Object Particle Spawner

By HoH Studios

Contents

Introduction

The Object Particle Spawner system allows you to utilize the standard particle system as an all-purpose object spawner for any situation. It uses a weighted-random spawning system and advanced object pooling to spawn GameObjects, or prefabs, for both 2D and 3D implementations, including animated and UI objects.

The basic way it works is the system pools the objects and activates them when particles are created by the particle system. This means that you can use all of the standard unity particle system modules and settings to provide unique and custom spawn behavior. The newly activated GameObjects are then controlled by the particle system to inherit properties from the particle like movement, rotation, scale, or easily scripted custom behavior like color. Then, the activated and controlled GameObjects are released from the system when the particle lifetime ends, when a rigidbody collision is sensed, or by manual release via code.

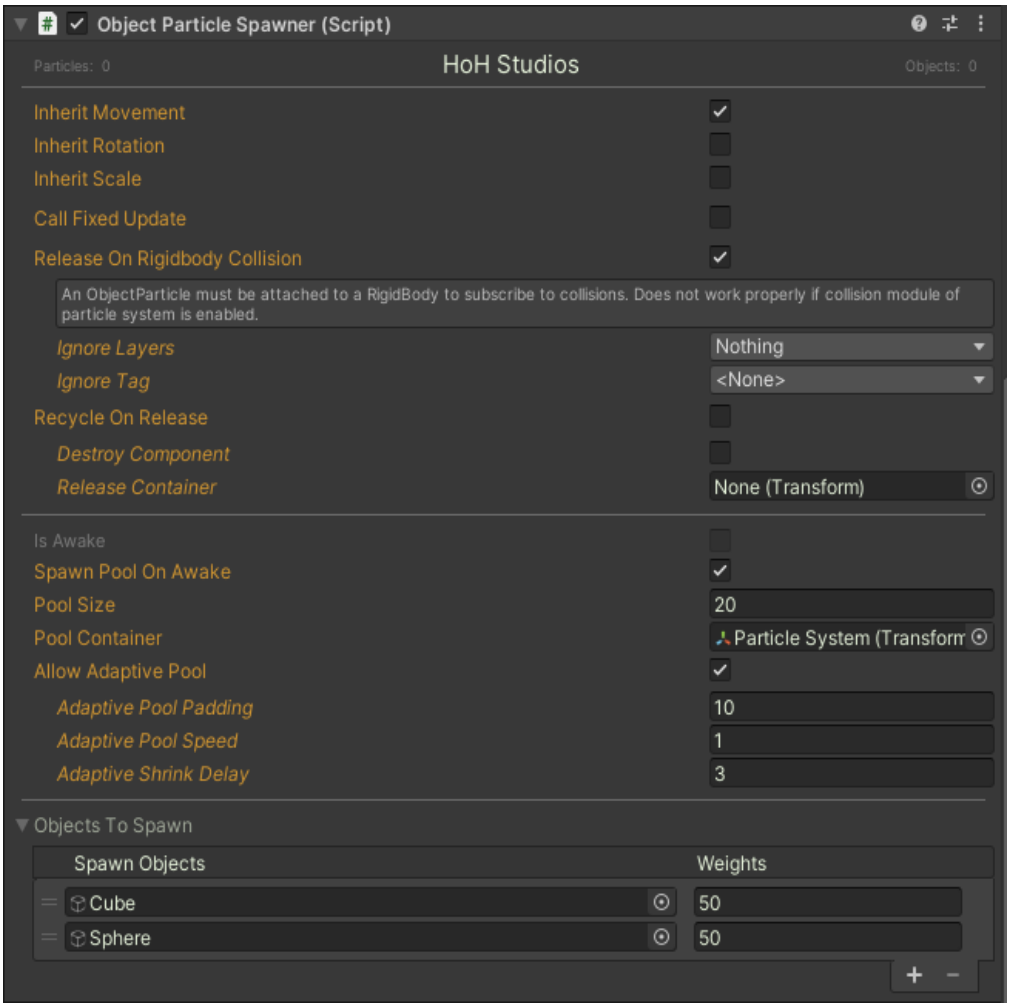
When a spawned object is released from the system, the objects can either be immediately recycled back into the object pool, or completely used for your own needs and behaviors as an independently spawned object. Then, as needed, the spawned objects can be recycled and returned back into the object pool at any point by a simple function call.

With this system, the standard unity particle system can now be completely utilized for any object spawning needs for 2D, 3D, animated, and UI objects. Additionally, fixed update support was added in v1.1 to provide more Rigidbody and physics-based capabilities.

The follow YouTube link takes you to our tutorial videos that provide more in-depth examples and tutorials for setting up and using the Object Particle Spawner Asset: https://www.youtube.com/playlist?list=PLcUXlBPTN-3V3uxfdb1B9N52Zgi4xAZNz.

Object Particle Spawner Component

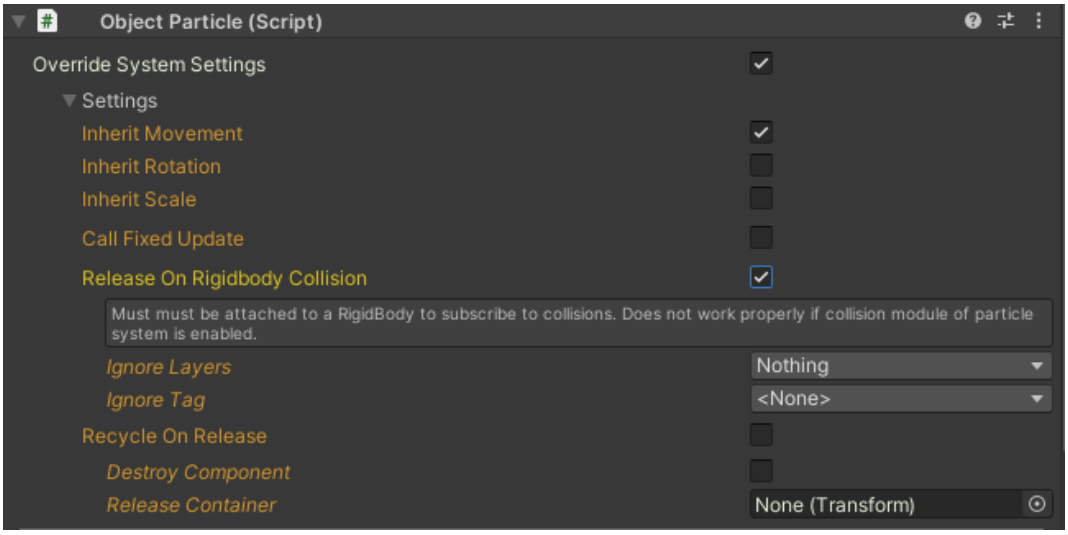
The ObjectParticleSpawner.cs component must be attached to a standard Unity Particle System to function. The spawner itself manages spawning and pooling the objects, as well as activating, updating, releasing, and providing the default spawned object particle behaviour. Let's take a deeper look at the component's settings in the inspector used in the system to provide these capabilities. Note the picture below as we step through each setting.



Object Particle Spawner	Setting Description
Inherit Movement	Allows the spawned objects to inherit movement and move with its particle throughout the particle's lifetime
Inherit Rotation	Allows the spawned objects to inherit rotation and rotate with its particle throughout the particle's lifetime
Inherit Scale	Allows the spawned objects to inherit the scale of the particle throughout the particle's lifetime
Call Fixed Update	Allow the system to call the OnFixedUpdate() function on the ObjectParticle components during fixed update. This function gets called in addition to OnUpdate() to provide support for physics and Rigidbody features
Release On Rigidbody Collision	Allows the option to release the object from the particle system when a Rigidbody collision is registered. An object particle must be attached to a Rigidbody on the spawn object for this functionality. This does not work properly if the collision module of the particle system is enabled. (Additionally, colliders must be present on the desired collision objects for unity to register the collision). See the Asset Demo scene for examples on implementing this feature.
Ignore Layers	Allows the Rigidbody detection to ignore certain layers: mainly used so that the spawned objects don't collide and release each other.
Ignore Tag	Allows the Rigidbody detection to ignore a certain tag: mainly used so that the spawned objects don't collide and release each other.
Recycle On Release	When enabled, automatically recycles objects back into the object pool when the object particle is released from the system. Objects are released from the system by either the particle lifetime ending, on rigidbody collision, or by manually calling Release() on the Object Particle component.
Destroy Component	Optionally destroys the ObjectParticle component from the spawned object if not recycled on release.
Release Container	Set a desired hierarchy object container to hold all of the released spawn objects that are not being recycled on release.
Spawn Pool On Awake	Spawns the object pool on awake.
Pool Size	The initial starting size of the object pool.
Pool Container	Set a desired hierarchy object container to hold all of the pool objects and remain organized.
Allow Adaptive Pool	When enabled, allows the pool to actively size itself to adapt to the supply and demand of the spawner system.
Adaptive Pool Padding	Padding allows additional pool objects to pad the adaptive pool. This can account for fluctuations in supply and demand.
Adaptive Pool Speed	Adaptive pool speed alters the speed at which objects are added or removed to the adaptive pool as it resizes itself. Useful for massive supply and demand fluctuations.
Spawn Objects	The game objects (or prefabs) to spawn. Drag and drop any game object into the array to feed it into the system. The spawner works with a weight-random spawning function to sustain multiple objects at the same time. Any weights that are greater than zero will be included in the spawner, and the higher the spawn weight will result in a higher chance of an object being spawned. **For instance:** A cube and a sphere with equal spawn weights will have an equal chance of spawning. A cube with a spawn weight of 75 and a sphere with a spawn weight of 25 will spawn the cube ¾ of the time and the sphere ¼ of the time.

Object Particle Component

The other component that helps drive the system is the ObjectParticle.cs component. Attach this component to the objects you intend to spawn. The Object Particle component is not required for the Object Particle Spawner to function, as the spawner will spawn any object given. The Object Particle component allows more fine-tuned custom behaviour, as you can override settings and callback functions to create unique and specific spawning features. Let's look deeper into the ObjectParticle.cs component.



These settings should look familiar, as they are identical to some of the settings in the ObjectParticleSpawner. Reference the description table above for more information if needed. By default, the *Override System Settings* option is disabled. This means that the spawned object will simply inherit the settings provided by the parent ObjectParticleSpawner component that spawned it. The component offers the option to override these settings on a per-object-specific basis, for unique spawn behaviour per-object and the ability to organize released objects in the hierarchy.

Object Particle Rigidbody Component

An additional object particle component called 'ObjectParticleRigidbody' was added to provide further Rigidbody support for 2D and 3D implementations. This new component, inheriting from ObjectParticle, utilizes the new Fixed Update features to provide smooth physics movement, but requires the 'Call Fixed Update' setting enabled on either the overridden object or the spawner system. Instead of using the basic 'ObjectParticle' component, attach the new Rigidbody component to Rigidbody objects that will be spawned to use these new features. The new features will handle Rigidbody physics to seamlessly transition the spawned object from the system to being released as an independent object. Be sure to use Interpolation or Extrapolation settings on the Rigidbody component for smooth physics.

Adding Custom Behaviour to Object Particles

Controlling the Object Particle Life-Cycle

The Object Particle component class allows fine-tuned control over your spawned object particles through some useful public functions. Notice the image below demonstrating the Object Particle's public fields and functions.

```
public class ObjectParticle : MonoBehaviour
{
    /// <summary>
    /// The most recent particle-pool information that is updated as the object lives its life cycle
    /// </summary>
    0 references
    public ObjectParticleSpawner.ParticlePoolInfo Info { get; private set; } = new ObjectParticleSpawner.ParticlePoolInfo();

    /// <summary>
    /// Releases the Object Particle from its parent Object Particle System early. Returns true if successful, false otherwise
    /// </summary>
    0 references
    public bool Release() {  }

    /// <summary>
    /// Returns the ObjectParticle to the object pool if the object pool existed and initialization references were made
    /// </summary>
    0 references
    public void Recycle() {  }
}
```

By referencing the ObjectParticle attached to a spawned object, you can easily call the *Release()* or *Recycle()* public functions at any time. As mentioned in the comments, the *Release()* function will release the object from the particle system's control early by manual function call. The *Recycle()* function allows you to manually recycle a released object back into the object pool when you are finished using it. Keep in mind that *Recycle()* can be manually called at any point on a released object as long as the spawner still exists.

The public *Info* field contains some of the essential fields that define the current ObjectParticle. The *Info* field is primarily used by the Object Particle Spawner component, but the fields may be useful in your own code.

```
public struct ParticlePoolInfo
{
    /// <summary>
    /// Returns true if this object was released from its particle spawner already, returns false if not released yet. Left as public field for performance
    /// </summary>
    public bool IsReleased;

    /// <summary>
    /// The target "instance ID" of the particle to keep track of, based on the particle's RandomSeed value. Left as public field for performance, DONT CHANGE THIS MANUALLY!
    /// </summary>
    public uint ParticleId;

    /// <summary>
    /// The spawn object ID associated with this Object, so we know which spawn object it was instantiated from, DONT CHANGE THIS MANUALLY!
    /// </summary>
    public int PoolId;
}
```

Custom Behaviour with Public Events

One way to add custom behaviour to your spawned objects is through public events revealed in the Object Particle class. These events make it easy to fully control your spawned objects without inheriting from the Object Particle class, if so desired. See these three public events in the Object Particle class below, and note that they correspond to the same overridable functions accessible by inheritance in the next section.

```
public class ObjectParticle : MonoBehaviour
{
    /// <summary>
    /// The handler for the object particle events to return both the spawner and the particle
    /// </summary>
    public delegate void ObjectParticleHandler(ObjectParticleSpawner.ParticleInfo particleInfo`, ObjectParticleSpawner spawner);

    /// <summary>
    /// An event handler that invokes its subscribers when the Object Particle is activated from its parent object pool
    /// </summary>
    public ObjectParticleHandler OnActivationEvent;

    /// <summary>
    /// An event handler that invokes its subscribers when the Object Particle is updated and controlled each frame by the Object Particle System
    /// </summary>
    public ObjectParticleHandler OnUpdateEvent;

    /// <summary>
    /// An event handler that invokes its subscribers when the Object Particle is updated on fixed update and controlled each frame by the Object Particle System
    /// </summary>
    public ObjectParticleHandler OnFixedUpdateEvent;

    /// <summary>
    /// An event handler that invokes its subscribers when the Object Particle is released from its parent Object Particle System.
    /// </summary>
    public ObjectParticleHandler OnReleaseEvent;
}
```

Note the comments above each event. An important event is the *OnReleaseEvent* which is invoked when the Object Particle Spawner is no longer controlling object. This event is important because it lets your spawned object know when the object is completely independent and controllable for your needs.

The other three events are useful for establishing unique spawn behaviour and retrieving information about your spawned object during its life-cycle. The *OnActivationEvent* is invoked when the object is initially activated from the object pool, and is useful for retrieving information and setting up data as needed. The *OnUpdateEvent* is invoked each frame that the object is being controlled by the system. This event is most useful for applying custom behaviour during the spawning sequence. The *OnFixedUpdateEvent* was added to be invoked each Fixed Update frame that the object is being controlled by the system. This event is most useful for applying custom physics and Rigidbody behaviour, and only gets called if ‘Call Fixed Update’ setting is enabled.

See the following example on how to add the custom behaviour of inheriting particle color through public events.

Example: Inherit Particle Color with Events

Let's take a deeper look at the example script provided in the asset demos. The *InheritColorEvents* script demonstrates how to add custom behaviour by subscribing to the public event handlers revealed by the Object Particle component. In this case, our *InheritColorEvents* script is a component that expects to be attached to an Object Particle component, in order to influence the spawned object's behaviour independently. Note the comments as you read the code in the following image.

```
public class InheritColorEvents : MonoBehaviour
{
    ObjectParticle _objectParticle;
    Material _objectMaterial;

    0 references
    private void Awake()
    {
        // Get the reference to the object particle component
        _objectParticle = GetComponent<ObjectParticle>();

        // Subscribe to the public events of the Object Particle
        _objectParticle.OnActivationEvent += ObjectActivated;
        _objectParticle.OnUpdateEvent += ObjectUpdated;
        _objectParticle.OnReleaseEvent += ObjectReleased;
    }

    /// <summary>
    /// Invoked when the object is activated from the object pool
    /// </summary>
    1 reference
    private void ObjectActivated(ObjectParticleSpawner.ParticleInfo particleInfo, ObjectParticleSpawner spawner)
    {
        // Add initial behaviour here
    }

    /// <summary>
    /// Invoked when the object is updated and controlled each frame from the Object Particle Spawner
    /// </summary>
    1 reference
    private void ObjectUpdated(ObjectParticleSpawner.ParticleInfo particleInfo, ObjectParticleSpawner spawner)
    {
        // Get the material for color update
        if (_objectMaterial == null)
            _objectMaterial = GetComponent<MeshRenderer>().material;

        // Set the object's color to the current color of the particle this frame
        _objectMaterial.color = particleInfo.Particle.GetCurrentColor(spawner.ParticleSystem);
    }

    /// <summary>
    /// Invoked when the object is released from the Object Particle Spawner and can act as an independent object
    /// </summary>
    1 reference
    private void ObjectReleased(ObjectParticleSpawner.ParticleInfo particleInfo, ObjectParticleSpawner spawner)
    {
        // Informs the custom behaviour when the object is now released and independent
    }
}
```

In order to inherit the particle's color via public events, we begin by gaining reference to the Object Particle component attached. Then, we subscribe to all three of the public events *OnAwake()*. In the *ObjectUpdated()* function, we begin by grabbing the object particle's material if needed. Next, we set up a reference to our *Info* variable that holds our important object particle references. Finally, we set the object material's color to the current color of the particle given, using the *Info* references to gain a reference to the parent particle system. This script demonstrates how easily we can listen for the life-cycle of the Object Particle and apply our custom behaviour to the spawned object as it is activated, updated, and released from the object pool to be used as its own independent object.

Inheriting from the Object Particle class

The Object Particle class can easily be inherited into a new class to provide even more advanced custom behaviour. Four virtual functions exist to be overridden that allow you to tap into the callback functions representing the life-cycle of the spawned object. Let's look into overriding these functions below

```
public class CustomBehaviour : ObjectParticle
{
    /// <summary>
    /// Allows for manual override to add custom behavior on object pool activation from parent ObjectParticleSpawner
    /// </summary>
    3 references
    public override void OnActivation(ObjectParticleSpawner.ParticleInfo particleInfo, ObjectParticleSpawner spawner) { }

    /// <summary>
    /// Allows for manual override of the Object's Behavior, called on late update. Call base function to apply default behaviour,
    /// or call ApplyDefaultBehaviour functions in the object particle spawner to apply modular behaviour.
    /// </summary>
    8 references
    public override void OnUpdate(ObjectParticleSpawner.ParticleInfo particleInfo, ObjectParticleSpawner spawner) { }

    /// <summary>
    /// Allows for manual physics override of the Object's Behavior, called on fixed update. No base behaviour supplied.
    /// </summary>
    4 references
    public override void OnFixedUpdate(ObjectParticleSpawner.ParticleInfo particleInfo, ObjectParticleSpawner spawner) { }

    /// <summary>
    /// Allows for manual override to add custom behavior on object particle released from parent ObjectParticleSpawner
    /// </summary>
    4 references
    public override void OnRelease(ObjectParticleSpawner.ParticleInfo particleInfo, ObjectParticleSpawner spawner) { }
}
```

Take note of the comments above each overridable method. By inheriting from the Object Particle, we can now add special behaviour to each spawned object at key points in the life-cycle. *OnActivation()* gets called once, the moment the object is activated from the pool in the spawner system. *OnUpdate()* gets called each frame that the object is alive and being controlled by the system, before being released. *OnFixedUpdate()* gets called each fixed update frame the object is alive and being controlled by the system before being released, only called if **'Call Fixed Update'** setting is enabled. *OnRelease()* gets called once, the moment the object is officially released from the system's control. Using these three functions we now have full control over our spawned objects.

A great example of inheriting from the ObjectParticle component is the new 'ObjectParticleRigidbody' component. This new component overrides some of these functions to provide extended functionality for Rigidbody behaviour for seamless and smooth physics behaviour. Look at the next section of this documentation to see more examples of how to successfully implement these ideas into practice.

Example: Inherit Particle Color with Inheritance

Let's take a deeper look at the example script provided in the asset demos, covered in one of our many tutorial videos. The *InheritColor* script demonstrates how to add custom behaviour to the object while it is being controlled by the spawner. In this case, we are inheriting from the ObjectParticle class, and we are only interested in overriding the *OnUpdate()* function.

```
public class InheritColor : ObjectParticle
{
    Material _objectMaterial;

    /// <summary>
    /// Gets called once the moment the object gets spawned
    /// </summary>
    2 references
    public override void OnActivation(ObjectParticleSpawner.ParticleInfo particleInfo, ObjectParticleSpawner spawner)
    {
        // Use this override for attaching behavior the moment the particle is activated from the pool into the system
    }

    /// <summary>
    /// Gets called each frame that the object particle is alive and being controlled by the system
    /// </summary>
    7 references
    public override void OnUpdate(ObjectParticleSpawner.ParticleInfo particleInfo, ObjectParticleSpawner spawner)
    {
        // We still want to inherit the particles movement, rotation, and scale in the base class function
        base.OnUpdate(particleInfo, spawner);

        // Get the material for color update
        if (_objectMaterial == null)
            _objectMaterial = GetComponent<MeshRenderer>().material;

        // Set the object's color to the current color of the particle this frame
        _objectMaterial.color = particleInfo.Particle.GetCurrentColor(spawner.ParticleSystem);
    }

    /// <summary>
    /// Gets called once when the particle is officially released from the system
    /// </summary>
    3 references
    public override void OnRelease(ObjectParticleSpawner.ParticleInfo particleInfo, ObjectParticleSpawner spawner)
    {
        // Use this override for attaching behavior the moment the particle is released from the system
    }
}
```

In order to inherit the particle's color, we begin by caching the object's material at the top as a private field. Then, in *OnUpdate()*, we call the base function so that we continue to inherit the default movement, rotation, and scale of the particle. Finally, we set the object material's color to the current color of the particle given, using the *Info* references to gain a reference to the parent particle system.

It's that simple to create custom behaviour for your spawned objects.

Example: Inherit Trajectory without Rigidbodies

As another example, the included demo script below provides a template on inheriting the particle's trajectory on release without Rigidbody physics.

```
/// <summary>
/// A small example demonstrating how to continue the object's movement and rotational trajectory after release, indefinitely, without Rigidbodies.
///
/// Meant to be used as reference or template for creating your own custom behaviours.
/// </summary>
0 references
public class InheritTrajectory : ObjectParticle
{
    // Establish logic variables
    private bool _startCustomBehaviour = false;
    private Vector3 _releaseVelocity;
    private Vector3 _releaseAngularVelocity;

    /// <summary>
    /// Begin custom trajectory behaviour once the object is released by setting the logic variables as needed
    /// </summary>
    3 references
    public override void OnRelease(ObjectParticleSpawner.ParticleInfo particleInfo, ObjectParticleSpawner spawner)
    {
        // Get the movement velocity on release
        _releaseVelocity = particleInfo.GetWorldVelocity(spawner);

        // Get the rotational angular velocity on release
        _releaseAngularVelocity = particleInfo.Particle.angularVelocity3D;

        // Start the custom trajectory inside of update
        _startCustomBehaviour = true;
    }

    /// <summary>
    /// Use update to move the transform position and rotation
    /// </summary>
    0 references
    private void Update()
    {
        // Don't do anything until the object is released
        if (!_startCustomBehaviour)
            return;

        var deltaTime = Time.deltaTime;

        // Translate and rotate the object to continue its trajectory as defined by OnRelease()
        transform.Translate(_releaseVelocity * deltaTime, Space.World);
        transform.Rotate(_releaseAngularVelocity * deltaTime, Space.Self);
    }
}
```

As shown above, the script does nothing until the object is released from the system. Then, on release, it caches the velocity and angular velocity of the particle. Then, in regular *Update()*, we move and rotate the object at the same velocity to continue its trajectory seamlessly and indefinitely. Now, the spawner is spawning objects that continue on their trajectory when released, providing example template on how to implement your own custom post-release behaviour.

Useful Extensions

Two useful extension methods were added to the *ObjectParticleSpawner.ParticleInfo* class, which is the main particle data structure passed as parameter to the Object Particle functions. These functions are shown below.

```
/// <summary>
/// Returns the "true" world velocity of the particle that is originally distorted by the particle system's modules.
/// Does NOT guarantee correct velocity if using "Random between two constants" or "Random between to curves"
/// for the "Speed Modifier" of "VelocityOverLifetime" module, since it would be influenced by a randomly generated number.
/// </summary>
2 references
public static Vector3 GetWorldVelocity(this ObjectParticleSpawner.ParticleInfo info, ObjectParticleSpawner spawner) { }

/// <summary>
/// Returns the world space position of the particle taking into account simulation space
/// </summary>
3 references
public static Vector3 GetWorldPosition(this ObjectParticleSpawner.ParticleInfo info, ObjectParticleSpawner spawner) { }
```

An example of using these functions can be found in the *InheritTrajectory* and *ObjectParticleRigidbody* example scripts in the project, and they provide the world velocity and position of the particle to make behaviour manipulation convenient regardless of the simulation space of the particle system. Below you can see a snippet from the *InheritTrajectory.cs* class to demonstrate how to effectively use the *GetWorldVelocity()* extension method.

```
public override void OnRelease(ObjectParticleSpawner.ParticleInfo particleInfo, ObjectParticleSpawner spawner)
{
    // Get the movement velocity on release
    _releaseVelocity = particleInfo.GetWorldVelocity(spawner);
```

In the snippet, the real-world velocity of the particle gets calculated and returned by the extension method. The same can be done for the *GetWorldPosition()* extension function to get the real-world position of the particle in the particle info regardless of the simulation space.

Optimizations

Beyond the object pooling mechanisms in the asset, great lengths were taken to micro-optimize the asset as far as reasonably possible. The main difficulty with these optimizations is that a small performance overhead spread over thousands of concurrently spawned object particles can easily become magnified. A vast majority of use cases for the spawner will never reach the need to perform any further optimizations.

However, for the sake of completeness, we will address the cases where performance becomes an issue. In most cases, especially on mobile, performance loss comes from the GPU actually rendering the objects rather than the spawner system code itself. To help you keep your systems optimized, here are some tips we recommend:

- Use dynamic batching whenever possible to help reduce GPU rendering performance
- Keep the particle system's emission as low as possible to fit your needs
- Use the Max Particles setting of the Particle System to limit performance loss
- Be sure that any code inside of the *OnUpdate()* override function of the Object Particle class is as optimized as possible, as performance loss in this function can easily magnify
- Inherit as little functionality as needed from the particles and spawner system
- Be sure that your Object Pool settings are optimally set up for your spawning needs to mitigate performance loss
- Manually recycle your released objects by calling the ObjectParticle's *Recycle()* function when the objects are no longer needed rather than destroying them
- Optimize your spawn objects and their scripts

Object Pool Class

Included with the Object Particle Spawner source code is our ObjectPool.cs class. The Object Pool class provides a robust foundation that you can use to meet your own object pooling needs. It is the base class that we use in all of our pooling needs at HoH Studios. For an example of its use, see the ObjectParticlePool.cs partial class to see how we use the ObjectPool class to manage the pooling of this asset. The Object Pool class itself contains many public functions that give the developer a lot of control for setting up and establishing a wide array of object pooling scenarios. *One of the main benefits of the Object Pool used in this asset is that you can manually set up the object pool through public functions if more fine-grained control over memory is required.*

Contact

Email us directly at hohstudiosllc@gmail.com for any additional information, help with the asset, constructive feedback, or general inquiries. Please contact us with issues you are having with any of our assets before posting negative reviews, as we are a small company that is always willing to go above and beyond for our clients and customers. Thank you, and enjoy!

Changelog

Version	Changelog
1.1 (2/1/2021)	- Restructured the data structures to minimize memory and GC impact and improve performance - Restructured the folder hierarchy of the project - Changed the '*Info'* variable contents of the Object Particle to only contain relevant information and maximize performance - Added *OnFixedUpdate()* virtual function as well as *OnFixedUpdateEvent* for fixed update physics support - Changed the parameter signatures for *OnActivation, OnUpdate, OnFixedUpdate, and OnRelease* functions and events to include particle info as well as the spawner - Added '*particleInfo.GetWorldVelocity(..)*' and '*particleInfo.GetWorldPosition(..)*' extension functions to the particle info structure to provide easy world conversions of useful information - Added *'ObjectParticleRigidbody.cs'* and *'ObjectParticleRigidbody2D.cs'* scripts to the project to provide additional Rigidbody support, meant to replace the ObjectParticle component when using rigid bodies. - Added '*InheritTrajectory'* demo script to provide a template for custom trajectory behaviour without using rigid bodies. - Added '*InheritSmoothMove'* demo script to the provide example on how to smooth the movement of non-rigid bodies OnUpdate. - Updated the previous demo scenes and added the 'Custom Behaviour Scene' demo scene to showcase custom behaviour implementations. - Updated the *'Object Pool'* pooling to minimize GC allocation, and removed pooling name-lookup utility - Removed the *'(Released)'* name suffix from released objects to minimize GC allocation and maximize performance - Changed transform parenting on-release to be managed in fixed update so there is no movement jitter when changing parent transforms - Added '*InfoField'* attribute to assist in editor displays - Added .asmdef assembly definition support to the asset to remove impact on project's re-compiling time