Table of contents

Introduction

Fast Unity Creation Kit is a framework designed to make creating games in Unity faster and easier especially for beginners and solo developers. It is designed to be easy to use and understand, and to be as flexible as possible. It is designed to be used with modern Unity versions, however most of the features should work with older versions as well.

Concepts

Interfacing

Most of the features in Fast Unity Creation Kit are accessed through C# interfaces. This is done to make it easier to use the features in your own code, and to make it easier to understand how the features work.

Interfaces are a way to implement multiple inheritance in C#, and are used to define a logic class that can be implemented by other classes. With this solution Unity Component layer and logic layer can be separated, making it easier to understand and maintain the code.

Components

Some of the features in Fast Unity Creation Kit are implemented as Unity Components, so it can be quickly added to GameObjects in the scene. These components are designed to be plug-and-play with minimal configuration required.

Scriptable Objects

Fast Unity Creation Kit tries to avoid using Scriptable Objects as much as possible, as they can lead to a lot of performance issues and are difficult to debug. However, some features are implemented as Scriptable Objects - especially those that need to be shared between multiple GameObjects like configuration data.

Systems

Introduction

There are many systems in Fast Unity Creation Kit that are designed to make creating games in Unity faster and easier. Most of them are based on other games or game development frameworks and used widely across multiple game genres.

This introduction contains a brief overview of the systems in Fast Unity Creation Kit. For more detailed information, please refer to the individual system documentation in this document.

Systems

- Core (Core) - The core system of Fast Unity Creation Kit. It contains the basic functionality that is used by other systems.
- - The building system is used to manage the building in the game. It is used to create and manage the building of structures, houses, etc.
- - The combat system is used to manage the combat in the game. It is used to create and manage the combat between the player and enemies.
- - The entities system is used to manage the entities in the game. It is used to create and manage the entities in the game - players, enemies, NPCs, in-world destructible objects, etc.
- - The farming system is used to manage the farming in the game. It is used to create and manage the farming of crops, animals, etc.
- - The dialogue system is used to manage the dialogue in the game. It is used to create conversations between characters in the game.
- - The economy system is used to manage the economy in the game. It is used to create and manage the economy of the game - gold, but also health is considered as a local currency of an entity.
- - The inventory system is used to manage the inventory of the player. It is used to store items, weapons, armor, etc.

- - The quest system is used to manage the quests in the game. It is used to create and manage quests for the player to complete.
- - The status system is used to manage the status of objects in the game. It is used mostly to provide status effects like poison, slow, etc.
- - The time system is used to manage the time in the game. It is used to call events when the time changes, like day-night cycle, turn-based combat, etc.
- - The UI system is used to manage the user interface in the game. It is used to create and manage the user interface of the game - health bars, inventory, etc.

Core

Core is a package of commonly used classes and interfaces that are used by other systems in Fast Unity Creation Kit.

The subsystems of the Core system are:

- Events (Events) - The events system is used to manage the events in the game. It is used to create and manage the events that can be triggered by the player or the game itself. This subsystem is designed to separate other systems from each other, so they can be easily replaced or extended.
- Identification (Identification) - The identification system is used to manage the identification of objects in the game. It is used to create and manage the identification of objects in the game - players, enemies, NPCs, in-world destructible objects, etc.
- Initialization (Initialization) - The initialization system is used to manage the initialization of objects in the game.
- - The numerics system is used to manage the numerical values in the game. It is used as an overlay to C# built-in types to provide easier and more flexible way to work with numbers via interfaces like `INumber`.
- - The priority system is used to manage the priority of objects in the game. It is used to create prioritized lists of objects, so they are automatically sorted.
- - The singleton system is used to manage the singleton objects in the game. It is used to create singleton objects that can be accessed from anywhere in the game.
- - The utility system is used to manage object properties. It can handle object with name, description, icons, prefabs, etc.
- - The values system is used to manage the values of objects in the game. It is used to create dynamically modifiable values - for example maximum health that can be affected by status effects or equipment.

Events

Events is a package of classes and interfaces that are used to manage the events in the game. It is used to create and manage the events that can be triggered by the player or the game itself. This subsystem is designed to separate other systems from each other, so they can be easily replaced or extended.

Concepts

Events are a way to communicate between different parts of the game. They are used to trigger actions in response to certain conditions, like player input, game state changes, etc. Events are a way to decouple different parts of the game, so they can be easily replaced or extended without affecting other parts of the game. Thanks to this system this project is completely modular and modules are independent of each other.

The core of Events system is the `EventChannel<TSelf>` class. It is a base class for all event channels in the game. It contains methods to register and unregister event listeners, and to trigger the event. It also contains separate generic type to add ability to pass data with the event.

```
public abstract class EventChannel :
EventChannelBase<EventChannelCallback>

public abstract class EventChannel<TChannelData> :
EventChannelBase<EventChannelCallback<TChannelData>>
        where TChannelData : IEventChannelData
```

This represents a local event channel that can be attached to an object.

There are also global event channels that are used to broadcast events to all objects in the game. They are implemented as singletons, so they can be accessed from anywhere in the game.

```
public abstract class GlobalEventChannel<TSelf> : EventChannel
        where TSelf : GlobalEventChannel<TSelf>, new()

public abstract class GlobalEventChannel<TSelf, TChannelData> :
EventChannel<TChannelData>
```

```
        where TSelf : GlobalEventChannel<TSelf, TChannelData>, new()
        where TChannelData : IEventChannelData
```

> In most cases you will use GlobalEventChannel, as it is more flexible and easier to use. Local events are a possibility if you need to trigger events only on a specific object or link events to a specific object when that event is used to communicate between components on the same object.

Creating Events

To create an event you need to extend `GlobalEventChannel<TSelf>` or `EventChannel<TSelf>` via a custom class. The `TSelf` type parameter should be the type of the class you are extending as.

Event Channels also support secondary type parameter `TData` which is the type of the data that will be passed with the event, that type needs to extend `IEventChannelData` interface.

```
public class MyGlobalEventChannel :
GlobalEventChannel<MyGlobalEventChannel>
{
    // You can also override Trigger method to add custom logic
    public override void Trigger()
    {
        Debug.Log("Hello world!");
    }
}
```

> If you don't call base Trigger method in your custom Trigger method, the event will not be triggered on listeners and thus the created event can be considered a virtual event (event without listeners).

You can also create an event that passes data with it.

```
public readonly struct MyEventData : IEventChannelData
{
    public readonly string message;

    public MyEventData(string message)
    {
        this.message = message;
    }
}

public class MyGlobalEventChannelWithData :
GlobalEventChannel<MyGlobalEventChannelWithData, MyEventData>
{

}
```

Listening to Events

To listen to an event you need to access the event channel and subscribe to it.

```
public class MyListener : MonoBehaviour
{
    private void Start()
    {
        MyGlobalEventChannel.RegisterEventListener(ExampleCallback);
    }

    private void ExampleCallback()
    {
        Debug.Log("Event triggered!");
    }
}
```

If you want to pass data with the event, you need to subscribe to the event with a callback that accepts the data as a parameter.

```
public class MyListener : MonoBehaviour
{
    private void Start()
    {

MyGlobalEventChannelWithData.RegisterEventListener(ExampleCallback);
    }

    private void ExampleCallback(MyEventData data)
    {
        Debug.Log("Event triggered with data: " + data.Message);
    }
}
```

Triggering Events

To trigger an event you need to access the event channel and call the `Trigger` method.

```
public class MyTrigger : MonoBehaviour
{
    private void Start()
    {
        MyGlobalEventChannel.Trigger();
    }
}
```

> If you want to pass data with the event, you need to call the `Trigger` method with the data as a parameter.

```
public class MyTrigger : MonoBehaviour
{
    private void Start()
    {
        MyGlobalEventChannelWithData.Trigger(new MyEventData("Hello
world!"));
```

```
    }
}
```

Handling local events

Local events are events that are only triggered on the object that the event channel is attached to. You do everything almost the same as with global events, but you need to use `EventChannel<TSelf>` instead of `GlobalEventChannel<TSelf>` and you need to call methods on the instance of the event channel.

Identification

Identification is a package that contains basic identifier and identifiable object framework. Identifiers are `readonly structs` to ensure that they are immutable and can be used as keys.

Core interfaces

IIdentifier

Represents an identifier - a unique or non-unique identifier of an object.

IUniqueIdentifier

Represents a unique identifier - an identifier that is unique within a context.

INumberIdentifier

Represents a number identifier - an identifier that is a number as a value. `TNumber` is a type of the number that inherits from `INumber` interface from `Core.Numerics` package.

IIdentifiable

Represents an object that can be identified by an identifier. `TIdentifier` is a type of the identifier which inherits from `IIdentifier` interface.

IUniqueIdentifiable

Represents an object that can be identified by a unique identifier. `TIdentifier` is a type of the identifier which inherits from `IIdentifier` interface. This interface does not require `TIdentifier` to inherit from `IUniqueIdentifier` interface as it is in programmer's responsibility to ensure that the identifier is unique within a given context.

> ⚠ While using IUniqueIdentifiable interface, it is programmer's responsibility to ensure that the identifier is unique within a given context.

Implemented identifiers

Snowflake128

A 128-bit identifier that is generated using the Snowflake algorithm. It is guaranteed to be unique within a given context. First 64 bits are a timestamp, next 32 bits are identifier data (e.g. worker ID), next 16 bits are another pack of identifier data (e.g. process ID), next 8 bits are reserved and remaining 8 bits indicates if the identifier is null.

Numeric identifiers

- ID8 - A simple 8-bit number used as an identifier.
- ID16 - A simple 16-bit number used as an identifier.
- ID32 - A simple 32-bit number used as an identifier.
- ID64 - A simple 64-bit number used as an identifier.

Creating custom identifiers

You can also create custom identifiers by implementing `IIdentifier` interface. An example of a custom identifier is shown below:

```
public readonly struct CustomIdentifier : IIdentifier
{
    // It is not recommended to use managed types within identifiers.
    // However this is just an example for demonstration purposes.
    public readonly string value;

    public CustomIdentifier(string value)
    {
        this.value = value;
    }
}
```

It is also recommended to implement IEquatable{TSelf} interface for custom identifiers.

⚠ It is not recommended to use managed types within identifiers as it may lead to performance issues.

Initialization

Initialization is a package that contains the `IInitializable` interface responsible for initializing objects in the game. Any initializable object should implement this interface and provide custom logic calls to request initialization when needed.

Implementation

You need to implement the `IInitializable` interface in your class to make it initializable.

Example for regular C# class:

```
public class MyInitializable : IInitializable
{
    bool IInitializable.InternalInitializationStatusStorage { get; set;
}

    void IInitializable.OnInitialize()
    {
        // Initialization logic
        Debug.Log("MyInitializable initialized");
    }

    public MyInitializable()
    {
        // Request initialization
        this.Initialize();
    }
}
```

Example for Unity MonoBehaviour:

```
public class MyInitializable : MonoBehaviour, IInitializable
{
    bool IInitializable.InternalInitializationStatusStorage { get; set;
}

    void IInitializable.OnInitialize()
```

```
    {
        // Initialization logic
        Debug.Log("MyInitializable initialized");
    }

    private void Awake()
    {
        // Request initialization
        this.Initialize();
    }
}
```

Property InternalInitializationStatusStorage on the interface is used to store the initialization status of the object. This is an internal property and should not be accessed directly, for that purposes you should use `IsInitialized` property.

Initialization with multiple entry paths

Initialization can be requested multiple times, but it will be executed only once. This is useful when you need to initialize an object from multiple entry points, and you don't know which one will be called first.

Also, if you want to avoid using this.Initialize() method you can use `EnsureInitialized()` method that will initialize the object if it's not initialized yet. This method is clearer and more readable in some cases - when you want to ensure that the object is initialized, but you are not sure if it was already initialized.

Using `EnsureInitialized()` method requires casing the object to `IInitializable` interface.