Script Demo Elaborated

This document will explain the three example scripts found in the Script Demo scene.

1. Scripted Effects (Vibration Effect)

The goal of this script is to illustrate how to create a simple haptic effect and control it via a Unity script. For simplicity, this script is written as a MonoBehavior, and assigned to a unity game object.

```
    public HapticPlugin HapticDevice = null;
    private bool vibrationOn;
    private int FXID = -1;
```

Three variables are required, the haptic device the effect will apply to, the current state (on or off) of the effect, and most importantly, the ID of the effect. Every "effect" function call will need the configName of the current haptic device, and the FXID of the current effect, so these variables are declared as member variables of the MonoBehavior.

This script is written assuming that it will apply only to a single haptic device, although it should not be hard to extend it to work with multiple devices. (Each device will need its own effect.)

```
        if (HapticDevice == null)
            HapticDevice = (HapticPlugin)FindObjectOfType(typeof(Hapti
cPlugin));
```

As there's only one haptic device, we can assign this pointer simply by searching for an object of that type.

In this very simple example, we will be toggling the effect on and off with a keypress. The update function checks for `Input.GetKeyDown("v")` and calls either `TurnEffectOn()` or `TurnEffectOff()`, depending, of course, on whether the effect is already running. The meat of the script is in the `TurnEffectOn()` function.

First, if there is no Effect ID, one is requested from OpenHaptics. This is the equivalent to the OpenHaptics call `hlGenEffects`.

```
    if (FXID == -1)
    {
        FXID = HapticPlugin.effects_assignEffect(HapticDevice.configNa
me);

        if (FXID == -1) // Still broken?
        {
```

```
            Debug.LogError("Unable to assign Haptic effect.");
            return;
        }
    }
```

Next we must assign the effect settings. This is done with a call to effect_settings(..). This call assigns all of the effect parameters at ounce. Two of the parameters are vectors that take the form of arrays of doubles, so those must be created first.

```
    // Send the effect settings to OpenHaptics.
    double[] pos = {0.0, 0.0, 0.0}; // Position (not used for vibratio
n)
    double[] dir = {0.0, 1.0, 0.0}; // Direction of vibration

    HapticPlugin.effects_settings(
        HapticDevice.configName,
        FXID,
        0.33, // Gain
        0.33, // Magnitude
        300,  // Frequency
        pos,  // Position (not used for vibration)
        dir); // Direction.
```

Notice that vibration effects do not require position, so a dummy value is assigned.

Next we set the effect type, and we start the effect.

```
    HapticPlugin.effects_type( HapticDevice.configName,    FXID,4 ); /
/ Vibration effect == 4
    HapticPlugin.effects_startEffect(HapticDevice.configName, FXID );
```

This is all that should be needed to create and trigger an OpenHaptic effect that makes the device vibrate. However, you'll probably also want a way to **stop** the effect.

```
    void TurnEffectOff()
    {
        HapticPlugin.effects_stopEffect(HapticDevice.configName, FXID
);
    }
```

And that's it. Attach this MonoBehavior to any object in your scene, and you should be able to turn a vibration effect on and off by hitting the 'v' key.

To see and feel this in action, try the included "Script Demo" scene, and to see slightly more advanced use of OpenHaptics effects, you can examine the included HapticEffect.cs script. This script allows the developer to assign "zones" of haptic effects within the environment and is a good starting point for any effect-related scripting.

2. Useful variables (Meters Demo)

This simple script illustrates the use of some of the member variables of the HapticPlugin MonoBehavior. For simplicity, this explanation assumes you are familiar with creating a UI in Unity.

```
    public HapticPlugin HapticDevice = null;
    public GameObject Bunny = null;
```

Again, a member variable referencing the current HapticDevice is needed, along with references a model of a bunny and several UI elements (not shown here for clarity.)

```
    if (HapticDevice == null)
        HapticDevice = (HapticPlugin)FindObjectOfType(typeof(HapticPlu
gin));
```

As before, the start function locates the haptic device by searching for an object of that type. If there's more than one, this function will only find one of them, and it's undefined which one.

The first UI element controlled by this script is the "Bunny Poking" meter. Basically it detects whether or not you're touching the bunny, and if you are, it displays how hard you're poking it.

```
    if (HapticDevice.touching == Bunny)
    {
        float depth = (1.0f / depthMax) * HapticDevice.touchingDepth;
        depthMeter.fillAmount = depth;
    }
    else
        depthMeter.fillAmount = 0;
```

The HapticDevice.touching member variable is a reference to the touchable game object the stylus is currently touching, or `null` if nothing is being touched. HapticDevice.touchingDepth, is the distance between the real world stylus tip, and the haptic proxy. This directly corresponds to the amount of pressure being applied by the user. Here we convert that value to a value between 0.0 and 1.0, and assign that to a fill meter. Of course, if we're not touching the bunny, the value is zero.

Note : Only objects that are tagged "touchable" and are therefore rendered via OpenHaptics' shape rendering system will work with this method. If your object is *not* "touchable" and you're simulating touching using the Unity collision detection system, you will need to detect which object you're touching through more traditional Unity collision detection handling.

The next UI element is the "Speed Meter". This one is pretty straightforward.

```
    speedMeter.fillAmount = (1.0f / speedMax) * HapticDevice.stylusVel
ocityRaw.magnitude;
```

The member variable stylusVelocityRaw is the literal velocity of the physical haptic stylus. (Not the proxy, or the in-game object.) Here it's converted to a value between 0.0 and 1.0 to fill a meter.

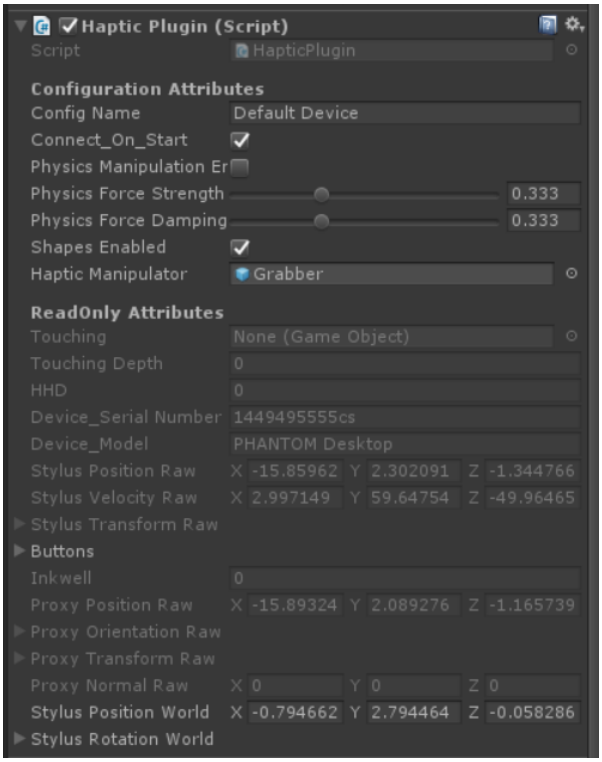
And finally, we toggle the visibility of some UI elements based on the current status of the stylus buttons.

```
if (HapticDevice.Buttons [0] == 1)
    button1Text.enabled = true;
else
    button1Text.enabled = false;

if (HapticDevice.Buttons [1] == 1)
    button2Text.enabled = true;
else
    button2Text.enabled = false;
```

The Buttons member variable is an array of four integer values. The values are either 1 or 0, depending on whether or not the corresponding button has been pressed. All currently available devices have two or less buttons, so don't worry about the third and fourth elements in that array.

To see and feel this in action, try the included "Script Demo" scene, and remember that you can look at all these variables in real time in the Inspector panel of the Unity debugger.



3. Scripted Surface Materials

The goal of this script is to illustrate how to create more complex haptic surface effects. OpenHaptics only supports one material type per surface, so this will be achieved by updating the surface effects in real-time. In this demo, white-colored regions of the object will be slippery, and black-colored regions will be high-friction. This should illustrate how OpenHaptics parameters can be manipulated and combined to achieve much more complex effects.

Once again, this script is a MonoBehavior intended to be attached to a gameObject, and once again we'll need a reference to the haptic device itself. We'll also need a reference to the texture that will modulate the surface parameters. (This needn't be the same texture that is used for rendering, but is in the example scene for clarity.)

```
    HapticPlugin device = null;
    public Texture2D FrictionTexture = null;
```

There's also, as seen in the screenshot above, a bunch of variables defining the two haptic surfaces.

The magic of this demonstration is in transitioning between these two sets of values based on the color of the texel nearest the stylus tip.

```
    // Find the pointer to the collider that defines the "zone".
    Collider collider = gameObject.GetComponent<Collider>();

    // Find point on collider closest to the stylus.
    Vector3 StylusPos = device.stylusPositionWorld;    //World Coordin
ates
    Vector3 CP = collider.ClosestPointOnBounds(StylusPos);     //World
 Coordinates
    float delta = (CP - StylusPos).magnitude;
```

First we grab a reference to the Unity Collider of the GameObject. We use ClosestPointOnBounds to find our "Closest Point". If the stylus is touching or inside the collider, this point will be exactly the same as the stylus position. Otherwise it will simply be the nearest point. We calculate the delta between these two points to determine if the stylus is close enough to the surface that we care about it. We do this with a simple `if (delta < 1.0f)`. If we *are* within range of the surface, we perform a raycast between that point and the object's center. (Concave shapes may need a slightly more complicated algorithm.)

```
    Vector3 direction =  transform.position - CP;
    direction.Normalize();

    // Cast a ray between the stylus and the center of the collider
    RaycastHit[] hits = Physics.RaycastAll(CP, direction);
```

It may seem strange to perform a raycast operation when we already *have* a closest point, but there's a good reason. The Unity RaycastHit object contains texture coordinates, so this is the easiest way to retrieve them.

```
    foreach (RaycastHit H in hits)
    {
        if (H.collider == collider)
        {
            // This is the correct hit, so retrieve the UV values...
            Vector2 UV = H.textureCoord;

            // Scale the UV to the size of the texture...
            int U = (int)(UV.x * FrictionTexture.width);
            int V = (int)(UV.y * FrictionTexture.height);

            // Retrieve the color of that pixel.
            Color C = FrictionTexture.GetPixel(U, V);
            luminosity = C.grayscale;
            break;
        }
    }
```

We loop through the array of hits because this will be the list of *all* raycast hits, and we're only looking for where the ray intersects the particular collider we're working with. When we find the correct hit, we take the UV coordinates and multiply them by texture resolution to get pixel

coordinates. GetPixel retrieves the color of that pixel. Unity color objects have a built in grayscale member, so we store that as our “luminosity”.

If the luminosity fraction is our value, then 1.0-luminosity must be the inverse fraction.

```
float Value = luminosity;
float inVal = 1.0f-Value;

HapticPlugin.shape_settings(gameObject.GetInstanceID(),
    hlStiffness * Value + hlStiffness2 * inVal,
    hlDamping * Value + hlDamping2 * inVal,
    hlStaticFriction * Value + hlStaticFriction2 * inVal,
    hlDynamicFriction * Value + hlDynamicFriction2 * inVal,
    hlPopThrough * Value + hlPopThrough2 * inVal);
```

Here we finally set the parameters of the haptic surface. If the luminosity is 1.0, only the values for the white surface will be used, if the luminosity is 0.0, only the values for the black surface will be used, and grey pixels will be a mixture of the two sets of values.

You can see and feel this script in action in the Script Demos scene. You could use this same general technique to create a number of interesting effects, including simulating wetness, varying friction based on how hard the user is pressing, or just have surfaces that change over time.