splines plugin code default

```
using System;
using System.Collections.Generic;
using System.Linq;
using Unity.Collections;
using Unity.Mathematics;
using UnityEngine;
using UnityEngine.Serialization;
using UnityEngine.Splines;
using Random = UnityEngine.Random;

#if UNITY_EDITOR
using UnityEditor;
#endif

namespace UnityEngine.Splines
{
    /// <summary>
    /// SplineInstantiate is used to automatically instantiate prefabs or objects along a spline.
    /// </summary>
    [ExecuteInEditMode]
    [AddComponentMenu("Splines/Spline Instantiate")]
    public class SplineInstantiate : SplineComponent
    {
        /// <summary>
        /// The space in which to interpret the offset, this can be different from the orientation space used to
instantiate objects.
        /// </summary>
        public enum OffsetSpace
        {
            /// <summary> Use the spline space to orient instances.</summary>
            [InspectorName("Spline Element")]
            Spline = Space.Spline,
            /// <summary> Use the spline GameObject space to orient instances.</summary>
            [InspectorName("Spline Object")]
            Local = Space.Local,
```

```
    /// <summary> Use world space to orient instances.</summary>
    [InspectorName("World Space")]
    World = Space.World,
    /// <summary> Use the original object space to orient instances.</summary>
    [InspectorName("Instantiated Object")]
    Object
}

[Serializable]
internal struct Vector3Offset
{
    [Flags]
    public enum Setup
    {
        None = 0x0,
        HasOffset = 0x1,
        HasCustomSpace = 0x2
    }

    public Setup setup;
    public Vector3 min;
    public Vector3 max;

    public bool randomX;
    public bool randomY;
    public bool randomZ;

    public OffsetSpace space;

    public bool hasOffset => (setup & Setup.HasOffset) != 0;
    public bool hasCustomSpace => (setup & Setup.HasCustomSpace) != 0;

    internal Vector3 GetNextOffset()
    {
        if ((setup & Setup.HasOffset) != 0)
        {
            return new Vector3(
                randomX ? Random.Range(min.x, max.x) : min.x,
                randomY ? Random.Range(min.y, max.y) : min.y,
                randomZ ? Random.Range(min.z, max.z) : min.z);
```

```
        }

        return Vector3.zero;
    }

    internal void CheckMinMaxValidity()
    {
        max.x = Mathf.Max(min.x, max.x);
        max.y = Mathf.Max(min.y, max.y);
        max.z = Mathf.Max(min.z, max.z);
    }

    internal void CheckMinMax()
    {
        CheckMinMaxValidity();
        if (max.magnitude > 0)
            setup |= Setup.HasOffset;
        else
            setup &= ~Setup.HasOffset;
    }

    internal void CheckCustomSpace(Space instanceSpace)
    {
        if ((int)space == (int)instanceSpace)
            setup &= ~Setup.HasCustomSpace;
        else
            setup |= Setup.HasCustomSpace;
    }
}

/// <summary>
/// Describe the item prefab to instantiate and associate it with a probability
/// </summary>
[Serializable]
public struct InstantiableItem
{
    /// <summary> The prefab to instantiate.</summary>
    [HideInInspector]
    [Obsolete("Use Prefab instead.", false)]
    public GameObject prefab;
```

```
    /// <summary> The prefab to instantiate.</summary>
    [FormerlySerializedAs("prefab")]
    public GameObject Prefab;
    /// <summary> Probability for this prefab. </summary>
    [HideInInspector]
    [Obsolete("Use Probability instead.", false)]
    public float probability;
    /// <summary> Probability for this prefab. </summary>
    [FormerlySerializedAs("probability")]
    public float Probability;
}

/// <summary>
/// Describe the possible methods to instantiate instances along the spline.
/// </summary>
public enum Method
{
    /// <summary> Use exact number of instances.</summary>
    [InspectorName("Instance Count")]
    InstanceCount,
    /// <summary> Use distance along the spline between 2 instances.</summary>
    [InspectorName("Spline Distance")]
    SpacingDistance,
    /// <summary> Use distance in straight line between 2 instances.</summary>
    [InspectorName("Linear Distance")]
    LinearDistance
}

/// <summary>
/// Describes the coordinate space that is used to orient the instantiated object.
/// </summary>
public enum Space
{
    /// <summary> Use the spline space to orient instances.</summary>
    [InspectorName("Spline Element")]
    Spline,
    /// <summary> Use the spline GameObject space to orient instances.</summary>
    [InspectorName("Spline Object")]
    Local,
    /// <summary> Use world space to orient instances.</summary>
```

```
    [InspectorName("World Space")]
    World,
}

[SerializeField]
SplineContainer m_Container;

/// <summary>
/// The SplineContainer containing the targeted spline.
/// </summary>
[Obsolete("Use Container instead.", false)]
public SplineContainer container => Container;
/// <summary>
/// The SplineContainer containing the targeted spline.
/// </summary>
public SplineContainer Container
{
    get => m_Container;
    set => m_Container = value;
}

[SerializeField]
List<InstantiableItem> m_ItemsToInstantiate = new List<InstantiableItem>();

/// <summary>
/// The items to use in the instantiation.
/// </summary>
public InstantiableItem[] itemsToInstantiate
{
    get => m_ItemsToInstantiate.ToArray();
    set
    {
        m_ItemsToInstantiate.Clear();
        m_ItemsToInstantiate.AddRange(value);
    }
}

[SerializeField]
Method m_Method = Method.SpacingDistance;
```

```
/// <summary>
/// The instantiation method to use.
/// </summary>
[Obsolete("Use InstantiateMethod instead.", false)]
public Method method => InstantiateMethod;
/// <summary>
/// The instantiation method to use.
/// </summary>
public Method InstantiateMethod
{
    get => m_Method;
    set => m_Method = value;
}

[SerializeField]
Space m_Space = Space.Spline;

/// <summary>
/// The coordinate space in which to orient the instanced object.
/// </summary>
[Obsolete("Use CoordinateSpace instead.", false)]
public Space space => CoordinateSpace;
/// <summary>
/// The coordinate space in which to orient the instanced object.
/// </summary>
public Space CoordinateSpace
{
    get => m_Space;
    set => m_Space = value;
}

[SerializeField]
Vector2 m_Spacing = new Vector2(1f, 1f);

/// <summary>
/// Minimum spacing between 2 generated instances,
/// if equal to the maxSpacing, then all instances will have the exact same spacing.
/// </summary>
public float MinSpacing
{
```

```
    get => m_Spacing.x;
    set
    {
        m_Spacing = new Vector2(value, m_Spacing.y);
        ValidateSpacing();
    }
}

/// <summary>
/// Maximum spacing between 2 generated instances,
/// if equal to the minSpacing, then all instances will have the exact same spacing
/// </summary>
public float MaxSpacing
{
    get => m_Spacing.y;
    set
    {
        m_Spacing = new Vector2(m_Spacing.x, value);
        ValidateSpacing();
    }
}

[SerializeField]
AlignAxis m_Up = AlignAxis.YAxis;

/// <summary>
/// Up axis of the object, by default set to the y-axis.
/// </summary>
[Obsolete("Use UpAxis instead.", false)]
public AlignAxis upAxis => UpAxis;
/// <summary>
/// Up axis of the object, by default set to the y-axis.
/// </summary>
public AlignAxis UpAxis
{
    get => m_Up;
    set => m_Up = value;
}

[SerializeField]
```

```
AlignAxis m_Forward = AlignAxis.ZAxis;

/// <summary>
/// Forward axis of the object, by default set to the Z Axis
/// </summary>
[Obsolete("Use ForwardAxis instead.", false)]
public AlignAxis forwardAxis => ForwardAxis;
/// <summary>
/// Forward axis of the object, by default set to the Z Axis
/// </summary>
public AlignAxis ForwardAxis
{
    get => m_Forward;
    set
    {
        m_Forward = value;
        ValidateAxis();
    }
}

[SerializeField]
Vector3Offset m_PositionOffset;

/// <summary>
/// Minimum (X,Y,Z) position offset to randomize instanced objects positions.
/// (X,Y and Z) values have to be lower to the ones of maxPositionOffset.
/// </summary>
[Obsolete("Use MinPositionOffset instead.", false)]
public Vector3 minPositionOffset => MinPositionOffset;
/// <summary>
/// Minimum (X,Y,Z) position offset to randomize instanced objects positions.
/// (X,Y and Z) values have to be lower to the ones of maxPositionOffset.
/// </summary>
public Vector3 MinPositionOffset
{
    get => m_PositionOffset.min;
    set
    {
        m_PositionOffset.min = value;
        m_PositionOffset.CheckMinMax();
```

```
    }
}

/// <summary>
/// Maximum (X,Y,Z) position offset to randomize instanced objects positions.
/// (X,Y and Z) values have to be higher to the ones of minPositionOffset.
/// </summary>
[Obsolete("Use MaxPositionOffset instead.", false)]
public Vector3 maxPositionOffset => MaxPositionOffset;
/// <summary>
/// Maximum (X,Y,Z) position offset to randomize instanced objects positions.
/// (X,Y and Z) values have to be higher to the ones of minPositionOffset.
/// </summary>
public Vector3 MaxPositionOffset
{
    get => m_PositionOffset.max;
    set
    {
        m_PositionOffset.max = value;
        m_PositionOffset.CheckMinMax();
    }
}

/// <summary>
/// Coordinate space to use to offset positions of the instances.
/// </summary>
[Obsolete("Use PositionSpace instead.", false)]
public OffsetSpace positionSpace => PositionSpace;
/// <summary>
/// Coordinate space to use to offset positions of the instances.
/// </summary>
public OffsetSpace PositionSpace
{
    get => m_PositionOffset.space;
    set
    {
        m_PositionOffset.space = value;
        m_PositionOffset.CheckCustomSpace(m_Space);
    }
}
```

```
[SerializeField]
Vector3Offset m_RotationOffset;

/// <summary>
/// Minimum (X,Y,Z) euler rotation offset to randomize instanced objects rotations.
/// (X,Y and Z) values have to be lower to the ones of maxRotationOffset.
/// </summary>
[Obsolete("Use MinRotationOffset instead.", false)]
public Vector3 minRotationOffset => MinRotationOffset;
/// <summary>
/// Minimum (X,Y,Z) euler rotation offset to randomize instanced objects rotations.
/// (X,Y and Z) values have to be lower to the ones of maxRotationOffset.
/// </summary>
public Vector3 MinRotationOffset
{
    get => m_RotationOffset.min;
    set
    {
        m_RotationOffset.min = value;
        m_RotationOffset.CheckMinMax();
    }
}

/// <summary>
/// Maximum (X,Y,Z) euler rotation offset to randomize instanced objects rotations.
/// (X,Y and Z) values have to be higher to the ones of minRotationOffset.
/// </summary>
[Obsolete("Use MaxRotationOffset instead.", false)]
public Vector3 maxRotationOffset => MaxRotationOffset;
/// <summary>
/// Maximum (X,Y,Z) euler rotation offset to randomize instanced objects rotations.
/// (X,Y and Z) values have to be higher to the ones of minRotationOffset.
/// </summary>
public Vector3 MaxRotationOffset
{
    get => m_RotationOffset.max;
    set
    {
        m_RotationOffset.max = value;
```

```
            m_RotationOffset.CheckMinMax();
        }
    }

    /// <summary>
    /// Coordinate space to use to offset rotations of the instances.
    /// </summary>
    [Obsolete("Use RotationSpace instead.", false)]
    public OffsetSpace rotationSpace => RotationSpace;
    /// <summary>
    /// Coordinate space to use to offset rotations of the instances.
    /// </summary>
    public OffsetSpace RotationSpace
    {
        get => m_RotationOffset.space;
        set
        {
            m_RotationOffset.space = value;
            m_RotationOffset.CheckCustomSpace(m_Space);
        }
    }

    [SerializeField]
    Vector3Offset m_ScaleOffset;

    /// <summary>
    /// Minimum (X,Y,Z) scale offset to randomize instanced objects scales.
    /// (X,Y and Z) values have to be lower to the ones of maxScaleOffset.
    /// </summary>
    [Obsolete("Use MinScaleOffset instead.", false)]
    public Vector3 minScaleOffset => MinScaleOffset;
    /// <summary>
    /// Minimum (X,Y,Z) scale offset to randomize instanced objects scales.
    /// (X,Y and Z) values have to be lower to the ones of maxScaleOffset.
    /// </summary>
    public Vector3 MinScaleOffset
    {
        get => m_ScaleOffset.min;
        set
        {
```

```
            m_ScaleOffset.min = value;
            m_ScaleOffset.CheckMinMax();
        }
    }

    /// <summary>
    /// Maximum (X,Y,Z) scale offset to randomize instanced objects scales.
    /// (X,Y and Z) values have to be higher to the ones of minScaleOffset.
    /// </summary>
    [Obsolete("Use MaxScaleOffset instead.", false)]
    public Vector3 maxScaleOffset => MaxScaleOffset;
    /// <summary>
    /// Maximum (X,Y,Z) scale offset to randomize instanced objects scales.
    /// (X,Y and Z) values have to be higher to the ones of minScaleOffset.
    /// </summary>
    public Vector3 MaxScaleOffset
    {
        get => m_ScaleOffset.max;
        set
        {
            m_ScaleOffset.max = value;
            m_ScaleOffset.CheckMinMax();
        }
    }

    /// <summary>
    /// Coordinate space to use to offset rotations of the instances (usually OffsetSpace.Object).
    /// </summary>
    [Obsolete("Use ScaleSpace instead.", false)]
    public OffsetSpace scaleSpace => ScaleSpace;
    /// <summary>
    /// Coordinate space to use to offset rotations of the instances (usually OffsetSpace.Object).
    /// </summary>
    public OffsetSpace ScaleSpace
    {
        get => m_ScaleOffset.space;
        set
        {
            m_ScaleOffset.space = value;
            m_ScaleOffset.CheckCustomSpace(m_Space);
```

```
    }
}

// Keep old serialization of instances to ensure that no zombie instances will remain serialized.
[SerializeField, HideInInspector, FormerlySerializedAs("m_Instances")]
List<GameObject> m_DeprecatedInstances = new List<GameObject>();

const string k_InstancesRootName = "root-";
GameObject m_InstancesRoot;

Transform instancesRootTransform
{
    get
    {
        if (m_InstancesRoot == null)
        {
            m_InstancesRoot = new GameObject(k_InstancesRootName+GetInstanceID());
            m_InstancesRoot.hideFlags |= HideFlags.HideAndDontSave;
            m_InstancesRoot.transform.parent = transform;
            m_InstancesRoot.transform.localPosition = Vector3.zero;
            m_InstancesRoot.transform.localRotation = Quaternion.identity;
        }
        return m_InstancesRoot.transform;
    }
}
readonly List<GameObject> m_Instances = new List<GameObject>();
internal List<GameObject> instances => m_Instances;
bool m_InstancesCacheDirty = false;

[SerializeField]
bool m_AutoRefresh = true;

InstantiableItem m_CurrentItem;

bool m_SplineDirty = false;
float m_MaxProbability = 1f;

float maxProbability
{
    get => m_MaxProbability;
```

```
            set
            {
                if (m_MaxProbability != value)
                {
                    m_MaxProbability = value;
                    m_InstancesCacheDirty = true;
                }
            }
        }

        [HideInInspector]
        [SerializeField]
        int m_Seed = 0;

        int seed
        {
            get => m_Seed;
            set
            {
                m_Seed = value;
                m_InstancesCacheDirty = true;
                Random.InitState(m_Seed);
            }
        }

        List<float> m_TimesCache = new();
        List<float> m_LengthsCache = new();

        void OnEnable()
        {
            if (m_Seed == 0)
                m_Seed = GetInstanceID();
#if UNITY_EDITOR
            Undo.undoRedoPerformed += UndoRedoPerformed;
#endif

            //Bugfix for SPLB-107: Duplicating a SplineInstantiate is making children visible
            //This ensure to delete the invalid children.
            CheckChildrenValidity();
```

```
            Spline.Changed += OnSplineChanged;
            UpdateInstances();
        }

        void OnDisable()
        {
#if UNITY_EDITOR
            Undo.undoRedoPerformed -= UndoRedoPerformed;
#endif
            Spline.Changed -= OnSplineChanged;
            Clear();
        }

        void UndoRedoPerformed()
        {
            m_InstancesCacheDirty = true;
            m_SplineDirty = true;
        }

        void OnValidate()
        {
            ValidateSpacing();

            m_SplineDirty = m_AutoRefresh;

            EnsureItemsValidity();

            m_PositionOffset.CheckMinMaxValidity();
            m_RotationOffset.CheckMinMaxValidity();
            m_ScaleOffset.CheckMinMaxValidity();

        }

        void EnsureItemsValidity()
        {
            float probability = 0;
            for (int i = 0; i < m_ItemsToInstantiate.Count; i++)
            {
                var item = m_ItemsToInstantiate[i];
```

```
            if (item.Prefab != null)
            {
                if (transform.IsChildOf(item.Prefab.transform))
                {
                    Debug.LogWarning("Instantiating a parent of the SplineInstantiate object itself is not permitted"
+
                        $" ({item.Prefab.name} is a parent of {transform.gameObject.name}).");
                    item.Prefab = null;
                    m_ItemsToInstantiate[i] = item;
                }
                else
                    probability += item.Probability;
            }
        }
        maxProbability = probability;
    }

    void CheckChildrenValidity()
    {
        // All the children have to be checked in case multiple SplineInstantiate components are used on the
same GameObject.
        // We want to be able to have multiple components as it allows for example to instantiate grass and
        // trees with different parameters on the same object.
        var ids = GetComponents<SplineInstantiate>().Select(sInstantiate =>
sInstantiate.GetInstanceID()).ToList();
        var childCount = transform.childCount;
        for (int i = childCount - 1; i >= 0; --i)
        {
            var child = transform.GetChild(i).gameObject;
            if (child.name.StartsWith(k_InstancesRootName))
            {
                var invalid = true;
                foreach (var instanceID  in ids)
                {
                    if (child.name.Equals(k_InstancesRootName + instanceID))
                    {
                        invalid = false;
                        break;
                    }
                }
```

```
                if (invalid)
#if UNITY_EDITOR
                    DestroyImmediate(child);
#else
                    Destroy(child);
#endif
            }
        }
    }

    void ValidateSpacing()
    {
        var xSpacing = Mathf.Max(0.1f, m_Spacing.x);
        if (m_Method != Method.LinearDistance)
        {
            var ySpacing = float.IsNaN(m_Spacing.y) ? xSpacing : Mathf.Max(0.1f, m_Spacing.y);
            m_Spacing = new Vector2(xSpacing, Mathf.Max(xSpacing, ySpacing));
        }
        else if (m_Method == Method.LinearDistance)
        {
            var ySpacing = float.IsNaN(m_Spacing.y) ? m_Spacing.y : xSpacing;
            m_Spacing = new Vector2(xSpacing, ySpacing);
        }
    }

    /// <summary>
    /// This method prevents Up and Forward axis to be aligned.
    /// Up axis will always be kept as the prioritized one.
    /// If Forward axis is in the same direction than the Up (or -Up) it'll be changed to the next axis.
    /// </summary>
    void ValidateAxis()
    {
        if (m_Forward == m_Up || (int)m_Forward == ((int)m_Up + 3) % 6)
            m_Forward = (AlignAxis)(((int)m_Forward + 1) % 6);
    }

    internal void SetSplineDirty(Spline spline)
    {
        if (m_Container != null && m_Container.Splines.Contains(spline) && m_AutoRefresh)
```

```
        UpdateInstances();
    }

    void InitContainer()
    {
        if (m_Container == null)
            m_Container = GetComponent<SplineContainer>();
    }

    /// <summary>
    /// Clear all the created instances along the spline
    /// </summary>
    public void Clear()
    {
        SetDirty();
        TryClearCache();
    }

    /// <summary>
    /// Set the created instances dirty to erase them next time instances will be generated
    /// (otherwise the next generation will reuse cached instances)
    /// </summary>
    public void SetDirty()
    {
        m_InstancesCacheDirty = true;
    }

    void TryClearCache()
    {
        if (!m_InstancesCacheDirty)
        {
            for (int i = 0; i < m_Instances.Count; i++)
            {
                if (m_Instances[i] == null)
                {
                    m_InstancesCacheDirty = true;
                    break;
                }
            }
        }
```

```
            if (m_InstancesCacheDirty)
            {
                for (int i = m_Instances.Count - 1; i >= 0; --i)
                {
#if UNITY_EDITOR
                    DestroyImmediate(m_Instances[i]);
#else
                    Destroy(m_Instances[i]);
#endif
                }

#if UNITY_EDITOR
                DestroyImmediate(m_InstancesRoot);
#else
                Destroy(m_InstancesRoot);
#endif
                m_Instances.Clear();
                m_InstancesCacheDirty = false;
            }
        }

        void ClearDeprecatedInstances()
        {
            foreach (var instance in m_DeprecatedInstances)
            {
#if UNITY_EDITOR
                DestroyImmediate(instance);
#else
                Destroy(instance);
#endif
            }

            m_DeprecatedInstances.Clear();
        }

        /// <summary>
        /// Change the Random seed to obtain a new generation along the Spline
        /// </summary>
        public void Randomize()
```

```
{
    seed = Random.Range(int.MinValue, int.MaxValue);
    m_SplineDirty = true;
}

void Update()
{
    if (m_SplineDirty)
        UpdateInstances();
}

/// <summary>
/// Create and update all instances along the spline based on the list of available prefabs/objects.
/// </summary>
public void UpdateInstances()
{
    ClearDeprecatedInstances();
    TryClearCache();

    if (m_Container == null)
        InitContainer();

    if (m_Container == null || m_ItemsToInstantiate.Count == 0)
        return;

    const float k_Epsilon = 0.001f;
    Random.InitState(m_Seed);
    int index = 0;
    int indexOffset = 0;

    m_LengthsCache.Clear();
    var splineLength = 0f;
    var totalSplineLength = 0f;
    for (int splineIndex = 0; splineIndex < m_Container.Splines.Count; splineIndex++)
    {
        var length = m_Container.CalculateLength(splineIndex);
        m_LengthsCache.Add(length);
        totalSplineLength += length;
    }
```

```
            var spacing = Random.Range(m_Spacing.x, m_Spacing.y);
            var currentDist = 0f;
            var instanceCountModeStep = 0f;

            if (m_Method == Method.InstanceCount)
            {
                // Advance dist by half length if we only need to spawn one item - we want to spawn it mid total spline length
                if (spacing == 1)
                    currentDist = totalSplineLength / 2f;
                else if (spacing < 1) // Using less operator here as the spacing setters always clamp spacing to a minimum value of 0.1
                {
                    // If there's nothing to spawn, make currentDist larger than length to effectively skip prefab spawnning but still trigger previous spawn cleanup
                    currentDist = totalSplineLength + 1f;
                }

                // Take into account the Closed property only if there's one spline in container
                if (m_Container.Splines.Count == 1)
                    instanceCountModeStep = totalSplineLength / (m_Container.Splines[0].Closed ? (int)spacing : (int)spacing - 1);
                else
                    instanceCountModeStep = totalSplineLength / ((int)spacing - 1);
            }

            //Needs to ensure the validity of the items to instantiate to be certain we don't have a parent of the hierarchy in these.
            EnsureItemsValidity();

            for (int splineIndex = 0; splineIndex < m_Container.Splines.Count; splineIndex++)
            {
                var spline = m_Container.Splines[splineIndex];
                using (var nativeSpline = new NativeSpline(spline, m_Container.transform.localToWorldMatrix, Allocator.TempJob))
                {
                    splineLength = m_LengthsCache[splineIndex];
                    var terminateSpawning = false;

                    if (m_Method == Method.InstanceCount)
```

```
{
    if (currentDist > (splineLength + k_Epsilon) && currentDist <= (totalSplineLength + k_Epsilon))
    {
        currentDist -= splineLength;
        terminateSpawning = true;
    }
}
else
    currentDist = 0f;

m_TimesCache.Clear();

while (currentDist <= (splineLength + k_Epsilon) && !terminateSpawning)
{
    if (!SpawnPrefab(index))
        break;

    m_TimesCache.Add(currentDist / splineLength);

    if (m_Method == Method.SpacingDistance)
    {
        spacing = Random.Range(m_Spacing.x, m_Spacing.y);
        currentDist += spacing;
    }
    else if (m_Method == Method.InstanceCount)
    {
        if (spacing > 1)
        {
            var previousDist = currentDist;

            currentDist += instanceCountModeStep;

            if (previousDist < splineLength && currentDist > (splineLength + k_Epsilon))
            {
                currentDist -= splineLength;
                terminateSpawning = true;
            }
        }
        // If we're here, we're spawning 1 object or none, therefore add total length to currentDist
        // so that we no longer enter the while loop as the object has been spawned already
```

```
                else
                    currentDist += totalSplineLength;
            }
            else if (m_Method == Method.LinearDistance)
            {
                //m_Spacing.y is set to NaN to trigger automatic computation
                if (float.IsNaN(m_Spacing.y))
                {
                    var meshfilter = m_Instances[index].GetComponent<MeshFilter>();
                    var axis = Vector3.right;
                    if (m_Forward == AlignAxis.ZAxis || m_Forward == AlignAxis.NegativeZAxis)
                        axis = Vector3.forward;
                    if (m_Forward == AlignAxis.YAxis || m_Forward == AlignAxis.NegativeYAxis)
                        axis = Vector3.up;

                    if (meshfilter == null)
                    {
                        meshfilter = m_Instances[index].GetComponentInChildren<MeshFilter>();
                        if (meshfilter != null)
                            axis =
Vector3.Scale(meshfilter.transform.InverseTransformDirection(m_Instances[index].transform.TransformDirection
(axis)), meshfilter.transform.lossyScale);
                    }

                    if (meshfilter != null)
                    {
                        var bounds = meshfilter.sharedMesh.bounds;
                        var filters = meshfilter.GetComponentsInChildren<MeshFilter>();
                        foreach (var filter in filters)
                        {
                            var localBounds = filter.sharedMesh.bounds;
                            bounds.size = new Vector3(Mathf.Max(bounds.size.x, localBounds.size.x),
                                Mathf.Max(bounds.size.z, localBounds.size.z),
                                Mathf.Max(bounds.size.z, localBounds.size.z));
                        }

                        spacing = Vector3.Scale(bounds.size, axis).magnitude;
                    }
                }
                else
```

```
                spacing = Random.Range(m_Spacing.x, m_Spacing.y);

                nativeSpline.GetPointAtLinearDistance(m_TimesCache[index], spacing, out var nextT);
                currentDist = nextT >= 1f ? splineLength + 1f : nextT * splineLength;
            }

            index++;
        }

        //removing extra unnecessary instances
        for (int i = m_Instances.Count - 1; i >= index; i--)
        {
            if (m_Instances[i] != null)
            {
#if UNITY_EDITOR
                DestroyImmediate(m_Instances[i]);
#else
                Destroy(m_Instances[i]);
#endif
                m_Instances.RemoveAt(i);
            }
        }

        //Positioning elements
        for (int i = indexOffset; i < index; i++)
        {
            var instance = m_Instances[i];
            var splineT = m_TimesCache[i - indexOffset];

            nativeSpline.Evaluate(splineT, out var position, out var direction, out var splineUp);
            instance.transform.position = position;

            if (m_Method == Method.LinearDistance)
            {
                var nextPosition = nativeSpline.EvaluatePosition(i + 1 < index ? m_TimesCache[i + 1 -
indexOffset] : 1f);
                direction = nextPosition - position;
            }

            var up = math.normalizesafe(splineUp);
```

```
                var forward = math.normalizesafe(direction);
                if (m_Space == Space.World)
                {
                    up = Vector3.up;
                    forward = Vector3.forward;
                }
                else if (m_Space == Space.Local)
                {
                    up = transform.TransformDirection(Vector3.up);
                    forward = transform.TransformDirection(Vector3.forward);
                }

                // Correct forward and up vectors based on axis remapping parameters
                var remappedForward = math.normalizesafe(GetAxis(m_Forward));
                var remappedUp = math.normalizesafe(GetAxis(m_Up));
                var axisRemapRotation = Quaternion.Inverse(Quaternion.LookRotation(remappedForward,
remappedUp));

                instance.transform.rotation = Quaternion.LookRotation(forward, up) * axisRemapRotation;

                var customUp = up;
                var customForward = forward;
                if (m_PositionOffset.hasOffset)
                {
                    if (m_PositionOffset.hasCustomSpace)
                        GetCustomSpaceAxis(m_PositionOffset.space, splineUp, direction, instance.transform, out
customUp, out customForward);

                    var offset = m_PositionOffset.GetNextOffset();
                    var right = Vector3.Cross(customUp, customForward).normalized;
                    instance.transform.position += offset.x * right + offset.y * (Vector3)customUp + offset.z *
(Vector3)customForward;
                }

                if (m_ScaleOffset.hasOffset)
                {
                    customUp = up;
                    customForward = forward;
                    if (m_ScaleOffset.hasCustomSpace)
                        GetCustomSpaceAxis(m_ScaleOffset.space, splineUp, direction, instance.transform, out
```

```
customUp, out customForward);

                    customUp = instance.transform.InverseTransformDirection(customUp).normalized;
                    customForward = instance.transform.InverseTransformDirection(customForward).normalized;

                    var offset = m_ScaleOffset.GetNextOffset();
                    var right = Vector3.Cross(customUp, customForward).normalized;
                    instance.transform.localScale += offset.x * right + offset.y * (Vector3)customUp + offset.z *
(Vector3)customForward;
                }

                if (m_RotationOffset.hasOffset)
                {
                    customUp = up;
                    customForward = forward;
                    if (m_RotationOffset.hasCustomSpace)
                    {
                        GetCustomSpaceAxis(m_RotationOffset.space, splineUp, direction, instance.transform, out
customUp, out customForward);
                        if (m_RotationOffset.space == OffsetSpace.Object)
                            axisRemapRotation = quaternion.identity;
                    }

                    var offset = m_RotationOffset.GetNextOffset();

                    var right = Vector3.Cross(customUp, customForward).normalized;
                    customForward = Quaternion.AngleAxis(offset.y, customUp) * Quaternion.AngleAxis(offset.x,
right) * customForward;
                    customUp = Quaternion.AngleAxis(offset.x, right) * Quaternion.AngleAxis(offset.z,
customForward) * customUp;
                    instance.transform.rotation = Quaternion.LookRotation(customForward, customUp) *
axisRemapRotation;
                }
            }

            indexOffset = index;
        }
    }

    m_SplineDirty = false;
```

```
    }

    bool SpawnPrefab(int index)
    {
        var prefabIndex = m_ItemsToInstantiate.Count == 1 ? 0 : GetPrefabIndex();
        m_CurrentItem = m_ItemsToInstantiate[prefabIndex];

        if (m_CurrentItem.Prefab == null)
            return false;

        if (index >= m_Instances.Count)
        {
#if UNITY_EDITOR
            var assetType = PrefabUtility.GetPrefabAssetType(m_CurrentItem.Prefab);
            if (assetType == PrefabAssetType.MissingAsset)
            {
                Debug.LogError($"Trying to instantiate a missing asset for item index [{prefabIndex}].");
                return false;
            }

            if (assetType != PrefabAssetType.NotAPrefab && !Application.isPlaying)
                m_Instances.Add(PrefabUtility.InstantiatePrefab(m_CurrentItem.Prefab, instancesRootTransform) as
GameObject);
            else
#endif
                m_Instances.Add(Instantiate(m_CurrentItem.Prefab, instancesRootTransform));

            m_Instances[index].hideFlags |= HideFlags.HideAndDontSave;
        }

        m_Instances[index].transform.localPosition = m_CurrentItem.Prefab.transform.localPosition;
        m_Instances[index].transform.localRotation = m_CurrentItem.Prefab.transform.localRotation;
        m_Instances[index].transform.localScale = m_CurrentItem.Prefab.transform.localScale;

        return true;
    }

    void GetCustomSpaceAxis(OffsetSpace space, float3 splineUp, float3 direction, Transform
instanceTransform, out float3 customUp, out float3 customForward)
    {
```

```
    customUp = Vector3.up;
    customForward = Vector3.forward;
    if (space == OffsetSpace.Local)
    {
        customUp = transform.TransformDirection(Vector3.up);
        customForward = transform.TransformDirection(Vector3.forward);
    }
    else if (space == OffsetSpace.Spline)
    {
        customUp = splineUp;
        customForward = direction;
    }
    else if (space == OffsetSpace.Object)
    {
        customUp = instanceTransform.TransformDirection(Vector3.up);
        customForward = instanceTransform.TransformDirection(Vector3.forward);
    }
}

int GetPrefabIndex()
{
    var prefabChoice = Random.Range(0, m_MaxProbability);
    var currentProbability = 0f;
    for (int i = 0; i < m_ItemsToInstantiate.Count; i++)
    {
        if (m_ItemsToInstantiate[i].Prefab == null)
            continue;

        var itemProbability = m_ItemsToInstantiate[i].Probability;
        if (prefabChoice < currentProbability + itemProbability)
            return i;

        currentProbability += itemProbability;
    }

    return 0;
}

void OnSplineChanged(Spline spline, int knotIndex, SplineModification modificationType)
{
```

```
            if (m_Container != null && m_Container.Spline == spline)
                m_SplineDirty = m_AutoRefresh;
        }
    }
}
```

Revision #1
Created 31 July 2023 19:16:17 by naruzkurai
Updated 13 October 2023 04:09:36 by naruzkurai