GameSense™ Client for Unity Engine

GameSense™ Client for Unity Engine is an interface simplifying the process of integrating support for GameSense™ in your project. It minimizes the amount of work required to communicate with the GameSense™ server, exposes a simple to use API and provides GUI editor tools.

Visit SteelSeries Techblog (http://techblog.steelseries.com/2015/06/29/introducing-gamesense.html) to learn more about GameSense™.

For more information about GameSense™ SDK, visit SteelSeries Developer Portal (http://developer.steelseries.com/gamesense) .

Installation

Simply place the contents of this repository somewhere in your project's Assets directory, preferably inside `Assets\Scripts\GameSense`. The client is also available on the Unity Asset Store (https://www.assetstore.unity3d.com/#!/content/58312) .

The client relies on the presence of the SteelSeries Engine 3 (aka the GameSense™ server). You can download it from here (https://steelseries.com/engine) .

At this time, only Windows and OSX platforms are supported.

Usage

NOTE: It is strongly advised you familiarize yourself with how GameSense™ works first! Please make sure to visit the SDK pages (link at the top) to learn the high-level concepts.

There are two ways to use the client. One way involves the use of GUI editors, working with assets and a minimum amount of scripting. The other way is to use the client entirely through script. Choose your poison. In any case, the process boils down to the following steps:

- registering your game
- registering (or binding) events (with handlers)
- sending update events

Using GUI Editors

You can enable this method by dragging `GameSenseManager.prefab` into the scene hierarchy

and selecting from there. In the inspector window, you should be able to see several properties allowing you to configure GameSense™.

Registering a Game

The first three input fields are responsible for registering your game with the server. They are defined as follows:

- **Game Name** - the unique string identifying your game. Legal character set for this field is limited to uppercase A-Z, the digits 0-9, hyphen, and underscore.
- **Game Display Name** - this name will appear under GameSense tab in SteelSeries Engine 3. You can use any characters here.
- **Icon Color** - the color of the icon that will be displayed along with the game name in SteelSeries Engine 3.

Defining Events

Among the properties of `GameSenseManager` there is a field `Events`. It represents an array of events that will be registered or bound with the server. Please set the number of events you want to define. For each event, you will need to specify the following:

- **Event Name** - the unique string identifying the event. The same rules apply as for the **Game Name** field: only uppercase A-Z, 0-9, hyphen, and underscore are allowed.
- **Min/Max Value** - these two fields define an integer domain for the event. Typically, you will specify 0-1 for events that toggle state or 0-100 for events like health status.
- **Icon Id** - you can select the default (no icon) or one of the available icon identifiers. The corresponding icon will be displayed along with the event in SteelSeries Engine 3.
- **Handlers** - this field is optional. You can omit it if you want to defer the handler specification. This way, the user will be responsible for defining the behavior for your events. Otherwise, please refer to the following section.

Defining Handlers

You can choose to provide a default behavior for each event you define. There is a number of inspector tools at your disposal to accomplish the task. Again, please brush up on the basics of handler specification (https://github.com/SteelSeries/gamesense-sdk/blob/master/doc/api/writing-handlers-in-json.md) . This section assumes you understand the structure of handlers. In order to keep things clean and organized, please create the following directory hierarchy somewhere in your `Assets` directory:

```
GameSenseData/
├── Handler/
├── DeviceZone/
├── ColorEffect/
├── TactilePattern/
└── Rate/
```

These directories will store our assets that will be building blocks of our handlers. The

remaining part of this section will demonstrate how to create a handler for Apex M800 keyboard.

Navigate to `Assets/GameSenseData/Handler/` in Unity's asset explorer, right-click on background of the asset viewer and highlight `Create`. You should be able to see `GameSense` context menu. From there, select `Handlers/Color Handler`. This will create an asset for our handler. Please take the opportunity to give the asset a meaningful name since Unity does not allow you to change it later. Let us name it ***M800FunctionKeysHealthHandler***. Select this asset and notice the properties in the inspector window. Each illumination handler will have the following fields:

- **Device Zone** - a device-zone pair identifying a particular device or device family and an illumination zone on this device.
- **Mode** - describes how the value we send with each event is represented on the zone. **Note**: `Count` and `Percent` modes are meaningful only in the context of `RGBPerkey` device-zone. Refer to handler specification (https://github.com/SteelSeries/gamesense-sdk/blob/master/doc/api/writing-handlers-in-json.md) for more information.
- **Color** - defines a color effect we wish to use for this handler.
- **Rate** - describes flashing frequency and repeat limit

Please set **Mode** to `Percent`. For the remaining properties, we need to create more assets.

Navigate to `Assets/GameSenseData/DeviceZone/` and, using the `Create` context menu, select `GameSense/Device - Zone/Illumination/Generic/RGB Perkey Zone/Function Keys`.

Navigate to `Assets/GameSenseData/ColorEffect/` and, using the `Create` context menu, select `GameSense/Color Effects/Gradient`. Name the new asset ***RedToGreenGradient***. Select it and, using the inspector, set the top color to red and the bottom color to green.

Navigate to `Assets/GameSenseData/Rate/` and, using the `Create` context menu, select `GameSense/Rate/Static`. Name the new asset ***Repeat3Times5Hz***. Select the asset and, using the inspector, set **Frequency** to `5` and **Repeat Limit** to `3`.

We now have all the assets we need. Navigate back to `Assets/GameSenseData/Handler/` and select our handler. Using the inspector, assign our new assets to their corresponding fields:

- **Device Zone** - *RGBPerkeyZoneFunctionKeys*
- **Color** - *RedToGreenGradient*
- **Rate** - *Repeat3Times5Hz*

Now, select `GameSenseManager` object from the scene hierarchy. In the inspector, set `Events` array size to `1`. Configure the new event as follows:

- **Event Name** - "HEALTH"
- **Min Value** - `0`
- **Max Value** - `100`
- **Icon Id** - `Health`
- **Handlers** - set size to `1`, and assign ***M800FunctionKeysHealthHandler*** to the new element.

Set appropriate values to **Game Name**, **Game Display Name** and **Icon Color** if you haven't done so yet. The last thing to do is to send status updates from your own scripts. This can be achieved by inserting a one-liner of the following form somewhere in your code:

```
SteelSeries.GameSense.GSClient.Instance.SendEvent("HEALTH", currentHealth);
```

At this point, you can run your project. GameSense client script will carry out all the necessary setup for you. If everything was setup properly, you should see the health status being updated on function keys of your Apex M800. In case you do not own the device, you can configure the handler to update health status on any supported SteelSeries device you might own by assigning appropriate DeviceZone asset to the handler. For instance, on any two-zone mouse, you can use `GameSense/Device - Zone/Illumination/Mouse/Wheel` asset from the context menu, to see health updates on the wheel LED.

Tips

- Remember that you can specify script execution order in your project. Make sure `GSClient.cs` starts before anything that calls `SendEvent()`
- If you use `GameSenseManager.prefab` to register your game and setup events, please refrain from calling `RegisterGame()`, `RegisterEvent()` or `BindEvent()` manually.

Using Scripts

You have an option of carrying out the initial setup procedure entirely from code. Please make sure that no instance of `GameSenseManager.prefab` is present in the scene hierarchy. You can put the setup code in any `MonoBehaviour` script that you use. The following snippet demonstrates how it is typically done:

```
// somewhere at the top of the script
using SteelSeries.GameSense;
using SteelSeries.GameSense.DeviceZone;
.
.
.

// MonoBehaviour Awake() method
void Awake() {

    // GameSense setup

    // register our game
    GSClient.Instance.RegisterGame(
        "DEMO",             // game name
        "Unity Demo",       // game display name
        IconColor.Orange    // icon color
        );

    // function keys zone on perkey device (such as Apex M800)
    RGBPerkeyZoneFunctionKeys functionKeysZone =
```

```
        ScriptableObject.CreateInstance< RGBPerkeyZoneFunctionKeys >();

    // gradient color effect
    ColorGradient effectRedToGreenGradient = ColorGradient.Create(
        new RGB( 255, 0, 0 ), // red
        new RGB( 0, 255, 0 )  // green
        );

    // flash 3 times with the frequency of 5Hz on each update
    RateStatic rateRepeat3Times5Hz = RateStatic.Create( 5, 3 );

    // create our handler for health event
    ColorHandler healthHandler = ColorHandler.Create(
        functionKeysZone,         // device-zone
        IlluminationMode.Percent, // mode
        effectRedToGreenGradient, // color effect
        rateRepeat3Times5Hz       // rate
        );

    // finally, we bind our event with the handler
    GSClient.Instance.BindEvent(
        "HEALTH",                                 // event name
        0,                                        // min value
        100,                                      // max value
        EventIconId.Health,                       // icon id
        new AbstractHandler[] { healthHandler } // array of handlers
        );
}
```

Then, somewhere else in your code, you can send updates with the following:

```
SteelSeries.GameSense.GSClient.Instance.SendEvent("HEALTH", currentHealth);
```

Tips

- Remember that you can specify script execution order in your project. Make sure your setup script runs before anything that calls `SendEvent()`
- If you use the client through script, make sure you do not instantiate `GameSenseManager.prefab` in your scene hierarchy. Also, do not attach any client's scripts to any game objects.

General Tips

- Be aware of where you put your `SendEvent()` calls. If you really insist on putting them somewhere inside MonoBehaviour's `Update()` methods, at least guard them with logic to send event updates only when the variables you use for updates change values. Under no circumstance should you spam updates every frame or send redundant data.

- You generally should consider sending initial values for each event before the game starts running regular updates. The best place to do this is inside MonoBehaviour's `Start()`.
- Remember to bind your handlers for a particular event all at once. Any consecutive call to `BindEvent()` for the same event will override all previously bound handlers.
- The client will output warnings to Unity's console when something goes wrong. In case you wish to disable these warnings, you can define `SS_NWARN` preprocessor directive.
- You can enable lots of debug output by defining `SS_DEBUG` preprocessor directive.
- To statically disable the client, define `SS_GAMESENSE_DISABLED` preprocessor directive. This will effectively prevent the bodies of initialization routine and API functions from getting compiled.
- In the event of a critical failure, such as when the client can't communicate with the GameSense™ server, the client assumes an inactive state, where all the API calls have no effect.
- The event handlers you bind are the defaults for your game. The end user can customize or disable the handler behavior in SteelSeries Engine 3.
- This document by no means is meant to be a comprehensive GameSense™ guide. You should refer to SteelSeries Developer Portal (http://developer.steelseries.com/gamesense) for more information.

License