# Game Mechanics and Game Play Programming
# Using the Wiimote in Unity Indie

**Warning:**
This code has only been tested and I highly doubt it will work on Mac due to one or two system calls. If anyone desperately needs it in a Mac version, let me know and I will try to modify it.

Also, this code is far from perfect and was done in a bit of a rush. If you notice any crippling errors please let me know and I will try to change them. Otherwise, feel free to modify the code and improve it to your liking.

**Solution Summary:**
The indie version of Unity does not allow the use of plug-ins or importing DLLs. Also, there is some .Net functionality that Unity doesn't support. This means that there is no simple way to get any of the existing Wiimote-to-PC libraries to work in Unity.

The solution here is to have a separate process, outside of Unity, reading in the data from the Wiimote and passing it to Unity. This is typically quite easy as .Net has functionality to handle inter-process communication. Unfortunately, Unity doesn't support this either. So, wrapper presented here does the following.

- A UDP server is setup on the localhost.
- The server also interfaces with the WiimoteLib library that is used to access the wiimote in C#.
- The server waits for connections from a client and then waits for the client to request the wiimote data.
- When the client (Unity script) wants to update the wiimote data, it sends a message to the server, who polls the wiimote, compiles the update, and messages it back to the client who then updates its own representation of the wiimote.

**Installation:**
1) In Unity, create a folder in your Project panel called "Wiimote". Note, the folder must be called this for the some of the files to run properly.
2) Next, simply import the following four files into the Wiimote folder you just created.
    a. WiiUnity_Client.cs
    b. ClientWiiState.cs
    c. WiimoteServer.exe
    d. WiimoteLib.dll
3) The source code for the server is provided so that you can examine / change it if you wish but it you don't need it to use the source code

**Usage:**
There are only a handful of functions that you need to know to get this working in straight away. A polling technique is used so that you can get the data from the Wiimote whenever you feel is appropriate.

1) **WiiUnity_Client class** - This class is used as your main connection to the Wiimote (well kind of, see the Implementation Details below if you want the whole story).
    a. **Constructor** – Pretty much just an empty constructor (only one line). No arguments.

```
> WiiUnity_Client client = new WiiUnity_Client();
```

    b. **Initialising method** – This method should be called right after the constructor. It starts the server by creating a new process and running the WiimoteServer.exe. It then connects the client to the server and does some handshacking (e.g. it figures out how many wiimotes are currently connected to the computer). If any part of this fails, a boolean false value will be returned, after which calling any other method of the object (such as the update methods below) will result in an connection exception.

```
> public bool StartClient()
```

c. **Update methods** – These methods should be called every time you want to poll the wiimote and find out the state of the buttons, accelerometer, etc. They take as an argument the ID of the wiimote you want to poll. For example, for your player one's wiimote, pass an int value of "1", and "2" for player two, etc. This number corresponds to the LED that is lit up at the bottom of the controller. The data is stored in the ClientWiiState class (see below). There is no return type, if you call an update method of a nonexistent wiimote ID, simply nothing will happen except waste processing cycles.

```
> public void UpdateButtons(int wiimoteID)
> public void UpdateAccel(int wiimoteID)
> public void UpdateNunchuck(int wiimoteID)
> public void ToggleIR(int wiimoteID)
> public void UpdateIR(int wiimoteID)
> public void ToggleRumble(int wiimoteID)
```

d. **Accessors** – There is only one accessor method that you need to use. This method returns the current state of a wiimote as the client perseves it. This method should be called after the above update methods are called. It returns a ClientWiiState object, which will be explained further below.

```
> public ClientWiiState GetWiiState(int wiimoteID)
```

e. **Closing method** – It is good practise to call the closing method below just before your program execution stops (in Unity, put this in the `OnApplicationQuit()`). This will close the server and client in a clean way. If you don't make this call, you will have to manually stop the server after each run by closing the console window that the server is running in.

```
> public void EndClient()
```

**ClientWiiState class** – This class holds all the current information about a wiimote. There are no methods that you need to worry about using, all the methods there are simply to interface with the WiiUnity_Client class. There is a data member for each button, accelerometer axis, and joystick axis on the wiimote. The data members are all public and can just be accessed directly.

This is a list of data members that are publicly accessible in the ClientWiiState:

```
// All the button states of the wiimote. True if button is pressed down, false if released
public bool A, B, Up, Down, Left, Right, Plus, Minus, One, Two, Home;

// The accelerometer states
public float accelX, accelY, accelZ;

// Infrared states. Floats between (0,1)
public bool irActive, irVisible;
public float irMidpointX, irMidpointY, ir1PosX, ir1PosY, ir2PosX, ir2PosY;

public bool Rumble;

// Numchuck data
public bool ncC, ncZ;
public float ncJoyX, ncJoyY;

// Nunchuck accelerometer states
public float ncAccelX, ncAccelY, ncAccelZ;
```

**Example:**
This is an example to be used in Unity. Simply create a new script, copy the code below, attach the script to an object in the game (say the Camera), and run. This script will simply print out the value (as a Boolean) in every frame.

```
public class WiiUnityTest : MonoBehaviour {
	private WiiUnity_Client client;

	void Start() {
		client = new WiiUnity_Client();
	}

	void Update() {
		client.UpdateButons(1);
		client.UpdateAccel(1);
		ClientWiiState wiimote = client.GetWiiState(1);
		Debug.Log(wiimote.A);
	}

	void OnApplicationQuit() {
		client.EndClient();
	}
}
```

**Device Support:**

1) Wiimote Buttons – Every button on the standard wiimote can be accessed. Values are Boolean, false if the button is released, true if it is held down.

2) Wiimote Accelerometer – Values are stored as three floats and range between [-1, 1] for each of the 3D axis (x, y, z). Note that for the standard wiimote, the values are only really accurate to two decimal places. If the WiiMotionPlus is used, the granularity is much better. Also to note is that the accelerometers are not only sensitive to motion but also gravity.

3) Wiimote Infrared –
   a. To turn on and off infrared support, call the WiiUnity_Client.ToggleIR(int wiimoteID) .
   b. Once this is one, you can check if it is properly active by looking at the ClientWiiState.irActive variable.
   c. To quickly see if the wiimote is properly pointed at the Infrared Bar, check the ClientWiiState.irVisible variable. This package only works with two infrared points and both must be visible to the wiimote infrared camera.
   d. If the Infrared Bar is visible then you can get the position of each infrared point in 2D relative to the wiimotes infrared camera. This is through `ir1PosX, ir1PosY, ir2PosX, ir2PosY`. Each of these values are from 0-1. (0,0) is the top left of the IR Camera, and (1,1) is the bottom right of the IR Camera.
      i. With these values, its predicted that you can detect if the controller is rolling left or right by seeing if one IR point is higher than the other. This value may be used to negate the effects of the gravity on at least one axis of the accelerometer.
   e. `irMidpointX, irMidpointY` Are quick accessors for the centre of these two infrared points, and so is the direct centre of where the wiimote is pointing.

4) Wiimote Rumble - Simply toggle the vibration on and off.

5) Wiimote Audio NOT SUPORTED.

6) Nunchuck Buttons – The buttons on the nunchuck are also supported in a Boolean way.

7) Nunchuck Joystick – The joystick values are stored as two floats for 2D (x, y) coordinate space. Values are between [-0.5, 0.5]. Again, the data is only really reliable to two decimal places and this doesn't seem to change whether its stored as a double or a float.

8) Nunchuck Accelerometer supported

9) Multiple Wiimotes – The maximum four wiimotes are supported. The server and client are designed to automatically detect these. Checks are done in both client and server to make sure an incorrect wiimote ID wont cause the program to crash. This, along with function return values for notification of failure to find a wiimote, are structured such that if you have coded for two wiimotes and only one is used during execution then your program can still run (as long as you use the return values wisely).