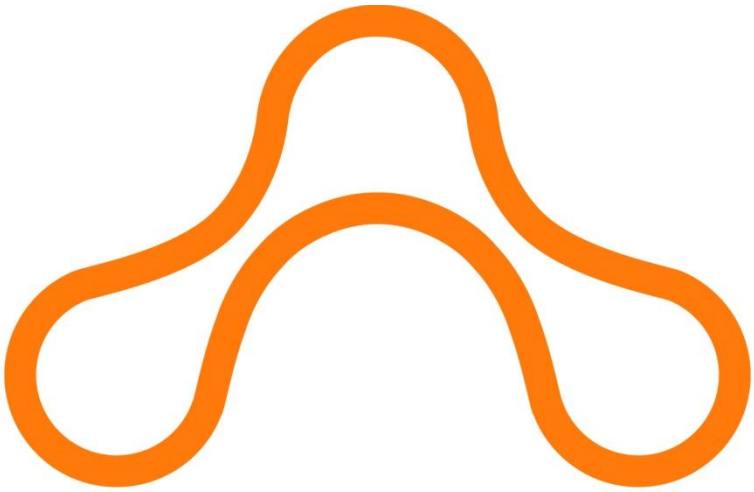

APEX PATH

Extensibility

Contents

Version History

Version 1.0	Initial Version
Version 1.1	Portals added
Version 1.2	New section on turning units and update to input receivers.

Introduction

Apex Path offers a number of extensibility points allowing you to customize a variety of the standard functionality.

This document will provide a short introduction to each of the main extensibility points.

For a full reference on all base classes and interfaces, please refer to the Apex Path API – Extensibility Digest help file.

Attributes

Defining an Attribute Set

As you may have noticed, trying to select an attribute for a unit or other attributed component only offers the option 'None', on a clean install of Apex Path.

This is because it is up to you to define the set of attributes that you wish to use.

Attributes are defined by an enum (C#). You have two options for setting up your attributes enum.

The first option is to use the Attributes Utility from within Unity. You access this via the Tools -> Apex menu. This allows you to create and manage your attribute set.

The second option is to manually create the enum.

To manually create your own attributes enum follow these steps:

1. Create a new C# file somewhere in your project and give it an appropriate name.
2. In the file define the enum.
3. Assign each value of the enum a value of 2^n where n is a value from 0 to 31, e.g. 1, 2, 4, 8, 16, 32, 64, etc.
4. You probably want to include a 'None' option with a value of 0 (zero).
5. Decorate the enum with the Flags attribute
6. Decorate the enum with the EntityAttributesEnum attribute

Example:

```
[Flags, EntityAttributesEnum]
public enum ExampleAttributeSet
{
    None = 0,
    OnRedTeam = 1,
    OnBlueTeam = 2,
    OnYellowTeam = 4,

    SpecialDoorsAvailable = 512
}
```

In the Unity editor you will now be able to select from your set of attributes when setting the attributes of components that support it.

AttributedComponent and IHaveAttributes

You can extend the attribute feature to your own classes in two ways.

If your class is a MonoBehaviour, you can simply derive from `AttributedComponent` instead.
This way the component will be ready to use, since there is already an Editor in place for such entities.

If you do not wish to derive from `AttributedComponent`, for whatever reason, you can instead implement the `IHaveAttributes` interface. Just be sure to mark each attribute field with the `AttributeProperty` attribute (yes a little confusing with these two same name terms).

AttributeSensitiveTrigger

To create triggers whose behavior depends on the attributes of those that enter and/or exit, simply derive your trigger behavior from `AttributeSensitiveTrigger`.

You can find a simple implementation of such a trigger `AttributeSensitiveTriggerExample` in the *Apex Examples/Scripts/Extensibility* folder.

There are also a few slightly more advanced examples if this type in the *Apex Examples/Misc* folder.

Path finding and Steering

Move Cost

In case you should find a need to use another move cost and heuristics provider than the default (`DiagonalDistance`), it's simple to do so.

If you simply want to use one of the other providers that come with Apex Path, you just select it on the Path Service Component on the Game World.

Creating your own provider

Create a class and implement the `IMoveCost` interface.
Alternatively you can derive your class from either `MoveCostBase` or `MoveCostDiagonalBase`, and override the relevant methods.

Specifying what provider to use

To use another provider, you need to define a factory.

Create a new MonoBehaviour and implement the `IMoveCostFactory` interface.
The implementation is simple; you just return an instance of the provider you want the path finder engine to use.

Add the factory as a component to the Game World (as a sibling to the Path Service Component).

An example factory `MoveCostFactoryExample` can be found in the *Apex Examples/Scripts/Extensibility* folder.

Path Smoothing

Apex Path already has an efficient path smoother, but should you want to write and use your own, you can do so quite easily.

Creating your own Path Smoother

Create a class and implement the `ISmoothPaths` interface.
How you choose to implement your own path smoother is of course up to you. A detailed description of the interface can be found in the Apex Path API – Extensibility Digest help file.

Specifying which Path Smoother to use

To use your own path smoother you need to define a factory.

Create a new MonoBehaviour and implement the `IPathSmootherFactory` interface.
The implementation is simple; you just return an instance of the path smoother you want the path finder engine to use.

Add the factory as a component to the Game World (as a sibling to the Path Service Component).

An example factory `PathSmootherFactoryExample` can be found in the *Apex Examples/Scripts/Extensibility* folder.

Portals

Portals need an associated action to actually move the unit when it enters the portal.
Apex Path ships with the teleport action, but you can easily create your own actions.

Creating your own Portal Action

The easiest approach is to implement `IPortalAction` on a MonoBehaviour derived class.

If you want additional control or want to use an action that is not a MonoBehaviour, then you can implement the `IPortalActionFactory` on a MonoBehaviour derived class and specify that as the action. You then create the actual action in your factory implementation.

An example of both implementations `PortalActionJumpComponent` and `PortalActionRandomComponent` can be found in the *Apex Examples/Scripts/Extensibility* folder.

IDefineSpeed

Units need a speed setting to be able to move.

Apex Path includes a default speed component for humanoids, called `HumanoidSpeedComponent.`

To create a speed component for another unit type, say a car, you need to create a MonoBehaviour class and implement the `IDefineSpeed` interface.

Add your new speed component in place of the Humanoid Speed Component.

An example speed component `CarSpeedComponent` can be found in the *Apex Examples/Scripts/Extensibility* folder. There is also a tutorial video on Speed Components on the web site.

SteeringComponent

Apex Path includes two steering components out of the box, the important of which is the `SteerForPathComponent` which is the one that enables a unit to follow a path.

However steering is not all about following paths. There are multiple other scenarios where steering can be applied. Examples include steering for separation, avoidance, cohesion, tether, flee etc.

While Apex will release a separate module with steering components, you can also create your own, which will then seamlessly integrate with the framework.

Steering works by combining the steering requests of all steering components that are attached to a unit, and this combined result will then cause the unit to move in a certain direction.

There is also the concept of steering controllers, such as patrol, wander, follow etc.

Controllers control components, e.g. the patrol behavior controls which path the unit travels along, but it is the steering component (in this case Steer for Path) that does the actual steering.

Creating your own steering component

To create a new steering component you must create a new class and derive it from `SteeringComponent`.

The component has a single method that must be overridden, and a few more that can be overridden.

In addition there are a few interfaces that may be implemented to further augment the component. The detailed information on these can be found in the Apex Path API – Extensibility Digest help file.

An example steering component `SteerToAvoidSpecificOther` can be found in the *Apex Examples/Scripts/Extensibility* folder.

Also watch the Steering and Orientation video on the web site for an in depth look at how steering works.

Moving Units

The default behavior of the `SteerableUnitComponent` is to move the unit using either the transform or its RigidBody if one is attached.

However you may want to control how movement takes place, and doing so is easy.

The easiest approach is to implement `IMoveUnits` on a MonoBehaviour derived class and attach it to the unit.

In case you want additional control or want to use an implementation that is not a MonoBehaviour, you can instead implement the `IMoveUnitsFactory` on a MonoBehaviour derived class and attach that to the unit instead. The factory must then create the actual `IMoveUnits` implementation.

An example of an `IMoveUnits` implementation `CharacterControllerMover` can be found in the *Apex Examples/Scripts/Extensibility* folder.

Turning Units

An `OrientationComponent` is responbsible for turning a unit to face in a direction, typically in the direction of movement.

However, as with movement, you may need to control how turning takes place.

To do so, derive a class from `OrientationComponent` and attach it to the unit.

Watch the Steering and Orientation video on the web site for an in depth look at how orientation works.

SteerForPathResultProcessorComponent

When the `SteerForPathComponent` receives a result back from the path finder, the default behavior is to stop if the result does not report success. Possible causes of a result not being a success can be that the destination is blocked or otherwise inaccessible.

Sometimes you may want to override the default behavior in those scenarios to take some other action instead of just stopping.

One way of doing this is to simply set the 'Navigate to Nearest if Blocked' setting on the Path Options Component to true.

Apex Path also ships with a couple such components to override the default behavior for certain scenarios, the `SteerForPathReplanWhenBlocked` and `SteerForPathReplanWhenNoRoute` respectively.

However you may want to implement your own in addition to or instead of those two.

The components are called result processors.
Once a result comes back to the steer for path component it enters a pipeline where each registered result processor gets the opportunity to handle the result.

If none of them do, the default handler is used, which is to either start along the path on success or stop on all other result outcomes.

Creating your own Result Processor

To create your own result processor, create a class derived from `SteerForPathResultProcessorComponent.` Then override a single method in which you can handle one or more of the possible result outcomes.

Add the processor to the unit (as a sibling to the Steer for Path Component).

Refer to the two aforementioned processors that ship with Apex Path for an example.

Operations

Input Receiver / Controller

The input receiver is the class responsible for processing the input received from various input devices, such as a mouse, keyboard, pad etc. It then calls methods on its associated input controller to carry out the actions.

The input controller is the class that exposes all actions that are available to be invoked by means of user input. It knows nothing about what device is used; it simply knows how to do various operations.

User input is required to have units move (of course it can also be done through scripting).

Apex Path ships with a very basic input receiver aimed for standalone windows, but this is just for ease of getting started.

You will certainly want to implement your own input receiver and most likely also your own input controller.

Apex Path includes the `InputController` class, which encapsulates the various methods used to select and move things around.

Creating your own Input Controller

It is recommended that you derive your own input controller from the `InputController` class. This way you have all your input related functionality in one place. It is however not a requirement.

Creating your own Input Receiver

To create an input receiver, create a MonoBehaviour derived class and mark it with the `InputReceiverAttribute`.
Implement your receive logic in the update method.

Add your receiver in place of the default receiver on the Game World game object.

An example input receiver that makes use of the Unity™ Input Manager and uses the `InputController` can be found in the *Apex Examples/Input* folder.

Load Balancing

One of the hurdles in creating games, is to keep a respectable frame rate. If too many things execute in the same frame things get choppy, i.e. the frame rate goes down.

In order to keep things smooth, Apex Path uses its own load balancer to evenly spread the workload across frames.

While it is not possible to directly extend the load balancer used by the Apex Game Tools, you can easily create your own load balancer class that you can then use to load balance pretty much anything.

The Apex load balancer does include a load balancer that you can use to execute your own actions; sort of like a coroutine, only it is load balanced.

This particular load balancer queue is `LoadBalancer.defaultBalancer`. You can find out more about it and the various action types in the Apex Path API – Extensibility Digest help file.

Creating your own load balancer

A load balancer is simply a class that exposes one or more `LoadBalancedQueue`s.

An example of a custom load balancer `CustomLoadBalancer` can be found in the *Apex Examples/Scripts/Extensibility* folder.

Creating your own type of load balanced items

You may also want to define additional types that can be load balanced.

For a type to be eligible for load balancing, it must implement the `ILoadBalanced` interface.

Once implemented, you can add instances of this type to a load balancer and they will be updated at the requested time interval or one of the following frames depending on system load.

You don't need to define a new type to make use of load balancing, any type can be load balanced, including MonoBehaviours, as long as the aforementioned interface is implemented.

An example of a custom load balanced item `CustomLoadBalancedItem` can be found in the *Apex Examples/Scripts/Extensibility* folder.

Messaging

The ability to loosely couple various entities is key to any software product, including games.

Apex Path ships with two message buses that allow entities to communicate without them knowing of each other directly. This is also known as the Event Aggregator pattern.

The `BasicMessageBus` is the default message bus used. It provides all the necessary functionality, but it is neither thread safe nor leak proof. Thread safety is not likely to be an issue with Unity™ but not being leak proof means that entities that subscribe to one or more messages, must remember to unsubscribe before exiting scope or being destroyed, otherwise they will not be garbage collected.

With a MonoBehaviour this means that unsubscription should be done in the OnDisable method.

The `AdvancedMessageBus` is identical to the basic version with regards to functionality. It is however both thread safe and leak proof, at the cost of some performance.

If you want to use the `AdvancedMessageBus` instead of the `BasicMessageBus` all you need to do is to add an `AdvancedMessageBusFactoryComponent` to the game world (the same game object that hosts the `GameServicesInitializerComponent`) in your scene.

Using the message bus is straightforward.

To subscribe to a given type of message, first make sure to implement the `IHandleMessage<T>` interface in the subscribing class (T being the type of message). If you want to subscribe to multiple message types, simply implement the mentioned interface for each type of message.
Then call `GameServices.messageBus.Subscribe(this)`.

If subscribing from a MonoBehaviour class you must make the subscription call in the Start or OnEnable methods, not in Awake, since the bus is not ready at that time.

To post a message, call `GameServices.messageBus.Post(new SomeMessage())`.

A message can be of any type, there are neither restrictions nor any interfaces that are required to implement.

An example of using the message bus (`MessageSubscriber` , `MessagePoster` and `MessageAttributesChanged`) can be found in the *Apex Examples/Scripts/Extensibility* folder.