using UnityEngine;

using System.Collections.Generic;

/// <summary>

/// 车辆行驶中需要避让的障碍物

/// </summary>

public class Barrier : MonoBehaviour

{

public Vector3 position;

public float radius = 4;

public float s;

public LinkedList<Car> waitCars;

}

using System.Collections.Generic;

using UnityEngine;

public class Car : OCar

{

const float MaxVelocityNoRoad = 30;

public Barrier barrier = null;

/\*

public enum State

{

inLine,

crossing,

changing,

prepareCross

}

\*/

public Cross cross;

public Cross preCross;

public Vector3[] crossLine;

/\*

public State state = State.inLine;

/// <summary>

/// 加速度,单位m/s

/// </summary>

public float accel = 0;

/// <summary>

/// 最大加速度

/// </summary>

public float maxAccel;

/// <summary>

/// 速度,单位km/h

/// </summary>

public float velocity = 0;

/// <summary>

/// 路径长度

/// </summary>

public float s = 0;

\*/

/// <summary>

/// 车辆行驶所在车道

/// </summary>

public new Line line

{

get { return (Line)base.line; }

set { base.line = value; }

}

/\*

/// <summary>

/// 路线的T参数

/// </summary>

public float lineT;

/// <summary>

/// 路点信息

/// </summary>

///

public Vector3[] linePoints;

public float segment;

/// <summary>

/// 目标点

/// </summary>

public Vector3 target = new Vector3(1, 0, 0);

public float expectVelocity;

public static float[] expects = { 30, 40, 50, 60, 70 };

/// <summary>

/// 修改该属性发出换道指令

/// </summary>

public bool lineChange = false;

/// <summary>

/// 车辆突然停下的测试

/// </summary>

public bool stopTest = false;

/// <summary>

/// 车辆跟驰对象，通过该对象限制加速度（包括路内跟驰，换道，路口内所有情形）

/// 解决冲突问题，车辆与阻碍其行驶的车辆构成一个跟驰行为，确保不会相撞

/// 维护一个临近范围车辆集，当触发车辆进入时即加入该集合，集合内寻找一个disOfForward最短的做跟驰

/// </summary>

public Car followCar;

\*/

void Awake()

{

this.expectVelocity = Car.expects[(int)Random.Range(0f, (float)Car.expects.Length)];

}

void Start()

{

GameEvents.current.OnLoadEvent += DestroyCar;

if (RectangleSelector.current != null)

RectangleSelector.current.Selectable.Add(this.gameObject);

}

void OnDestroy()

{

if (RectangleSelector.current != null)

{

RectangleSelector.current.Selectable.Remove(this.gameObject);

RectangleSelector.current.Selected.Remove(this.gameObject);

}

}

/\*

/// <summary>

/// 将车辆速度转换为m/s

/// </summary>

public new float Km2m()

{

return this.velocity / 3.6f;

}

\*/

/\*

public new Car PreCar()

{

return (Car)base.PreCar();

}

\*/

public new Car PreCar()

{

if (this.line.cars.Find(this) == null)

{

return null;

}

if (this.line.cars.Find(this).Previous == null)

{

return null;

}

return (Car)this.line.cars.Find(this).Previous.Value;

}

/\*

public new Car NextCar()

{

return (Car)base.NextCar();

}\*/

public new Car NextCar()

{

if (this.line.cars.Find(this) == null)

{

return null;

}

if (this.line.cars.Find(this).Next == null)

{

return null;

}

return (Car)this.line.cars.Find(this).Next.Value;

}

public Car CarClosest(Line line)

{

if (line.cars.First == null)

{

return null;

}

Car pointer = (Car)line.cars.First.Value;

while (pointer.NextCar() != null)

{

if (Vector3.Distance(this.transform.position, pointer.transform.position) < Vector3.Distance(this.transform.position, pointer.NextCar().transform.position))

{

break;

}

else

{

pointer = pointer.NextCar();

}

}

return pointer;

}

public void DestroyCar()

{

if (this.line != null)

{

this.line.cars.Remove(this);

}

Destroy(this.gameObject);

}

public void setLine(Line l)

{

if (this.line != null)

{

this.line.cars.Remove(this);

}

line = l;

lineT = 0;

linePoints = l.points;

segment = Line.segmentNum;

l.cars.AddLast(this);

}

/// <summary>

/// target在pointer之后则返回真

/// </summary>

public static bool judgeLocation(Car pointer, Car target)

{

if (pointer == null || target == null)

{

return true;

}

Vector3 dir1 = pointer.transform.forward.normalized;

Vector3 dir2 = (target.transform.position - pointer.transform.position).normalized;

if (Vector3.Dot(dir1, dir2) < 0)

{

return true;

}

else

{

return false;

}

}

public Car findNextCar(Line l)

{

LinkedListNode<OCar> pointer = l.cars.First;

while (pointer != null)

{

//在该车流中找到第一个在car后面的车辆

if (!judgeLocation(pointer.Value, this))

{

return (Car)pointer.Value;

}

pointer = pointer.Next;

}

return null;

}

public void changeLine(Line l)

{

Car target = findNextCar(l);

if (target == null)

{

//车流未找到插入位置，在末端插入

l.cars.AddLast(this);

this.line = l;

this.linePoints = l.points;

}

else

{

l.cars.AddBefore(l.cars.Find(target), this);

this.line = l;

this.linePoints = l.points;

}

}

/// <summary>

/// 找到路口内路径，以点组形式返回

/// 会改变车辆的line与linePoints

/// </summary>

public void findPath()

{

if (state == State.inLine) return;

int rdm1 = Random.Range(0, line.nextRoads.Count - 1);//确定道路

int rdm2 = 0;

//找到车辆数最少的车道

for (int i = 0; i < line.nextRoads[rdm1].lines.Length; i++)

{

rdm2 = line.nextRoads[rdm1].lines[i].cars.Count < line.nextRoads[rdm1].lines[rdm2].cars.Count ? i : rdm2;

}

//int rdm2 = Random.Range(0, line.nextRoads[rdm1].lines.Length-1);//确定车道

Line nextLine = line.nextRoads[rdm1].lines[rdm2];

//linePoints = Line.Interpolation(line, nextLine);

linePoints = Line.linkLine(line, nextLine);

line = nextLine;

}

public void driving()

{

//更新linT与下一个目标点

while (Vector3.Distance(target, transform.position) <= 2f)

{

lineT += (float)1 / (float)segment;

target = Line.Bezier(lineT, linePoints);

}

//车辆朝向目标点

if (target != transform.position)

transform.LookAt(target);

if (stopTest == true)

{

velocity = 0;

return;

}

//道路限速；车辆期望速度；正常行驶速度；取最小值

velocity = Mathf.Min(this.line == null ? Car.MaxVelocityNoRoad : this.line.maxVelocity, velocity + 3.6f \* accel \* Time.deltaTime, expectVelocity);

//屏蔽掉速度小于0的倒车行为

velocity = Mathf.Max(0, velocity);

s += Km2m() \* Time.deltaTime;

this.transform.Translate(Vector3.forward \* Km2m() \* Time.deltaTime);

}

public float disOfForward(Car other)

{

Vector3 forward = this.transform.forward.normalized;

Vector3 spacing = other.transform.position - this.transform.position;

return Vector3.Dot(forward, spacing);

}

}

using UnityEngine;

using System.Collections.Generic;

/// <summary>

/// 处理车辆间冲突的一个全局冲突系统

/// </summary>

public static class CollisionSystem

{

/// <summary>

/// 在全局维护的冲突域map

/// </summary>

public static Dictionary<KeyValuePair<Line, Line>, Barrier> newbarriers = new Dictionary<KeyValuePair<Line, Line>, Barrier>();

/// <summary>

/// 用一个map缓存路口内的行驶路径

/// </summary>

public static Dictionary<KeyValuePair<Line, Line>, Vector3[]> crossRunPoints = new Dictionary<KeyValuePair<Line, Line>, Vector3[]>();

/// <summary>

/// 匹配行驶路径与路口

/// </summary>

public static Dictionary<Vector3[], Cross> runPoints2Cross = new Dictionary<Vector3[], Cross>();

private static Dictionary<KeyValuePair<Car, Car>, Vector3> barriers = new Dictionary<KeyValuePair<Car, Car>, Vector3>();

//阻碍车辆的障碍物可能有多种，从每一个障碍物都可以获取一个加速度，在所有加速度中取最小值做最小加速度

/// <summary>

/// 维护特定车辆的所有障碍物集合

/// </summary>

private static Dictionary<Car, List<Vector3>> car2barriers = new Dictionary<Car, List<Vector3>>();

/// <summary>

/// 当两个车辆行驶路径发生冲突时，选择一个行驶条件更好的车辆继续行驶，另一车辆让行

/// </summary>

/// <param name="car1"></param>

/// <param name="car2"></param>

/// <returns></returns>

public static Car ChooseLucky(Car car1,Car car2)

{

//判断车辆的前后关系，前车继续行驶，后车让行

//“车辆在车道方向的前后关系”采取一种近似手段

//即将冲突车辆行驶方向相加得到一个夹角方向，按两车在该夹角方向投影前后判断车辆前后关系

var targetLine = car1.transform.forward + car2.transform.forward;

var car1Len = Vector3.Dot(car1.transform.forward, targetLine)/targetLine.magnitude;

var car2Len = Vector3.Dot(car2.transform.forward, targetLine)/targetLine.magnitude;

return car1Len > car2Len ? car1 : car2;

}

//Calculate the intersection point of two lines. Returns true if lines intersect, otherwise false.

//Note that in 3d, two lines do not intersect most of the time. So if the two lines are not in the

//same plane, use ClosestPointsOnTwoLines() instead.

/// <summary>

/// 计算两射线交点

/// </summary>

/// <param name="linePoint1"></param>

/// <param name="lineVec1"></param>

/// <param name="linePoint2"></param>

/// <param name="lineVec2"></param>

/// <returns></returns>

public static Vector3 LineLineIntersection(Vector3 linePoint1, Vector3 lineVec1, Vector3 linePoint2, Vector3 lineVec2)

{

Vector3 lineVec3 = linePoint2 - linePoint1;

Vector3 crossVec1and2 = Vector3.Cross(lineVec1, lineVec2);

Vector3 crossVec3and2 = Vector3.Cross(lineVec3, lineVec2);

Vector3 intersection;

float planarFactor = Vector3.Dot(lineVec3, crossVec1and2);

//is coplanar, and not parrallel

if (Mathf.Abs(planarFactor) < 0.0001f && crossVec1and2.sqrMagnitude > 0.0001f)

{

float s = Vector3.Dot(crossVec3and2, crossVec1and2) / crossVec1and2.sqrMagnitude;

intersection = linePoint1 + (lineVec1 \* s);

}

else

{

intersection = Vector3.zero;

}

return intersection;

}

/// <summary>

/// car1阻碍car2，求出car2可能的barrier

/// </summary>

/// <param name="car1"></param>

/// <param name="car2"></param>

/// <returns></returns>

private static Vector3 GetNewBarrier(Car car1,Car car2)

{

if (Vector3.Dot(car1.transform.forward, car2.transform.forward) / car1.transform.forward.magnitude / car2.transform.forward.magnitude <= 5)

{

return Vector3.zero;

}

//车辆朝向向量的交点即为barrier

return LineLineIntersection(car1.transform.position,car1.transform.forward,car2.transform.position,car2.transform.forward);

}

/// <summary>

/// 判断两辆车的行驶情况，判断是谁阻碍了交通

/// </summary>

/// <param name="car1"></param>

/// <param name="car2"></param>

public static void GetBarrier(Car car1,Car car2)

{

var key1 = new KeyValuePair<Car, Car>(car1, car2);

var key2 = new KeyValuePair<Car, Car>(car2, car1);

//一对车辆只需要一个车辆让行

if ( !barriers.ContainsKey(key1) && !barriers.ContainsKey(key2) ){

//判断两辆车谁先行驶

//若car2先行，则交换car次序，保证car1先行

if(ChooseLucky(car1, car2) == car2)

{

var temp = car1;

car1 = car2;

car2 = temp;

}

Vector3 newBarrier = GetNewBarrier(car1, car2);

//两车行驶方向平行

if(newBarrier == Vector3.zero)

{

return;

}

barriers.Add(new KeyValuePair<Car, Car>(car1,car2), newBarrier);

if (!car2barriers.ContainsKey(car2))

{

car2barriers.Add(car2, new List<Vector3>());

}

car2barriers[car2].Add(newBarrier);

return;

}

}

/// <summary>

/// 前车行驶一段时间后，后车即失去前车的阻碍，可以删除前车对后车产生的一个障碍物

/// </summary>

/// <param name="car1"></param>

/// <param name="car2"></param>

public static void DelBarrier(Car car1,Car car2)

{

if( !barriers.ContainsKey(new KeyValuePair<Car, Car>(car1, car2)) ){

var temp = car1;

car1 = car2;

car2 = temp;

}

if( !barriers.ContainsKey(new KeyValuePair<Car, Car>(car1, car2)))

{

return;

}

var delVarriers = barriers[new KeyValuePair<Car, Car>(car1, car2)];

barriers.Remove(new KeyValuePair<Car, Car>(car1, car2));

/\*if (car2barriers.ContainsKey(car2))

{\*/

car2barriers[car2].Remove(delVarriers);

//}

}

/// <summary>

/// 求出在存在障碍物的情况下，让行车辆所能获得的最大加速度

/// </summary>

/// <param name="car"></param>

/// <returns></returns>

public static float ToGiveWay(Car car)

{

if(car.state != Car.State.crossing)

{

return car.accel;

}

float ret = car.accel;

if( car2barriers.ContainsKey(car))

{

foreach (Vector3 barrier in car2barriers[car])

{

ret = Mathf.Min(ret, Mathf.Pow(car.Km2m(), 1.5f) \* (-car.Km2m()) / Mathf.Pow(Vector3.Distance(barrier, car.transform.position), 1));

}

}

return ret;

}

}

using UnityEngine;

using System.Collections.Generic;

//路口实例为一个大型的平面透明物体，负责处理车辆在道路之间的调度

//路口类需要根据路口进入车流量对车辆进行概率上的分发（确定车辆进入车道与驶出车道）

//车辆调用链:进入路口区域->选择想要驶出的道路（权重随机）-> 选择能够到达驶出道路的Line并行驶 -> 在目标Road中选择一条目标Line（随机选择或是选择价值系数最优）

//->由所在Line与目标Line算出行驶曲线

//需要注意的是，在路口中并不允许换道超车等行为，于是其中只有跟驰与避让行为，避让可以在车辆前进路线上设置一虚拟车辆使其跟驰。

//车辆流量的设置，采用交叉口流量设置的话，是在每一个RoadIn设置到每一个方向Road的流量，故对于RoadIn->RoadOut的映射需要附带一个流量数据集

public class Cross : MonoBehaviour

{

/// <summary>

/// 已经进入路口中的车辆集合

/// </summary>

public LinkedList<Car> cars;

/// <summary>

/// 道路类与道路包装类的key—value集合

/// </summary>

Dictionary<Road, RoadIn> RoadMap;

/// <summary>

/// 记录进入路口车辆驶出的目标道路

/// </summary>

public Dictionary<Car, Road> carRoadOut;

/// <summary>

/// 车辆与路口碰撞题碰撞中触发，道路在路口中时自动初始化

/// </summary>

/// <param name="other"></param>

private void OnTriggerStay(Collider other)

{

//与路口碰撞的是道路，需要将该道路映射入路口中

if (other.gameObject.GetComponent<Road>() != null)

{

Road road = other.gameObject.GetComponent<Road>();

//已经初始化过该道路，快速返回

if (RoadMap.ContainsKey(road))

{

return;

}

Vector3 pos = road.lines[0].points[road.lines[0].points.Length - 1];

//道路尾端接入路口，需要初始化该道路在路口类中的映射

if (this.gameObject.GetComponent<SphereCollider>().bounds.Contains(pos))

{

Dictionary<Road, int> stream = new Dictionary<Road, int>();

int totalCar = 0;

foreach (Line line in road.lines)

{

foreach (Road nextRoad in line.nextRoads)

{

if (stream.ContainsKey(nextRoad))

{

stream[nextRoad] += 10;

}

else

{

stream.Add(nextRoad, 10);

}

}

}

foreach (KeyValuePair<Road, int> rvi in stream)

{

totalCar += rvi.Value;

}

RoadMap[road] = new RoadIn(stream, totalCar);

}

}

//车辆与路口碰撞，说明车辆进入路口了，需要为路口中的行驶做准备

if (other.gameObject.GetComponent<Car>() != null)

{

Car car = other.gameObject.GetComponent<Car>();

//车辆行驶的车道没有下一条路

if (car.line != null && car.line.nextRoads.Count == 0)

{

return;

}

//已经驶出路口

if (car.preCross == this)

{

return;

}

//判断之前车辆是否已经为驶入路口做好初始化

if (car.crossLine != null && car.crossLine.Length != 0)

{

return;

}

//如果车辆将要进入路口时处于转弯状态，暂不初始化

if (car.state != Car.State.inLine)

{

return;

}

//车辆即将进入路口，初始化一些数据为车辆路口中的行驶做好准备

cars.AddLast(car);

car.state = Car.State.prepareCross;

car.cross = this;

return;

}

}

/// <summary>

/// 车辆进入路口区域触发，为路口中的行驶做好初始化

/// </summary>

/// <param name="other"></param>

void OnTriggerEnter(Collider other)

{

//如果与路口碰撞的不是车辆，跳过该流程

//其他初始化过程与OnTriggerStay中相同

if (other.gameObject.GetComponent<Car>() != null) {

Car car = other.gameObject.GetComponent<Car>();

if (car.line != null && car.line.nextRoads.Count == 0)

{

return;

}

cars.AddLast(car);

car.cross = this;

if (car.state != Car.State.inLine)

{

return;

}

car.state = Car.State.prepareCross;

return;

}

}

/// <summary>

/// 根据道路车流量权重随机为驶入路口的车辆选择选择驶出道路

/// </summary>

/// <param name="car"></param>

/// <returns></returns>

public Road FindRoadOut(Car car)

{

int a = Random.Range(0, 100);

if (a < 70)

{

return car.line.nextRoads[Random.Range(0, car.line.nextRoads.Count)];

}

RoadIn roadIn = RoadMap[car.line.fatherRoad];

float rand = Random.Range(0,roadIn.totalCars);

int temp = 0;

foreach (KeyValuePair<Road,int> kvp in roadIn.roadOutStream)

{

temp += kvp.Value;

if (rand <= temp)

{

carRoadOut[car] = kvp.Key;

return kvp.Key;

}

}

Debug.LogError("RoadIn streams set error !");

return null;

}

/// <summary>

/// 车辆权重随机选择驶出道路后，需要行驶到能够前往驶出道路的车道

/// </summary>

/// <param name="car"></param>

/// <param name="roadOut"></param>

/// <returns></returns>

public Line FindLineIn(Car car,Road roadOut)

{

Road roadIn = car.line.fatherRoad;

//循环遍历Car行驶Road中的每一条Line，判断是否能够行驶到roadOut

for (int i=0; i < roadIn.lines.Length; i++)

{

//判断起始位置从当前车道开始

int j = (i + car.line.indexInRoad()) % roadIn.lines.Length;

foreach(Road nextRoad in roadIn.lines[j].nextRoads)

{

if(nextRoad == roadOut)

{

return roadIn.lines[j];

}

}

}

Debug.LogError("Cross.FindLineIn error");

return null;

}

/// <summary>

/// 从已经选择好的roadOut中选择一条“较好”的道路

/// </summary>

/// <param name="roadOut"></param>

/// <param name="linein"></param>

/// <returns></returns>

public Line FindLineOut(Road roadOut,Line linein)

{

//暂时选择随机选取

//考虑到同源车辆在路口内避免碰撞的行为，我们让车辆只会行驶到对应道路的半区

if(linein.indexInRoad() < linein.fatherRoad.lines.Length / 2)

{

return roadOut.lines[Random.Range(0, roadOut.lines.Length/2)];

}

else

{

return roadOut.lines[Random.Range(roadOut.lines.Length/2, roadOut.lines.Length)];

}

}

//路口的信号交通配时方案中，考虑到Line末尾的信号灯实际上是控制这一条Line的车辆进入路口的限制

//而不会分辨驶出道路

//所以只需要在路口中维护一个Line与其末尾Light的映射集合，Light随着时间变化改变自身状态

//Line中的车辆只需要在进入路口之前访问Light状态

//映射关系与Light变化逻辑都放在Cross中，只需要暴露一个根据Line查看Light的接口

/// <summary>

/// 维护LineOut对应的交通配时方案

/// </summary>

Dictionary<Line, LightTimeSet> LightSet;

/// <summary>

/// 维护末端接入路口的Line与Light映射关系

/// </summary>

Dictionary<Line, CrossLight> LineLight;

/// <summary>

/// 随时间增长的线性值

/// </summary>

float LightTime;

/// <summary>

/// 该交叉口的信号周期配时

/// </summary>

float CycleTime;

/// <summary>

/// 向前端暴露设置信号周期方案的接口

/// </summary>

/// <param name="line"></param>

/// <param name="lightSet"></param>

/// <param name="timeSet"></param>

public void SetLineLightTime(Line line,List<CrossLight> lightSet, List<float> timeSet)

{

LightTimeSet set= new LightTimeSet();

set.lightSet = lightSet;

set.timeSet = timeSet;

LightSet.Add(line, set);

}

/// <summary>

/// 向外暴露获取车道红绿灯状态的接口

/// </summary>

/// <param name="line"></param>

/// <returns></returns>

public CrossLight JudgeLight(Line line)

{

if (!LineLight.ContainsKey(line))

{

Debug.LogError("this line no light");

}

return LineLight[line];

}

/// <summary>

/// 初始化路口中的对象

/// </summary>

void Start()

{

cars = new LinkedList<Car>();

RoadMap = new Dictionary<Road, RoadIn>();

carRoadOut = new Dictionary<Car, Road>();

LightSet = new Dictionary<Line, LightTimeSet>();

LineLight = new Dictionary<Line, CrossLight>();

}

/// <summary>

/// 控制路口中的红绿灯随时间变化

/// </summary>

private void Update()

{

LightTime += Time.deltaTime;

LightTime %= CycleTime;

//路口中信号灯变化逻辑

foreach ( KeyValuePair<Line,LightTimeSet> line in LightSet){

//一个周期内的红绿灯信号变化

if(LightTime > line.Value.timeSet[line.Value.index])

{

line.Value.index++;

LineLight[line.Key] = line.Value.lightSet[line.Value.index];

}

//开启一个新的周期

if(LightTime < line.Value.timeSet[0] && line.Value.index == line.Value.lightSet.Count)

{

line.Value.index = 0;

LineLight[line.Key] = line.Value.lightSet[line.Value.index];

}

}

}

}

/// <summary>

/// 对接入路口的道路构建一个包装类

/// </summary>

class RoadIn

{

/// <summary>

/// 接入路口道路的权重车流

/// </summary>

public Dictionary<Road, int> roadOutStream;

/// <summary>

/// 总权值

/// </summary>

public int totalCars;

public RoadIn(Dictionary<Road, int> stream, int totalCar)

{

this.roadOutStream = stream;

this.totalCars = totalCar;

}

}

/// <summary>

/// 红绿灯状态

/// </summary>

public enum CrossLight

{

RED,

YELLOW,

GREEN

}

/// <summary>

/// 红绿灯配时方案

/// </summary>

class LightTimeSet

{

public List<CrossLight> lightSet;

public List<float> timeSet;

public int index;

}

using System;

using UnityEngine;

[DisallowMultipleComponent]

public class GameEvents : MonoBehaviour

{

public static GameEvents current;

public event Action OnLoadEvent;

public event Action<int> OnDeleteEvent;

void Awake()

{

current = this;

}

public void DispatchOnLoad()

{

if (OnLoadEvent != null)

{

OnLoadEvent();

}

}

public void OnDelete(int id)

{

if (OnDeleteEvent != null)

{

OnDeleteEvent(id);

}

}

}

using System;

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// 车道生成车辆预制件的脚本

/// </summary>

public class GeneratePoint : MonoBehaviour

{

public GameObject car;

public Car[] cars = new Car[50];

private int t = 0;

private Line line;

public static GeneratePoint m\_generatePoint;

private float timer = 0;

private float intervalTime = 1;//生成间隔

public Car latestCar = null;

/// <summary>

/// 初始化生产点

/// </summary>

private void Start()

{

m\_generatePoint = this;

line = this.GetComponent<Line>();

}

/// <summary>

/// 每帧更新计时器

/// </summary>

private void Update()

{

timer += Time.deltaTime;

if(timer>=intervalTime )

{

GenerateCar();

timer = 0;

}

}

/// <summary>

/// 生成车辆方法

/// </summary>

public void GenerateCar()

{

GameObject go = GameObject.Instantiate(car, line.lineStart, Quaternion.identity);

go.GetComponent<Car>().setLine(line);

if (t < 50)

{

cars[t++] = go.GetComponent<Car>();

}

}

}

using System.Collections;

using System.Collections.Generic;

using System.Data;

using UnityEngine;

public class Line : OLine

{

/// <summary>

/// 车道渲染器

/// </summary>

private LineRenderer lineRenderer;

/// <summary>

/// 车道的贝塞尔控制点

/// </summary>

public Vector3[] points;

/// <summary>

/// 车道离散化的轨迹点个数

/// </summary>

public const int segmentNum = 100;

/// <summary>

/// 车道所属道路

/// </summary>

public Road fatherRoad;

/// <summary>

/// 车道限速

/// </summary>

public float maxVelocity;

public Car lineLock;

/// <summary>

/// 车道所能通向的下一条道路

/// </summary>

public List<Road> nextRoads;

public Vector3 lineStart { get => points[0]; }

public Vector3 lineEnd { get => points[points.Length -1]; }

public Vector3 startVector{ get => (points[1] - points[0]).normalized; }

public Vector3 endVector { get => (points[points.Length - 1] - points[points.Length - 2]).normalized; }

/// <summary>

/// 获取车道在道路中的下标

/// </summary>

/// <returns></returns>

public int indexInRoad()

{

for (int i = 0; i < fatherRoad.lines.Length; i++)

{

if (fatherRoad.lines[i] == this)

{

return i;

}

}

return -1;

}

/// <summary>

/// 初始化车道相关字段

/// </summary>

private void Start()

{

lineRenderer = GetComponent<LineRenderer>();

lineRenderer.sortingLayerID = 0;

cars = new LinkedList<OCar>();

maxVelocity = 70;

if (RectangleSelector.current != null)

{

RectangleSelector.current.Selectable.Add(this.gameObject);

}

}

/// <summary>

/// 渲染车道，使其可见

/// </summary>

private void Update()

{

//获取子点对象的transform

Transform[] pointTran = GetComponentsInChildren<Transform>();

points = new Vector3[pointTran.Length - 1];

for (int i = 1; i < pointTran.Length; i++)

{

//获取子点对象的坐标

points[i - 1] = pointTran[i].position;

}

DrawCurve();

}

private void OnDestroy()

{

RectangleSelector.current.Selectable.Remove(this.gameObject);

GameEvents.current.OnDeleteEvent -= DestroySelf;

}

private void DestroySelf(int id)

{

if (id == gameObject.GetInstanceID())

{

RectangleSelector.current.Selectable.Remove(this.gameObject);

Destroy(this.gameObject);

}

}

/// <summary>

/// 根据车道的贝塞尔控制点绘制车道

/// </summary>

private void DrawCurve()

{

for (int i = 1; i <= segmentNum; i++)

{

float t = (float)i / (float)segmentNum;

Vector3 pixel = Bezier(t, points);

lineRenderer.positionCount = i;

lineRenderer.SetPosition(i - 1, pixel);

}

}

/// <summary>

/// 根据贝塞尔曲线控制点与T值得到轨迹中的点

/// </summary>

/// <param name="t"></param>

/// <param name="p"></param>

/// <returns></returns>

public static Vector3 Bezier(float t, Vector3[] p)

{

Vector3 ans = Vector3.zero;

int n = p.Length;

for (int i = 0; i < n; i++)

{

ans += p[i] \* C(i, n - 1) \* Mathf.Pow(t, i) \* Mathf.Pow(1 - t, n - 1 - i);

}

return ans;

}

/// <summary>

/// 求出点在贝塞尔曲线上对应的T值，简单二分法

/// </summary>

public static float CalculateT(Vector3 point, Vector3[] p)

{

float start = 0f;

float end = 1f;

float mid = (start + end) / 2;

while (Vector3.Distance(point, Bezier(mid, p)) >= 2f)

{

if (Vector3.Distance(point, Bezier(start, p)) > Vector3.Distance(point, Bezier(end, p)))

{

start = mid + 0.01f;

}

else

{

end = mid;

}

mid = (start + end) / 2;

}

return mid;

}

/// <returns>

/// n! / m!(n-m)!

/// </returns>

private static int C(int m, int n)

{

int ans = 1;

for (int i = 1; i <= n; i++)

ans = ans \* i;

for (int i = 1; i <= m; i++)

ans = ans / i;

for (int i = 1; i <= n - m; i++)

ans = ans / i;

return ans;

}

/// <summary>

/// 两条Line之间自动生成一条平滑曲线，以路点形式返回

/// </summary>

public static Vector3[] Interpolation(Line line1, Line line2)

{

Vector3[] ans = new Vector3[3];

float b = ((line2.lineStart.x - line1.lineEnd.x) \* line1.endVector.z

- (line2.lineStart.z - line1.lineEnd.z) \* line1.endVector.x) / (line2.startVector.z \* line1.endVector.x - line2.startVector.x \* line1.endVector.z);

//float a = (line2.lineStart.x - line1.lineEnd.y + b \* line2.startVector.x) / line1.endVector.x;

ans[0] = line1.lineEnd;

if (b > 999999 || line2.startVector.z \* line1.endVector.x - line2.startVector.x \* line1.endVector.z == 0)

ans[1] = (line1.lineEnd + line2.lineStart) / 2;

else

ans[1] = new Vector3(line2.lineStart.x + b \* line2.startVector.x, (line1.lineEnd.y + line2.lineStart.y) / 2, line2.lineStart.z + b \* line2.startVector.z);

ans[2] = line2.lineStart;

return ans;

}

/// <summary>

/// 路径生成算法需要的贝塞尔控制点，供linkLine调用

/// </summary>

private static Vector3 controlPoint(Vector3 now1, Vector3 now2, Vector3 target1, Vector3 target2)

{

if (target1[2] - target2[2] == 0)

{

return new Vector3(now2.x, 0, (now2.z + target1.z) / 2);

}

else if (target1[0] - target2[0] == 0)

{

return new Vector3((now2.x + target1.x) / 2, 0, now2.z);

}

else

{

double a1 = (now1[2] - now2[2]) / (now1[0] - now2[0]);

double b1 = now1[2] - a1 \* now1[0];

double a2 = (target1[2] - target2[2]) / (target1[0] - target2[0]);

double b2 = target1[2] - a2 \* target1[0];

double x = (b2 - b1) / (a1 - a2);

double y = a1 \* x + b1;

return new Vector3((float)(x + now2[0]) / 2, 0, (float)(y + now2[2]) / 2);

}

}

/// <summary>

/// 作用同Interpolation, 算法不同

/// </summary>

public static Vector3[] linkLine(Line now, Line target)

{

Vector3[] result = new Vector3[4];

result[0] = now.points[now.points.Length - 1];

result[1] = controlPoint(now.points[now.points.Length - 2], now.points[now.points.Length - 1], target.points[0], target.points[1]);

result[2] = controlPoint(target.points[1], target.points[0], now.points[now.points.Length - 1], now.points[now.points.Length - 2]);

result[3] = target.points[0];

return result;

}

/// <summary>

/// 新的路径生成算法，适用于更普适的场景

/// </summary>

/// <param name="now">当前道路</param>

/// <param name="target">目标道路</param>

/// <returns>生成曲线</returns>

public static Vector3[] linkLine2(Line now,Line target)

{

Vector3[] result = new Vector3[4];

var len = Vector3.Distance(now.points[now.points.Length - 1], target.points[0])/3 ;

result[0] = now.points[now.points.Length - 1];

//计算第一个非端点贝塞尔控制点

var a1 = now.points[now.points.Length - 2];

var a2 = now.points[now.points.Length - 1];

var x = len / Vector3.Distance(a1, a2)\*(a2.x-a1.x);

var z = len / Vector3.Distance(a1, a2)\*(a2.z-a1.z);

result[1].x = a2.x + x;

result[1].z = a2.z + z;

//计算第二个非端点贝塞尔控制点

a1 = target.points[0];

a2 = target.points[1];

x = len / Vector3.Distance(a1, a2) \* (a2.x - a1.x);

z = len / Vector3.Distance(a1, a2) \* (a2.z - a1.z);

result[2].x = a1.x - x;

result[2].z = a1.z - z;

result[3] = target.points[0];

return result;

}

}

using UnityEngine;

/// <summary>

/// OriginRoad作为生产车辆的脚本，生成的车辆按照泊松分布

/// </summary>

[DisallowMultipleComponent]

public class OriginRoad : MonoBehaviour//泊松分布的交通流

{

/// <summary>

/// 生产对象

/// </summary>

public GameObject Car;

/// <summary>

/// 附着对象上的road脚本

/// </summary>

public Road originRoad;

private float z;//单位时间内平均到达数（辆/s）

private float m;//泊松分布下车辆到达率( m = zt)

private float t = 0;

/// <summary>

/// 初始化道路生产脚本

/// </summary>

void Start()

{

this.originRoad = this.GetComponent<Road>();

z = 1f;

}

/// <summary>

/// 每一帧按照泊松分布概率生成车辆

/// </summary>

void Update()

{

/\*t += Time.deltaTime;

m = t \* z;\*/

m = Time.deltaTime \* z;

double p = m \* Mathf.Exp(-m);

if(p> UnityEngine.Random.Range(0f,1f))

{

GenerateCar();

//t = 0;

}

}

/// <summary>

/// 生产车辆预制件的方法，每帧调用

/// </summary>

private void GenerateCar()

{

//道路中随机选择一个车道

Line line = originRoad.lines[UnityEngine.Random.Range(0, originRoad.lines.Length)];

//上次同车道生产的车辆未走远时，放弃生产车辆

if (line.cars.Last != null && line.cars.Last.Value.s <= Car.transform.localScale.z)

{

return;

}

//生成车辆预制件实例

GameObject go = GameObject.Instantiate(Car, line.lineStart, Quaternion.identity);

Car car = go.GetComponent<Car>();

//初始化车辆相关属性

car.line = line;

car.lineT = 0;

car.linePoints = line.points;

car.segment = Line.segmentNum;

line.cars.AddLast(car);

//车辆获得一个随机初速度

car.velocity = UnityEngine.Random.Range(20, 30);

}

}

using UnityEngine;

public class Point : MonoBehaviour

{

/// <summary>

/// 可视化设置道路起终点所需要保留的属性

/// </summary>

private float scaleX;

private float scaleY;

private float scaleZ;

private Vector3 mouseOffset;

private Vector3 screenPoint;

/// <summary>

///

/// </summary>

void Update()

{

// fix size

transform.localScale = new Vector3(0.5f / transform.parent.localScale.x, 0.5f / transform.parent.localScale.y, 0.5f / transform.parent.localScale.z);

}

/// <summary>

/// 按下鼠标时触发

/// </summary>

void OnMouseDown()

{

screenPoint = Camera.main.WorldToScreenPoint(transform.position);

mouseOffset = gameObject.transform.position - Camera.main.ScreenToWorldPoint(new Vector3(Input.mousePosition.x, Input.mousePosition.y, screenPoint.z));

}

/// <summary>

/// 鼠标按住并拖动改变贝塞尔控制点位置

/// </summary>

void OnMouseDrag()

{

//改变贝塞尔控制点位置

Vector3 currentScreenPoint = new Vector3(Input.mousePosition.x, Input.mousePosition.y, screenPoint.z);

Vector3 currentPosition = Camera.main.ScreenToWorldPoint(currentScreenPoint) + mouseOffset;

transform.position = currentPosition;

GetComponentInParent<LaneMesh>().RecalculateVerticesPosition();

}

}

using UnityEngine;

public class Road : MonoBehaviour

{

/// <summary>

/// 记录道路位置与ID

/// </summary>

private int id;

public int Id { get => id; set => id = value; }

private Vector3 prevPosition;

private Vector3 nowPosition;

public GameObject roadObject;

/// <summary>

/// 道路中的车道

/// </summary>

public Line[] lines;

/// <summary>

/// 标志道路类别

/// </summary>

public RoadTypeEnum roadType;

private ObjectData objectData = new ObjectData();

/// <summary>

/// 道路脚本初始化，判断是否为生产车辆的道路，生产唯一ID

/// </summary>

private void Start()

{

//判断道路是否被选中为产生车辆的起始道路

if (GetComponent<OriginRoad>() != null)

{

roadType = RoadTypeEnum.SOURCE;

}

else

{

roadType = RoadTypeEnum.NORMAL;

}

//为道路生产一个专属ID

var random = new System.Random();

id = random.Next(1, int.MaxValue);

//重置道路ID，将道路加入到一个存储列表

if (objectData.id == 0)

{

objectData.id = id;

SaveData.current.objects.Add(objectData);

}

if (RectangleSelector.current != null)

{

RectangleSelector.current.Selectable.Add(this.gameObject);

}

GameEvents.current.OnLoadEvent += DestorySelf;

GameEvents.current.OnDeleteEvent += DestroySelf;

//为道路中的所有车道设置父类引用

lines = GetComponentsInChildren<Line>();

foreach (Line line in lines)

{

line.fatherRoad = this;

}

}

private void Update()

{

objectData.position = transform.position;

objectData.rotation = transform.rotation;

objectData.roadType = roadType;

foreach (Line line in lines)

{

line.fatherRoad = this;

}

}

/// <summary>

/// 删除道路脚本时调用，更新一些全局资源

/// </summary>

private void OnDestroy()

{

RectangleSelector.current.Selectable.Remove(this.gameObject);

SaveData.current.objects.Remove(objectData);

GameEvents.current.OnLoadEvent -= DestorySelf;

GameEvents.current.OnDeleteEvent -= DestroySelf;

}

/// <summary>

/// 删除道路预制件接口

/// </summary>

private void DestorySelf()

{

Destroy(this.gameObject);

}

/// <summary>

/// 通过道路ID删除道路预制件接口

/// </summary>

/// <param name="id"></param>

private void DestroySelf(int id)

{

if (id == gameObject.GetInstanceID())

{

Destroy(this.gameObject);

}

}

}

/// <summary>

/// 道路种类

/// </summary>

public enum RoadTypeEnum

{

NORMAL,

SOURCE

}

using UnityEngine;

public class Util

{

/// <summary>

/// 修改传入浮点型变量的保留位数

/// </summary>

/// <param name="f"></param>

/// <returns></returns>

public static float limit(float f)

{

return Mathf.Floor(f \* 1000) / 1000;

}

}

using UnityEngine;

using System.Collections;

using System.Collections.Generic;

using BehaviorDesigner.Runtime;

using BehaviorDesigner.Runtime.Tasks;

/// <summary>

/// 任何情况下车辆的减速停车行为

/// </summary>

public class CarStop : Action

{

Car car;

public override void OnStart()

{

car = gameObject.GetComponent<Car>();

}

/// <summary>

/// 车辆的让行行为

/// </summary>

/// <returns></returns>

public override TaskStatus OnUpdate()

{

if(Vector3.Distance(car.barrier.position,car.transform.position)-car.barrier.radius <= 0.5f+car.transform.localScale.z)

{

car.accel = 0;

car.velocity = 0;

}

else

{

car.accel = Mathf.Pow(car.Km2m(), 1.5f) \* (0 - car.Km2m()) / Mathf.Pow(car.barrier.s - car.barrier.radius - car.s, 0.9f);

}

car.driving();

return TaskStatus.Running;

}

/// <summary>

/// 冲突结束时车辆开始尝试加速行驶

/// </summary>

public override void OnEnd()

{

//停止的时间结束时开始加速

car.accel = 30;

}

}

using UnityEngine;

using BehaviorDesigner.Runtime;

using BehaviorDesigner.Runtime.Tasks;

/// <summary>

/// 车辆的换道行为

/// </summary>

public class ChangeLine : Action

{

Car car;

/// <summary>

/// 换道的目标车道

/// </summary>

Line targetLine;

/// <summary>

/// 目标车道在当前道路的下标

/// </summary>

public SharedInt targetLineIndex;

/// <summary>

/// 换道路径的生成算法，提供二次开发接口

/// </summary>

public static OriginCustom.CP cp=CalculatePath;

/// <summary>

/// 软件自身提供的简易换道路径生成算法

/// </summary>

public static Vector3[] CalculatePath(OCar car,OLine targetLine)

{

Vector3[] ret =new Vector3[4];

ret[0] = car.transform.position;

ret[1] = ret[0] + car.transform.forward.normalized \* 3f;

ret[2] = ret[1] + targetLine.transform.position - car.line.transform.position;

ret[3] = ret[2] + ret[1] - ret[0];

return ret;

}

/// <summary>

/// 硬性换道路径的计算方法，用于纯模拟环境

/// </summary>

/// <returns></returns>

public Vector3[] Move()

{

Vector3[] ret = new Vector3[2];

ret[0] = car.transform.position;

ret[1] = ret[0] + targetLine.transform.position - car.line.transform.position;

return ret;

}

/// <summary>

/// 车辆换道信息初始化

/// </summary>

public override void OnStart()

{

car = gameObject.GetComponent<Car>();

targetLine = car.line.fatherRoad.lines[targetLineIndex.Value];

car.linePoints = cp(car,targetLine);

//car.linePoints = Move();

car.line.cars.Remove(car);

//行驶路径初始化

car.line = null;

car.lineT = 0;

car.target = car.linePoints[0];

}

/// <summary>

/// 车辆在换道信息初始化之后开始进入换道过程

/// </summary>

/// <returns></returns>

public override TaskStatus OnUpdate()

{

car.accel = 5;

car.driving();

if(car.lineT >= 1)

{

End();

return TaskStatus.Success;

}

return TaskStatus.Running;

}

/// <summary>

/// 换道结束时调用，重置车辆的形式信息

/// </summary>

private void End()

{

car.changeLine(targetLine);

car.lineT = Line.CalculateT(car.transform.position, car.line.points);

car.state = Car.State.inLine;

car.lineChange = false;

car.driving();

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using BehaviorDesigner.Runtime;

using BehaviorDesigner.Runtime.Tasks;

/// <summary>

/// 车辆在同一车道上的跟驰行驶行为

/// </summary>

public class Following : Action

{

Car car;

/// <summary>

/// 车辆跟驰算法的二次开发接口

/// </summary>

public static OriginCustom.GM gm=OriginGM;

//public delegate float GM(float c, float m, float l, OCar previous);

/// <summary>

/// GM跟驰模型

/// </summary>

/// <param name="c">车辆灵敏度</param>

private static float OriginGM(OCar m\_car,float c,float m,float l,OCar previous)

{

return c \* Mathf.Pow(m\_car.Km2m(), m) \* (previous.Km2m() - m\_car.Km2m()) / Mathf.Pow(previous.s - m\_car.s, l);

}

public override void OnStart()

{

car = gameObject.GetComponent<Car>();

car.target = car.transform.position;

}

/// <summary>

/// 车辆在车道中行驶时默认处于跟驰行为中，确保不会碰撞前车

/// </summary>

/// <returns></returns>

public override TaskStatus OnUpdate()

{

if (car.state != Car.State.inLine)

{

car.accel = 0;

car.velocity = 30;

}

else

car.crossLine = Line.linkLine2(lineIn, lineOut);

//传递进入路口钱确定的初始信息

targetLineIndex.Value = lineIn.indexInRoad();

LineOutIndex.Value = lineOut.indexInRoad();

return TaskStatus.Success;

}

}

using UnityEngine;

using BehaviorDesigner.Runtime.Tasks;

using BehaviorDesigner.Runtime;

/// <summary>

/// 车辆在路口内行驶的行为

/// </summary>

public class RunCross : Action

{

Car car;

/// <summary>

/// 接受prepareCross中传递来的参数

/// </summary>

public SharedInt LineOutIndex;

/// <summary>

/// 初始化车辆在路口内行驶所需要的信息

/// </summary>

public override void OnStart()

{

car = gameObject.GetComponent<Car>();

car.state = Car.State.crossing;

car.linePoints = car.crossLine;

}

/// <summary>

/// 车辆按计算好的路径行驶

/// </summary>

/// <returns></returns>

public override TaskStatus OnUpdate()

{

car.driving();

//车辆的lineT超过1，说明路口中的路径已经行驶完，需要更迭车辆状态，重置行驶信息

if(car.lineT >= 1)

{

car.crossLine = null;

car.lineT = 0;

car.setLine(car.cross.carRoadOut[car].lines[LineOutIndex.Value]);

car.state = Car.State.inLine;

car.cross.cars.Remove(car);

car.preCross = car.cross;

car.cross = null;

car.driving();

return TaskStatus.Success;

}

return TaskStatus.Running;

}

}

using UnityEngine;

using System.Collections;

using System.Collections.Generic;

using BehaviorDesigner.Runtime;

using BehaviorDesigner.Runtime.Tasks;

/// <summary>

/// 穿越路口的行为

/// </summary>

public class ThoughtCrossing : Action

{

Car car;

Line runLine;

public override void OnStart()

{

car = gameObject.GetComponent<Car>();

}

//当前的问题在于，当多次调用该脚本时，车辆line的判断问题

public override TaskStatus OnUpdate()

{

car.findPath();

return TaskStatus.Success;

}

}

using UnityEngine;

using BehaviorDesigner.Runtime;

using BehaviorDesigner.Runtime.Tasks;

/// <summary>

/// 车辆是否换道的判断脚本

/// </summary>

public class ChangeLineInstruction : Conditional

{

Car car;

/// <summary>

/// 限制车辆换道频率

/// </summary>

float changTime;

public SharedInt targetLineIndex;

/// <summary>

/// 为车道行驶价值系数提供二次开发接口

/// </summary>

public static OriginCustom.JV jv=JudgeValue;

/// <summary>

/// 判断换道产生的收益值,返回值[0,1]

/// </summary>

/// <param name="car"></param>

/// <param name="line"></param>

/// <returns></returns>

public static float JudgeValue(OCar car, OLine line)

{

//目标车道不存在车辆

if (line.cars.First == null)

{

return 1;

}

OCar near = car.CarClosest(line);

float PreSpeed;//间隙前车速度

float nextS;//间隙后车的行驶间距

float preS;//间隙前车的行驶间距

//near在car之后

if (OCar.judgeLocation(car, near))

{

//换道之后成为头车且间隙允许换道

if (near.PreCar() == null)

{

preS = 10000;

PreSpeed = 10000;

}

else

{

OCar nearPre = near.PreCar();//间隙前车

PreSpeed = nearPre.velocity;

preS = nearPre.s - car.s;

}

nextS = car.s - near.s;

}

else//near在car之前

{

OCar nearNext = near.NextCar();

PreSpeed = near.velocity;

preS = near.s - car.s;

if (nearNext == null)

{

nextS = 10000;//设立一个很大的数表示无穷

}

else

{

nextS = car.s - nearNext.s;

}

}

if (preS <= car.transform.localScale.z || nextS <= car.transform.lossyScale.z \* 1.5)

{

return 0;

}

if (car.PreCar() == null || (PreSpeed <= car.PreCar().velocity && preS - car.PreCar().s < car.transform.lossyScale.z))

{

return 0;

}

return 1;

}

/// <summary>

/// 车道选择策略

/// 从该车道临近车道选择一个正收益车道

/// </summary>

private Line NearLinePick()

{

Line[] lines = car.line.fatherRoad.lines;

int nowIndex = car.line.indexInRoad();

if (nowIndex == 0)

{

return jv(car, lines[1]) > 0 ? lines[1] : car.line;

}

else if (nowIndex == lines.Length - 1)

{

return jv(car, lines[lines.Length - 2]) > 0 ? lines[lines.Length - 2] : car.line;

}

else

{

if (jv(car, lines[nowIndex - 1]) > 0)

{

return lines[nowIndex - 1];

}

else if (jv(car, lines[nowIndex + 1]) > 0)

{

return lines[nowIndex + 1];

}

else

{

return car.line;//选择本身

}

}

}

/// <summary>

/// 选择左侧车道换道

/// </summary>

/// <returns></returns>

private Line LeftPick()

{

Line[] lines = car.line.fatherRoad.lines;

int nowIndex = car.line.indexInRoad();

if (nowIndex == 0)

{

return car.line;

}

return jv(car, lines[nowIndex - 1]) > 0 ? lines[nowIndex - 1] : car.line;

}

/// <summary>

/// 选择右侧车道换道

/// </summary>

/// <returns></returns>

private Line RightPick()

{

Line[] lines = car.line.fatherRoad.lines;

int nowIndex = car.line.indexInRoad();

if (nowIndex == lines.Length - 1)

{

return car.line;

}

return jv(car, lines[nowIndex + 1]) > 0 ? lines[nowIndex + 1] : car.line;

}

/// <summary>

/// 车辆不满足当前行驶环境时寻求主动换道时修改car.lineChange

/// </summary>

private void PositiveChange()

{

//换道条件判断

if (car.line != null && car.line.cars.First != null && car.line.cars.First.Value != car && car.s > car.transform.localScale.z)

{

if (car.line.cars.Find(car) == null)

{

return;

}

Car pre = (Car)car.line.cars.Find(car).Previous.Value;

//当前车道运行速度低于预期时，寻求车道换道

if ((pre.velocity <= 40 && car.expectVelocity - car.velocity > 5 && car.accel < 5) || (pre.velocity >= 40 && car.expectVelocity - car.velocity > 15 && car.accel < 5))

{

car.lineChange = true;

}

}

}

/// <summary>

/// 换道目标不为自身，正式开始换道

/// </summary>

/// <returns></returns>

private bool SuccessChange()

{

Line line = PickLine();

if (!line.Equals(car.line) && CanPreGap(car,line) && CanNextGap(car,line))

{

targetLineIndex.Value = line.indexInRoad();

return true;

}

return false;

}

/// <summary>

/// 选择一个行驶价值最高的车道

/// </summary>

/// <returns></returns>

private Line PickLine()

{

if(JudgeRightValue(car) >= JudgeLeftValue(car))

{

if(JudgeLineValue(car) >= JudgeRightValue(car)) {

return car.line;

}

else

{

return car.line.fatherRoad.lines[car.line.indexInRoad() + 1];

}

}

else

{

if(JudgeLineValue(car) >= JudgeLeftValue(car))

{

return car.line;

}

else

{

return car.line.fatherRoad.lines[car.line.indexInRoad() - 1];

}

}

}

/// <summary>

/// 左侧车道价值: 0.57-0.32V1

/// V1: 决策车辆与左车道后车相对速度

/// </summary>

/// <param name="car"></param>

/// <returns></returns>

private static double JudgeLeftValue(Car car)

{

if (car.line.indexInRoad() == 0)//当前车道为最左车道

{

return double.MinValue;

}

var LeftLine = car.line.fatherRoad.lines[car.line.indexInRoad() - 1];

var cars = LeftLine.cars;

var nearCarV = 0.0;

foreach (var nearCar in cars)

{

if (((Car)nearCar).s < car.s)

{

nearCarV = ((Car)nearCar).velocity;

}

}

return 0.57 - 0.32 \* (car.velocity - nearCarV);

}

/// <summary>

/// 当前车道价值: 0.28\*V2 + 0.36\*V3 + 0.21S

/// V2：目标车与前车相对速度

/// V3：目标车与后车相对速度

/// S：目标车与当前车道前车相对位置

/// </summary>

/// <param name="car"></param>

/// <returns></returns>

private static double JudgeLineValue(Car car)

{

var preV = double.MaxValue;

var preS = double.MaxValue;

if (car.PreCar() != null)

{

preV = car.PreCar().velocity;

preS = car.PreCar().s;

}

var nextV = 0.0;

if (car.NextCar() != null)

{

nextV = car.NextCar().velocity;

}

return 0.28 \* (preV - car.velocity) + 0.36 \* (car.velocity - nextV) + 0.21 \* (preS - car.s);

}

/// <summary>

/// 右侧车道价值: 0.17 - 0.22\*V4

/// V4 ：决策车辆与右车道后车相对速度

/// </summary>

/// <param name="car"></param>

/// <returns></returns>

private static double JudgeRightValue(Car car)

{

if (car.line.indexInRoad() == car.line.fatherRoad.lines.Length - 1)//当前车道为最右车道

{

return double.MinValue;

}

var RightLine = car.line.fatherRoad.lines[car.line.indexInRoad() + 1];

var cars = RightLine.cars;

var nearCarV = 0.0;

foreach (var nearCar in cars)

{

if (((Car)nearCar).s < car.s)

{

nearCarV = ((Car)nearCar).velocity;

}

}

return 0.17 - 0.22 \* (car.velocity - nearCarV);

}

/// <summary>

/// 判断前车间隙是否能够换道

/// 换道临界前车间隙： G1 = exp{1.23 - 0.34\*max(0,V5) - 0.21\*min(0,V5)}

/// V5: 与目标车道前车相对速度

/// </summary>

/// <param name="car"></param>

/// <param name="line"></param>

/// <returns></returns>

private static bool CanPreGap(Car car, Line line)//上述公式条件过于严苛，尝试放宽条件

{

//目标车道没有前车

if (line.cars.First == null || line.cars.First.Value.s < car.s)

{

return true;

}

foreach (var node in line.cars)

{

var preCar = (Car)node;

//找到前车

if (preCar.NextCar() == null || preCar.NextCar().s < car.s)

{

var V5 = preCar.velocity - car.velocity;

var G1 = Mathf.Exp((float)(1.23 - 0.34 \* Mathf.Max(0, V5) - 0.21 \* Mathf.Min(0, V5)));

//尝试缩小临界前车间隙

return preCar.s - car.s >= G1/2;

}

}

return true;

}

/// <summary>

/// 换道临界后车间隙： G2 = exp{1.35 - 0.41\*max(0,V6) - 0.28\*min(0,V6)}

/// V6: 与目标车道后车相对速度

/// </summary>

/// <param name="car"></param>

/// <param name="line"></param>

/// <returns></returns>

private static bool CanNextGap(Car car, Line line)

{

foreach (var node in line.cars)

{

var nextCar = (Car)node;

//找到前车

if (nextCar.s < car.s)

{

var V6 = car.velocity - nextCar.velocity;

var G2 = Mathf.Exp((float)(1.35 - 0.41 \* Mathf.Max(0, V6) - 0.28 \* Mathf.Min(0, V6)));

return car.s - nextCar.s >= G2/2;

}

}

return true;

}

public override void OnStart()

{

car = gameObject.GetComponent<Car>();

}

/// <summary>

/// 每一帧都判断当前车辆是否需要换道

/// </summary>

/// <returns></returns>

public override TaskStatus OnUpdate()

{

if (car.state == Car.State.prepareCross)

{

return TaskStatus.Failure;

}

//将要进入路口时就禁止主动换道

if (car.cross != null && car.state == OCar.State.inLine)

{

return TaskStatus.Failure;

}

if (car.state == Car.State.changing)

{

//changTime = 0;

return TaskStatus.Success;

}

/\*

if (changTime < 1)

{

changTime += Time.deltaTime;

return TaskStatus.Failure;

}\*/

//PositiveChange();

/\*if (car.lineChange == true)

{\*/

//即将驶入路口，不予换道

if (car.state == Car.State.prepareCross)

{

return TaskStatus.Failure;

}

if (SuccessChange())

{

Debug.LogWarning("count");

car.state = Car.State.changing;

return TaskStatus.Success;

}

/\*}\*/

/\*//最终判断不支持换道，驳回换道请求

car.lineChange = false;\*/

return TaskStatus.Failure;

}

}

using BehaviorDesigner.Runtime.Tasks;

/// <summary>

/// 车辆是否位于路口中的判断

/// </summary>

public class IsInCross : Conditional

{

Car car;

public override void OnStart()

{

car = gameObject.GetComponent<Car>();

}

/// <summary>

/// 判断车辆是否处于或即将处于路口之中

/// </summary>

/// <returns></returns>

public override TaskStatus OnUpdate()

{

//车辆所处车道没有下一条道路

if(car.line != null && car.line.nextRoads.Count == 0)

{

return TaskStatus.Failure;

}

//车俩处于路口准备或路口中的状态位，则该车辆在路口中

if(car.state == Car.State.prepareCross || car.state == Car.State.crossing)

{

return TaskStatus.Success;

}

//车辆已经计算好了将要在路口中行驶的路径，说明已经做好了路口行驶的准备

if(car.state == Car.State.inLine && car.crossLine != null && car.crossLine.Length != 0)

{

return TaskStatus.Success;

}

return TaskStatus.Failure;

}

}

using UnityEngine;

using BehaviorDesigner.Runtime.Tasks;

public class JudgeConflict : Conditional

{

Car car;

/// <summary>

/// 标志是否冲突的标志位

/// </summary>

bool conflict = false;

/// <summary>

/// 车辆的碰撞体触发，说明发生了冲突

/// </summary>

/// <param name="other"></param>

public override void OnTriggerEnter(Collider other)

{

if(other.gameObject.GetComponent<Car>() != null)

{

Car otherCar = other.gameObject.GetComponent<Car>();

//在道路中行驶的车辆不会碰撞

if (car.state == OCar.State.inLine && otherCar.state == OCar.State.inLine)

{

return;

}

//发生冲突时，选择一辆车继续行驶，一辆让行

Car lucky = CollisionSystem.ChooseLucky(car, otherCar);

var value = CollisionSystem.LineLineIntersection(car.transform.position, car.transform.forward, otherCar.transform.position, otherCar.transform.forward);

//尾部碰撞器，两车不会碰撞

if (value == Vector3.zero)

{

return;

}

//为让行车辆构造碰撞体

var newBarrier = new Barrier();

newBarrier.position = value;

//设置otherCar的barrier

if (lucky == car)

{

if(otherCar.barrier == null || Vector3.Distance(otherCar.barrier.position,otherCar.transform.position) >= Vector3.Distance(newBarrier.position, otherCar.transform.position)){

otherCar.barrier = newBarrier;

}

}

//设置Car的barrier

if (lucky == otherCar)

{

if(car.barrier == null || Vector3.Distance(car.barrier.position, car.transform.position) >= Vector3.Distance(newBarrier.position, car.transform.position))

{

car.barrier = newBarrier;

}

}

}

}

public override void OnTriggerExit(Collider other)

{

if(other.gameObject.GetComponent<Car>() != null && conflict == true)

{

//Debug.LogWarning("冲突结束");

conflict = false;

}

}

public override void OnStart()

{

car = gameObject.GetComponent<Car>();

}

/// <summary>

/// 发生冲突即将车辆碰撞标志位置为真

/// </summary>

/// <returns></returns>

public override TaskStatus OnUpdate()

{

if (car.barrier != null)

{

conflict = true;

}

if(conflict == true)

{

return TaskStatus.Success;

}

else

{

return TaskStatus.Failure;

}

}

}

using UnityEngine;

using System.Collections;

using System.Collections.Generic;

using BehaviorDesigner.Runtime;

using BehaviorDesigner.Runtime.Tasks;

public class JudgeLineLock : Conditional

{

Car car;

// Use this for initialization

public override void OnStart()

{

car = gameObject.GetComponent<Car>();

}

// Update is called once per frame

public override TaskStatus OnUpdate()

{

//车辆判断锁持有

//锁无人持有或锁自己持有，则加锁并继续运行

if(car.lineT >= 0.8 && (car.line.lineLock == null||car.line.lineLock == car))

{

car.line.lineLock = car;

return TaskStatus.Success;

}

//锁非本车辆持有，放弃抢占

else if (car.lineT >= 0.8 && car.line.lineLock != car)

{

return TaskStatus.Failure;

}

return TaskStatus.Success;

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using System;

using System.IO;

using System.Text;

using System.Reflection;

public class DllReader

{

public static GameObject go;

public static OriginCustom currentCustom;

public static Type type;

/// <summary>

/// 读取dll文件

/// </summary>

/// <param name="className"></param>类名

/// <param name="filePath"></param>文件在的地址

/// <returns></returns> 类型

public static Type ReadDll(string className, string filePath)

{

FileStream fs = new FileStream(filePath, FileMode.OpenOrCreate);

byte[] b = new byte[fs.Length];

fs.Read(b, 0, b.Length);

fs.Dispose();

fs.Close();

Assembly assembly = Assembly.Load(b);

Type type1 = assembly.GetType(className);

return type1;

}

public static GameObject CreateManager(Type type)

{

GameObject go = new GameObject();

go.name = "CustomManager";

go.AddComponent(type);

//GameObject.Instantiate(go);

return go;

}

public static void LoadDll(string path)

{

type = ReadDll(@"Custom", path);

go = CreateManager(type);

currentCustom = go.GetComponent<OriginCustom>();

if (currentCustom != null)

{

Following.gm = currentCustom.CustomGM;

ChangeLine.cp = currentCustom.CustomCP;

ChangeLineInstruction.jv = currentCustom.CustomJV;

}

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public abstract class OrginCustom

{

public abstract float CustomGM(float c, float m, float l, Car previous);

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class TestButtonAdder : MonoBehaviour

{

public void AddDll()

{

DllReader.LoadDll("");

}

}

using System.Collections;

using System.Collections.Generic;

using System.Threading.Tasks;

using UnityEngine;

[RequireComponent(typeof(MeshFilter), typeof(MeshRenderer)), DisallowMultipleComponent]

public class LaneMesh : MonoBehaviour

{

private Mesh mesh;

private Vector3[] vertices;

void Awake()

{

var task = GenerateMeshAsync();

}

private async Task GenerateMeshAsync()

{

try

{

await Task.Yield();

GetComponent<MeshFilter>().mesh = mesh = new Mesh();

mesh.name = "Lane";

// Generate vertices of the road, 44 vertices in total

int v = 0;

vertices = new Vector3[44];

for (float y = 0; y <= 0.1f; y += 0.1f)

{

for (int x = 0; x <= 10; x++)

{

vertices[v++] = new Vector3(x, y, 0);

}

for (int x = 0; x <= 10; x++)

{

vertices[v++] = new Vector3(10 - x, y, 3.5f);

}

}

mesh.vertices = vertices;

// Generate triangles, 84 triangles, 252 vertices of triangles in total

int[] triangles = new int[252];

int t = 0;

// side faces

for (int i = 0; i < 22; i++)

{

triangles[t] = i % 22;

triangles[t + 1] = triangles[t + 4] = (22 + i) % 22 + 22;

triangles[t + 2] = triangles[t + 3] = (i + 1) % 22;

triangles[t + 5] = (22 + i + 1) % 22 + 22;

t += 6;

}

// bottom faces

for (int i = 0; i < 10; i++)

{

triangles[t] = 21 - i;

triangles[t + 1] = triangles[t + 4] = i;

triangles[t + 2] = triangles[t + 3] = 20 - i;

triangles[t + 5] = 1 + i;

t += 6;

}

// top faces

for (int i = 0; i < 10; i++)

{

triangles[t] = 22 + i;

triangles[t + 1] = triangles[t + 4] = 43 - i;

triangles[t + 2] = triangles[t + 3] = 23 + i;

triangles[t + 5] = 42 - i;

t += 6;

}

mesh.triangles = triangles;

mesh.RecalculateNormals();

var collider = gameObject.AddComponent<MeshCollider>();

var outline = gameObject.AddComponent<Outline>();

}

catch(System.Exception error)

{

Debug.Log(error);

}

}

public void RecalculateVerticesPosition()

{

var childTransforms = this.GetComponentsInChildren<Transform>();

var childPositions = new Vector3[3];

var controlPoints = new Vector3[3];

var offsets = new Vector3[3];

var t = 0f;

for (int i = 0; i < 3; i++)

{

childPositions[i] = childTransforms[i + 1].position;

}

offsets[0] = Quaternion.Euler(0, 90, 0) \* (childPositions[1] - childPositions[0]).normalized;

offsets[1] = Quaternion.Euler(0, 90, 0) \* (childPositions[2] - childPositions[0]).normalized;

offsets[2] = Quaternion.Euler(0, 90, 0) \* (childPositions[2] - childPositions[1]).normalized;

// calculate vertices from 2 edges of the front face

for (int i = 0; i < 3; i++)

{

controlPoints[i] = childPositions[i] + 1.75f \* offsets[i];

}

for (int i = 0; i < 11; i++)

{

vertices[i] = QuadraicBezier(controlPoints[0], controlPoints[1], controlPoints[2], t);

vertices[22 + i] = vertices[i] + new Vector3(0, 0.1f, 0);

t += 0.1f;

}

// calculate vertices from 2 edges of the back face

for (int i = 0; i < 3; i++)

{

controlPoints[i] = childPositions[i] - 1.75f \* offsets[i];

}

t = 0f;

for (int i = 0; i < 11; i++)

{

vertices[21 - i] = QuadraicBezier(controlPoints[0], controlPoints[1], controlPoints[2], t);

vertices[43 - i] = vertices[21 - i] + new Vector3(0, 0.1f, 0);

t += 0.1f;

}

mesh.vertices = vertices;

mesh.RecalculateNormals();

}

private Vector3 QuadraicBezier(Vector3 position1, Vector3 position2, Vector3 position3, float t)

{

var temp1 = (1 - t) \* position1 + t \* position2;

var temp2 = (1 - t) \* position2 + t \* position3;

var position = (1 - t) \* temp1 + t \* temp2;

return position;

}

// for debug

private void OnDrawGizmos()

{

if (vertices == null)

{

return;

}

Gizmos.color = Color.black;

for(int i = 0; i < vertices.Length; i++)

{

Gizmos.DrawSphere(vertices[i], 0.02f);

}

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using TMPro;

using UI.Panel.FileSelet;

public class ButtonHandler : MonoBehaviour

{

[SerializeField] private TextMeshProUGUI statusBar;

[SerializeField] private GameObject roadPrefeb;

[SerializeField] private GameObject carPrefeb;

[SerializeField] private GameObject crossPrefab;

[SerializeField] private FileSelectPanel fileSelectPanel;

private GameObject[] roadList;

private Vector3[] roadPosition;

void Awake()

{

Time.timeScale = 0;

}

void Start()

{

StartCoroutine(DestroyObjectOnLick());

Debug.Log(Application.persistentDataPath);

}

// 设置道路

private IEnumerator SetUpRoad()

{

int status = 1;

var roadPosition = new Vector3[2];

statusBar.SetText("请点击道路的第一个点（起点），按ESC键退出");

while (status == 1)

{

if (Input.GetKeyDown(KeyCode.Escape))

{

statusBar.SetText("");

yield break;

}

if (Input.GetMouseButtonDown(0))

{

var plane = new Plane(Vector3.up, Vector3.zero);

var ray = Camera.main.ScreenPointToRay(Input.mousePosition);

float entry;

if (plane.Raycast(ray, out entry))

{

roadPosition[0] = ray.GetPoint(entry);

}

status += 1;

}

yield return null;

}

statusBar.SetText("请点击道路的第二个点（终点），按ESC键退出");

while (status == 2)

{

if (Input.GetKeyDown(KeyCode.Escape))

{

statusBar.SetText("");

yield break;

}

if (Input.GetMouseButtonDown(0))

{

var plane = new Plane(Vector3.up, Vector3.zero);

var ray = Camera.main.ScreenPointToRay(Input.mousePosition);

float entry;

if (plane.Raycast(ray, out entry))

{

roadPosition[1] = ray.GetPoint(entry);

}

status += 1;

}

yield return null;

}

Vector3 position = (roadPosition[1] + roadPosition[0]) / 2.0f;

var scale = Vector3.Distance(roadPosition[1], roadPosition[0]) / 60.0f;

Quaternion rotation = Quaternion.LookRotation(roadPosition[1] - roadPosition[0], Vector3.up);

rotation \*= Quaternion.Euler(0, -90f, 0);

GameObject newRoad = Instantiate(roadPrefeb, position, rotation);

newRoad.transform.localScale = new Vector3(scale, 1.0f, 1.0f);

statusBar.SetText("");

}

// 连接道路

private IEnumerator ConnectLane()

{

int status = 1;

var roadList = new GameObject[2];

statusBar.SetText("请点击需要被连接的前一条车道， 按ESC退出");

while (status == 1)

{

if (Input.GetKey(KeyCode.Escape))

{

statusBar.SetText("");

yield break;

}

if (Input.GetMouseButtonDown(0))

{

roadList[0] = this.SelectObjectOnClick();

if (roadList[0] != null)

{

status += 1;

}

}

yield return null;

}

statusBar.SetText("请点击需要被连接的后一条车道， 按ESC退出");

while (status == 2)

{

if (Input.GetKey(KeyCode.Escape))

{

statusBar.SetText("");

yield break;

}

if (Input.GetMouseButtonDown(0))

{

roadList[1] = this.SelectObjectOnClick();

if (roadList[1] != null)

{

status += 1;

}

}

yield return null;

}

var nextRoad = roadList[1].GetComponentInParent<Road>();

roadList[0].GetComponent<Line>().nextRoads.Add(nextRoad);

statusBar.SetText("");

}

// 设置车辆源

private IEnumerator SetCarSource()

{

bool isDone = false;

statusBar.SetText("请点击需要设置为车辆源的道路");

while (!isDone)

{

if (Input.GetKeyDown(KeyCode.Escape))

{

statusBar.SetText("");

yield break;

}

if (Input.GetMouseButtonDown(0))

{

var selectedRoad = this.SelectObjectOnClick();

if (selectedRoad != null)

{

OriginRoad originRoad = selectedRoad.transform.parent.gameObject.AddComponent<OriginRoad>();

originRoad.Car = carPrefeb;

selectedRoad.GetComponentInParent<Road>().roadType = RoadTypeEnum.SOURCE;

isDone = true;

}

}

yield return null;

}

statusBar.SetText("");

}

// 用于选中鼠标点击的物件

private GameObject SelectObjectOnClick()

{

GameObject selectedObject = null;

var ray = Camera.main.ScreenPointToRay(Input.mousePosition);

RaycastHit rayHit;

if (Physics.Raycast(ray, out rayHit))

{

selectedObject = rayHit.collider.gameObject;

}

return selectedObject;

}

// del键删除选中的物件

private IEnumerator DestroyObjectOnLick()

{

while (true)

{

if (Input.GetKeyDown(KeyCode.Delete))

{

var deleteList = RectangleSelector.current.Selected;

var propertyTable = Inspector.current.PropertyTableList;

for (int i = 0; i < deleteList.Count; i++)

{

GameEvents.current.OnDelete(deleteList[i].GetInstanceID());

}

for (int i = 0; i < propertyTable.Count; i++)

{

Destroy(propertyTable[i]);

}

RectangleSelector.current.Selected.Clear();

}

yield return null;

}

}

// 设置路口

private IEnumerator SetCross()

{

var isDone = false;

while (!isDone)

{

if (Input.GetMouseButtonDown(0))

{

var plane = new Plane(Vector3.up, Vector3.zero);

var ray = Camera.main.ScreenPointToRay(Input.mousePosition);

var position = new Vector3();

float entry;

if (plane.Raycast(ray, out entry))

{

position = ray.GetPoint(entry);

}

GameObject gameObject = Instantiate(crossPrefab, position, Quaternion.identity);

isDone = true;

}

yield return null;

}

}

// 下列函数均用于与按钮关联，用于触发上面的函数

public void OnStopButtonClick()

{

Time.timeScale = 0;

var cars = GameObject.FindGameObjectsWithTag("Car");

for (int i = 0; i < cars.Length; i++)

{

Destroy(cars[i]);

}

}

public void OnGenerateRoadButtonClick()

{

if (Time.timeScale == 0)

{

StartCoroutine(SetUpRoad());

}

else

{

statusBar.SetText("正在播放中，请先停止再进行编辑");

}

}

public void OnLinkLaneButtonClick()

{

if (Time.timeScale == 0)

{

StartCoroutine(ConnectLane());

}

else

{

statusBar.SetText("正在播放中，请先停止再进行编辑");

}

}

public void OnSetCarSourceButtonClick()

{

if (Time.timeScale == 0)

{

StartCoroutine(SetCarSource());

}

else

{

statusBar.SetText("正在播放中，请先停止再进行编辑");

}

}

public void OnLoadModelButtonClick()

{

//DllReader.testInit();

if (fileSelectPanel != null)

fileSelectPanel.Activate(DllSelect);

}

public void OnSetCrossButtonClick()

{

if (Time.timeScale == 0)

{

StartCoroutine(SetCross());

}

else

{

statusBar.SetText("正在播放中，请先停止再进行编辑");

}

}

private void DllSelect()

{

DllReader.LoadDll(UI.Panel.FileSelet.FileSelectPanel.currentFileSelectPanel.filePath);

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using TMPro;

// 探测器面板脚本，目前使用反射实现

// 在未来确定了路，车，路口的模式之后会改用更好的方法实现

public class Inspector : MonoBehaviour

{

public static Inspector current;

private int tableInList;

[SerializeField] private GameObject content;

[SerializeField] private GameObject propertyPrefab;

[SerializeField] private GameObject warningText;

[SerializeField] private GameObject scrollView;

private List<GameObject> propertyTableList = new List<GameObject>();

public List<GameObject> PropertyTableList { get => propertyTableList; set => propertyTableList = value; }

void Start()

{

current = this;

}

void Update()

{

// 当没有任何物件被选中的时候，显示提示

if (RectangleSelector.current.Selected.Count == 0)

{

warningText.SetActive(true);

scrollView.SetActive(false);

}

else

{

warningText.SetActive(false);

scrollView.SetActive(true);

}

// 使用数量确认是否有东西选中

if(RectangleSelector.current.Selected.Count != tableInList)

{

ShowProperty();

tableInList = RectangleSelector.current.Selected.Count;

}

// 在探测器中更新选中物件的属性

if(Time.timeScale != 0)

{

UpdateProperty();

}

}

// 当任何物件被选中时，在探测器面板展示其内部属性

public void ShowProperty()

{

for(int i = 0; i < propertyTableList.Count; i++)

{

Destroy(propertyTableList[i]);

propertyTableList.Remove(propertyTableList[i]);

}

foreach(var gameObject in RectangleSelector.current.Selected)

{

if(gameObject.tag == "Car")

{

var properties = gameObject.GetComponent<Car>().GetType().GetFields();

foreach(var property in properties)

{

GameObject propertyTable = Instantiate(propertyPrefab);

propertyTable.transform.SetParent(content.transform, false);

propertyTable.name = gameObject.GetInstanceID().ToString() + ' ' + property.Name;

propertyTableList.Add(propertyTable);

propertyTable.transform.Find("Name").gameObject.GetComponent<TextMeshProUGUI>().SetText(property.Name);

if(property.GetValue(gameObject.GetComponent<Car>()) != null){

propertyTable.transform.Find("Value").gameObject.GetComponent<TextMeshProUGUI>().SetText(property.GetValue(gameObject.GetComponent<Car>()).ToString());

Else{

propertyTable.transform.Find("Value").gameObject.GetComponent<TextMeshProUGUI>().SetText("null");

}

}

if(gameObject.tag == "Road")

{

var properties = gameObject.GetComponent<Road>().GetType().GetFields();

foreach(var property in properties)

{

GameObject propertyTable = Instantiate(propertyPrefab);

propertyTable.transform.SetParent(content.transform, false);

propertyTable.name = gameObject.GetInstanceID().ToString() + ' ' + property.Name;

propertyTableList.Add(propertyTable);

propertyTable.transform.Find("Name").gameObject.GetComponent<TextMeshProUGUI>().SetText(property.Name);

if(property.GetValue(gameObject.GetComponent<Road>()) != null)

propertyTable.transform.Find("Value").gameObject.GetComponent<TextMeshProUGUI>().SetText(property.GetValue(gameObject.GetComponent<Road>()).ToString());

else

propertyTable.transform.Find("Value").gameObject.GetComponent<TextMeshProUGUI>().SetText("null");

}

}

if(gameObject.tag == "Lane")

{

var properties = gameObject.GetComponent<Line>().GetType().GetFields();

foreach(var property in properties)

{

GameObject propertyTable = Instantiate(propertyPrefab);

propertyTable.transform.SetParent(content.transform, false);

propertyTable.name = gameObject.GetInstanceID().ToString() + ' ' + property.Name;

propertyTableList.Add(propertyTable);

propertyTable.transform.Find("Name").gameObject.GetComponent<TextMeshProUGUI>().SetText(property.Name);

if(property.GetValue(gameObject.GetComponent<Line>()) != null){

propertyTable.transform.Find("Value").gameObject.GetComponent<TextMeshProUGUI>().SetText(property.GetValue(gameObject.GetComponent<Line>()).ToString());

else

propertyTable.transform.Find("Value").gameObject.GetComponent<TextMeshProUGUI>().SetText("null");

}

}

}

}

// 更新已经显示的属性

public void UpdateProperty()

{

foreach (var gameObject in RectangleSelector.current.Selected)

{

if(gameObject.tag == "Car")

{

var fields = gameObject.GetComponent<Car>().GetType().GetFields();

foreach(var field in fields)

{

foreach (var propertyTable in propertyTableList)

{

if(propertyTable.name == gameObject.GetInstanceID().ToString() + ' ' + field.Name)

{

if(field.GetValue(gameObject.GetComponent<Car>()) != null){

propertyTable.transform.Find("Value").gameObject.GetComponent<TextMeshProUGUI>().SetText(field.GetValue(gameObject.GetComponent<Car>()).ToString());

}

}

}

}

if(gameObject.tag == "Road")

{

var fields = gameObject.GetComponent<Road>().GetType().GetFields();

foreach(var field in fields)

{

foreach (var propertyTable in propertyTableList)

{

if(propertyTable.name == gameObject.GetInstanceID().ToString() + ' ' + field.Name)

{

if(field.GetValue(gameObject.GetComponent<Road>()) != null)

propertyTable.transform.Find("Value").gameObject.GetComponent<TextMeshProUGUI>().SetText(field.GetValue(gameObject.GetComponent<Road>()).ToString());

}

}

}

}

if(gameObject.tag == "Lane")

{

var fields = gameObject.GetComponent<Line>().GetType().GetFields();

foreach(var field in fields)

{

foreach (var propertyTable in propertyTableList)

{

if(propertyTable.name == gameObject.GetInstanceID().ToString() + ' ' + field.Name)

{

if(field.GetValue(gameObject.GetComponent<Line>()) != null){

propertyTable.transform.Find("Value").gameObject.GetComponent<TextMeshProUGUI>().SetText(field.GetValue(gameObject.GetComponent<Line>()).ToString());

}

}

}

}

}

}

}

using System;

using System.Collections;

using System.Collections.Generic;

using System.Threading.Tasks;

using UnityEngine;

using UnityEngine.EventSystems;

// 控制照相机的脚本，实现视角移动

public class CameraController : MonoBehaviour

{

public Transform cameraTransform;

public float moveSpeed;

public float moveTime;

public float rotationSpeed;

public float zoomAmount;

private Vector3 newPosition;

private Vector3 dragStartPosition;

private Vector3 dragStopPosition;

private Vector3 rotationStartPosition;

private Vector3 rotationStopPosition;

private Vector3 newZoom;

private Quaternion newRotation;

void Start()

{

newPosition = transform.position;

newRotation = transform.rotation;

newZoom = cameraTransform.localPosition;

}

async Task Update()

{

// 确认是否有其他面板打开

if(!EventSystem.current.IsPointerOverGameObject())

{

// 创建异步工作

Task moveCamera = MoveCamera();

Task zoomCamera = ZoomCamera();

Task rotateCamera = RotateCamera();

// 等待完成

await Task.WhenAll(moveCamera, zoomCamera, rotateCamera);

// 计算移动/旋转的量值，并以此改变照相机的位置

transform.position = Vector3.Lerp(transform.position, newPosition, moveTime);

transform.rotation = Quaternion.Lerp(transform.rotation, newRotation, moveTime);

cameraTransform.localPosition = Vector3.Lerp(cameraTransform.localPosition, newZoom, moveTime);

}

}

// 使用wsad键/鼠标中键拖拽移动照相机

private async Task MoveCamera()

{

await Task.Yield();

if (Input.GetKey(KeyCode.UpArrow))

{

newPosition += (transform.forward \* moveSpeed);

}

if (Input.GetKey(KeyCode.DownArrow))

{

newPosition += (transform.forward \* -moveSpeed);

}

if (Input.GetKey(KeyCode.LeftArrow))

{

newPosition += (transform.right \* -moveSpeed);

}

if (Input.GetKey(KeyCode.RightArrow))

{

newPosition += (transform.right \* moveSpeed);

}

if (Input.GetMouseButtonDown(2))

{

Plane Plane = new Plane(Vector3.up, Vector3.zero);

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

float entry;

if (Plane.Raycast(Ray, out entry))

{

dragStartPosition = Ray.GetPoint(entry);

}

}

if (Input.GetMouseButton(2))

{

Plane Plane = new Plane(Vector3.up, Vector3.zero);

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

float entry;

if (Plane.Raycast(Ray, out entry))

{

dragStopPosition = Ray.GetPoint(entry);

newPosition = transform.position + dragStartPosition - dragStopPosition;

}

}

}

// 使用鼠标滚轮进行缩放

private async Task ZoomCamera()

{

await Task.Yield();

if (Input.mouseScrollDelta.y != 0)

{

newZoom.Set(newZoom.x, newZoom.y + zoomAmount \* Input.mouseScrollDelta.y, newZoom.z);

}

}

// 用ctrl+右键旋转摄像机

private async Task RotateCamera()

{

await Task.Yield();

if (Input.GetMouseButtonDown(1) & Input.GetKey(KeyCode.LeftControl))

{

rotationStartPosition = Input.mousePosition;

}

if (Input.GetMouseButton(1) & Input.GetKey(KeyCode.LeftControl))

{

rotationStopPosition = Input.mousePosition;

Vector3 rotationDiff = rotationStopPosition - rotationStartPosition;

rotationStartPosition = rotationStopPosition;

newRotation \*= Quaternion.Euler(Vector3.up \* (-rotationDiff.x \* rotationSpeed));

}

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.EventSystems;

// 继承IBeginDragHandler, IEndDragHandler接口，保证在滚动条拖拽时不会触发其他操作

public class ScrollBar : MonoBehaviour, IBeginDragHandler, IEndDragHandler

{

public static bool isGUIActive = false;

public void OnBeginDrag(PointerEventData eventData)

{

isGUIActive = true;

}

public void OnEndDrag(PointerEventData eventData)

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using System.IO;

using UnityEngine.UI;

namespace UI.Panel.FileSelet

{

public class FileSelectPanel : MonoBehaviour

{

public static FileSelectPanel currentFileSelectPanel;

public delegate void Work4String();

public Work4String work4String;

public bool isReady=false;

public string filePath;

private Vector2 contentSize;

private Vector2 filePosition;

private DirectoryInfo currentDirectory;

private DirectoryInfo[] directoryInfos;

private FileInfo[] fileInfos;

private float fileHeight;

[SerializeField]

private Button fileButton;

[SerializeField]

private RectTransform content;

[SerializeField]

private Scrollbar scrollbar;

[SerializeField]

private InputField pathInputField;

[SerializeField]

private InputField fileInputField;

private void Awake()

{

currentFileSelectPanel = this;

contentSize = content.sizeDelta;

filePosition = fileButton.GetComponent<RectTransform>().anchoredPosition;

currentDirectory = new DirectoryInfo(Application.persistentDataPath);

pathInputField.text = currentDirectory.FullName;

fileHeight = fileButton.GetComponent<RectTransform>().rect.height;

UpdatePath();

}

public void UpdatePath()

{

content.sizeDelta = contentSize;

fileButton.GetComponent<RectTransform>().anchoredPosition = filePosition;

Button[] childrenTransforms = content.GetComponentsInChildren<Button>();

foreach (Button btn in childrenTransforms)

{

if (btn.gameObject.activeSelf)

Destroy(btn.gameObject);

}

if (Directory.Exists(pathInputField.text))

{

currentDirectory = new DirectoryInfo(pathInputField.text);

}

else

{

}

pathInputField.text = currentDirectory.FullName;

directoryInfos = currentDirectory.GetDirectories();

fileInfos = currentDirectory.GetFiles();

foreach (DirectoryInfo di in directoryInfos)

{

Button nbtn = Instantiate(fileButton, content);

nbtn.gameObject.SetActive(true);

nbtn.GetComponentInChildren<Text>().text = " [文件夹] " + di.Name;

fileButton.GetComponent<RectTransform>().anchoredPosition -= new Vector2(0, fileHeight + 5);

content.sizeDelta += new Vector2(0, fileHeight + 5);

nbtn.onClick.AddListener(delegate () { pathInputField.text += @"\" + di.Name; UpdatePath(); });

}

foreach (FileInfo fi in fileInfos)

{

Button nbtn = Instantiate(fileButton, content);

nbtn.gameObject.SetActive(true);

nbtn.GetComponentInChildren<Text>().text = " [文件] " + fi.Name;

fileButton.GetComponent<RectTransform>().anchoredPosition -= new Vector2(0, fileHeight + 5);

content.sizeDelta = new Vector2(0, fileHeight + 5);

nbtn.onClick.AddListener(delegate () { fileInputField.text = fi.Name; });

}

scrollbar.value = 1;

}

public void BackUp()

{

try

{

pathInputField.text = currentDirectory.Parent.FullName;

}

catch

{

}

UpdatePath();

}

public void Activate(Work4String w4s)

{

work4String = w4s;

gameObject.SetActive(true);

isReady = false;

}

public void Affirm()

{

if (File.Exists(currentDirectory.FullName + @"\" + fileInputField.text))

{

isReady = true;

filePath = currentDirectory.FullName + @"\" + fileInputField.text;

gameObject.SetActive(false);

work4String();

}

}

public void Cancel()

{

gameObject.SetActive(false);

}

}

}