using System;

using System.Collections;

using UnityEngine;

#if UNITY\_EDITOR

using UnityEditor;

#endif

namespace UnityStandardAssets.Utility

{

public class WaypointCircuit : MonoBehaviour

{

public WaypointList waypointList = new WaypointList();

[SerializeField] private bool smoothRoute = true;

private int numPoints;

private Vector3[] points;

private float[] distances;

public float editorVisualisationSubsteps = 100;

public float Length { get; private set; }

public Transform[] Waypoints

{

get { return waypointList.items; }

}

//this being here will save GC allocs

private int p0n;

private int p1n;

private int p2n;

private int p3n;

private float i;

private Vector3 P0;

private Vector3 P1;

private Vector3 P2;

private Vector3 P3;

// Use this for initialization

private void Awake()

{

if (Waypoints.Length > 1)

{

CachePositionsAndDistances();

}

numPoints = Waypoints.Length;

}

public RoutePoint GetRoutePoint(float dist)

{

// position and direction

Vector3 p1 = GetRoutePosition(dist);

Vector3 p2 = GetRoutePosition(dist + 0.1f);

Vector3 delta = p2 - p1;

return new RoutePoint(p1, delta.normalized);

}

public Vector3 GetRoutePosition(float dist)

{

int point = 0;

if (Length == 0)

{

Length = distances[distances.Length - 1];

}

dist = Mathf.Repeat(dist, Length);

while (distances[point] < dist)

{

++point;

}

// get nearest two points, ensuring points wrap-around start & end of circuit

p1n = ((point - 1) + numPoints)%numPoints;

p2n = point;

// found point numbers, now find interpolation value between the two middle points

i = Mathf.InverseLerp(distances[p1n], distances[p2n], dist);

if (smoothRoute)

{

// smooth catmull-rom calculation between the two relevant points

// get indices for the surrounding 2 points, because

// four points are required by the catmull-rom function

p0n = ((point - 2) + numPoints)%numPoints;

p3n = (point + 1)%numPoints;

// 2nd point may have been the 'last' point - a dupe of the first,

// (to give a value of max track distance instead of zero)

// but now it must be wrapped back to zero if that was the case.

p2n = p2n%numPoints;

P0 = points[p0n];

P1 = points[p1n];

P2 = points[p2n];

P3 = points[p3n];

return CatmullRom(P0, P1, P2, P3, i);

}

else

{

// simple linear lerp between the two points:

p1n = ((point - 1) + numPoints)%numPoints;

p2n = point;

return Vector3.Lerp(points[p1n], points[p2n], i);

}

}

private Vector3 CatmullRom(Vector3 p0, Vector3 p1, Vector3 p2, Vector3 p3, float i)

{

// comments are no use here... it's the catmull-rom equation.

// Un-magic this, lord vector!

return 0.5f\*

((2\*p1) + (-p0 + p2)\*i + (2\*p0 - 5\*p1 + 4\*p2 - p3)\*i\*i +

(-p0 + 3\*p1 - 3\*p2 + p3)\*i\*i\*i);

}

private void CachePositionsAndDistances()

{

// transfer the position of each point and distances between points to arrays for

// speed of lookup at runtime

points = new Vector3[Waypoints.Length + 1];

distances = new float[Waypoints.Length + 1];

float accumulateDistance = 0;

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

{

var t1 = Waypoints[(i)%Waypoints.Length];

var t2 = Waypoints[(i + 1)%Waypoints.Length];

if (t1 != null && t2 != null)

{

Vector3 p1 = t1.position;

Vector3 p2 = t2.position;

points[i] = Waypoints[i%Waypoints.Length].position;

distances[i] = accumulateDistance;

accumulateDistance += (p1 - p2).magnitude;

}

}

}

private void OnDrawGizmos()

{

DrawGizmos(false);

}

private void OnDrawGizmosSelected()

{

DrawGizmos(true);

}

private void DrawGizmos(bool selected)

{

waypointList.circuit = this;

if (Waypoints.Length > 1)

{

numPoints = Waypoints.Length;

CachePositionsAndDistances();

Length = distances[distances.Length - 1];

Gizmos.color = selected ? Color.yellow : new Color(1, 1, 0, 0.5f);

Vector3 prev = Waypoints[0].position;

if (smoothRoute)

{

for (float dist = 0; dist < Length; dist += Length/editorVisualisationSubsteps)

{

Vector3 next = GetRoutePosition(dist + 1);

Gizmos.DrawLine(prev, next);

prev = next;

}

Gizmos.DrawLine(prev, Waypoints[0].position);

}

else

{

for (int n = 0; n < Waypoints.Length; ++n)

{

Vector3 next = Waypoints[(n + 1)%Waypoints.Length].position;

Gizmos.DrawLine(prev, next);

prev = next;

}

}

}

}

[Serializable]

public class WaypointList

{

public WaypointCircuit circuit;

public Transform[] items = new Transform[0];

}

public struct RoutePoint

{

public Vector3 position;

public Vector3 direction;

public RoutePoint(Vector3 position, Vector3 direction)

{

this.position = position;

this.direction = direction;

}

}

}

}

namespace UnityStandardAssets.Utility.Inspector

{

#if UNITY\_EDITOR

[CustomPropertyDrawer(typeof (WaypointCircuit.WaypointList))]

public class WaypointListDrawer : PropertyDrawer

{

private float lineHeight = 18;

private float spacing = 4;

public override void OnGUI(Rect position, SerializedProperty property, GUIContent label)

{

EditorGUI.BeginProperty(position, label, property);

float x = position.x;

float y = position.y;

float inspectorWidth = position.width;

// Draw label

// Don't make child fields be indented

var indent = EditorGUI.indentLevel;

EditorGUI.indentLevel = 0;

var items = property.FindPropertyRelative("items");

var titles = new string[] {"Transform", "", "", ""};

var props = new string[] {"transform", "^", "v", "-"};

var widths = new float[] {.7f, .1f, .1f, .1f};

float lineHeight = 18;

bool changedLength = false;

if (items.arraySize > 0)

{

for (int i = -1; i < items.arraySize; ++i)

{

var item = items.GetArrayElementAtIndex(i);

float rowX = x;

for (int n = 0; n < props.Length; ++n)

{

float w = widths[n]\*inspectorWidth;

// Calculate rects

Rect rect = new Rect(rowX, y, w, lineHeight);

rowX += w;

if (i == -1)

{

EditorGUI.LabelField(rect, titles[n]);

}

else

{

if (n == 0)

{

EditorGUI.ObjectField(rect, item.objectReferenceValue, typeof (Transform), true);

}

else

{

if (GUI.Button(rect, props[n]))

{

switch (props[n])

{

case "-":

items.DeleteArrayElementAtIndex(i);

items.DeleteArrayElementAtIndex(i);

changedLength = true;

break;

case "v":

if (i > 0)

{

items.MoveArrayElement(i, i + 1);

}

break;

case "^":

if (i < items.arraySize - 1)

{

items.MoveArrayElement(i, i - 1);

}

break;

}

}

}

}

}

y += lineHeight + spacing;

if (changedLength)

{

break;

}

}

}

else

{

// add button

var addButtonRect = new Rect((x + position.width) - widths[widths.Length - 1]\*inspectorWidth, y,

widths[widths.Length - 1]\*inspectorWidth, lineHeight);

if (GUI.Button(addButtonRect, "+"))

{

items.InsertArrayElementAtIndex(items.arraySize);

}

y += lineHeight + spacing;

}

// add all button

var addAllButtonRect = new Rect(x, y, inspectorWidth, lineHeight);

if (GUI.Button(addAllButtonRect, "Assign using all child objects"))

{

var circuit = property.FindPropertyRelative("circuit").objectReferenceValue as WaypointCircuit;

var children = new Transform[circuit.transform.childCount];

int n = 0;

foreach (Transform child in circuit.transform)

{

children[n++] = child;

}

Array.Sort(children, new TransformNameComparer());

circuit.waypointList.items = new Transform[children.Length];

for (n = 0; n < children.Length; ++n)

{

circuit.waypointList.items[n] = children[n];

}

}

y += lineHeight + spacing;

// rename all button

var renameButtonRect = new Rect(x, y, inspectorWidth, lineHeight);

if (GUI.Button(renameButtonRect, "Auto Rename numerically from this order"))

{

var circuit = property.FindPropertyRelative("circuit").objectReferenceValue as WaypointCircuit;

int n = 0;

foreach (Transform child in circuit.waypointList.items)

{

child.name = "Waypoint " + (n++).ToString("000");

}

}

y += lineHeight + spacing;

// Set indent back to what it was

EditorGUI.indentLevel = indent;

EditorGUI.EndProperty();

}

public override float GetPropertyHeight(SerializedProperty property, GUIContent label)

{

SerializedProperty items = property.FindPropertyRelative("items");

float lineAndSpace = lineHeight + spacing;

return 40 + (items.arraySize\*lineAndSpace) + lineAndSpace;

}

// comparer for check distances in ray cast hits

public class TransformNameComparer : IComparer

{

public int Compare(object x, object y)

{

return ((Transform) x).name.CompareTo(((Transform) y).name);

}

}

}

#endif

}