СОДЕРЖАНИЕ

[1 Bike 3](#_Toc76906097)

[2 BikeHudViewConroller 7](#_Toc76906098)

[3 GazeInput 8](#_Toc76906099)

[4 KeyboardInput 8](#_Toc76906100)

[5 NewInput 9](#_Toc76906101)

[6 VrDeviceInput 9](#_Toc76906102)

[7 MainMenuViewConroller 10](#_Toc76906103)

[8 OptionsViewConroller 10](#_Toc76906104)

[9 TrackDescription 11](#_Toc76906105)

[10 TrackEntryViewController 12](#_Toc76906106)

[11 TrackSelectionViewController 12](#_Toc76906107)

[12 Obstacle 13](#_Toc76906108)

[13 Powerup 14](#_Toc76906109)

[14 PowerupCoolant 15](#_Toc76906110)

[15 PowerupFuel 15](#_Toc76906111)

[16 PowerupSlowdown 15](#_Toc76906112)

[17 BotController 16](#_Toc76906113)

[18 CameraController 18](#_Toc76906114)

[19 Player 19](#_Toc76906115)

[20 RaceCondition 20](#_Toc76906116)

[21 RaceConditionLaps 20](#_Toc76906117)

[22 RaceController 20](#_Toc76906118)

[23 RaceResultsViewController 23](#_Toc76906119)

[24 CountdownViewController 23](#_Toc76906120)

[25 PauseViewController 24](#_Toc76906121)

[26 ComplexEngineSfxController 25](#_Toc76906122)

[27 EngineSfxController 26](#_Toc76906123)

[28 BezierTrackPoint 26](#_Toc76906124)

[29 Track 27](#_Toc76906125)

[30 TrackCircle 27](#_Toc76906126)

[31 TrackCloseBezierCurve 30](#_Toc76906127)

[32 TrackLine 35](#_Toc76906128)

[33 ObjectPlacer 35](#_Toc76906129)

[34 SplineMeshProxy 36](#_Toc76906130)

[35 ControlDevice 37](#_Toc76906131)

[36 GazePointer 37](#_Toc76906132)

[37 HandController 38](#_Toc76906133)

[38 Lever 40](#_Toc76906134)

[39 NavigationPanel 40](#_Toc76906135)

1. Bike

using System;  
using System.Collections.Generic;  
using UnityEngine;  
  
namespace TubeRace  
{  
 [Serializable]  
 public class BikeParametersInitial  
 {  
 [Range(100.0f, 1000.0f)] public float **maxVelocity**;  
 [Range(0.0f, 100.0f)] public float **thrust**;  
 [Range(0.0f, 1.0f)] public float **bounceFactor**;  
  
 [Range(100.0f, 1000.0f)] public float **maxAngularVelocity**;  
 [Range(0.0f, 100.0f)] public float **angularThrust**;  
 [Range(0.0f, 1.0f)] public float **angularDrag**;  
  
 public float **afterburnerThrust**;  
 public float **afterburnerMaxVelocityBonus**;  
  
 public float **afterburnerCoolSpeed**;  
 public float **afterburnerHeatSpeed**;  
 public float **afterburnerMaxHeat**;  
 }  
  
 public class **Bike** : MonoBehaviour  
 {  
 private const string **Tag** = "Bike";  
 public static GameObject[] BikesAsGameObjects;  
  
 #region Сериализуемые поля  
  
 [SerializeField] private Track **track**;  
 public Track Track => track;  
  
 [SerializeField] private AudioSource **collisionSfx**;  
 [SerializeField] private AnimationCurve **collisionVolumeCurve**;  
  
 [SerializeField] private bool **isPlayerBike**;  
 public bool IsPlayerBike => isPlayerBike;  
  
 [SerializeField] private BikeParametersInitial **initial**;  
  
 #endregion  
  
 #region Управление  
  
 public bool IsMovementControlsActive { get; set; }  
  
 private float forwardThrustAxis;  
  
 public void SetForwardThrustAxis(float val)  
 {  
 forwardThrustAxis = val;  
 }  
  
 private float horizontalThrustAxis;  
  
 public void SetHorizontalThrustAxis(float val)  
 {  
 horizontalThrustAxis = val;  
 }  
  
 #endregion  
  
 #region Топливо  
  
 public float Fuel { get; private set; }  
  
 public void AddFuel(float amount)  
 {  
 Fuel += amount;  
 Fuel = Mathf.Clamp(Fuel, 0, 100);  
 }  
  
 private bool CanConsumeFuel(float amount)  
 {  
 if (Fuel < amount)  
 return false;  
  
 Fuel -= amount;  
 return true;  
 }  
  
 #endregion  
  
 #region Скорости  
  
 public float Velocity { get; private set; }  
  
 public float NormalizedVelocity()  
 {  
 return Mathf.Clamp01(Velocity / initial.maxVelocity);  
 }  
  
 public void Slowdown(int percent)  
 {  
 Velocity -= Velocity \* percent / 100.0f;  
 }  
  
 public float Angle { get; private set; }  
 private float angularVelocity;  
  
 #endregion  
  
 #region Форсаж  
  
 public bool EnableAfterburner { get; set; }  
 private float afterburnerHeat;  
  
 public float NormalizedHeat  
 {  
 get  
 {  
 if (initial.afterburnerMaxHeat > 0)  
 return afterburnerHeat / initial.afterburnerMaxHeat;  
  
 return 0.0f;  
 }  
 }  
  
 private void HeatAfterburner()  
 {  
 afterburnerHeat += Velocity;  
 }  
  
 public void CoolAfterburner()  
 {  
 afterburnerHeat = 0;  
 }  
  
 #endregion  
  
 #region Статистика  
  
 public class BikeStatistics  
 {  
 public int RacePlace;  
 public float BestVelocity;  
 public float BestSeconds;  
 public float TotalSeconds;  
 }  
  
 public BikeStatistics Stats { get; private set; }  
  
 private float raceStartTime;  
  
 private int lapNum;  
 private List<float> lapDurations;  
 private float lapStartTime;  
  
 public float Distance { get; private set; }  
 public float PrevDistance { get; private set; }  
  
 #endregion  
  
 #region Гонка  
  
 public void OnRaceStart()  
 {  
 Stats.RacePlace = 0;  
 Stats.BestSeconds = 0;  
 Stats.TotalSeconds = 0;  
  
 raceStartTime = Time.time;  
 lapStartTime = raceStartTime;  
 }  
  
 public void OnRaceEnd()  
 {  
 Stats.TotalSeconds = Time.time - raceStartTime;  
 }  
  
 #endregion  
  
 #region Обновления  
  
 private void UpdateHeat()  
 {  
 afterburnerHeat -= initial.afterburnerCoolSpeed \* Time.deltaTime;  
  
 if (afterburnerHeat < 0)  
 afterburnerHeat = 0;  
  
 if (afterburnerHeat > initial.afterburnerMaxHeat)  
 Slowdown(100);  
 }  
  
 private void UpdateVelocity()  
 {  
 float dt = Time.deltaTime;  
  
 float forceThrustMax = initial.thrust;  
 float velocityMax = initial.maxVelocity;  
 float force = forwardThrustAxis \* initial.thrust;  
  
 if (EnableAfterburner  
 && CanConsumeFuel(1.0f \* Time.deltaTime))  
 {  
 afterburnerHeat += initial.afterburnerHeatSpeed \* Time.deltaTime;  
  
 force += initial.afterburnerThrust;  
 velocityMax += initial.afterburnerMaxVelocityBonus;  
 forceThrustMax += initial.afterburnerThrust;  
 }  
  
 float forceDrag = -Velocity \* (forceThrustMax / velocityMax);  
 force += forceDrag;  
  
 Velocity += force \* dt;  
 if (Stats.BestVelocity < Mathf.Abs(Velocity))  
 Stats.BestVelocity = Mathf.Abs(Velocity);  
  
 float ds = Velocity \* dt;  
 if (Physics.Raycast(transform.position, transform.forward, ds))  
 {  
 HeatAfterburner();  
  
 collisionSfx.volume = collisionVolumeCurve.Evaluate(NormalizedVelocity());  
 collisionSfx.Play();  
  
 Velocity = -Velocity \* initial.bounceFactor;  
 ds = Velocity \* dt;  
 }  
  
 PrevDistance = Distance;  
 Distance += ds;  
 }  
  
 private void UpdateAngle()  
 {  
 float dt = Time.deltaTime;  
 angularVelocity += horizontalThrustAxis \* initial.angularThrust;  
 Angle += angularVelocity \* dt;  
  
 if (Angle > 180.0f)  
 Angle -= 360.0f;  
 else if (Angle < -180.0f)  
 Angle += 360.0f;  
  
 angularVelocity += -angularVelocity \* initial.angularDrag \* dt;  
 angularVelocity = Mathf.Clamp(angularVelocity,  
 -initial.maxAngularVelocity, initial.maxAngularVelocity);  
 }  
  
 private void UpdatePhysics()  
 {  
 UpdateVelocity();  
 UpdateAngle();  
  
 if (Distance < 0)  
 Distance = 0;  
  
 Vector3 bikePos = track.Position(Distance);  
 transform.position = bikePos;  
 transform.rotation = track.Rotation(Distance);  
 transform.Rotate(Vector3.forward, Angle, Space.**Self**);  
 transform.Translate(-Vector3.up \* track.Radius, Space.**Self**);  
 }  
  
 private void UpdateBestTime()  
 {  
 int currLap = (int) (Distance / track.Length()) + 1;  
 if (currLap <= lapNum)  
 return;  
  
 float lapDuration = Time.time - lapStartTime;  
 lapStartTime = Time.time;  
  
 lapDurations.Add(lapDuration);  
 lapNum++;  
  
 if (lapDuration > Stats.BestSeconds)  
 Stats.BestSeconds = lapDuration;  
 }  
  
 #endregion  
  
 #region Юнити  
  
 private void **Awake**()  
 {  
 Stats = new BikeStatistics();  
 lapDurations = new List<float>();  
 }  
  
 private void **Start**()  
 {  
 BikesAsGameObjects = GameObject.FindGameObjectsWithTag(**Tag**);  
 }  
  
 private void **Update**()  
 {  
 UpdateHeat();  
 UpdatePhysics();  
 UpdateBestTime();  
 }  
  
 #endregion  
 }  
}

1. BikeHudViewConroller

using UnityEngine;  
using UnityEngine.UI;  
  
namespace TubeRace  
{  
 public class **BikeHudViewController** : MonoBehaviour  
 {  
 [SerializeField] private Bike **bike**;  
  
 [SerializeField] private Text **labelVelocity**;  
 [SerializeField] private Text **labelDistance**;  
 [SerializeField] private Text **labelLapNum**;  
  
 [SerializeField] private Text **labelRollAngle**;  
  
 [SerializeField] private Text **labelHeat**;  
 [SerializeField] private Text **labelFuel**;  
  
 private void **Update**()  
 {  
 labelVelocity.text = "Speed: " + (int) (bike.Velocity) + " m/s";  
 labelDistance.text = "Distance: " + (int) (bike.Distance) + " m";  
  
 int laps = (int) (bike.Distance / bike.Track.Length()) + 1;  
 labelLapNum.text = "Lap: " + laps;  
  
 labelRollAngle.text = "Angle: " + (int) (bike.Angle) + " deg";  
  
 labelHeat.text = "Heat: " + (int) (bike.NormalizedHeat \* 100.0f);  
 labelFuel.text = "Fuel: " + (int) bike.Fuel;  
 }  
 }  
}

1. GazeInput

using UnityEngine;  
  
namespace TubeRace  
{  
 public class **GazeInput** : NewInput  
 {  
 [SerializeField] private NavigationPanel **navigationPanel**;  
  
 public override Vector3 MoveDirection()  
 {  
 return navigationPanel.MoveDirection();  
 }  
  
 public override bool EnableAfterburner()  
 {  
 return false;  
 }  
 }  
}

1. KeyboardInput

using UnityEngine;  
  
namespace TubeRace  
{  
 public class **KeyboardInput** : NewInput  
 {  
 public override Vector3 MoveDirection()  
 {  
 Vector3 direction = new Vector3();  
  
 if (Input.GetKey(KeyCode.**W**))  
 direction.y = 1;  
  
 if (Input.GetKey(KeyCode.**S**))  
 direction.y = -1;  
  
 if (Input.GetKey(KeyCode.**A**))  
 direction.x = -1;  
  
 if (Input.GetKey(KeyCode.**D**))  
 direction.x = 1;  
  
 return direction;  
 }  
  
 public override bool EnableAfterburner()  
 {  
 return Input.GetKey(KeyCode.**Space**);  
 }  
 }  
}

1. NewInput

using UnityEngine;  
  
namespace TubeRace  
{  
 public abstract class **NewInput** : MonoBehaviour  
 {  
 public abstract Vector3 MoveDirection();  
  
 public abstract bool EnableAfterburner();  
 }  
}

1. VrDeviceInput

using System;  
using UnityEngine;  
  
namespace TubeRace  
{  
 public class **VrDeviceInput** : NewInput  
 {  
 [SerializeField] private Lever lever;  
  
 public override Vector3 MoveDirection()  
 {  
 throw new NotImplementedException();  
 }  
  
 public override bool EnableAfterburner()  
 {  
 throw new NotImplementedException();  
 }  
 }  
}

1. MainMenuViewConroller

using UnityEngine;  
  
namespace TubeRace  
{  
 public class **MainMenuViewController** : MonoBehaviour  
 {  
 [SerializeField] private TrackSelectionViewController **trackSelectionViewController**;  
 [SerializeField] private OptionsViewController **optionsViewController**;  
  
 public void **OnButtonNewGame**()  
 {  
 trackSelectionViewController.gameObject.SetActive(true);  
 gameObject.SetActive(false);  
 }  
  
 public void **OnButtonOptions**()  
 {  
 optionsViewController.gameObject.SetActive(true);  
 gameObject.SetActive(false);  
 }  
  
 public void **OnButtonExit**()  
 {  
 Application.Quit();  
 }  
 }  
}

1. OptionsViewConroller

using UnityEngine;  
using UnityEngine.UI;  
  
namespace TubeRace  
{  
 public class **OptionsViewController** : MonoBehaviour  
 {  
 [SerializeField] private MainMenuViewController **mainMenuViewController**;  
  
 [SerializeField] private bool **isFullScreen** = true;  
 [SerializeField] private Dropdown **dropdown**;  
  
 private readonly int[] width = {1920, 1366, 1440};  
 private readonly int[] height = {1080, 768, 900};  
  
 private void InitDropdownOptions()  
 {  
 for (int i = 0; i < height.Length; i++)  
 {  
 Dropdown.OptionData optionData = new Dropdown.OptionData  
 {  
 text = $"{width[i]} x {height[i]}"  
 };  
  
 dropdown.options.Add(optionData);  
 }  
 }  
  
 private void SetScreenResolution()  
 {  
 int option = dropdown.value;  
 Screen.SetResolution(width[option], height[option], isFullScreen);  
 }  
  
 public void **OnButtonExit**()  
 {  
 SetScreenResolution();  
  
 gameObject.SetActive(false);  
 mainMenuViewController.gameObject.SetActive(true);  
 }  
  
 private void **Awake**()  
 {  
 gameObject.SetActive(false);  
  
 InitDropdownOptions();  
 dropdown.value = 0;  
  
 SetScreenResolution();  
 }  
 }  
}

1. TrackDescription

using UnityEngine;  
  
namespace TubeRace  
{  
 [CreateAssetMenu]  
 public class **TrackDescription** : ScriptableObject  
 {  
 [SerializeField] private string **title**;  
 public string Title => title;  
  
 [SerializeField] private string **sceneName**;  
 public string SceneName => sceneName;  
  
 [SerializeField] private Sprite **preview**;  
 public Sprite Preview => preview;  
  
 [SerializeField] private float **length**;  
 public float Length => length;  
  
 public void SetLength(float newLength)  
 {  
 length = newLength;  
 }  
 }  
}

1. TrackEntryViewController

using System.Globalization;  
using UnityEngine;  
using UnityEngine.SceneManagement;  
using UnityEngine.UI;  
  
namespace TubeRace  
{  
 public class **TrackEntryViewController** : MonoBehaviour  
 {  
 [SerializeField] private TrackDescription **trackDescription**;  
 private TrackDescription activeDescription;  
  
 [SerializeField] private Text **labelTitle**;  
 [SerializeField] private Text **labelLength**;  
 [SerializeField] private GameObject **labelImage**;  
  
 private void SetViewFields(TrackDescription description)  
 {  
 activeDescription = description;  
  
 labelTitle.text = description.Title;  
 labelLength.text = description.Length.ToString(CultureInfo.InvariantCulture);  
 labelImage.GetComponent<Image>().sprite = description.Preview;  
 }  
  
 public void **OnButtonStartLevel**()  
 {  
 SceneManager.LoadScene(activeDescription.SceneName);  
 }  
  
 private void **Start**()  
 {  
 if (trackDescription != null)  
 SetViewFields(trackDescription);  
 }  
 }  
}

1. TrackSelectionViewController

using UnityEngine;  
  
namespace TubeRace  
{  
 public class **TrackSelectionViewController** : MonoBehaviour  
 {  
 [SerializeField] private MainMenuViewController **mainMenuViewController**;  
  
 public void **OnButtonExit**()  
 {  
 gameObject.SetActive(false);  
 mainMenuViewController.gameObject.SetActive(true);  
 }  
  
 private void **Awake**()  
 {  
 gameObject.SetActive(false);  
 }  
 }  
}

1. Obstacle

using UnityEngine;  
  
namespace TubeRace  
{  
 public class **Obstacle** : MonoBehaviour  
 {  
 [SerializeField] private Track **track**;  
 [SerializeField] private float **distance**;  
  
 [Range(0.0f, 20.0f)] [SerializeField] private float **radiusModifier** = 1.0f;  
 [SerializeField] private float **angle**;  
 [Range(0.0f, 100.0f)] public float **angularThrust**;  
  
 private Vector3 obstacleDirection;  
 private Vector3 trackPosition;  
  
 private Quaternion quater;  
 private Vector3 trackOffset;  
  
 private void UpdateAngle()  
 {  
 angle += angularThrust \* Time.deltaTime;  
 if (angle > 180.0f)  
 angle -= 360.0f;  
 else if (angle < -180.0f)  
 angle = 360.0f + angle;  
 }  
  
 private void UpdatePosition()  
 {  
 quater = Quaternion.AngleAxis(angle, Vector3.forward);  
 trackOffset = quater \* (Vector3.up \* (radiusModifier \* track.Radius));  
  
 transform.position = trackPosition - trackOffset;  
 transform.rotation = Quaternion.LookRotation(obstacleDirection, trackOffset);  
 }  
  
 private void **OnDrawGizmos**()  
 {  
 Gizmos.color = Color.red;  
  
 Vector3 centerlinePos = track.Position(distance);  
 Gizmos.DrawSphere(centerlinePos, track.Radius);  
 }  
  
 private void **OnValidate**()  
 {  
 obstacleDirection = track.Direction(distance);  
 trackPosition = track.Position(distance);  
 UpdatePosition();  
 }  
  
 private void **Update**()  
 {  
 UpdateAngle();  
 UpdatePosition();  
 }  
 }  
}

1. Powerup

using UnityEngine;  
  
namespace TubeRace  
{  
 public abstract class **Powerup** : MonoBehaviour  
 {  
 [SerializeField] private Track **track**;  
 [SerializeField] private float **distance**;  
 [SerializeField] private float **angle**;  
  
 protected abstract void OnPicked(Bike bike);  
  
 private void SetPosition()  
 {  
 Vector3 obstacleDir = track.Direction(distance);  
 Vector3 trackPosition = track.Position(distance);  
  
 Quaternion quater = Quaternion.AngleAxis(angle, Vector3.forward);  
 Vector3 trackOffset = quater \* (Vector3.up \* 0);  
  
 transform.position = trackPosition - trackOffset;  
 transform.rotation = Quaternion.LookRotation(obstacleDir, trackOffset);  
 }  
  
 private void UpdateBikes()  
 {  
 foreach (GameObject bikeGo in Bike.BikesAsGameObjects)  
 {  
 Bike bike = bikeGo.GetComponent<Bike>();  
  
 float prev = bike.PrevDistance;  
 float curr = bike.Distance;  
  
 if (prev < distance && curr > distance)  
 {  
 *// limit angles* OnPicked(bike);  
 }  
 }  
 }  
  
 private void **OnValidate**()  
 {  
 SetPosition();  
 }  
  
 private void **Update**()  
 {  
 UpdateBikes();  
 }  
 }  
}

1. PowerupCoolant

using UnityEngine;  
  
namespace TubeRace  
{  
 public class **PowerupCoolant** : Powerup  
 {  
 protected override void OnPicked(Bike bike)  
 {  
 bike.CoolAfterburner();  
 Debug.Log("PowerupCoolant picked up by " + bike.name);  
 }  
 }  
}

1. PowerupFuel

using UnityEngine;  
  
namespace TubeRace  
{  
 public class **PowerupFuel** : Powerup  
 {  
 [Range(0.0f, 100.0f)]  
 [SerializeField] private float **fuelAmount**;  
  
 protected override void OnPicked(Bike bike)  
 {  
 bike.AddFuel(fuelAmount);  
 }  
 }  
}

1. PowerupSlowdown

using UnityEngine;  
  
namespace TubeRace  
{  
 public class **PowerupSlowdown** : Powerup  
 {  
 [Range(0, 100)] [SerializeField] private int **slowdownPercent**;  
  
 protected override void OnPicked(Bike bike)  
 {  
 bike.Slowdown(slowdownPercent);  
 }  
 }  
}

1. BotController

using System;  
using UnityEngine;  
  
namespace TubeRace  
{  
 public enum BotBehaviour  
 {  
 **Nothing**,  
 **Behave** }  
  
 public enum TimerType  
 {  
 **Nothing**,  
  
 **ReactionDelay**,  
 **MoveOn**,  
 **Rotate**,  
  
 **MaxValues** }  
  
 [RequireComponent(typeof(Bike))]  
 public class **BotController** : MonoBehaviour  
 {  
 [SerializeField] private bool **isEnabled**;  
 [SerializeField] private BotBehaviour **behaviour**;  
  
 [Range(1, 100)] [SerializeField] private int **predictionTimeSteps**;  
  
 [Range(0.0f, 10.0f)] [SerializeField] private float **reactionDelayTime**;  
 [Range(0.0f, 10.0f)] [SerializeField] private float **moveForwardTime**;  
  
 private Bike bike;  
 private Transform bikeTransform;  
  
 private float[] timers;  
  
 private void InitTimers()  
 {  
 timers = new float[(int) TimerType.**MaxValues**];  
  
 SetTimer(TimerType.**ReactionDelay**, reactionDelayTime);  
 }  
  
 private void SetTimer(TimerType e, float time)  
 {  
 timers[(int) e] = time;  
 }  
  
 private bool IsTimerFinished(TimerType e)  
 {  
 return timers[(int) e] <= 0;  
 }  
  
 private void MoveForward()  
 {  
 bike.SetForwardThrustAxis(1);  
 }  
  
 private void StallForward()  
 {  
 bike.SetForwardThrustAxis(0);  
 }  
  
 private void MoveForwardOrStall()  
 {  
 if (!IsTimerFinished(TimerType.**MoveOn**))  
 {  
 MoveForward();  
 }  
 else  
 {  
 StallForward();  
 SetTimer(TimerType.**ReactionDelay**, reactionDelayTime);  
 }  
 }  
  
 private void MoveHorizontal()  
 {  
 bike.SetHorizontalThrustAxis(1);  
 }  
  
 private void StallHorizontal()  
 {  
 bike.SetHorizontalThrustAxis(0);  
 }  
  
 private void Move()  
 {  
 float dt = Time.deltaTime \* predictionTimeSteps;  
 float ds = bike.Velocity \* dt;  
 bool isCollision = Physics.Raycast(bikeTransform.position, bikeTransform.forward, ds);  
  
 if (!isCollision)  
 {  
 StallHorizontal();  
 MoveForwardOrStall();  
 }  
 else  
 {  
 MoveForward();  
 MoveHorizontal();  
 }  
 }  
  
 private void Behave()  
 {  
 if (!bike.IsMovementControlsActive)  
 return;  
  
 if (IsTimerFinished(TimerType.**ReactionDelay**))  
 Move();  
 else  
 SetTimer(TimerType.**MoveOn**, moveForwardTime);  
 }  
  
 private void UpdateBot()  
 {  
 if (!isEnabled)  
 return;  
  
 switch (behaviour)  
 {  
 case BotBehaviour.**Nothing**:  
 break;  
  
 case BotBehaviour.**Behave**:  
 Behave();  
 break;  
  
 default:  
 throw new ArgumentOutOfRangeException();  
 }  
 }  
  
 private void UpdateTimers()  
 {  
 for (int i = 0; i < timers.Length; i++)  
 if (timers[i] > 0)  
 timers[i] -= Time.deltaTime;  
 }  
  
 private void **Start**()  
 {  
 bike = GetComponent<Bike>();  
 bikeTransform = bike.transform;  
  
 InitTimers();  
 }  
  
 private void **Update**()  
 {  
 UpdateBot();  
 UpdateTimers();  
 }  
 }  
}

1. CameraController

using UnityEngine;  
  
namespace TubeRace  
{  
 public class **CameraController** : MonoBehaviour  
 {  
 [SerializeField] private Bike **bike**;  
  
 [SerializeField] private float **minViewField** = 60;  
 [SerializeField] private float **maxViewField** = 85;  
 [SerializeField] private float **shakeFactor**;  
 [SerializeField] private AnimationCurve **shakeCurve**;  
  
 private Camera thisCamera;  
 private Vector3 initialLocalPosition;  
  
 private void UpdateViewField()  
 {  
 float t = bike.NormalizedVelocity();  
 thisCamera.fieldOfView = Mathf.Lerp(minViewField, maxViewField, t);  
 }  
  
 private void UpdateCameraShake()  
 {  
 if (Time.timeScale <= 0)  
 return;  
  
 float t = bike.NormalizedVelocity();  
 float curveValue = shakeCurve.Evaluate(t);  
  
 Vector3 randomVector = Random.insideUnitSphere \* shakeFactor;  
 randomVector.z = 0;  
  
 thisCamera.transform.localPosition = initialLocalPosition + randomVector \* curveValue;  
 }  
  
 private void **Start**()  
 {  
 thisCamera = Camera.main;  
 initialLocalPosition = thisCamera.transform.localPosition;  
 }  
  
 private void **Update**()  
 {  
 UpdateViewField();  
 UpdateCameraShake();  
 }  
 }  
}

1. Player

using UnityEngine;  
  
namespace TubeRace  
{  
 public class **Player** : MonoBehaviour  
 {  
 [SerializeField] private NewInput **newInput**;  
  
 [SerializeField] private Bike **activeBike**;  
  
 private void CheckInput()  
 {  
 if (!activeBike.IsMovementControlsActive)  
 return;  
  
 Vector3 direction = newInput.MoveDirection();  
  
 activeBike.SetForwardThrustAxis(direction.y);  
 activeBike.SetHorizontalThrustAxis(direction.x);  
  
 activeBike.EnableAfterburner = newInput.EnableAfterburner();  
 }  
  
 private void **Update**()  
 {  
 CheckInput();  
 }  
 }  
}

1. RaceCondition

using UnityEngine;  
  
namespace TubeRace  
{  
 public abstract class **RaceCondition** : MonoBehaviour  
 {  
 public bool IsTriggered { get; protected set; }  
  
 public virtual void OnRaceStart()  
 {  
 }  
  
 public virtual void OnRaceEnd()  
 {  
 }  
 }  
}

1. RaceConditionLaps

using UnityEngine;  
  
namespace TubeRace  
{  
 public class **RaceConditionLaps** : RaceCondition  
 {  
 [SerializeField] private RaceController **raceController**;  
  
 private void UpdateConditionLaps()  
 {  
 if (!raceController.IsRaceActive && IsTriggered)  
 return;  
  
 var bikes = raceController.Bikes;  
 foreach (Bike bike in bikes)  
 {  
 int laps = (int) (bike.Distance / bike.Track.Length()) + 1;  
 if (laps < raceController.MaxLaps)  
 return;  
 }  
  
 IsTriggered = true;  
 }  
  
 private void **Update**()  
 {  
 UpdateConditionLaps();  
 }  
 }  
}

1. RaceController

using System.Collections.Generic;  
using UnityEngine;  
using UnityEngine.Events;  
  
namespace TubeRace  
{  
 public enum RaceMode  
 {  
 Laps,  
 Time,  
 LastStanding  
 }  
  
 public class **RaceController** : MonoBehaviour  
 {  
 [SerializeField] private RaceResultsViewController **raceResultsViewController**;  
 [SerializeField] private Track **track**;  
  
 [SerializeField] private Bike[] **bikes**;  
 private List<Bike> activeBikes;  
 private List<Bike> finishedBikes;  
 public IEnumerable<Bike> Bikes => bikes;  
  
 [SerializeField] private RaceMode raceMode;  
 [SerializeField] private RaceCondition[] **conditions**;  
  
 [SerializeField] private UnityEvent **eventRaceStart**;  
 [SerializeField] private UnityEvent **eventRaceFinished**;  
 public bool IsRaceActive { get; private set; }  
  
 [SerializeField] private int **maxLaps**;  
 public int MaxLaps => maxLaps;  
  
 [SerializeField] private int **countdownTimer**;  
 public float CountTimer { get; private set; }  
  
 private void StartRace()  
 {  
 activeBikes = new List<Bike>(bikes);  
 finishedBikes = new List<Bike>();  
  
 IsRaceActive = true;  
 CountTimer = countdownTimer;  
  
 foreach (RaceCondition condition in conditions)  
 condition.OnRaceStart();  
  
 foreach (Bike bike in bikes)  
 bike.OnRaceStart();  
  
 eventRaceStart?.Invoke();  
 }  
  
 private void EndRace()  
 {  
 IsRaceActive = false;  
  
 foreach (RaceCondition condition in conditions)  
 condition.OnRaceEnd();  
  
 eventRaceFinished?.Invoke();  
 }  
  
 private void UpdatePositions()  
 {  
 foreach (Bike bike in activeBikes)  
 {  
 if (finishedBikes.Contains(bike))  
 continue;  
  
 float currDistance = bike.Distance;  
 float totalRaceDistance = maxLaps \* track.Length();  
  
 if (currDistance > totalRaceDistance)  
 {  
 finishedBikes.Add(bike);  
 bike.Stats.RacePlace = finishedBikes.Count;  
 bike.OnRaceEnd();  
  
 if (bike.IsPlayerBike)  
 raceResultsViewController.Show(bike.Stats);  
 }  
 }  
 }  
  
 private void UpdatePrestart()  
 {  
 if (CountTimer > 0)  
 {  
 CountTimer -= Time.deltaTime;  
  
 if (CountTimer <= 0)  
 {  
 foreach (Bike bike in bikes)  
 bike.IsMovementControlsActive = true;  
 }  
 }  
 }  
  
 private void UpdateConditions()  
 {  
 if (IsRaceActive)  
 return;  
  
 foreach (RaceCondition condition in conditions)  
 {  
 if (!condition.IsTriggered)  
 return;  
 }  
  
 EndRace();  
 }  
  
  
 private void **Start**()  
 {  
 StartRace();  
 }  
  
 private void **Update**()  
 {  
 if (!IsRaceActive)  
 return;  
  
 UpdatePositions();  
 UpdatePrestart();  
 UpdateConditions();  
 }  
 }  
}

1. RaceResultsViewController

using System.Globalization;  
using UnityEngine;  
using UnityEngine.SceneManagement;  
using UnityEngine.UI;  
  
namespace TubeRace  
{  
 public class **RaceResultsViewController** : MonoBehaviour  
 {  
 [SerializeField] private Text **place**;  
 [SerializeField] private Text **topVelocity**;  
 [SerializeField] private Text **totalSeconds**;  
 [SerializeField] private Text **bestSeconds**;  
  
 public void Show(Bike.BikeStatistics stats)  
 {  
 gameObject.SetActive(true);  
  
 place.text = "Place: " + stats.RacePlace;  
 topVelocity.text = "Top speed: " + ((int) stats.BestVelocity) + " m/s";  
 totalSeconds.text = "Time: " + stats.TotalSeconds.ToString(CultureInfo.CurrentCulture) + " seconds";  
 bestSeconds.text = "Best lap: " + stats.BestSeconds.ToString(CultureInfo.CurrentCulture) + " seconds";  
 }  
  
 public void OnButtonQuit()  
 {  
 SceneManager.LoadScene(PauseViewController.**MainMenuScene**);  
 }  
  
 private void **Awake**()  
 {  
 gameObject.SetActive(false);  
 }  
 }  
}

1. CountdownViewController

using UnityEngine;  
using UnityEngine.UI;  
  
namespace TubeRace  
{  
 public class **CountdownViewController** : MonoBehaviour  
 {  
 [SerializeField] private RaceController **raceController**;  
 [SerializeField] private Text **label**;  
  
 private void DisableCountdown()  
 {  
 label.text = "";  
 gameObject.SetActive(false);  
 }  
  
 private void UpdateCountdown()  
 {  
 float currSeconds = raceController.CountTimer + 1;  
  
 if (currSeconds > 0.0f && currSeconds < 1.0f)  
 {  
 label.text = "GO";  
 Invoke(nameof(DisableCountdown), 1.0f);  
 }  
 else  
 {  
 label.text = ((int) currSeconds).ToString();  
 }  
 }  
  
 private void **Update**()  
 {  
 UpdateCountdown();  
 }  
 }  
}

1. PauseViewController

using UnityEngine;  
using UnityEngine.SceneManagement;  
  
namespace TubeRace  
{  
 public class **PauseViewController** : MonoBehaviour  
 {  
 public const string **MainMenuScene** = "scene\_main\_menu";  
  
 [SerializeField] private RaceController **raceController**;  
 [SerializeField] private RectTransform **content**;  
  
 public void **OnButtonContinue**()  
 {  
 Time.timeScale = 1;  
 content.gameObject.SetActive(false);  
 }  
  
 public void **OnButtonEndRace**()  
 {  
 SceneManager.LoadScene(**MainMenuScene**);  
 }  
  
 private void **Start**()  
 {  
 content.gameObject.SetActive(false);  
 }  
  
 private void **Update**()  
 {  
 if (Input.GetKeyDown(KeyCode.**Escape**))  
 {  
 if (raceController.IsRaceActive)  
 {  
 GameObject go = content.gameObject;  
 go.SetActive(!go.activeInHierarchy);  
  
 bool canUpdate = !go.activeInHierarchy;  
 Time.timeScale = canUpdate ? 1 : 0;  
 }  
 }  
 }  
 }  
}

1. ComplexEngineSfxController

using UnityEngine;  
  
namespace TubeRace  
{  
 public class **ComplexEngineSfxController** : MonoBehaviour  
 {  
 private const float **PitchFactor** = 2f;  
  
 [SerializeField] private Bike **bike**;  
  
 [SerializeField] private AudioSource **sfxLow**;  
 [SerializeField] private AudioSource **sfxHigh**;  
 [SerializeField] private AudioSource **sfxLoud**;  
 [SerializeField] private AudioSource **sfxSonicBoom**;  
  
 [SerializeField] private AnimationCurve **curveLow**;  
 [SerializeField] private AnimationCurve **curveHigh**;  
 [SerializeField] private AnimationCurve **curveLoud**;  
  
 [SerializeField] private float **superSonicSpeed**;  
 [SerializeField] private AnimationCurve **sonicCurve**;  
 private bool IsSuperSonic { get; set; }  
  
 private void SetSuperSonic(bool enable)  
 {  
 if (!IsSuperSonic && enable)  
 sfxSonicBoom.Play();  
  
 IsSuperSonic = enable;  
 }  
  
 private void UpdateSuperSonicSound()  
 {  
 SetSuperSonic(Mathf.Abs(bike.Velocity) > superSonicSpeed);  
  
 if (sfxSonicBoom.isPlaying)  
 {  
 float t = Mathf.Clamp01(sfxSonicBoom.time / sfxSonicBoom.clip.length);  
 sfxSonicBoom.volume = sonicCurve.Evaluate(t);  
 }  
 }  
  
 private void UpdateEngineSound()  
 {  
 if (IsSuperSonic)  
 {  
 sfxLow.volume = 0;  
 sfxHigh.volume = 0;  
 sfxLoud.volume = 0;  
  
 return;  
 }  
  
 float t = Mathf.Clamp01(bike.Velocity / superSonicSpeed);  
  
 sfxLow.volume = curveLow.Evaluate(t);  
 sfxLow.pitch = 1.0f + **PitchFactor** \* t;  
  
 sfxHigh.volume = curveHigh.Evaluate(t);  
 sfxHigh.pitch = 1.0f + **PitchFactor** \* t;  
  
 sfxLoud.volume = curveLoud.Evaluate(t);  
 }  
  
 private void **Update**()  
 {  
 *// UpdateSuperSonicSound();* UpdateEngineSound();  
 }  
 }  
}

1. EngineSfxController

using UnityEngine;  
  
namespace TubeRace  
{  
 public class **EngineSfxController** : MonoBehaviour  
 {  
 [SerializeField] private Bike **bike**;  
 [SerializeField] private AudioSource **engineSource**;  
  
 [Range(0.0f, 1.0f)] [SerializeField] private float **pitchModifier**;  
  
 private void UpdateEngineSound()  
 {  
 engineSource.pitch = 1.0f + pitchModifier \* bike.NormalizedVelocity();  
 }  
  
 private void **Update**()  
 {  
 UpdateEngineSound();  
 }  
 }  
}

1. BezierTrackPoint

using UnityEngine;  
  
namespace TubeRace  
{  
 public class **BezierTrackPoint** : MonoBehaviour  
 {  
 [SerializeField] private float **length** = 1.0f;  
 public float Length => length;  
  
 private void **OnDrawGizmos**()  
 {  
 Gizmos.color = Color.cyan;  
 Gizmos.DrawSphere(transform.position, 10.0f);  
 }  
 }  
}

1. Track

using UnityEngine;  
  
namespace TubeRace  
{  
 public abstract class **Track** : MonoBehaviour  
 {  
 [Header("Base track properties")] [SerializeField]  
 private float **radius**;  
  
 public float Radius => radius;  
  
 public abstract float Length();  
  
 public abstract Vector3 Position(float distance);  
  
 public abstract Vector3 Direction(float distance);  
  
 public virtual Quaternion Rotation(float distance)  
 {  
 return Quaternion.identity;  
 }  
 }  
}

1. TrackCircle

using System.Collections.Generic;  
using UnityEngine;  
#if UNITY\_EDITOR  
using UnityEditor;  
  
#endif  
  
namespace TubeRace  
{  
#if UNITY\_EDITOR  
 [CustomEditor(typeof(TrackCircle))]  
 public class **RaceTrackRoundEditor** : Editor  
 {  
 public override void OnInspectorGUI()  
 {  
 base.OnInspectorGUI();  
  
 if (GUILayout.Button("Generate"))  
 {  
 ((TrackCircle) target).GenerateTrackData();  
 }  
 }  
 }  
#endif  
  
 public class **TrackCircle** : Track  
 {  
 [SerializeField] private float **circleRadius**;  
 [SerializeField] private int **division**;  
  
 [SerializeField] private Quaternion[] **trackSampledRotations**;  
 [SerializeField] private Vector3[] **trackSampledPoints**;  
 [SerializeField] private float[] **trackSampledSegmentLengths**;  
 [SerializeField] private float **trackSampledLength**;  
  
 public override float Length()  
 {  
 return trackSampledLength;  
 }  
  
 public override Vector3 Position(float distance)  
 {  
 distance = Mathf.Repeat(distance, trackSampledLength);  
  
 for (int i = 0; i < trackSampledSegmentLengths.Length; i++)  
 {  
 float diff = distance - trackSampledSegmentLengths[i];  
  
 if (diff < 0)  
 {  
 float t = distance / trackSampledSegmentLengths[i];  
 return Vector3.Lerp(trackSampledPoints[i], trackSampledPoints[i + 1], t);  
 }  
  
 distance -= trackSampledSegmentLengths[i];  
 }  
  
 return Vector3.zero;  
 }  
  
 public override Vector3 Direction(float distance)  
 {  
 distance = Mathf.Repeat(distance, trackSampledLength);  
  
 for (int i = 0; i < trackSampledSegmentLengths.Length; i++)  
 {  
 float diff = distance - trackSampledSegmentLengths[i];  
 if (diff < 0)  
 return (trackSampledPoints[i + 1] - trackSampledPoints[i]).normalized;  
  
 distance -= trackSampledSegmentLengths[i];  
 }  
  
 return Vector3.forward;  
 }  
  
 public override Quaternion Rotation(float distance)  
 {  
 distance = Mathf.Repeat(distance, trackSampledLength);  
  
 for (int i = 0; i < trackSampledSegmentLengths.Length; i++)  
 {  
 float diff = distance - trackSampledSegmentLengths[i];  
 if (diff < 0)  
 {  
 float t = distance / trackSampledSegmentLengths[i];  
  
 return Quaternion.Slerp(  
 trackSampledRotations[i],  
 trackSampledRotations[i + 1], t  
 );  
 }  
  
 distance -= trackSampledSegmentLengths[i];  
 }  
  
 return Quaternion.identity;  
 }  
  
#if UNITY\_EDITOR  
 [SerializeField] private bool **debugDrawCircle**;  
 [SerializeField] private bool **debugDrawSampledPoints**;  
  
 private static Quaternion GenerateRotation(Vector3 a, Vector3 b, float t)  
 {  
 Vector3 dir = (b - a).normalized;  
 Vector3 up = Vector3.Lerp(a, b, t);  
  
 Quaternion rotation = Quaternion.LookRotation(dir, up);  
 return rotation;  
 }  
  
 private static IEnumerable<Quaternion> GenerateRotations(IReadOnlyList<Vector3> points)  
 {  
 var rotations = new List<Quaternion>();  
 float t = 0;  
  
 for (int i = 0; i < points.Count - 1; i++)  
 {  
 Quaternion rotation = GenerateRotation(points[i], points[i + 1], t);  
 rotations.Add(rotation);  
  
 t += 1.0f / (points.Count - 1);  
 }  
  
 rotations.Add(GenerateRotation(points[points.Count - 1], points[0], t));  
 return rotations.ToArray();  
 }  
  
 public void GenerateTrackData()  
 {  
 Debug.Log("Generating track data");  
  
 var points = new List<Vector3>();  
 var rotations = new List<Quaternion>();  
  
 float divsionf = division;  
 for (int i = 0; i < division; i++)  
 {  
 float angle = 2.0f \* Mathf.**PI** \* i / divsionf;  
 Vector3 newPoints = new Vector3(  
 Mathf.Cos(angle) \* circleRadius, 0, Mathf.Sin(angle) \* circleRadius);  
  
 points.Add(newPoints);  
 }  
  
 trackSampledPoints = points.ToArray();  
 rotations.AddRange(GenerateRotations(trackSampledPoints));  
 trackSampledRotations = rotations.ToArray();  
  
 trackSampledSegmentLengths = new float[trackSampledPoints.Length - 1];  
 trackSampledLength = 0;  
  
 for (int i = 0; i < trackSampledPoints.Length - 1; i++)  
 {  
 Vector3 a = trackSampledPoints[i];  
 Vector3 b = trackSampledPoints[i + 1];  
  
 float segmentLength = (b - a).magnitude;  
 trackSampledSegmentLengths[i] = segmentLength;  
 trackSampledLength += segmentLength;  
 }  
  
 EditorUtility.SetDirty(this);  
 }  
  
 private void DrawSampledTrackPoints()  
 {  
 Handles.DrawAAPolyLine(trackSampledPoints);  
 }  
  
 private void DrawCircleGizmos()  
 {  
 Handles.DrawWireDisc(Vector3.zero, Vector3.up, circleRadius);  
 }  
  
 private void **OnDrawGizmos**()  
 {  
 if (debugDrawCircle)  
 DrawCircleGizmos();  
  
 if (debugDrawSampledPoints)  
 DrawSampledTrackPoints();  
 }  
#endif  
 }  
}

1. TrackCloseBezierCurve

using System.Collections.Generic;  
using UnityEngine;  
#if UNITY\_EDITOR  
using UnityEditor;  
  
#endif  
  
namespace TubeRace  
{  
#if UNITY\_EDITOR  
 [CustomEditor(typeof(TrackClosedBezierCurve))]  
 public class **TrackClosedBezierCurveEditor** : Editor  
 {  
 public override void OnInspectorGUI()  
 {  
 base.OnInspectorGUI();  
  
 if (GUILayout.Button("Generate"))  
 {  
 ((TrackClosedBezierCurve) target).GenerateTrackData();  
 }  
 }  
 }  
#endif  
  
 public class **TrackClosedBezierCurve** : Track  
 {  
 [SerializeField] private TrackDescription **trackDescription**;  
 [SerializeField] private BezierTrackPoint[] **trackPoints**;  
  
 [SerializeField] private int **division**;  
  
 [SerializeField] private Quaternion[] **trackSampledRotations**;  
 [SerializeField] private Vector3[] **trackSampledPoints**;  
 [SerializeField] private float[] **trackSampledSegmentLengths**;  
 [SerializeField] private float **trackSampledLength**;  
  
 public override float Length()  
 {  
 return trackSampledLength;  
 }  
  
 public override Vector3 Position(float distance)  
 {  
 distance = Mathf.Repeat(distance, trackSampledLength);  
  
 for (int i = 0; i < trackSampledSegmentLengths.Length; i++)  
 {  
 float diff = distance - trackSampledSegmentLengths[i];  
  
 if (diff < 0)  
 {  
 float t = distance / trackSampledSegmentLengths[i];  
 return Vector3.Lerp(trackSampledPoints[i], trackSampledPoints[i + 1], t);  
 }  
  
 distance -= trackSampledSegmentLengths[i];  
 }  
  
 return Vector3.zero;  
 }  
  
 public override Vector3 Direction(float distance)  
 {  
 distance = Mathf.Repeat(distance, trackSampledLength);  
  
 for (int i = 0; i < trackSampledSegmentLengths.Length; i++)  
 {  
 float diff = distance - trackSampledSegmentLengths[i];  
 if (diff < 0)  
 return (trackSampledPoints[i + 1] - trackSampledPoints[i]).normalized;  
  
 distance -= trackSampledSegmentLengths[i];  
 }  
  
 return Vector3.forward;  
 }  
  
 public override Quaternion Rotation(float distance)  
 {  
 distance = Mathf.Repeat(distance, trackSampledLength);  
  
 for (int i = 0; i < trackSampledSegmentLengths.Length; i++)  
 {  
 float diff = distance - trackSampledSegmentLengths[i];  
  
 if (diff < 0)  
 {  
 float t = distance / trackSampledSegmentLengths[i];  
  
 return Quaternion.Slerp(  
 trackSampledRotations[i],  
 trackSampledRotations[i + 1],  
 t);  
 }  
  
 distance -= trackSampledSegmentLengths[i];  
 }  
  
 return Quaternion.identity;  
 }  
  
 private static IEnumerable<Quaternion> GenerateRotations(  
 Transform a,  
 Transform b,  
 IReadOnlyList<Vector3> points)  
 {  
 var rotations = new List<Quaternion>();  
 float t = 0;  
  
 for (int i = 0; i < points.Count - 1; i++)  
 {  
 Vector3 direction = (points[i + 1] - points[i]).normalized;  
 Vector3 up = Vector3.Lerp(a.up, b.up, t);  
  
 Quaternion rotation = Quaternion.LookRotation(direction, up);  
 rotations.Add(rotation);  
  
 t += 1.0f / (points.Count - 1);  
 }  
  
 rotations.Add(b.rotation);  
 return rotations.ToArray();  
 }  
  
 private void **Start**()  
 {  
 if (trackDescription != null)  
 trackDescription.SetLength(trackSampledLength);  
 }  
  
#if UNITY\_EDITOR  
 [SerializeField] private bool **debugDrawBezier**;  
 [SerializeField] private bool **debugDrawSampledPoints**;  
  
 private static Vector3[] GenerateBezierPoints(  
 BezierTrackPoint a,  
 BezierTrackPoint b,  
 int division)  
 {  
 Transform aTransform = a.transform;  
 Transform bTransform = b.transform;  
  
 Vector3 aPosition = aTransform.position;  
 Vector3 bPosition = bTransform.position;  
  
 float aLength = a.Length;  
 float bLength = b.Length;  
  
 return Handles.MakeBezierPoints(  
 aPosition,  
 bPosition,  
 aPosition + aTransform.forward \* aLength,  
 bPosition - bTransform.forward \* bLength,  
 division);  
 }  
  
 public void GenerateTrackData()  
 {  
 Debug.Log("Generating track data");  
  
 var points = new List<Vector3>();  
 var rotations = new List<Quaternion>();  
  
 if (trackPoints.Length < 3)  
 return;  
  
 for (int i = 0; i < trackPoints.Length - 1; i++)  
 {  
 var newPoints = GenerateBezierPoints(trackPoints[i], trackPoints[i + 1], division);  
 var newRotations = GenerateRotations(  
 trackPoints[i].transform,  
 trackPoints[i + 1].transform,  
 newPoints);  
  
 rotations.AddRange(newRotations);  
 points.AddRange(newPoints);  
 }  
  
 var lastNewPoints = GenerateBezierPoints(  
 trackPoints[trackPoints.Length - 1],  
 trackPoints[0],  
 division);  
 var lastNewRotations = GenerateRotations(  
 trackPoints[trackPoints.Length - 1].transform,  
 trackPoints[0].transform,  
 lastNewPoints);  
  
 points.AddRange(lastNewPoints);  
 rotations.AddRange(lastNewRotations);  
  
 trackSampledRotations = rotations.ToArray();  
 trackSampledPoints = points.ToArray();  
  
 trackSampledSegmentLengths = new float[trackSampledPoints.Length - 1];  
 trackSampledLength = 0;  
  
 for (int i = 0; i < trackSampledPoints.Length - 1; i++)  
 {  
 Vector3 a = trackSampledPoints[i];  
 Vector3 b = trackSampledPoints[i + 1];  
  
 float segmentLength = (b - a).magnitude;  
 trackSampledSegmentLengths[i] = segmentLength;  
 trackSampledLength += segmentLength;  
 }  
  
 if (trackDescription != null)  
 trackDescription.SetLength(trackSampledLength);  
  
 EditorUtility.SetDirty(this);  
 }  
  
 private static void DrawTrackPartGizmos(BezierTrackPoint a, BezierTrackPoint b)  
 {  
 Transform aTransform = a.transform;  
 Transform bTransform = b.transform;  
  
 Vector3 aPosition = aTransform.position;  
 Vector3 bPosition = bTransform.position;  
  
 float aLength = a.Length;  
 float bLength = b.Length;  
  
 Handles.DrawBezier(  
 aPosition,  
 bPosition,  
 aPosition + aTransform.forward \* aLength,  
 bPosition - bTransform.forward \* bLength,  
 Color.green, Texture2D.whiteTexture, 1.0f);  
 }  
  
 private void DrawBezierCurveGizmos()  
 {  
 if (trackPoints.Length < 3)  
 return;  
  
 for (int i = 0; i < trackPoints.Length - 1; i++)  
 DrawTrackPartGizmos(trackPoints[i], trackPoints[i + 1]);  
  
 DrawTrackPartGizmos(trackPoints[trackPoints.Length - 1], trackPoints[0]);  
 }  
  
 private void DrawSampledTrackPoints()  
 {  
 Handles.DrawAAPolyLine(trackSampledPoints);  
 }  
   
 private void **OnDrawGizmos**()  
 {  
 if (debugDrawBezier)  
 DrawBezierCurveGizmos();  
  
 if (debugDrawSampledPoints)  
 DrawSampledTrackPoints();  
 }  
#endif  
 }  
}

1. TrackLine

using UnityEngine;  
  
namespace TubeRace  
{  
 public class **TrackLine** : Track  
 {  
 [Header("Linear track properties")] [SerializeField]  
 private Transform **start**;  
  
 [SerializeField] private Transform **end**;  
  
 public override float Length()  
 {  
 return (end.position - start.position).magnitude;  
 }  
  
 public override Vector3 Position(float distance)  
 {  
 Vector3 startPosition = start.position;  
 Vector3 direction = end.position - startPosition;  
  
 return startPosition + direction.normalized \* distance;  
 }  
  
 public override Vector3 Direction(float distance)  
 {  
 Mathf.Clamp(distance, 0, Length());  
  
 return (end.position - start.position).normalized;  
 }  
  
 private void **OnDrawGizmos**()  
 {  
 Gizmos.color = Color.green;  
 Gizmos.DrawLine(start.position, end.position);  
 }  
 }  
}

1. ObjectPlacer

using UnityEngine;  
  
namespace TubeRace  
{  
 public class **ObjectPlacer** : MonoBehaviour  
 {  
 [SerializeField] private GameObject **prefab**;  
 [SerializeField] private int **numObjects**;  
 [SerializeField] private Track **track**;  
  
 [SerializeField] private int **seed**;  
 [SerializeField] private bool **canRadomizeRotation**;  
 [Range(0.0f, 1.0f)] [SerializeField] private float **skipProbability**;  
  
 private void **Start**()  
 {  
 Random.InitState(seed);  
 float distance = 0;  
  
 for (int i = 0; i < numObjects; i++)  
 {  
 if (!(Random.Range(0.0f, 1.0f) <= skipProbability))  
 {  
 GameObject go = Instantiate(prefab);  
 go.transform.position = track.Position(distance);  
 go.transform.rotation = track.Rotation(distance);  
  
 if (canRadomizeRotation)  
 go.transform.Rotate(Vector3.forward, Random.Range(0, 360), Space.**Self**);  
 }  
  
 distance += track.Length() / numObjects;  
 }  
 }  
 }  
}

1. SplineMeshProxy

using SplineMesh;  
using UnityEngine;  
#if UNITY\_EDITOR  
using UnityEditor;  
  
#endif  
  
namespace TubeRace  
{  
#if UNITY\_EDITOR  
 [CustomEditor(typeof(SplineMeshProxy))]  
 public class **SplineMeshProxyEditor** : Editor  
 {  
 public override void OnInspectorGUI()  
 {  
 base.OnInspectorGUI();  
  
 if (GUILayout.Button("Update"))  
 {  
 ((SplineMeshProxy) target).UpdatePoints();  
 }  
 }  
 }  
#endif  
  
 [RequireComponent(typeof(Spline))]  
 public class **SplineMeshProxy** : MonoBehaviour  
 {  
 [SerializeField] private BezierTrackPoint **pointA**;  
 [SerializeField] private BezierTrackPoint **pointB**;  
 private Spline spline;  
  
 public void UpdatePoints()  
 {  
 spline = GetComponent<Spline>();  
  
 SplineNode nodeA = spline.nodes[0];  
 Transform transformA = pointA.transform;  
 Vector3 positionA = transformA.position;  
  
 nodeA.Position = positionA;  
 nodeA.Direction = positionA + transformA.forward \* pointA.Length;  
  
 SplineNode nodeB = spline.nodes[1];  
 Transform transformB = pointB.transform;  
 Vector3 positionB = transformB.position;  
  
 nodeB.Position = positionB;  
 nodeB.Direction = positionB + transformB.forward \* pointB.Length;  
 }  
 }  
}

1. ControlDevice

using UnityEngine;  
  
namespace TubeRace  
{  
 public abstract class **ControlDevice** : MonoBehaviour  
 {  
 private HandController handController;  
  
 public void StartMovement(HandController hand)  
 {  
 handController = hand;  
 }  
  
 public void StopMovement()  
 {  
 handController = null;  
 }  
  
 protected abstract void UpdateMovement(HandController hand);  
  
 private void **Update**()  
 {  
 if (handController == null)  
 return;  
  
 UpdateMovement(handController);  
 }  
 }  
}

1. GazePointer

using UnityEngine;  
  
namespace TubeRace  
{  
 public class **GazePointer** : MonoBehaviour  
 {  
 [SerializeField] private Camera **thisCamera**;  
 [Range(1, 50)] [SerializeField] private float **gazeRange**;  
  
 public Vector3 DirectionRelativePlane(Vector3 planePosition)  
 {  
 Transform transformCamera = thisCamera.transform;  
 Ray ray = new Ray(transformCamera.position, transformCamera.forward);  
  
 bool isCollision = Physics.Raycast(  
 ray, out RaycastHit rayHit,  
 gazeRange,  
 LayerMask.GetMask("NavigationPanel")  
 );  
  
 if (isCollision)  
 {  
 Vector3 hitPosition = rayHit.point;  
 Debug.DrawLine(transform.position, hitPosition);  
  
 Vector3 relativePanel = hitPosition - planePosition;  
 return relativePanel;  
 }  
  
 return Vector3.zero;  
 }  
 }  
}

1. HandController

using System;  
using UnityEngine;  
using UnityEngine.InputSystem;  
  
namespace TubeRace  
{  
 public enum HandType  
 {  
 **Default** = 0,  
 **Right** = 1,  
 **Left** = 2  
 }  
  
 public class **HandController** : MonoBehaviour,  
 WrapperInputActions.IXRIRightHandActions,  
 WrapperInputActions.IXRILeftHandActions  
 {  
 [SerializeField] private HandType **handType**;  
  
 private Transform thisTransform;  
 private WrapperInputActions inputActions;  
  
 private Vector3 InputPosition { get; set; }  
 private Quaternion InputRotation { get; set; }  
  
 private float inputSelect;  
  
 private Vector3 lastFramePosition;  
 private bool canUpdatePositionAndRotation;  
  
 private ControlDevice pickedDevice;  
  
 public Vector3 DeltaS()  
 {  
 return InputPosition - lastFramePosition;  
 }  
  
 private void PickupDevice(ControlDevice device)  
 {  
 canUpdatePositionAndRotation = false;  
 pickedDevice = device;  
 device.StartMovement(this);  
 }  
  
 private void ReleaseDevice()  
 {  
 canUpdatePositionAndRotation = true;  
 pickedDevice.StopMovement();  
 pickedDevice = null;  
 }  
  
 public void OnPosition(InputAction.CallbackContext context)  
 {  
 if (lastFramePosition == Vector3.zero)  
 lastFramePosition = InputPosition;  
  
 lastFramePosition = InputPosition;  
 InputPosition = context.ReadValue<Vector3>();  
 }  
  
 public void OnRotation(InputAction.CallbackContext context)  
 {  
 InputRotation = context.ReadValue<Quaternion>();  
 }  
  
 public void OnSelect(InputAction.CallbackContext context)  
 {  
 inputSelect = context.ReadValue<float>();  
  
 if (pickedDevice && inputSelect == 0)  
 ReleaseDevice();  
 }  
  
 private void **OnTriggerStay**(Collider other)  
 {  
 if (pickedDevice != null)  
 return;  
  
 if (other.gameObject.CompareTag("ControlDevice") && inputSelect > 0)  
 {  
 ControlDevice controlDevice = other.GetComponentInParent<ControlDevice>();  
 PickupDevice(controlDevice);  
 }  
 }  
  
 private void **Awake**()  
 {  
 thisTransform = transform;  
 canUpdatePositionAndRotation = true;  
 }  
  
 private void **Start**()  
 {  
 inputActions = new WrapperInputActions();  
  
 switch (handType)  
 {  
 case HandType.**Default**:  
 throw new Exception("Choose hand type");  
 case HandType.**Right**:  
 inputActions.XRIRightHand.SetCallbacks(this);  
 break;  
 case HandType.**Left**:  
 inputActions.XRILeftHand.SetCallbacks(this);  
 break;  
 default:  
 throw new ArgumentOutOfRangeException();  
 }  
  
 inputActions.Enable();  
 }  
  
 private void **Update**()  
 {  
 if (!canUpdatePositionAndRotation)  
 return;  
  
 thisTransform.localPosition = InputPosition;  
 thisTransform.localRotation = InputRotation;  
 }  
 }  
}

1. Lever

using UnityEngine;  
  
namespace TubeRace  
{  
 public class **Lever** : ControlDevice  
 {  
 [SerializeField] private Transform **thisMoveDirection**;  
 [SerializeField] private Transform **thisRotationAxis**;  
 [SerializeField] private float **speed**;  
  
 protected override void UpdateMovement(HandController hand)  
 {  
 Vector3 handDeltaS = hand.DeltaS();  
  
 float moveCoeff = Vector3.Dot(  
 thisMoveDirection.forward,  
 handDeltaS);  
  
 transform.rotation \*= Quaternion.AngleAxis(  
 moveCoeff \* speed \* Time.deltaTime,  
 thisRotationAxis.forward);  
 }  
 }  
}

1. NavigationPanel

using UnityEngine;  
  
namespace TubeRace  
{  
 public class **NavigationPanel** : MonoBehaviour  
 {  
 [SerializeField] private GazePointer **gazePointer**;  
 [Range(0.05f, 5f)] [SerializeField] private float **deadZoneRadius**;  
 [Range(1f, 5f)] [SerializeField] private float **maxRadius**;  
  
 public Vector3 MoveDirection()  
 {  
 Vector3 relPos = gazePointer.DirectionRelativePlane(transform.position);  
 relPos = Vector3.ClampMagnitude(relPos, maxRadius);  
  
 if (relPos.magnitude < deadZoneRadius)  
 relPos = Vector3.zero;  
  
 relPos = new Vector3(  
 Vector3.Dot(transform.right, relPos),  
 Vector3.Dot(transform.up, relPos));  
  
 return relPos;  
 }  
  
 private void **OnDrawGizmos**()  
 {  
 Vector3 position = transform.position;  
  
 Gizmos.color = Color.red;  
 Gizmos.DrawSphere(position, deadZoneRadius);  
  
 Gizmos.color = new Color(0, 0, 1, 0.3f);  
 Gizmos.DrawSphere(position, maxRadius);  
 }  
 }  
}