Message System User Guide v2.1.0

Introduction

This guide provides a comprehensive overview of the Message System, detailing its features and usage. The system supports various messaging channels, allowing for robust communication within applications and games.

MessageChannels Enum

The `MessageChannels` enum defines various channels for message communication.

```
public enum MessageChannels
{
    System,
    Logging,
    Debug,
    Error,
    Warning,
    Info,
    Gameplay,
    Level,
    Quest,
    Achievement,
    Combat,
    AI,
    Physics,
    Animation,
    Player,
    PlayerStats,
    PlayerInventory,
    PlayerSkills,
    PlayerActions,
    PlayerHealth,
    PlayerMovement,
    Enemy,
    EnemyStats,
    EnemyAI,
    EnemyHealth,
    EnemyActions,
    Items,
    ItemPickup,
    ItemDrop,
    ItemUse,
    ItemCrafting,
    InventoryManagement,
    UI,
    UINotifications,
    UIMenus,
    UIButtons,
    UIDialogs,
    UILoadingScreen,
    UIPopup,
```

```
    Network,
    NetworkConnect,
    NetworkDisconnect,
    NetworkError,
    NetworkData,
    Audio,
    Music,
    SoundEffects,
    Voice,
    Input,
    KeyPress,
    MouseClick,
    Touch,
    Time,
    DayNightCycle,
    Timer,
    Weather,
    Environment,
    NPC,
    Scripting,
    SaveLoad,
    Cutscene,
    Tutorial,
    Economy,
    Trade,
    Dialogue,
    Camera,
    UI_HUD,
    UI_Inventory,
    UI_QuestLog,
    UI_SkillTree,
    Social,
    Chat,
    Mail,
    FriendRequest,
    Clan,
    Group
}
```

Core Components

IMessageChannel Interface

The `IMessageChannel` interface is the foundation for creating message channels. It allows for the definition of custom channels beyond the standard ones provided by the `MessageChannels` enum:

```
public interface IMessageChannel
{
    string Name { get; }
}
```

CustomChannel

A `CustomChannel` is used to define a custom message channel that may be specific to a particular use case or part of an application:

```
public class CustomChannel : IMessageChannel
{
    public string Name { get; private set; }

    public CustomChannel(string name)
    {
        Name = name;
    }
}
```

DefaultMessageChannel

The `DefaultMessageChannel` simplifies the usage of predefined channels from the `MessageChannels` enum, encapsulating them into an easy-to-use object:

```
public class DefaultMessageChannel : IMessageChannel
{
    public MessageChannels Channels { get; private set; }

    public DefaultMessageChannel(MessageChannels channels)
    {
        Channels = channels;
    }

    public string Name => Channels.ToString();
}
```

Message Envelope

The `IMessageEnvelope` interface and its implementation `MessageEnvelope` are used to wrap messages.

```
public interface IMessageEnvelope
{
    Type MessageType { get; }
}

public interface IMessageEnvelope<out T> : IMessageEnvelope
{
    T? Message { get; }
}

public class MessageEnvelope<T> : IMessageEnvelope, IMessageEnvelope<T?>
{
    public T? Message { get; private set; }
    public Type MessageType { get; private set; } = typeof(T);

    public MessageEnvelope(T? message)
```

```
    {
        Message = message;
    }
}
```

Message Manager

The `MessageManager` class handles the registration, sending, and processing of messages.

Registering Handlers

Register a handler for a specific channel with the MessageChannels enum.

```
MessageSystem.MessageManager.RegisterForChannel<GameplayMessage>
(MessageChannels.Gameplay, GameplayMessageHandler);
```

Register a handler for multiple channels with the MessageChannels enum.

```
MessageSystem.MessageManager.RegisterForChannel<StringMessage>
(StringMessageHandler, 0, MessageChannels.Gameplay, MessageChannels.System,
MessageChannels.UI);
```

Register a handler using custom channels.

```
protected const string AwesomeTime = "AwesomeTime";
protected const string TacoString = "Taco";
protected IMessageChannel awesomeCustomChannel = new CustomChannel(AwesomeTime);
protected IMessageChannel tacoCustomChannel = new CustomChannel(TacoString);

MessageSystem.MessageManager.RegisterForChannel<GameplayMessage>
(tacoCustomChannel, TacoCustomChannel);
MessageSystem.MessageManager.RegisterForChannel<SystemMessage>
(AwesomeCustomChannelHandler, 0, awesomeCustomChannel, gamePlayChannel);
```

Unregistering Handlers

Unregister a handler for a specific channel with the MessageChannels enum.

```
MessageSystem.MessageManager.UnregisterForChannel<GameplayMessage>
(MessageChannels.Gameplay, GameplayMessageHandler);
```

Unregister a handler for multiple channels with the MessageChannels enum

```
MessageSystem.MessageManager.UnregisterForChannel<GameplayMessage>
(StringMessageHandler, MessageChannels.System, MessageChannels.Gameplay,
MessageChannels.UI);
```

Unregister a handler using custom channels.

```
MessageSystem.MessageManager.UnregisterForChannel<SystemMessage>
(AwesomeCustomChannelHandler, awesomeCustomChannel, gamePlayChannel);
```

Sending Messages

Send a message immediately.

```
MessageSystem.MessageManager.SendImmediate(MessageChannels.Gameplay, new
GameplayMessage("Hello"));
```

Send a message to be processed later.

```
MessageSystem.MessageManager.Send(MessageChannels.Gameplay, new
GameplayMessage("Queued Message"));
```

Send a message immediately to multiple channels.

```
MessageSystem.MessageManager.SendImmediate(new StringMessage("Hello"),
MessageChannels.Gameplay, MessageChannels.System, MessageChannels.UI);
```

Broadcast a message to all channels immediately.

```
MessageSystem.MessageManager.BroadcastImmediate(new GameplayMessage("Broadcast
Message"));
```

Broadcast a message to all channels to be processed later.

```
MessageSystem.MessageManager.Broadcast(new GameplayMessage("Broadcast Message"));
```

Processing Messages

Process queued messages.

```
MessageSystem.MessageManager.ProcessMessages();
```

Process queued messages asynchronously.

```
await MessageSystem.MessageManager.ProcessMessagesAsync();
```

Sending Messages Asynchronously

Send a message immediately asynchronously.

```
await MessageSystem.MessageManager.SendImmediateAsync(MessageChannels.Gameplay,
new GameplayMessage("Hello Async"));
```

Send a message to be processed later asynchronously.

```
await MessageSystem.MessageManager.SendAsync(MessageChannels.Gameplay, new
GameplayMessage("Queued Message Async"));
```

Broadcast a message to all channels immediately asynchronously.

```
await MessageSystem.MessageManager.BroadcastImmediateAsync(new
GameplayMessage("Broadcast Message Immediate Async"));
```

Broadcast a message to all channels to be processed later asynchronously.

```
await MessageSystem.MessageManager.BroadcastAsync(new GameplayMessage("Broadcast
Message Async"));
```

Serialization

Serialize Message to JSON

```
var serializedMessage = MessageSystem.MessageManager.SerializeMessageToJson(new
GameplayMessage("Serialize Test"));
```

Deserialize Message from JSON

```
var deserializedMessage =
MessageSystem.MessageManager.DeserializeMessageFromJson<GameplayMessage>
(serializedMessage);
```

Examples

Sending a Complex Message

Here's an example of sending a complex message with multiple properties.

```
public struct ItemMessage
{
    public IItem Item { get; }
    public IActor? Source { get; }
    public IActor? Target { get; }

    public ItemMessage(IItem item, IActor? source, IActor? target)
    {
        Item = item;
        Source = source;
        Target = target;
    }
}

// Sending an ItemMessage
var item = new Item();
var sourceActor = new Actor();
var targetActor = new Actor();

MessageSystem.MessageManager.SendImmediate(MessageChannels.Items, new
ItemMessage(item, sourceActor, targetActor));
```

Message Handlers

Example message handlers.

```
private void TacoCustomChannel(IMessageEnvelope message)
{
    if (!message.Message<GameplayMessage>().HasValue) return;
    var data = message.Message<GameplayMessage>().GetValueOrDefault();
    testObject.StringProp = data.Message;
}

private void AwesomeCustomChannelHandler(IMessageEnvelope message)
{
    if (!message.Message<SystemMessage>().HasValue) return;
    var data = message.Message<SystemMessage>().GetValueOrDefault();
    testObject.intField = data.TestObject.intField;
    testObject.stringField = data.TestObject.stringField;
}

private void StringMessageHandler(IMessageEnvelope message)
{
    if(!message.Message<StringMessage>().HasValue) return;
    var data = message.Message<StringMessage>().GetValueOrDefault();
    testObject.StringProp = data.Message;
}

private void GamePlayMessageHandler2(IMessageEnvelope message)
{
    if(!message.Message<GameplayMessage>().HasValue) return;
    var data = message.Message<GameplayMessage>().GetValueOrDefault();
    testObject.StringProp = data.Message;
}

private void SystemMessageHandler(IMessageEnvelope message)
{
    if(!message.Message<SystemMessage>().HasValue) return;
    var data = message.Message<SystemMessage>().GetValueOrDefault();
    testObject.intField = data.TestObject.intField;
    testObject.stringField = data.TestObject.stringField;
}

private void AnotherSystemHandler(IMessageEnvelope message)
{
    if (!message.Message<SystemMessage>().HasValue) return;
    var data = message.Message<SystemMessage>().GetValueOrDefault();
    testObject.intField = data.TestObject.intField;
    testObject.stringField = data.TestObject.stringField;
}

private void GameplayMessageHandler(IMessageEnvelope message)
{
    if(!message.Message<GameplayMessage>().HasValue) return;
    var data = message.Message<GameplayMessage>().GetValueOrDefault();
    testObject.stringField = data.Message;
}
```

Conclusion

This guide provides an overview of the Message System, detailing its core components, usage, and examples. For more information and advanced usage, refer to the source code and tests.