



Heathen Systems

Core Developers Guide 2018.1

Overview

Heathen Systems Core is the foundation of Heathen Engineering's coded assets for the Unity 3D game engine. While the asset has been developed with Unity 2018 and later in mind the code should work with Unity 5.5 and later with little or no modification. Full source code is provided with the asset from the Unity Asset Store but remains the property of Heathen Engineering. Unity Asset Store is the only authorized distributor of Heathen Systems or any of its related components. Standard Unity Asset Store licensing conditions apply to all Heathen Systems resources and sources and its related components, additional provisions will be considered on a case by case bases.

Note that this document will be superseded by an online resource over time and is intended as a gap fill until a proper SDK can be created.

Support

Heathen Engineering Limited provides full support for Heathen Systems and related components as described in this document, on the Unity Asset Store page, the Heathen Engineering corporate site and limited to the versions of the Unity 3D game engine the asset package has been published under unless otherwise noted. You can contact support via Support@HeathenEngineering.com or the Heathen Engineering Support desk (visit [HeathenEngineering.com](https://heathenengineering.com) for details).

Join us on Discord at <https://discord.gg/RMGtDXV>

Table of Contents

Overview	2	SerializableTransform	11
Support	2	SerializableVector2	11
Table of Contents	3	SerializableVector2Int	11
Framework	4	SerializableVector3	11
Serializable	4	SerializableVector4	11
Scriptable	4	Scriptable Manifest	12
Modular	4	Game Events	13
Using Serializable Types	5	Variables	14
Extras	5	Reference Variables	15
Using Scriptable Types	6	List Variable	16
Using Modular Behaviours	7	Library Variables	17
Serializable Manifest	8	Toolbox	18
KeyedDataLibrary	8	Chronos	18
KeyedObject	10	Damage Handler	18
SerializableColor	10	Data Library Manager	18
SerializableQuaternion	10	Instance Renderer	19
SerializableRectTransform	11		

Framework

Heathen Systems Core framework overview.

Serializable

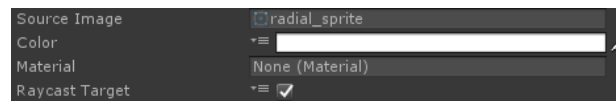
Systems Core defines serializable bridge classes for common Unity structures and classes that are not typically serializable. These structures are designed to facilitate save and load of such data and include implicit conversions to the related Unity type.

- Serializable Color
converts to and from Unity Color and Vector 4
- Serializable Quaternion
converts to and from Unity Quaternion and Vector 4
- Serializable Rect Transform
converts to and from Unity RectTransform and Transform
- Serializable Transform
converts to and from Unity Transform
- Serializable Vector 2
converts to and from Vector 2
- Serializable Vector 2 Int
converts to and from Vector 2
- Serializable Vector 3
converts to and from Vector 2 and 3
- Serializable Vector 4
converts to and from Vector 2, 3 and 4
- Serializable Data Library
Indexed collection of string keyed, object values

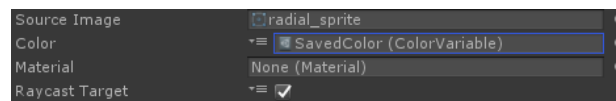
Scriptable

Systems Core defines Unity Scriptable Objects for most primitive types (string, float, etc.). Unity structures such as Color and Vectors and event systems (Unity Event with and without parameters) are also available and the Scriptable system includes base classes and inspectors for easy extension and integration with existing code.

The concept is that most if not all occurrences of data types meant for designers to populate at design-time e.g. *public float* (example); can be replaced with *public FloatReference* (example) enabling the designer to choose between constant or scriptable input values e.g. 'type it in' or 'reference it from elsewhere'.



Or



Using scriptable expressions for common and player data makes it faster and easier to maintain symmetry in design, carry data between scenes or even game sessions and enables designers to define save data, save it and load it with no engineer required via the Data Library File Manager.

Modular

Components and behaviours have been designed to server designers and developers as puzzles pieces. Each component developed for a specific discrete task without dependency or awareness of the wider system.

The modularity concept insures behaviours are easily tailored to your games specific needs, reduces the work impact of changing code by isolating functionality and capitalizes on reuse of both data (scriptable objects) and functionality (modular behaviours).

Systems Core provides a set of basic components and developer tools for easy extension with additional components available in Heathen Systems UIX and Heathen Systems Extensions with more expected as Heathen Engineering continues to mature the framework.

Using Serializable Types

Heathen Systems Core serializable variants of Vector2, Quaternion and others are meant for use on objects that must be serializable. The stock Unity equivalent of these type are not traditionally serializable (e.g. binary serializable). Each type includes implicit operators and where required UnityEvent variants to support there use in as seamless a manner as possible along side the native Unity types they represent.

Note that serializable variants do not include the native functionality of there Unity counter parts e.g. SerializableQuaternion cannot perform rotations for you it can however be assigned to a UnityEngine.Quaternion to create a quaternion rotation that can carry with it the x, y, z and w matrix values it was serialized with. Because scriptable variables must support binary serialization (to enable file saves) there underlying data type is the serializable variant of the related Unity type ... e.g. Vector2Variable has a base type of SerializableVector2 ... the base type is the value that will be reported on change events. E.g. UnityEvent<SerializableVector2> not UnityEvent<Vector2>.

In most cases a serializable value will implicitly convert the required Unity type such as when assigning variables

```
Public Vector2 MyVector = new SerializableVector2();
```

The above statement does compile and yields the desired results. Understanding what is happing is key to good performance; in this case a new SerializableVector2 is being initialized then converted to a Vector2 which creates a new Vector2. This example is not performant but does illustrate the case.

Similarly, equivalency tests undergo the same conversion where the Unity native type causes the serializable variant to first be converted (generating a new Unity value of that type) before performing the conversion.

It is recommended to only use Serializable types as data storage and to cache their Unity type values for use in any logic tests or processes where more than a single read is required.

Extras

Heathen Systems Core includes a collection of UnityEvent derived classes that will draw proper inspectors when used e.g. instead of UnityEvent<string> use UnityStringEvent. These are simple wrappers an add no functionality above the typical UnityEvent<type> declaration other than displaying correctly in the editor.

A thread safe console logger (ConsoleLogger) has been provided and can be accessed by ConsolLogger.Log(string message) as a static method.

Additional interfaces, delegates and attribute properties have been provided including [ShowOnly] and [EnumFlags] to improve the quality of default inspectors.

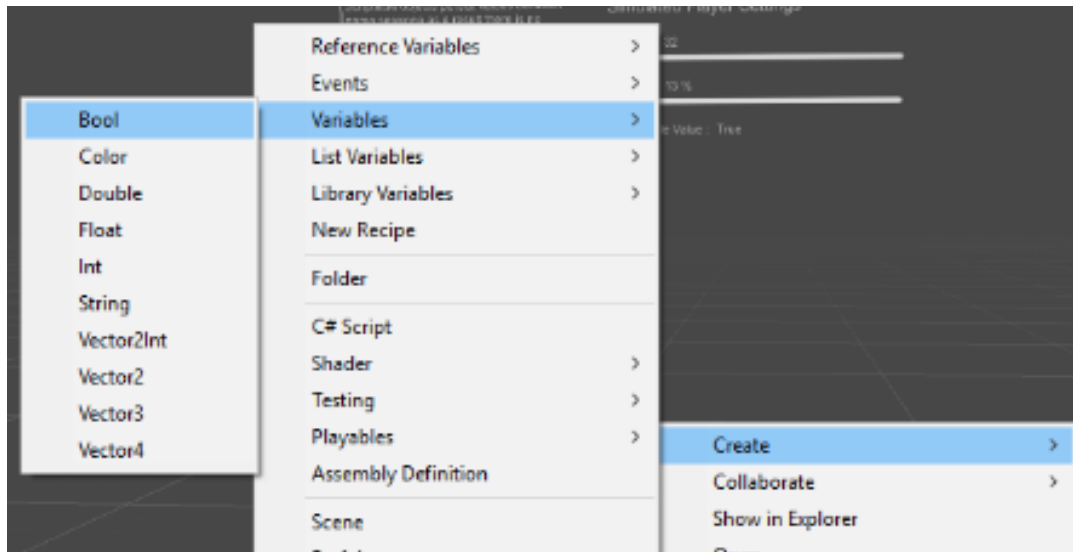
Using Scriptable Types

Scriptable Objects are a standard feature of Unity though often underutilized. Unity Learn has a freely available tutorial on the subject available in a basic (<https://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/scriptable-objects>) and intermediate (<https://unity3d.com/learn/tutorials/topics/scripting/ability-system-scriptable-objects>) presentation.

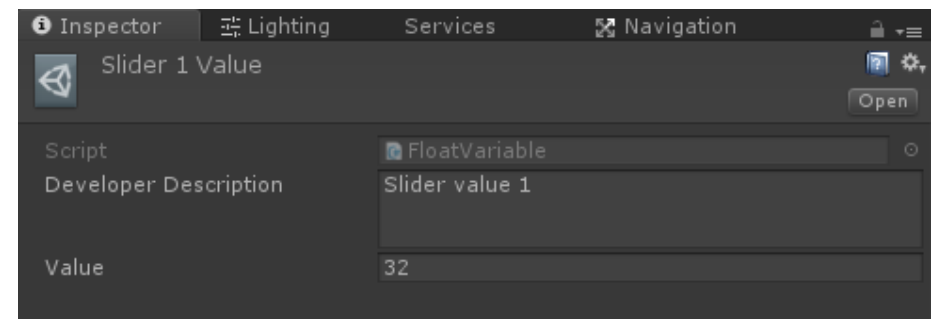
Heathen Systems Core defines common use scriptable objects representing simple data types, lists and events and provides companion behaviours to help designers leverage scriptable objects via the inspector just as they would any other field.

For developers Heathen Systems Core provides Interfaces, abstract classes and base classes along with custom inspectors and property drawers to insure fast seamless integration of custom objects and logic with Unity and Systems Core.

To create a new scriptable object right click in your project and select Create, along the top of the menu you will find categories of scriptable objects defined by Systems Core.




Once created you can select the new variable object in your project and edit its values via the inspector and assign it to references in your scenes.



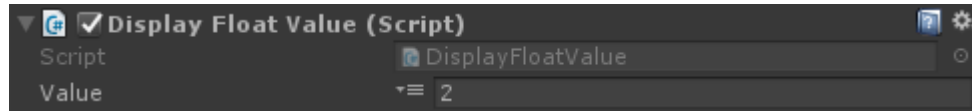
Using Modular Behaviours

Heathen Systems Core behaviours work like any other mono behaviour and can be added to game objects to build up functionality. All Heathen Systems Core behaviours leverage Variable References in place of primitive value fields this allows the designer to choose rather they will type in a constant value or reference a Scriptable Object Variable to provide the value to the behaviour.

 (menu) buttons appear beside Variable Reference fields and can be clicked to change the fields mode.

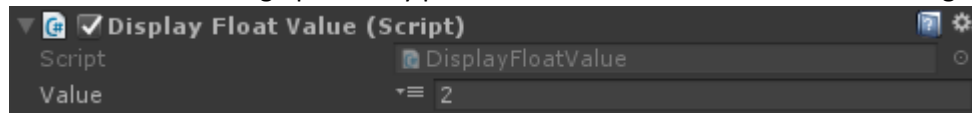
- Constant

Works like a primitive field allowing the designer to directly type in a value or use the standard inspector controls such as the color picker



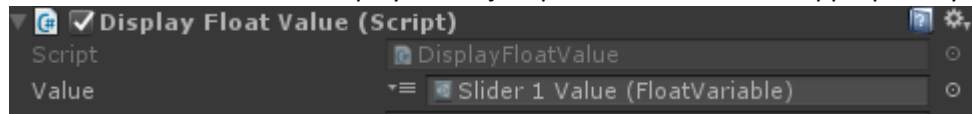
- Static

Works the same as Constant but at runtime will ignore updates to its value that aren't directly assigned to its Constant Value. This is useful to selectively prevent references from being updated by procedural events but shouldn't inhibit designers or developers from setting the value deliberately.



- Variable

In variable mode the field displays an object picker filtered for the appropriately typed Scriptable Object Variable.



The property drawer for Variable Reference is generic and will draw any class that inherits from Variable Reference in this manner. Note that the inspector does assume that the Reference has a public `<type> ConstantValue; member` and a public `DataVariable<type> Variable; member`. These will be used to draw the controls and calculate the changes in size. The property drawer also handles child values and list values and always displays string values as multi lined.

Additional component behaviours are available in other Heathen Engineering assets which leverage the Heathen Systems Core. Systems Core is provided with all such Unity Asset packs.

Serializable Manifest

KeyedDataLibrary

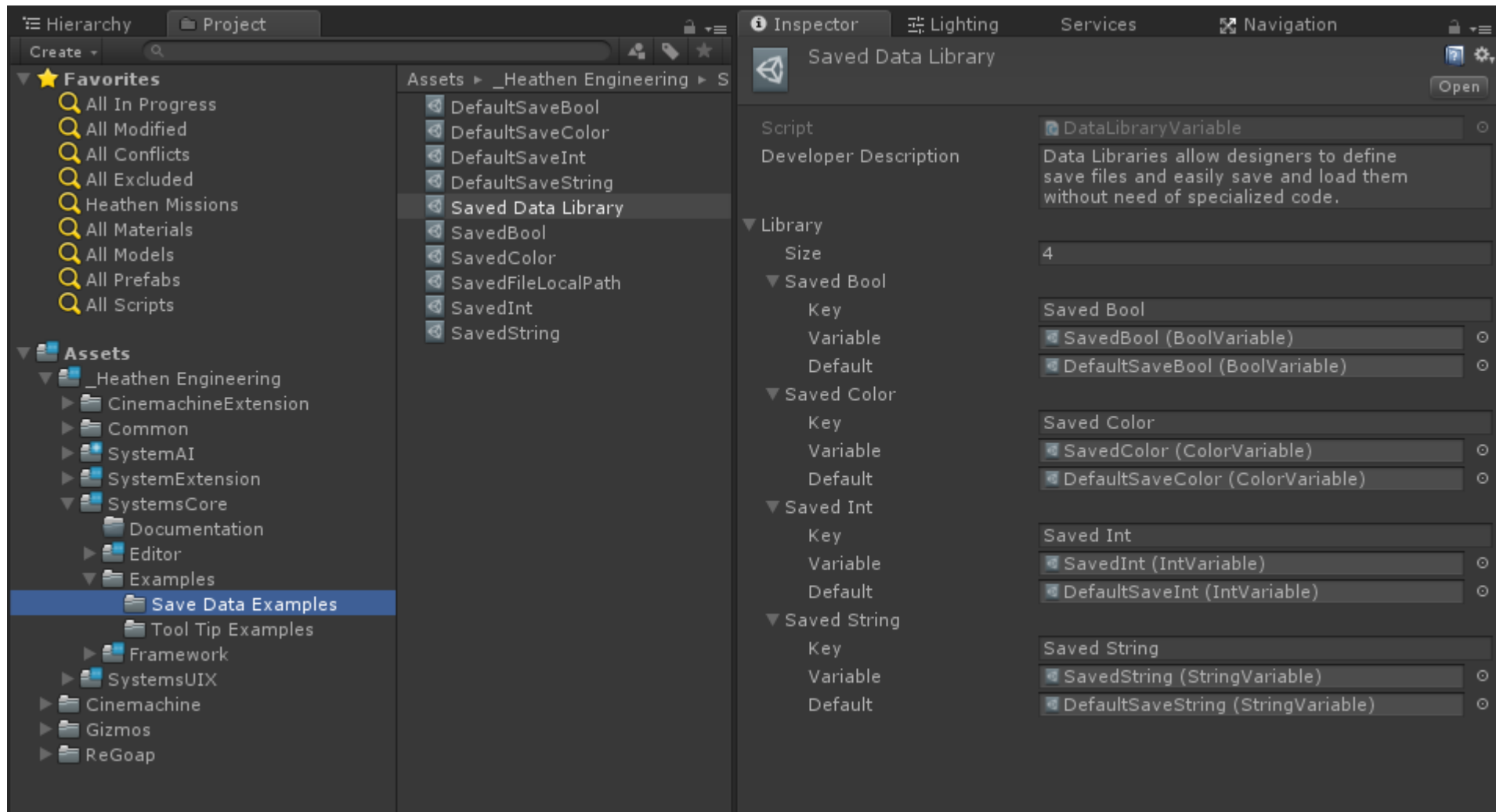
Stores a list of KeyedObject and provides typical dictionary like interfaces to developers. The KeyedDataLibrary populates a Dictionary<string,KeyedObject> and manages the references with the serializable list. This method uses slightly more memory than a simple list but offers vastly superior performance with look up operations in particular of large or complex data sets.

The intended use is to function as a generic save file structure and to this end KeyedDataLibrary includes ready to use static Serialize and Deserialize functions which operate on byte arrays for easy implementation with most data sources (Disk, Steamworks Cloud, etc.). A companion component 'DataLibraryFileManager' and a scriptable object 'DataLibraryVariable' allows the system to leverage Heathen Systems scriptable objects and enables designers to implement save file data models and save and load functions with no programming required.

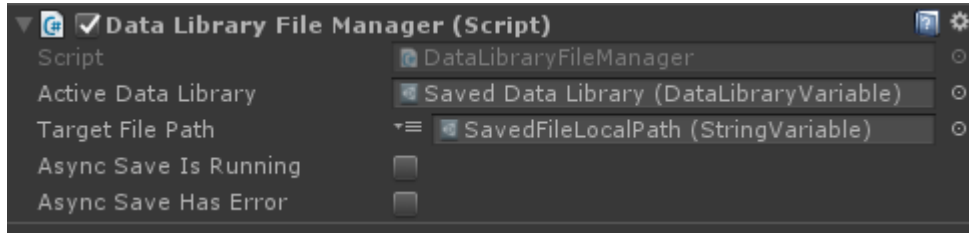
How to use

Right click in your Project panel and select "Create > Library Variable > Data Library"

This will create a new scriptable object where you can define the data you wish to save and load. You can define 2 scriptable variables for each data point in the library representing the 'Variable' or active value and the 'Default' value such as to be applied when initializing a new library. This Data Library scriptable object will represent your 'active profile' for lack of a better term, that is when you load a saved file the 'Variable's will be populated with values matched on 'Key'. Variables can be referenced via their individual scriptable objects or via the Data Library itself if you are more comfortable in code.



Once you have your library and its data model to your suited to your needs you are ready to use it, save it and load it at will. The Data Library File Manager is a ready to play solution which takes a reference to a Data Library Variable and a string (typed in or referenced by variable) and provides simple functions for saving and loading.



Simply add a standard `UnityEngine.UI.Button` and assign the `DataLibraryFileManager.AsyncSyncToFile` to the click event for you 'Save' button and `DataLibraryFileManager.SyncFromFile` to the click event for a 'Load' button.

Note save can be performed asynchronously (won't make the main thread wait) or synchronously (main will wait till save complete) but the 'SyncFrom' functions will require main thread.

Remember you can have multiple Data Libraries representing different structures, while Data Library can handle very large data sets its often advantageous to break save files up by the frequency of change e.g. System Settings which don't change often are typically saved separately from 'Profiles' which generally change many times per game session.

KeyedObject

Keyed Object is the string/object pair used by the `KeyedDataLibrary`. While it can technically store any value, you should only store values that are serializable if you wish to save and load the library to and from disk. Note that all of Heathen Systems 'Variables' are binary serializable for this reason. Its also important to note that the Data Library object and its File Manager expect the values stored to implement 'DataVariable' class.

How to use

This object is used by the system in the 'back end' and shouldn't need to be accessed directly. With regards to developing new custom classes that can be managed by Data Library you only need insure that your class object is serializable (e.g. insure you remember `[Serializable]` on your class declaration) and that you provide a `DataVariable` such that your designers can reference the type as a variable. See Data Variable section below.

SerializableColor

Serializable Color is a minimalistic expression of Unity's Color meant to be used with serialization processes. The class its self has very little functionality and simply stores the required data to construct a color.

Serializable Color is derived from `SerializableVector4` and includes implicit operators to convert between `UnityEngine.Color` and `SerializableColor`.

`SerializableColor` should only be used on objects that need to serialize a color value.

SerializableQuaternion

`SerializableQuaternion` is a minimalistic expression of Unity's Quaternion meant to be used with serialization processes. The class its self has very little functionality and is derived from `SerializableVector4` similar to Color.

SerializableQuaternion should only be used on objects that need to serialize a quaternion value.

SerializableRectTransform

A composite class leveraging Serializable Vector2, 3 and 4 by inheritance from SerializableTransform. The class defines an implicit operator for easy conversion between UnityEngine RectTransform and SerializableRectTransform and should only be used on objects that need to serialize a full RectTransform.

Note that serialized transforms of any kind only store the position, rotation and scale of an object they do not store the in-game reference, that is storing a transform is only useful for setting the position, rotation and scale of objects after a game load, other processes must handle instantiating the game objects themselves.

SerializableTransform

A composite class leveraging Serializable Vector2, 3 and 4. The class defines an implicit operator for easy conversion between UnityEngine Transform and SerializableTransform and should only be used on objects that need to serialize a full Transform.

Note that serialized transforms of any kind only store the position, rotation and scale of an object they do not store the in-game reference, that is storing a transform is only useful for setting the position, rotation and scale of objects after a game load, other processes must handle instantiating the game objects themselves.

SerializableVector2

Serializable class representation of UnityEngine.Vector2 structure. The class defines implicit operators for easy conversion between UnityEngine.Vector2 and SerializableVector2.

SerializableVector2Int

Serializable class representation of UnityEngine.Vector2Int structure. The class defines implicit operators for easy conversion between UnityEngine.Vector2Int and SerializableVector2Int.

SerializableVector3

Serializable class representation of UnityEngine.Vector3 structure. The class defines implicit operators for easy conversion between UnityEngine.Vector3 and SerializableVector3.

Derived from SerializableVector2.

SerializableVector4

Serializable class representation of UnityEngine.Vector4 structure. The class defines implicit operators for easy conversion between UnityEngine.Vector4 and SerializableVector4.

Derived from SerializableVector3.

Scriptable Manifest

Note that all Data Variables and Reference Variables include a 'Reference' serializable expression such that they can easily be references in place of the corresponding 'primitive'.

For example: `public float MyFloat = 1.25f;`

becomes: `public FloatReference MyFloat = new FloatReference(1.25f);`

Note Variable References are class objects thus follow reference semantics whereas float and other primitives are immutable structs and thus follow value semantics, the difference is important to remember when dealing with Boolean logic in particular e.g. imagine 2 variable references storing the same value, performing a direct bool check of if `a == b` is performing a reference test not a value test and will result in false, use `a.Value == b.Value` to test the values equivalency.

Primitive reference types such as 'string', 'float', 'int', etc. and Unity structures such as 'Color', 'AnimationCurve', 'Vector3', etc. derive from the base class `VariableReference`. `VariableReference` defines a property drawer allowing the designer to choose between 'constant', 'static' and 'variable' values.

- **Constant**
The field will display in inspector according to the default property drawer of its type e.g. float in the case of a `FloatReference`
The value will behave the same as a 'primitive' field e.g. can be read from and written to and does not store against a scriptable object.
- **Static**
The field will display in inspector according to the default property drawer of its type e.g. float in the case of a `FloatReference`
The value will behave the same as a Constant but will ignore attempts to set its 'Value' member e.g. read only. This can be bypassed in code by directly setting the 'ConstantValue' member. The purpose is to insure events and other procedural mechanisms cant update a value that should remain protected. It is not meant to prevent designers or developers from setting the value deliberately.
- **Variable**
The field will display in inspector as a reference slot of the related Variable type (scriptable objects)
The value can be read from or written to via the same mechanisms and will update the value of the linked scriptable object accordingly.

Data Variables also include change event notification e.g. fire events passing their value to the listener when the value is updated. Data Variable events are of type `Game Event`; you can use Listeners (provided components) to register objects to listen for specific variable change events or alternatively define your own game events and trigger them or listen to them as required.

Game Events

Game events are defined as scriptable objects and as such can be easily referenced at design or run-time across multiple scenes if required. Game Events simply record the listeners which register to the event via 'Listener' behaviours, game events can be easily triggered from typical Unity Actions (designer) or from code (programmer). All Data Variable types are also game events of their corresponding type and automatically trigger their events on change of their value each variable type includes a specific data typed listener which supports a single parameter event of that type.

Events are particularly useful for reducing the number of Boolean tests. By Boolean test we are referring to the practice of testing for change on each loop e.g. if(a == b) then else; unless the value is expected to change many times a second such as position this incurs unnecessary overhead. Registering on an event and calling its listener is not 'free' of overhead but in many cases offers a lower processing cost compared to testing on every update in particular for settings values such as volume, UI styling, etc..

In addition to the DataVariable change events the following are generic events that can be created and triggered as needed.

Bool

The process that triggers the event provides a bool value which will be passed to all listeners

Collision

The process that triggers the event provides a Collision object defining a collision state which will be passed to all listeners

Double

The process that triggers the event provides a double value which will be passed to all listeners

Float

The process that triggers the event provides a float value which will be passed to all listeners

Int

The process that triggers the event provides a int value which will be passed to all listeners

String

The process that triggers the event provides a string value which will be passed to all listeners

Trigger

The process that triggers the event provides a Collider which will be passed to all listeners

Game

Generic event which does not pass any data to listeners

Variables

These are serializable and can be used with Data Libraries. Each variable defines a Game Event on change of its value Change Event Listeners are available for each type which register themselves on enable and unregister themselves on disable optionally raising the event internally when they bind to the event source (the Variable). Change events can be used to effectively bind a set of actions to the state of a variable's value this can be lower impact than setters which update on each frame in cases where values change infrequently.

Note that composite types such as Vector2, Vector2Int, etc. will send Serializable variants of their values. The change events do test for any change of a member value not for change of reference value e.g. if vector2.x or vector2.y has changed but not if Vector2 a is not Vector2 b when a and b have the same x and y values.

Bool

Allows designers to reference bool as a variable and listen for change events

Color

Allows designers to reference color as a variable and listen for change events

Double

Allows designers to reference double as a variable and listen for change events

Float

Allows designers to reference float as a variable and listen for change events

Int

Allows designers to reference int as a variable and listen for change events

String

Allows designers to reference string as a variable and listen for change events

Vector2

Allows designers to reference Vector2 as a variable and listen for change events

Vector2Int

Allows designers to reference Vector2Int as a variable and listen for change events

Vector3

Allows designers to reference Vector3 as a variable and listen for change events

Vector4

Allows designers to reference Vector4 as a variable and listen for change events

Reference Variables

These are not serializable and do not report change events but share the same Variable Reference property drawer enabling designers the freedom of choice with regards to constant or variable reference.

In general these are used to provide shared access to common object references for example, the Main Camera which is classically accessed via tag (poor performance), most modern games however have multiple cameras and need to reference them as well (UI Camera, Effects Camera, etc.) singletons are a classic example but this breaks the modularity rule, a better approach is to register cameras with Scriptable Objects at run time, these scriptable objects can be referenced at design time and don't need any direct understanding of what cameras they will be referencing as a result components that leverage these Reference Variables remain isolated from the components that may have populated them making them easier to test and maintain.

The same example can be used for colors, animation curves, lists of game objects and more. The concept is simply to store the references in a previously agreed upon location that is independent of any context, scene or game object.

AnimationCurve

Allows designers to reference animation curves as variables

Camera

Allows designers to reference cameras as variables

Canvas

Allows designers to reference Canvas as variables

GameObject

Allows designers to reference game objects as variables

RectTransform

Allows designers to reference RectTransforms as variables

Transform

Allows designers to reference Transforms as variables

List Variable

List Variables are specialized collections which can be expressed as a single scriptable object. Typically List Variables can only be referenced and do not include a Variable Reference component.

List variables such as Object and Transform include companion 'Lister' components that can be attached to game objects causing the game object to be registered to the list and removed from it on destroy.

Component

Stores a collection of 'component' which is the base type for all mono behaviours, this list can be used to carry a list of references to any mono behaviours.

Keyed String

Stores two strings representing key and variable. This list type is used by Swap Key allowing designers to define keys such as Player Name and have the game swap them for the appropriate variable value at runtime. This can also be used for bespoke multi-language implementations.

Object

Stores a list of Game Objects; has a companion 'Lister' allowing the designer to add the Lister to any game object such that at run time the game object will be registered to the list and on destruction removed from it.

String

A simple string list, uses StringReference e.g. can be populated with string constants or variables.

Transform

Stores a list of Transform; has a companion 'Lister' allowing the designer to add the Lister to any game object such that at run time the game object's transform will be registered to the list and on destruction removed from it.

Library Variables

Systems Core includes one Library Variable, a library variable is effectively a list variable but has specialized relationships with other component behaviours to fill some specific role. The Data Library and its Data Library File Manager as described in the Serializable Keyed Data Library section are expressed as scriptable objects via the Data Library together they provide a binary serializable save file system that can leverage Data Variables as described above.

Toolbox

Included is a set of 'tools' developed by Heathen Engineering which leverage the Systems Core to perform some common game functions such as moving, rotating, spawning, etc. The Toolbox folder is where you will find the Console Logger, Listers, Value Setters, Chronos and other components noted earlier in the document.

Chronos

Chronos very simply sets the current delta time for scaled and unscaled and fixed and non-fixed times. The use of this component becomes apparent when you have many time sensitive processes, as Time.deltaTime and similar do have some overhead. This component is storing the values on Update and should be set to execute before any other script. The result is that Scriptable Variables of type Float will be populated and keep up to date with the current frame time on a single pass. These variables can then be used by other components without causing additional reads of Time.deltaTime or similar. This is like caching delta time before some iterative process e.g. float time = Time.deltaTime; and should have the same considerations.

Damage Handler

The damage handler system consists of an interface and a base implementation of it as a mono behaviour. The concept is that game elements that can receive damage can implement this interface or include a component of the base implementation Damage Handler enabling game elements that cause damage to easily issue 'damage reports'. A 'damage report' is simply the structured data representing damage in your game and can be as simple as a float value or as complex as a unique class object with internal references of its own.

Examples using the IDamageHandler interface are available in the Spawner Example.

Use case

Say we damager 'bullet' and a few damageable types such as 'player', 'enemy' and 'wall'. A typical approach is to test for each type on collision from the bullet or from the damageable trigger on collision to see if it's a bullet. This works okay in simple cases but requires additional processing to then apply any kind of damage, it also requires the bullet, player, enemy and wall to all be 'aware' of each other breaking the concept of modularity. With the damage handler system elements (bullet, player, etc.) only need be aware of 'damage report' which as noted can be as simple as a primitive float value or as complex as your game requires. Now when a bullet collides with a wall it can simply call Apply Damage and pass in its damage report this will route the damage report to the element which is responsible for applying any change the damage might have caused.

This system is particularly useful when you have various damage types that may or may not be applicable to all potential damagables and when you have many different types of things that should handle damaged.

Data Library Manager

This component is designed to facilitate asynchronous save and load of game data via Scriptable Variables, screen shots are shown in the Scriptable Manifest section of Keyed Data Library. The component takes a reference to a Data Library Variable which houses references to all Scriptable Variables you wish to save and load as well as a to a string path that should be used for reading and write. The component includes convenience functions enabling a basic save and load system with zero code required and is demonstrated in the General Examples scene under the Core Examples.

Instance Renderer

The instance renderer system consists of a generic interface and a default implementation of it for rendering GPU instanced objects. This technique is highly performant and is capable of rendering 1023 objects of the same mesh and material in a single batch / call regardless of the position, rotation or scale. This technique requires that the material used has Instanced Render enabled and it does not support skinned meshes.

The value of this approach is most apparent with projectiles, swarm mobs and similar non-skinned, high volume (many of them), rendered, moving instances. Bullets are a common use case, in particular in 3D bullet hells where traditional spawn pools break down after a few hundred instances primarily due to enable / disable of game objects which has a relatively high-performance cost. With Instance Renderer thousands and 10s of thousands of objects can be rendered and manipulated (moved, rotated, etc.) including collision detection with very little or no frame rate impact.

This has been an available feature of Unity for some time though it requires a good bit of handling on the part of the developer. The Instance Renderer interface and components aim to make this quick and easy to work with even if you have no coding experience. The basic implementation name 'Instance Renderer' is a ready to use game object component that takes a reference to a mesh and material and provides Spawn functions for easy use. All that needs to be provided is a Matrix Transform Data class to describe how the renderer should get transform data for each instance in order to render it in the correct transform.

The Projectile Spawner is an example use case of the Instance Renderer which describes a Matrix Transform Data structure that includes damage, expiration and speed to facilitate simulation of a projectile. The Spawner Example scene demonstrates its use.

Interfaces of Note

`IInstanceRenderer<T>`

Note that T must implement `IMatrixTransformData` and must support a default constructor.

`IInstanceRenderer<T>` defines the following members

`T Spawn();`

This should create a new entry of T and store it in a collection for rendering, returning a reference to it for manipulation by the caller.

`Void Spawn(T data);`

This should take an instance of T and add it to the collection for rendering ... e.g. for use when the caller built T and simply needs to register it for rendering.

`Void Remove(T data);`

This should remove the instance of T from the collection such that it wont be rendered any more.

`Void Render();`

This should handle batching of T elements in the collection and passing them to `Graphics.RenderInstance` for rendering. Note the Instance Renderer basic implementation provided offers a good option for managing the matrix transform data collection and batching them into properly sized groups for rendering. This can be used as an example or a base class to derive from as is done in the Projectile Spawner with its internal `ProjectileInstanceRenderer` class.

IMatrixTransformData

This is for use as the T of the `IInstanceRenderer`; the only technical requirement is that it define a `Matrix4xx4 TRS { get; }` e.g. such that the Instance Renderer can get the Transform Rotation Scale Matrix for the given instance.

A basic `MatrixTransformData` implementation is provided that will define the position, rotation and scale for populating the `Matrix4x4`. This can be used as a base or simply as an example for your own implementations. It is the base of the `ProjectileData` used by `ProjectileSpawner`.