Drakkar Trail Documentation

Drakkar Trail

Drakkar Trail is an add-on for Unity, designed to create smooth dynamic trails for swords, axes, and other melee weapons. With a focus on top-tier performance and user-friendly implementation.

Drakkar Trail requires a very basic knowledge of C# programming.

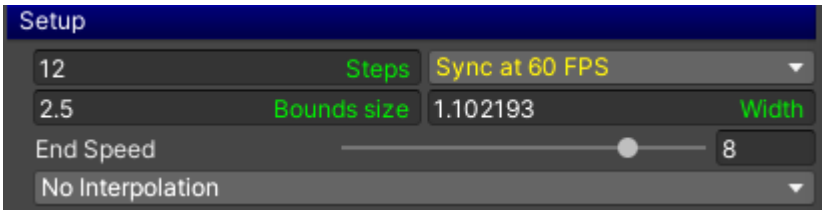
Drakkar Trail also integrates with **Drakkar Events** allowing developers to dynamically activate/deactivate trails through the use of **DrakkarActions**

You can find **Drakkar Events** here.

DrakkarTrail Component

When you add a DrakkarTrail component to your gameobject you see something like this:

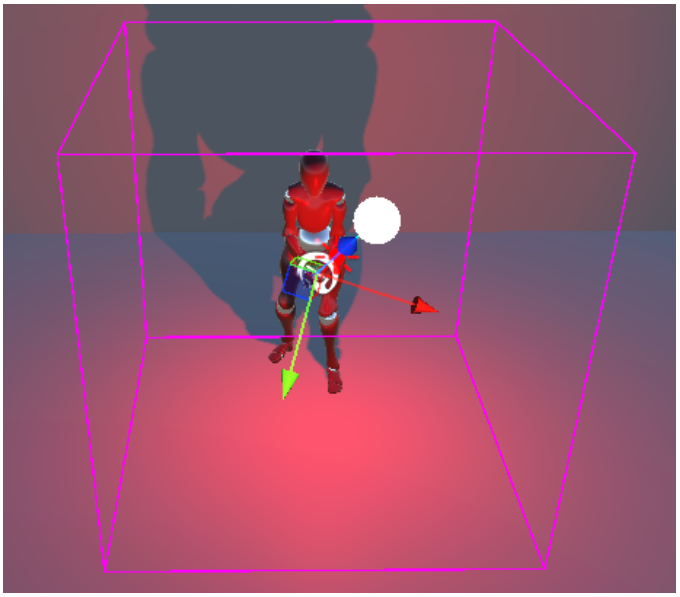
Setup



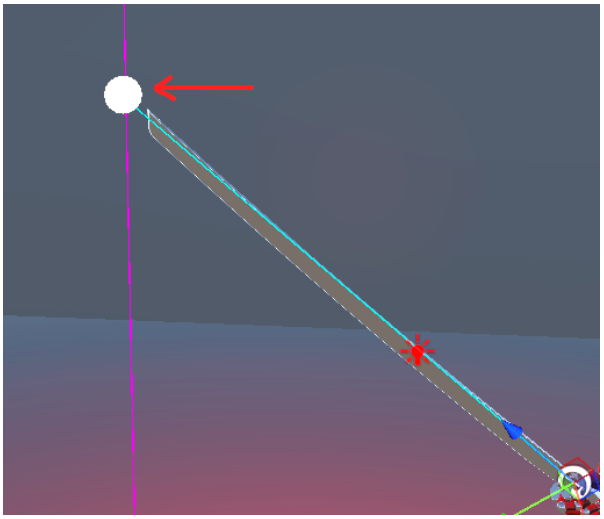
Steps represent the number of "animation frames" the system will take into account when determining the length of the trail. The higher the number the longer the trail. It can't be lower than 4.

Sync… can make the trail generation sync to a specific framerate.
Usually you don't want it to update the trail all frames because, if your game framerate will vary between, let's say, 60 and 30 fps, you would see your trail looking longer or shorter. Syncing to a specific framerate (**the slower one**) the trail will look the same at both framerates. Use **Sync at 60 FPS** only if you are totally sure your game will never drop below 60fps.

Bounds Size sets the size of the bounding box Unity uses to determine the trail's frustum visibility. If the box goes offscreen the trail will be culled.

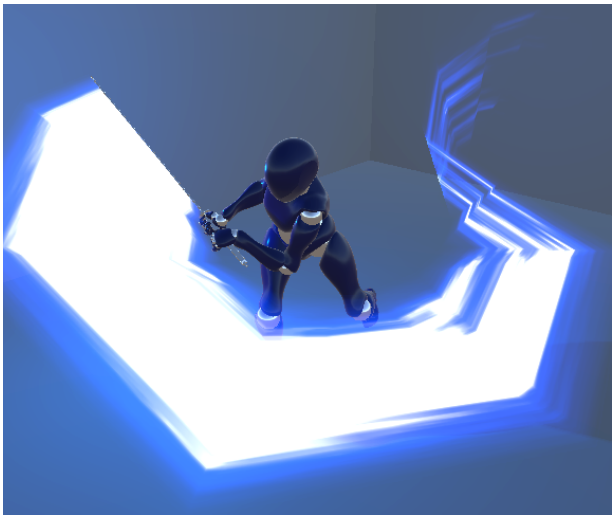


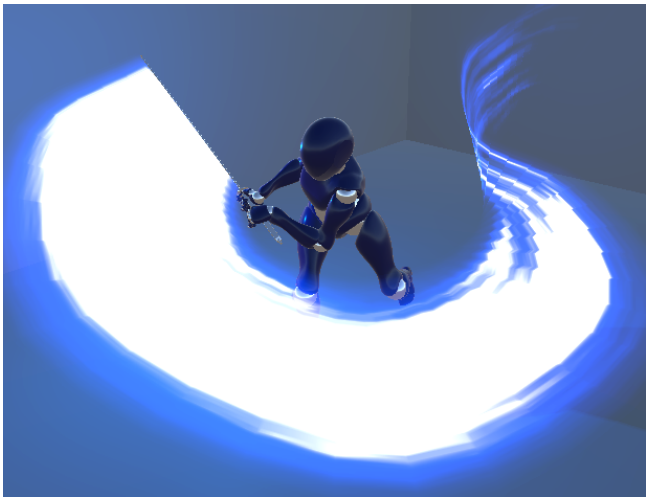
Width represents the width of the trail itself. It can be set numerically or using the handle in the scene view.



End Speed when the trail stops emitting it will quickly shorten to zero in the next frames. EndSpeed determines how fast it will shorten.

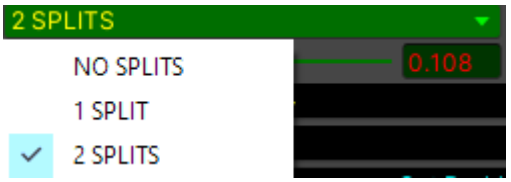
Interpolation determines how many interpolation steps the system will add to smooth the trail. With **No Interpolation** the trail could look choppy if the weapon movement is fast enough.





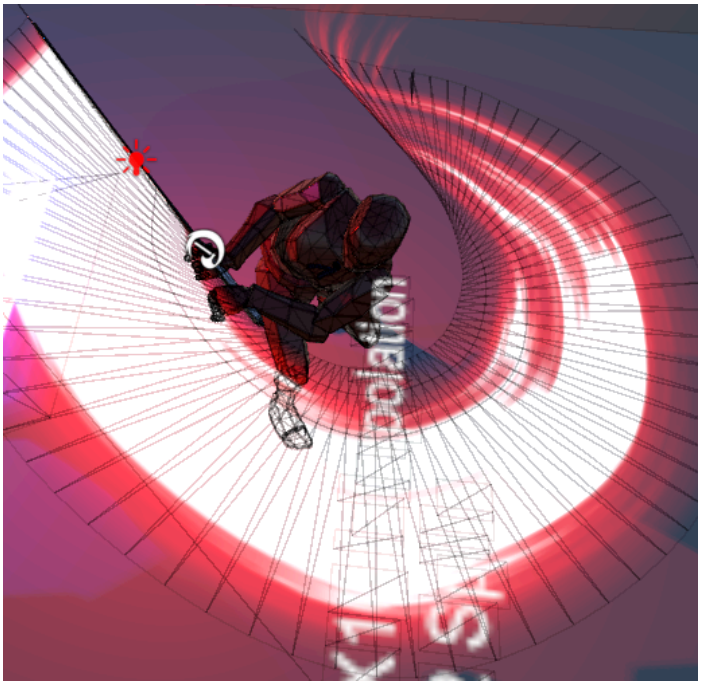
The interpolation value can vary from **No Interpolation** to **X10** when the system generates 10 more points for each trail step to smooth the trail.

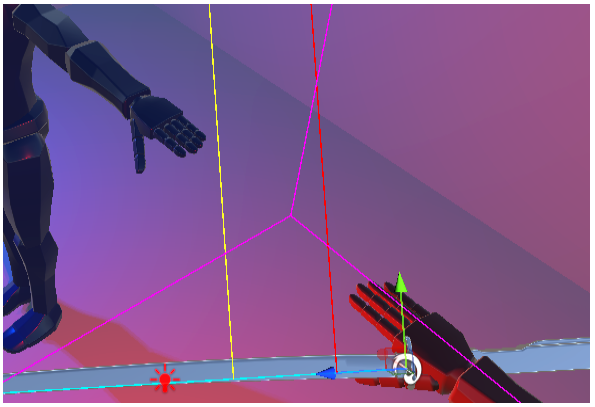
Splits determine how many edges the resulting geometry must include.
In a wide and very dense trail you might experience UV warping due to many triangles that are very long and thin. Splitting the geometry with one or two edges contributes to better distribute the vertices resulting in a more constant UV distribution.



When **Splits** are present, one or two other parameters show:

They control the position of the splits along the trail as shown by the colored gizmos.

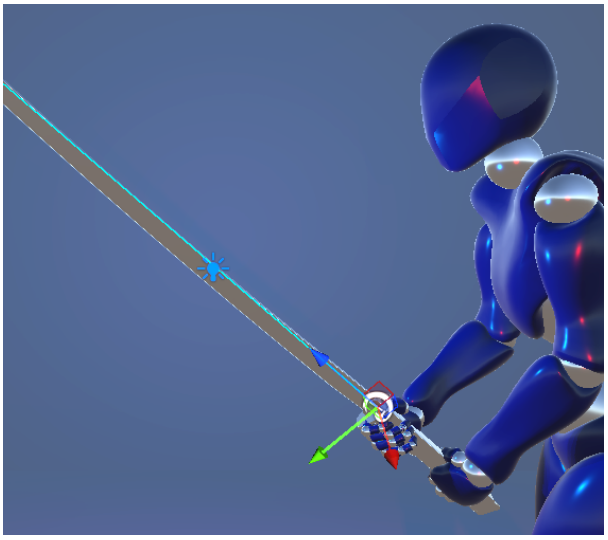




The resulting edges can be seen by using the wireframe visualization.

Rendering

Please note that the system requires the **DrakkarTrail** object to be aligned so that its **Y-axis** points in the forward direction of the weapon/blade, as shown in the picture.



Vertices just tell you how many vertices are going to be used by the trail.

Material specifies the material the trail will be rendered with.

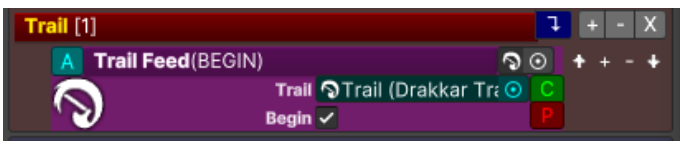
Layer specifies the layer the trail will be rendered in.

Events

Events are only available if **Drakkar Events** is properly installed. **Drakkar Events** package is available on the **Asset Store**.

On Begin, On End and **On Clear** DrakkarEvents are executed when the trail starts, when it ends and when it is cleared.

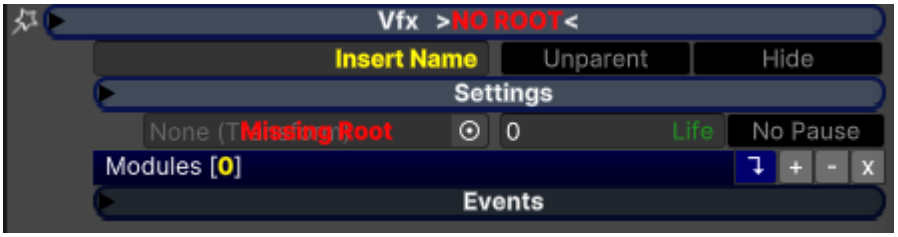
If **DrakkarEvents** is installed you can use the **TrailFeedAction** and **TrailFeedClearAction** DrakkarActions to interact with **DrakkarTrail** like the example:



Vfx

Specifies a **DrakkarVFX** that will start and stop together with the trail.

Vfx is only available if **DrakkarVFX** is properly installed. The **DrakkarVFX** package is available on the **Asset Store**.



For **DrakkarVFX** specific usage please refer to **DrakkarVFX** documentation.

Using DrakkarTrail via script

You can reference a **DrakkarTrail** in your script this way.

```
using Drakkar.GameUtils;

public DrakkarTrail Trail;
```

To make a trail start generating:

```
Trail.Begin();
```

To make a trail stop generating:

```
Trail.End();
```

To clear the trail:

```
Trail.Clear();
```

You can arrange these calls in a script and then link them through Unity's animation events:

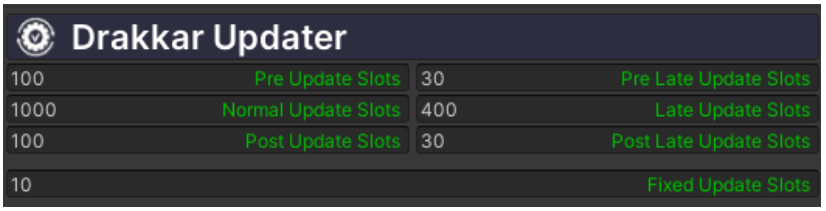
```
public class AnimationEvents : MonoBehaviour
{
	public DrakkarTrail Trail;

	public void StartTrail()
	{
		Trail.Begin();
	}

	public void StopTrail()
	{
		Trail.End();
	}
}
```

Basically calling **StartTrail** at the point you want the trail to begin and **StopTrail** when you want it to end (see the provided example scene).

Video Tutorials

Setup a new trail

Drakkar Updater



The DrakkarUpdater component is a service any script can subscribe to that helps calling many “updates” in a row without the processing overhead of Monobehaviour’s Update.

It defines how many slots can be used per event type.

The scripts can subscribe to:

- **Normal update** (similar to Unity’s Update)
- **Pre-Update** (are called BEFORE the Normal update)
- **Post-Update** (are called AFTER the Normal update)
- **LateUpdate** (similar to Unity’s LateUpdate)
- **Pre-Late update** (are called BEFORE the Late update)
- **Post-Late update** (are called AFTER the Late update)
- **Fixed update** (similar to Unity’s FixedUpdate)

Another advantage is that **ANY** class or struct can subscribe to the DrakkarUpdater, **not just Monobehaviours**.

DrakkarUpdater registration is handled by implementing the **interfaces**:

- **IUpdatable**
- **IPreUpdatable**
- **IPostUpdatable**
- **ILateUpdatable**
- **IPreLateUpdatable**
- **IPostLateUpdatable**
- **IFixedUpdatable**

A script can subscribe to one or more interfaces. Depending on the interface implemented the user must provide the following implementations:

- **OnUpdate()**
- **OnPreUpdate()**
- **OnPostUpdate()**
- **OnLateUpdate()**
- **OnPreLateUpdate()**
- **OnPostLateUpdate()**
- **OnFixedUpdate()**

Examples:

```
public class UpdateExample : MonoBehaviour, IUpdatable
{
	private void Start()
	{
		DrakkarUpdater.Add(this);	// subscribe to the DrakkarUpdater
	}
	public void OnUpdate()
	{
		// write here your update code
	}

	private void OnDestroy()
	{
		DrakkarUpdater.Remove(this);	// don't forget to unsubscribe
	}
}

public class LateUpdateExample : MonoBehaviour, ILateUpdatable
{
	private void Start()
	{
		DrakkarUpdater.AddLate(this);		// subscribe to the DrakkarUpdater
	}
	public void OnLateUpdate()
	{
		// write here your update code
	}

	private void OnDestroy()
	{
		DrakkarUpdater.RemoveLate(this);	// don't forget to unsubscribe
	}
}
```

The same pattern applies to PreUpdate, PostUpdate, FixedUpdate, etc.

Please consider a script can implement more than one interface so there can be a **OnUpdate** together with a **OnPreLateUpdate,** and so on.

Links

DRAKKARDEV.COM
EMAIL
DISCORD
FACEBOOK
YOUTUBE
TWITTER