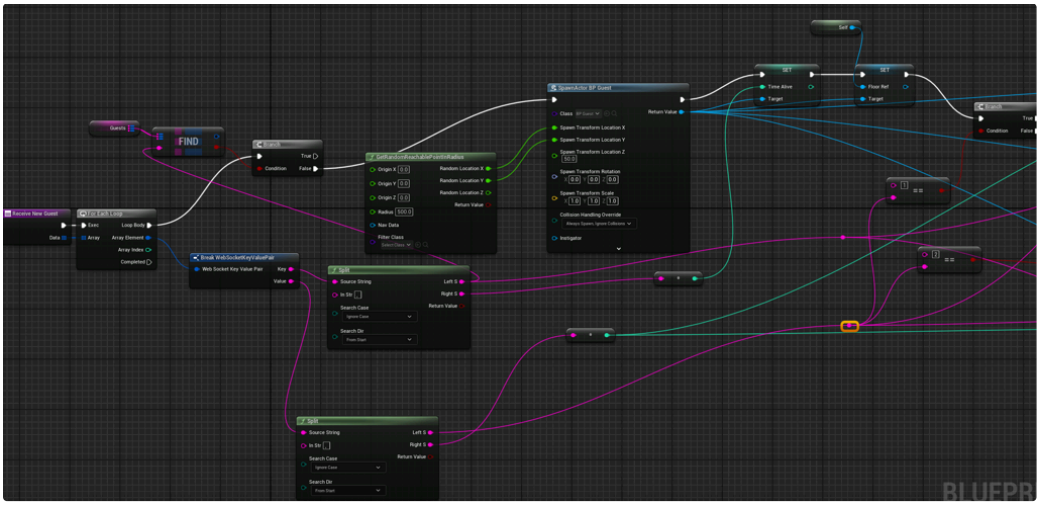


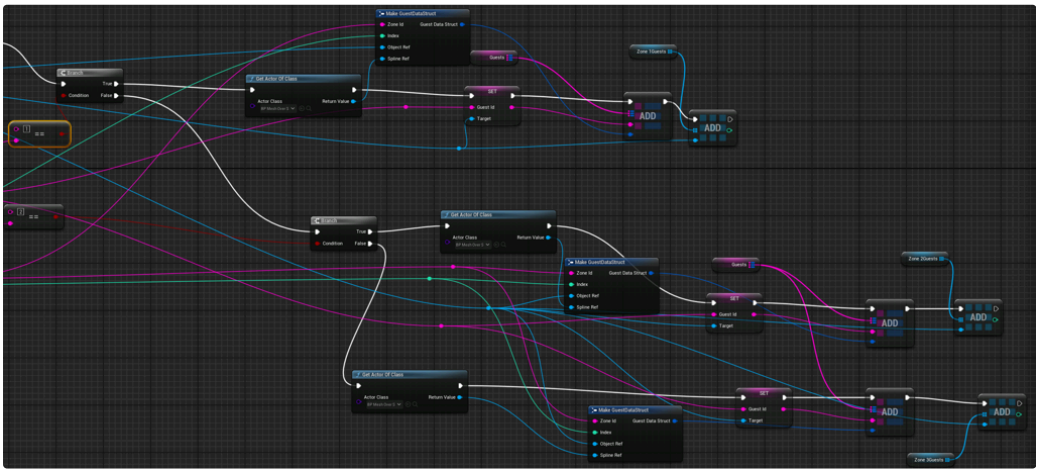
Add Guests To Ride Function



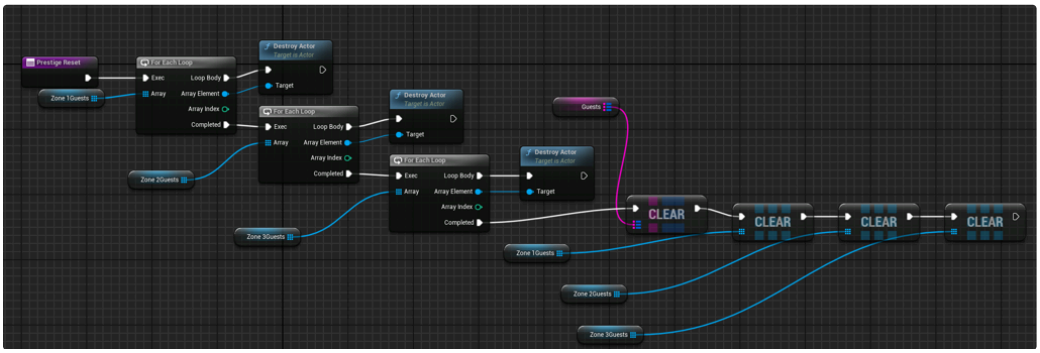


Receive New Guest Function





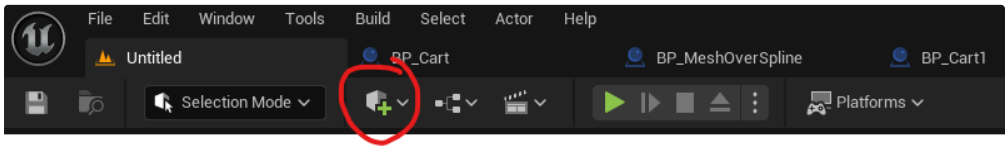
Prestige Reset Function



Event Graph

Once the simulation begins the **BP_Floor** class gets the locations of the beginning of each queue for the rides you create. In my example I have three rides so for each ride there must be an actor named **BP_Zone{NUM}QueueStart** so that the position data can be sent to the WSS for the queue positions to be created automatically based on the queue's capacity.

After the queue slots have been created the **BP_Floor** class becomes your main WebSocket listener because the floor stretches the navigable space for the **BP_Guest AI's**. So your level needs a **Nav Mesh Bounds Volume** that covers your queue lines, spawn point, and the walking area from the spawn point to each queue.

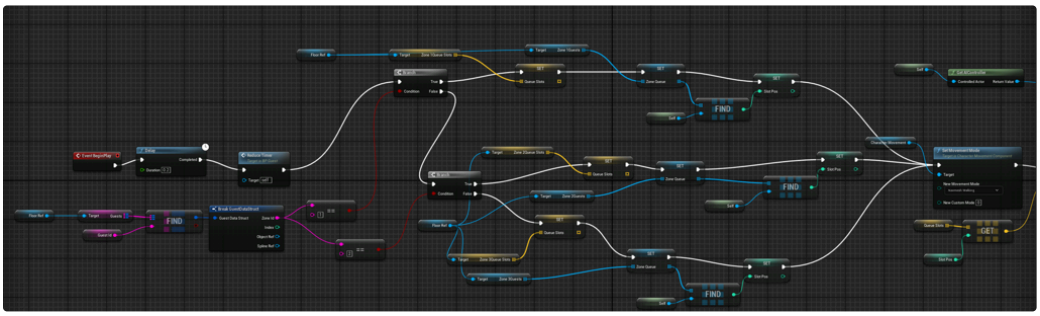
1. 

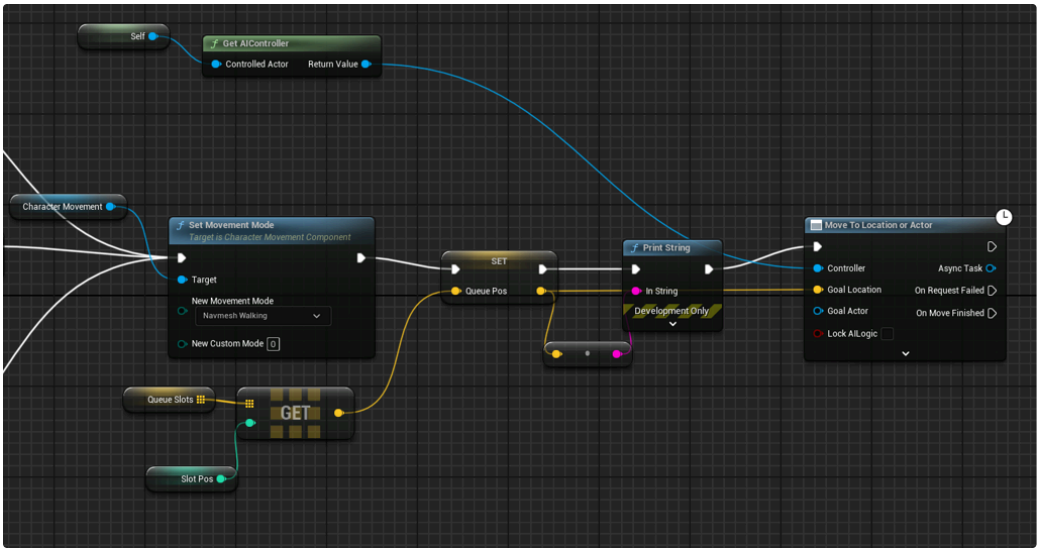
2. Click the highlighted button above and from the dropdown select **Volumes**
3. Then select Nav Mesh Bounds Volume and adjust the scale so that the X and Y fits the area but also increase the Z value so that the height of the BP_Guest objects are encompassed

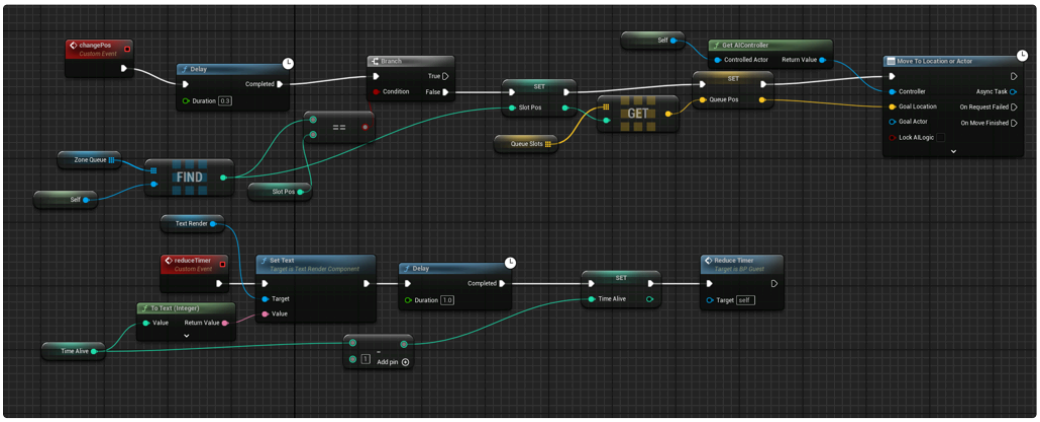
The gameloop from the WebSocket Server is compressed as the unreal simulation handles the bare bones of the project. Moreover, for brevity I will describe the high level system architecture as each function interacts with a Zone, the Zone's guests, and the Zone's cart object.

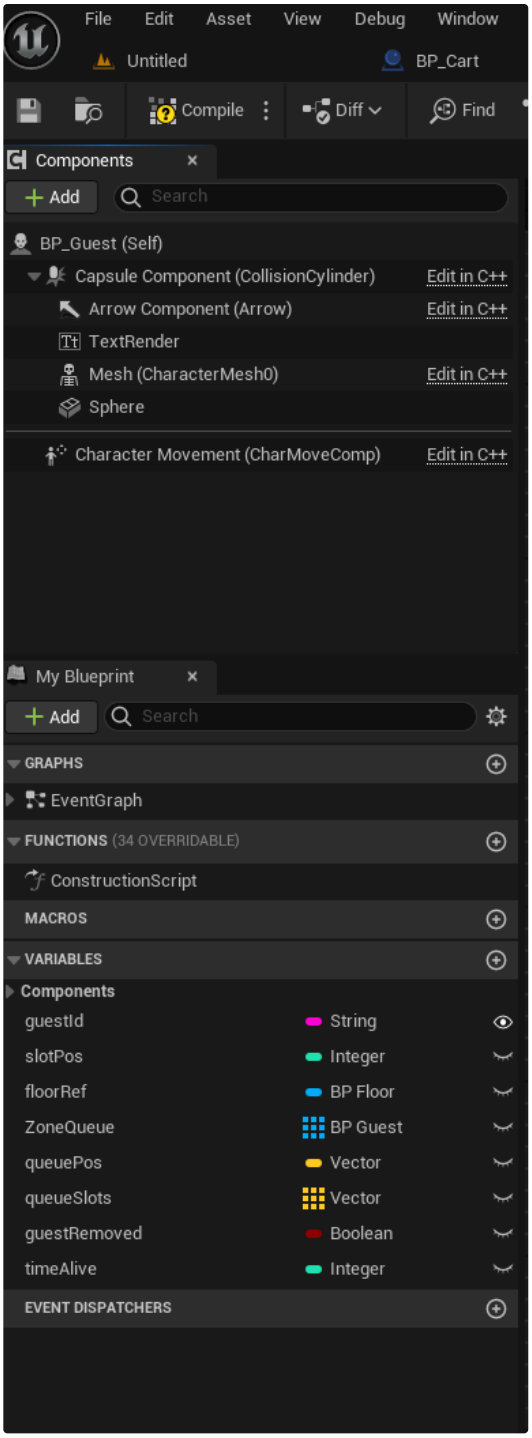
Once the system receives guests they get processed into their appropriate Zone's BP_Guest Object Reference Array. Each guest is given a random spawn location around the origin of the map and receives the **BP_Floor** object reference and the guest's time until frustration. The guest then uses it's position in its respective ZoneGuest array to get the vector of it's position in the queue. Once the guest moves to it's spot in queue it either moves up the line or onto the ride once added. After the guest is added it is placed onto the loading dock. If the guest does not make it onto the ride before it's timer runs out, displayed above the guests mesh, then the guest is removed from the scene. Once the ride starts all guests in the loading dock are removed from the scene and the cart makes its way around the track using the rideTimer value from the WebSocket Server, determining the playrate for the cart's timeline.

BP_Guest Event Graph

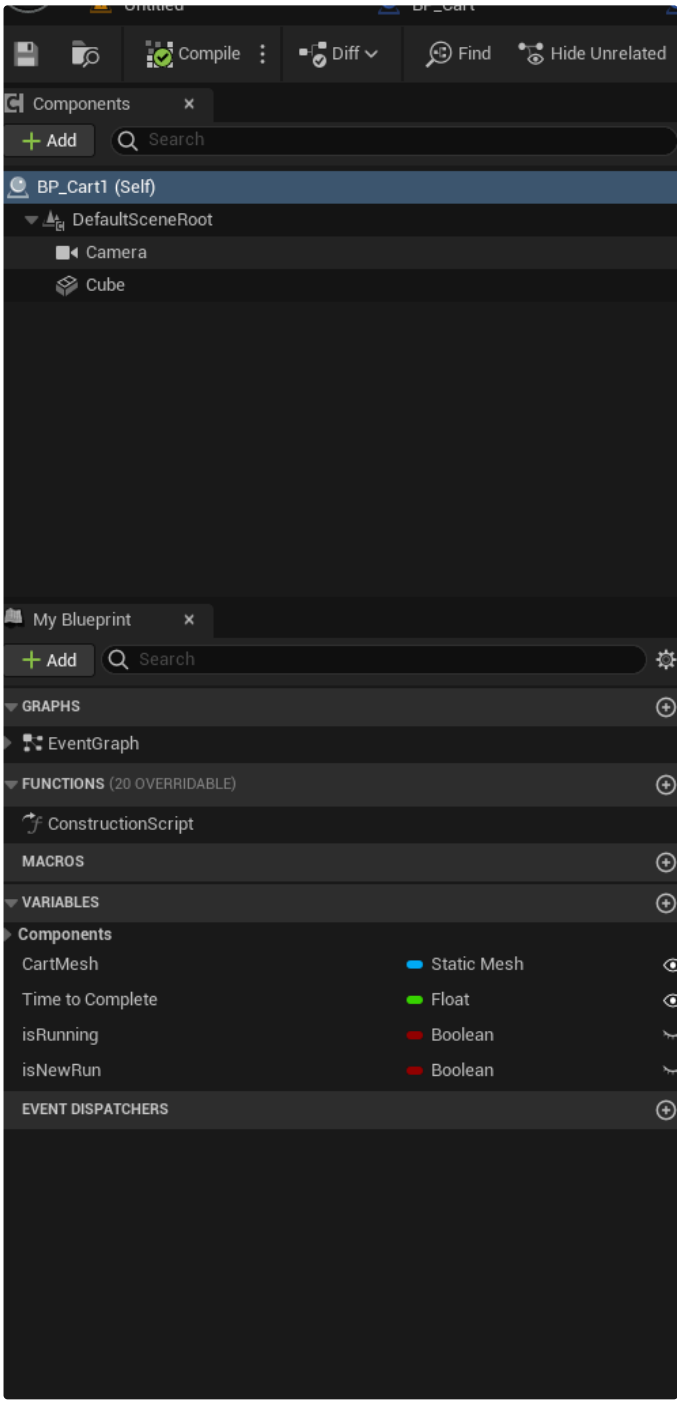


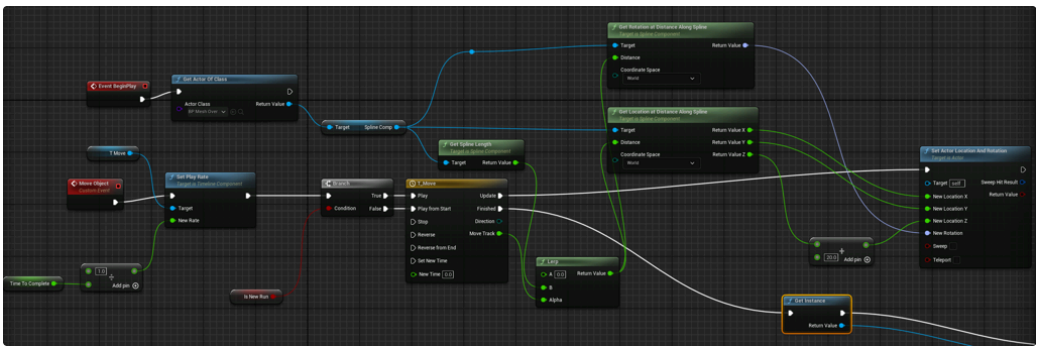

Self
Get AIController
Controlled Actor
Return Value
Character Movement
Set Movement Mode
Target is Character Movement Component
Target
New Movement Mode
Navmesh Walking
New Custom Mode
0
SET
Queue Pos
Print String
In String
Development Only
Move To Location or Actor
Controller
Async Task
Goal Location
On Request Failed
Goal Actor
On Move Finished
Lock AILogic
Queue Slots
GET
Slot Pos

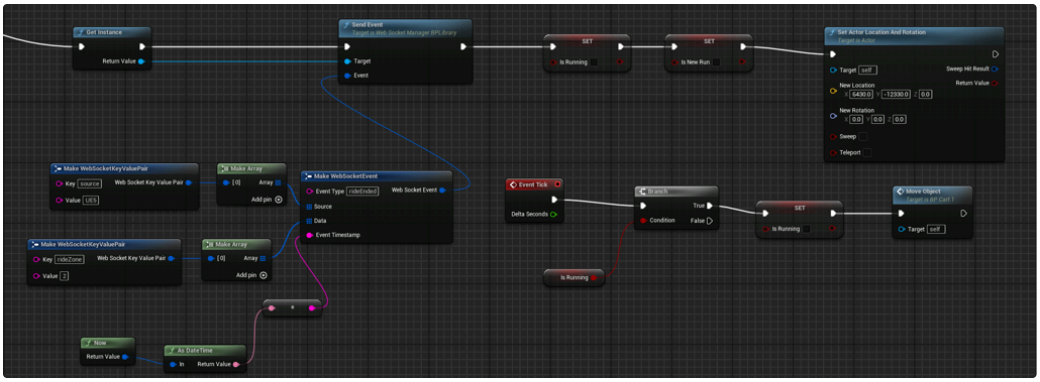

changePos
Custom Event
Delay
Duration 0.3
Completed
Branch
Condition
True
False
SET
Slot Pos
Queue Slots
GET
Queue Pos
Self
Get AIController
Controlled Actor
Return Value
Move To Location or Actor
Controller
Goal Location
Goal Actor
Lock AILogic
Async Task
On Request Failed
On Move Finished
Zone Queue
FIND
Slot Pos
Text Render
reduceTimer
Custom Event
Set Text
Target is Text Render Component
Target
Value
Delay
Duration 1.0
Completed
SET
Time Alive
Reduce Timer
Target is BP_Guard
Target self
To Text (Integer)
Value
Return Value
Time Alive
Add pin


File
Edit
Asset
View
Debug
Window
Untitled
BP_Cart
Compile
Diff
Find
Components
Add
Search
BP_Guest (Self)
Capsule Component (CollisionCylinder)
Edit in C++
Arrow Component (Arrow)
Edit in C++
TextRender
Mesh (CharacterMesh0)
Edit in C++
Sphere
Character Movement (CharMoveComp)
Edit in C++
My Blueprint
Add
Search
GRAPHS
EventGraph
FUNCTIONS (34 OVERRIDABLE)
ConstructionScript
MACROS
VARIABLES
Components
guestId
String
slotPos
Integer
floorRef
BP Floor
ZoneQueue
BP Guest
queuePos
Vector
queueSlots
Vector
guestRemoved
Boolean
timeAlive
Integer
EVENT DISPATCHERS

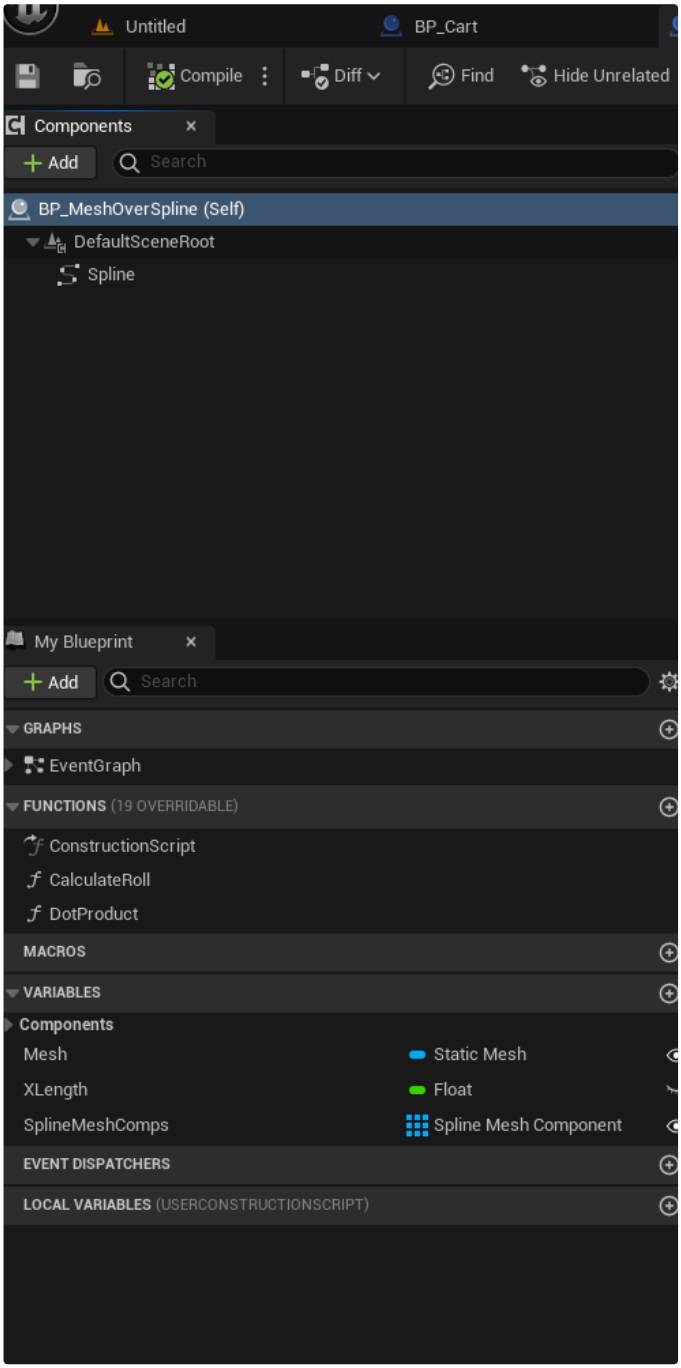
BP_Cart Event Graph

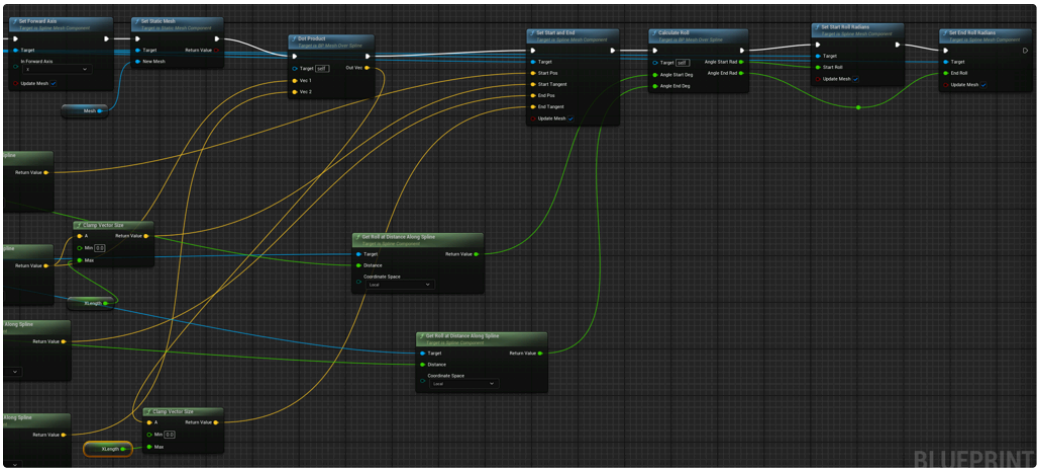


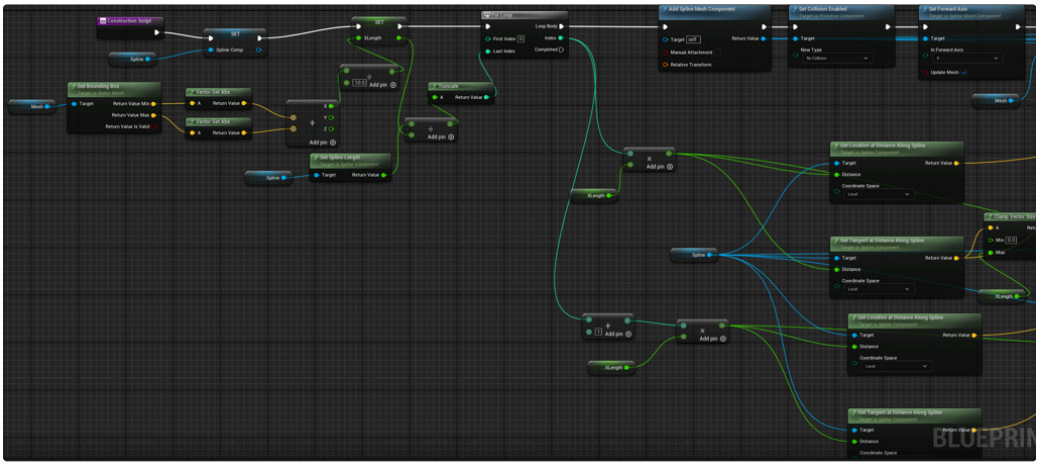




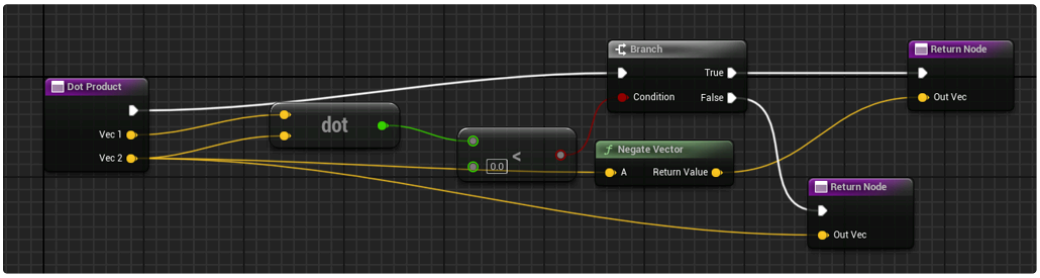
BP_MeshOverSpline

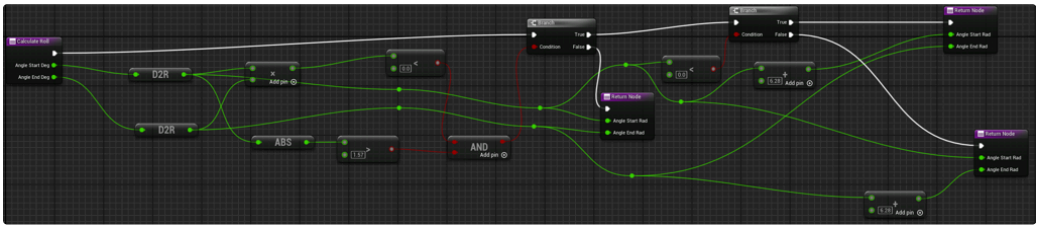






The design for the spline blueprint is built for creating infinitely scalable and customizable rollercoasters as the spline builds itself with the mesh attached so that the rollercoaster can be built and adjusted in real time. However, the normal splines in unreal engine do not handle loops/rotating tracks so two functions must be implemented for the track's orientation to stay normal.

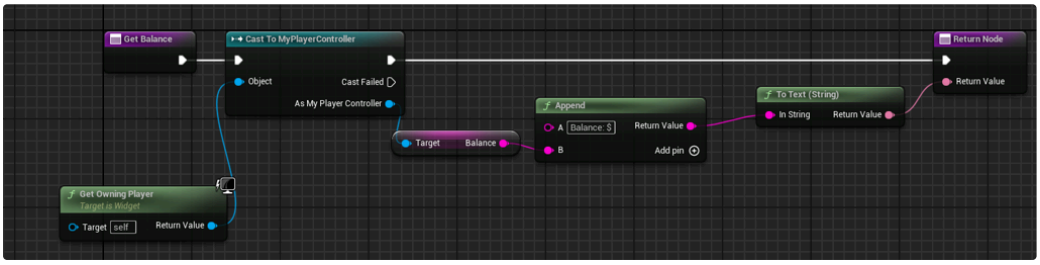




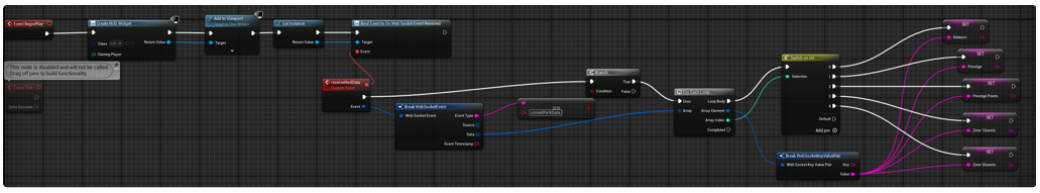
The original blueprint for this can be found here, but in summary the function takes the starting and ending angle in degrees then checks if the total rotation between the two becomes negative such as 350 degrees to 10 degrees is +20 degrees not -340 degrees. This function ensures the shortest rotation possible is taken so that the spline rotation stays smooth and oriented properly. Moreover, the dot product function ensures, that during corkscrews or loops, the spline stays oriented upright and the roll maintains itself throughout the loop. Furthermore the original blueprints for the spline and cart can be found in this youtube video.

HUD/Game Mode

In order to create a HUD that will display when piloting the default player camera, from the blank unreal engine project, a Game Mode and Player Controller class must be created from adding a new blueprint class in the content browser then selecting the respective classes. The Game Mode class will be set as the GameMode Override for the level with the only other change being the Player Controller Class. Then a HUD can be created from the User Interface section when adding a new item in the content browser. From here for every text block added, on top of the canvas in the HUD, you must bind the Text attribute.



This is the basic setup for displaying the text from a variable that will be held inside of the Player Controller class from earlier.



On event play the HUD widget will be created and added to the Viewport, but in my implementation the data is grabbed by the WebSocket plugin and broken into a switch statement so that for any new variables that you wish to display can be added into the HUD without changing the base blueprint.

Future Changes/Revisions

The Unreal Engine implementation showcases a bare bones simulation for the purposes of visualizing the movement inside of a real park. However, there is still huge room for improvement from the landscape, rollercoaster carts, guest blobs, and the UI.

The biggest changes, in terms of functionality, would be updating the UI to support all of the functions controlled by the React UI. This would eliminate the need for two clients and also reduce the amount of delay between actions as removing a guest/adding a guest to the ride has to go from the React UI →WebSocket→Unreal Engine.

Visually, the park landscape could be expanded to include more realistic terrain, props, and architectural elements that better represent a fully built theme park. The rollercoaster carts could be replaced with proper mesh models and animations to give the ride a sense of movement and weight. Likewise, the current guest blobs could be swapped out for low-poly character meshes with basic idle or walking animations to make the simulation feel more alive and immersive.