Using the ThinkGear Native Library in Unity

Features

- Step-by-step guide to integrate brainwave-sensing functionality into your Unity-based game
- A downloadable sample Unity project that demonstrates a simple implementation
- Develop and deploy ThinkGear-enabled applications on both Mac and PC

Introduction

Unity has gained considerable traction for being an incredibly easy-to-use, yet powerful game development tool. NeuroSky's MindKit ships ready out-of-the-box to work with both Mac and PC versions of Unity. Using the ThinkGear software library in the MindKit, a developer using Unity can easily integrate brainwave-sensing functionality into their Unity projects. This application note will walk you through this procedure.

Important: Because utilization of the ThinkGear native library relies on Unity's plugin functionality, you *must* have Unity Pro. Furthermore, Unity restricts plugin functionality to standalone builds of the player, so projects built for the Web Player will not be able to use the ThinkGear library.

A developer can work around these restrictions by utilizing the ThinkGear Socket Protocol (via the ThinkGear Connector) rather than the ThinkGear native library. This method requires the installation of daemon software on the user's computer, but allows more restrictive languages and frameworks to integrate brainwave-reading functionality.

Setting Up

Before dropping the ThinkGear library binaries into your Unity project, make sure that your project is set up to accept plugins. Unity expects all plugins to be placed in a `Plugins` folder in the root level of your project folder, so if the folder isn't already there, create it.

On the MDT CD, there is a `develop` directory that contains the requisite libraries and sample code — we're interested in the content inside `develop/macosx` and `develop/win32`. Copy the following files into the `Plugins` folder in your Unity project:

- `develop/macosx/ThinkGearBundle.bundle` — Mac ThinkGear library
- `develop/win32/thinkgear.dll` — PC ThinkGear library
- `develop/win32/ThinkGear.cs` — C# wrapper script

Unity assumes a consistent naming scheme across Mac and PC versions of a plugin, so some quick renaming is in order:

- Rename `ThinkGearBundle.bundle` to `ThinkGear.bundle`
- Rename `thinkgear.dll` to `ThinkGear.dll`

By now, the **Project** panel in your Unity project should look something like Figure 1.

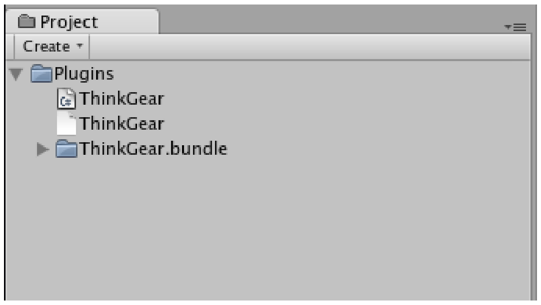

Figure 1: Unity project panel

Using ThinkGear

The `ThinkGear.cs` wrapper essentially imports all of the ThinkGear functions exposed in the library as static `ThinkGear` class methods. By virtue of being in the `Plugins` directory, Unity makes this class available at runtime so you can invoke the static methods directly without having to drop it into a `GameObject`.

Both the ThinkGear API documentation (included on the MDT CD) and the `ThinkGear.cs` file contain descriptions of the various functions available to you, so you should browse through those to get a feel for the API. In general, though, the control flow for the function calls is as shown in Figure 2.

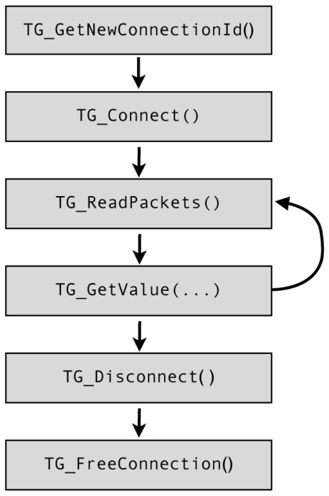

Figure 2: ThinkGear control flow

Connecting

Establishing a connection via the ThinkGear library involves the first two blocks of the control flow diagram shown in Figure 3.

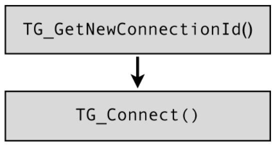

Figure 3: Connection flow

For the most part, the connection code is fairly straightforward:

```
// generate a handle to a ThinkGear connection
int handleID = ThinkGear.TG_GetNewConnectionId();

// perform the actual connection
int connectStatus = ThinkGear.TG_Connect(handleID,
                                         "/dev/tty.MindSet",
                                         ThinkGear.BAUD_9600,
                                         ThinkGear.STREAM_PAKCETS);
```

However, we also need to make sure that the data coming back from the headset is valid. This is done by idling for a period of time, until we know a functional MindSet would **definitely** have returned valid data:

```
if(connectStatus >= 0){
    // sleep for 1.5 seconds
    yield return new WaitForSeconds(1.5f);

    // read all of the data in the buffer
    int packetCount = ThinkGear.TG_ReadPackets(handleID, -1);

    // we've received some data, thus we've connected to a valid headset
    if(packetCount > 0){
       // implement some behavior here
    }
    // no valid headset data received, so close the connection
    else {
       ...

       ThinkGear.TG_FreeConnection(handleID);
    }
}
else {
    // the connection attempt was unsuccessful
    ThinkGear.TG_FreeConnection(handleID);
}
```

Note: The MindSet, in its standard configuration, transmits brainwave data every second. Thus, an idle period of 1.5s is a sufficient amount of time to wait before checking on the data received.

Reading Data

Reading data involves the third and fourth blocks in the control flow diagram shown in Figure 4.

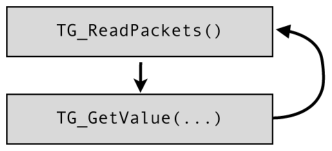

Figure 4: Data reading flow

It involves a continuously running loop, consisting of:

- A single `TG_ReadPackets()` call, which parses packet data from the buffer and then validates it
- Multiple `TG_GetDataValue()` calls, which returns interpreted data from these packets

In Unity, this is best achieved by using the `InvokeRepeating()` method, which continuously calls a named method at specific intervals. Since the headset broadcasts data at 1Hz, using a callback interval of 1s is appropriate.

In the code sample in the Connecting section, there was an `if` statement that checked whether the headset connection was successful. It makes sense for the `InvokeRepeating()` to be placed in this statement:

```
// we've received some data, thus we've connected to a valid headset
if(packetCount > 0){
    InvokeRepeating("UpdateHeadsetData", 0.0f, 1.0f);
}
```

We'll also need to define the callback method:

```
// Repeating callback method to retrieve data from the headset
void UpdateHeadsetData(){
    int packetCount = ThinkGear.TG_ReadPackets(handleID, -1);

    if(packetCount > 0){
        float attention = ThinkGear.TG_GetDataValue(handleID,
                                            ThinkGear.DATA_ATTENTION);

        float meditation = ThinkGear.TG_GetDataValue(handleID,
                                            ThinkGear.DATA_MEDITATION);

        ...
    }
}
```

From here, it is up to your application to do something meaningful with the brainwave data received from the headset.

Disconnecting

Disconnecting from the headset involves the last two blocks in the control flow diagram shown in Figure 5.

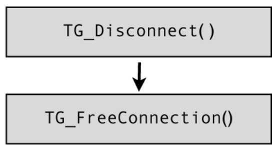

Figure 5: Disconnection flow

It involves a simple:

```
ThinkGear.TG_Disconnect(handleID);
ThinkGear.TG_FreeConnection(handleID);
```

In general, though, one can simply call:

```
ThinkGear.TG_FreeConnection(handleID);
```

The `TG_FreeConnection()` function implicitly calls `TG_Disconnect()`. `TG_Disconnect()` is only really useful if you want to retain the assigned ThinkGear handle for reuse.

Sample Project

The sample Unity project demonstrates a simple application that lets a user connect to the headset and view the data being transmitted by it. It involves a simple GUI for handling user input and output, and a controller that handles data flow between the GUI and the ThinkGear.

The controller — `ThinkGearController` — was designed around an event-driven mechanism, so it sends and receives messages to query or change the state of the headset. The implementation largely follows the code samples above, save for a few trivial differences. For the most part, you can drop this controller class into your Unity project and set up your `MonoBehaviour` instances to trigger and listen to its events.

Note: Keep in mind that there is a fair amount of overhead to `SendMessage()` calls. `ThinkGearController` uses `SendMessage()` to send events to all `GameObject` instances in the scene, so this may impose a fairly hefty performance hit if you have a large project.

If performance is a huge concern, a polling-based mechanism (where the `ThinkGearController` maintains state, and another `MonoBehaviour` continuously polls the `ThinkGearController` instance for state changes) should prove to be far more efficient and scalable.

Alternatively, you can modify the `TriggerEvent()` method to restrict the scope of its message sending, by invoking `SendMessage()` or `BroadcastMessage()` *only* on the local GO instance (i.e. `gameObject.SendMessage("SomeEvent")`).

The events that `ThinkGearController` utilizes or recognizes are as follows:

Received and Handled Events

Event name	Parameters	Description
`OnHeadsetConnectionRequest()`	None	Initiate a headset connection request
`OnHeadsetDisconnectionRequest()`	None	Initiate a headset disconnection request

Broadcasted Events

Event name	Parameters	Description
`OnHeadsetConnected()`	None	Broadcast when the headset has successfully connected
`OnHeadsetDisconnected()`	None	Broadcast when the headset has successfully disconnected
`OnHeadsetDataReceived()`	`Hashtable data`	Broadcast when data is received from the headset
`OnHeadsetConnectionError()`	None	Broadcast when a connection attempt was unsuccessful

Other Niceties

The sample Unity project implements the bare essentials to enable MindSet connectivity. For a customer-facing application, considerations should be made to improve the user experience.

Auto-connect on Startup

To save the user the task of having to explicitly connect to the headset, the application can, on startup, automatically connect to the last seen headset. In Unity, this is simply a matter of storing and loading the last-used serial port via a `PlayerPrefs` parameter, and then connecting to the headset in the `Start()` method of a `MonoBehaviour`.

Port scanning

The application can implement logic to perform a port scan of available serial ports, saving the user from having to type one in. This is useful in Windows, where serial ports are consistently named (e.g. `COM1` to `COMxx`), though not so much in OS X, where serial ports are arbitrarily named.

Important: In Windows, COM port names should have a `\\.\` prepended to them. It is a required prefix for addressing serial ports above `COM9`, but is optional otherwise. Read this MSDN document for more details.

Auto-disconnect

The headset doesn't do any connection cleanup on power-down, so if a user turns off the headset while the software still has an open connection, the software may end up in an inconsistent state. It is prudent to continuously check that the headset is still receiving data, and to timeout the connection if data hasn't been received in a period of time (generally 3s or so).

Conclusion

After reading this document, you should have a good idea on how to integrate brainwave-sensing functionality into your Unity project. The sample Unity project offers a quick-start foundation on which to build your ThinkGear-enabled games, and the feature suggestions offered towards the end of this document should get you thinking about the sorts of usability improvements that could be made.

References

- Unity Pro plugin documentation

Corporate Address
NeuroSky, Inc.
125 S. Market St., Ste. 900
San Jose, CA 95113
United States
(408) 600-0129

Questions/Support: http://support.neurosky.com
or email: support@neurosky.com

Community Forum: http://developer.neurosky.com/forum

NeuroSky
Brain-Computer Interface Technologies