# WearScript Documentation

***Release .0.1.0***

**Brandyn White**

January 05, 2015

Contents

WearScript combines the power of Android development on Glass with the learning curve of a website. Go from concept to demo in a fraction of the time. For an overview check out the intro video and sample script below. Visit https://github.com/wearscript for the goods.

One-Line Installer(Linux/OSX): Execute the following in a shell to install WearScript on your Glass and authenticate with our default server.

```
curl -L http://goo.gl/U1RIHm > install.py && python install.py
```

```
<!-- Sample WearScript -->
<html style="width:100%; height:100%; overflow:hidden">
<body style="width:100%; height:100%; overflow:hidden; margin:0">
<canvas id="canvas" width="640" height="360" style="display:block"></canvas>
<script>
function server() {
    WS.log('Welcome to WearScript');  // Write to Android Log and Playground console
    WS.say('Welcome to WearScript');  // Text-to-Speech
    WS.sound('SUCCESS')

    // Changes canvas color with head rotation
    WS.sensorOn('orientation', .15, function (data) {
        ctx.fillStyle = 'hsl(' + data['values'][0] + ', 90%, 50%)'
        ctx.fillRect(0, 0, 640, 360);
    });

    // Stream several sensors (can view in the Sensors tab)
    var sensors = ['gps', 'accelerometer', 'magneticField', 'gyroscope',
                   'light', 'gravity', 'linearAcceleration', 'rotationVector'];
    for (var i = 0; i < sensors.length; i++)
        WS.sensorOn(sensors[i], .15);

    // Stream camera frames (can view in the Images tab)
    WS.cameraOn(.25);
    WS.dataLog(false, true, .15);

    // Hookup touch and eye gesture callbacks
    WS.gestureCallback('onTwoFingerScroll', function (v, v2, v3) {
        WS.log('onTwoFingerScroll: ' + v + ', ' + v2 + ', ' + v3);
    });
    WS.gestureCallback('onEyeGesture', function (name) {
        WS.log('onEyeGesture: ' + name);
    });
    WS.gestureCallback('onGesture', function (name) {
        WS.log('onGesture: ' + name);
    });
    WS.gestureCallback('onFingerCountChanged', function (i, i2) {
        WS.log('onFingerCountChanged: ' + i + ', ' + i2);
    });
    WS.gestureCallback('onScroll', function (v, v2, v3) {
        WS.log('onScroll: ' + v + ', ' + v2 + ', ' + v3);
    });
}
function main() {
    if (WS.scriptVersion(1)) return;
    ctx = document.getElementById('canvas').getContext("2d");
    WS.serverConnect('{{WSUrl}}', server);
}
window.onload = main;
</script></body></html>
```

CHAPTER 1

# Basics

In your Javascript environment, an object *WS* is initialized and injected for you with the methods below. WearScripts run on Glass in a WebView which supports standard browser functionality (e.g., dom manipulation, html5 canvas, cookies, local storage, etc.).

To start WearScript from the "Ok Glass" menu say "Ok Glass, Wear a Script". You can also tap the "Ok Glass" card and scroll to the "Wear a Script" card. Tap playground to connect to the playground, scroll to see other options (e.g., Stop, Setup, Gist Sync, etc.).

**scriptVersion(int version)** [boolean] Checks if the webview is running on a specific version.

```
if (WS.scriptVersion(1)) return;
```

**say(String message)** [void] Uses Text-to-Speach to read text

```
WS.say('Welcome to wearscript');
```

**serverConnect(String server, Function callback)** [void] Connects to the WearScript server, if given '{{WSUrl}}' as the server it will substitute the user configured server. Some commands require a server connection.

- Callback takes no parameters and is called when a connection is made, if there is a reconnection it will be called again.

```
WS.serverConnect('{{WSUrl}}', function () {
    WS.say('connected');
});
```

**log(String message)** [void] Log a message to the Android log and the JavaScript console of the webapp (if connected to a server).

```
WS.log('Welcome to wearscript');
```

**sound(String type)** [void] Play a stock sound on Glass. One of TAP, DISALLOWED, DISMISSED, ERROR, SELECTED, SUCCESS.

```
WS.sound('SUCCESS');
```

**shutdown()** [void] Shuts down wearscript

```
WS.shutdown();
```

**activityCreate()** [void] Creates a new activity in the foreground and replaces any existing activity (useful for bringing window to the foreground)

```
WS.activityCreate();
```

**activityDestroy()** [void] Destroys the current activity.

```
WS.activityDestroy();
```

**wake()** [void] Wake the screen if it is off, shows whatever was there before (good in combination with WS.activityCreate() to bring it forward).

```
WS.wake();
```

**liveCardCreate(boolean nonSilent, double period)** [void] Creates a live card of your activity, if nonSilent is true then the live card is given focus. Live cards are updated by polling the current activity, creating a rendering, and drawing on the card. The poll rate is set by the period. Live cards can be clicked to open a menu that allows for opening the activity or closing it.

**liveCardDestroy()** [void] Destroys the live card.

**displayWebView()** [void] Display the WebView activity (this is the default, reserved for future use when we may have alternate views).

CHAPTER 2

# Gist Integration

WearScript uses Github's Gist service to facilitate storing and sharing scripts. The Go server (wearscript-server) communicates with the Gist API and only allows operation on gists that have [wearscript] as the prefix of the description. The default script name for Glass devices is glass.html and there should be a manifest.json file with a field "name" that is the script's name. Gists can be synced to Glass by using the Gist Sync option in the menu when you start WearScript. In the playground gists can be opened, forked from others, and modified.

CHAPTER 3

# GDK Cards

WearScript uses an abstraction called a CardTree that allows for a hierarchy of cards where a node in the tree can optionally have either a menu or another set of cards beneath it and every card can have a tap/select callback. The syntax is overloaded to make common functionality concise.

## 3.1 Examples

The following displays a GDK card (consists of a body and footer)

```
var tree = new WS.Cards();
tree.add('Body text', 'Footer text');
WS.cardTree(tree);
WS.displayCardTree();
```

Lets add another card

```
var tree = new WS.Cards();
tree.add('Body 0', 'Footer 0');
tree.add('Body 1', 'Footer 1');
WS.cardTree(tree);
WS.displayCardTree();
```

Select and tap callbacks are optional arguments. If a card doesn't have any action on tap (i.e., a tap callback, subtree, or menu) it will use the "disallowed" sound.

```
var tree = new WS.Cards();
tree.add('Body 0', 'Footer 0', function () {WS.say('Selected')});
tree.add('Body 1', 'Footer 1', undefined, function () {WS.say('Tapped')});
WS.cardTree(tree);
WS.displayCardTree();
```

A menu is added with alternating title and callback arguments at the end of the parameter list. Tap/select parameters (if present) precede them.

```
var tree = new WS.Cards();
tree.add('Body 0', 'Footer 0', 'Say 0', function () {WS.say('Say 0')}, 'Say 1', function () {WS.say('
tree.add('Body 1', 'Footer 1', function () {WS.say('Selected')}, 'Say 0', function () {WS.say('Say 0
tree.add('Body 2', 'Footer 2', function () {WS.say('Selected')}, function () {WS.say('Tapped')}, 'Say
WS.cardTree(tree);
WS.displayCardTree();
```

A subtree of cards is added by creating another set of cards and placing it as the last parameter (may only have a menu or a subtree for a card). There is no depth limit for subtrees.

```
var tree = new WS.Cards();
tree.add('Body 0', 'Footer 0');
var subtree = new WS.Cards();
subtree.add('Sub Body 0', 'Sub Footer 0');
subtree.add('Sub Body 1', 'Sub Footer 1');
tree.add('Body 1', 'Footer 1', subtree);
WS.cardTree(tree);
WS.displayCardTree();
```

The GDK cards are limited in the layouts provided and the Mirror API provides more formatting options with HTML. We've ported much of this functionality without using Mirror so that more complex layouts (e.g., lists) can be used. Currently unsupported features include auto-paginate and auto-size. The syntax is that you create a <script> tag with the content (same format as Mirror accepts), then pass the ID of that tag to tree.addHTML(id) which can take all of the same options previously described (e.g., callbacks, menus, subtrees). Both card types can co-exist in the CardTree as illustrated below. For several examples of HTML cards checkout https://api.wearscript.com/#/gist/9477514/glass.html

First put the html in a script tag

```
<script type="text/html" id="tpl_card0">
    <article>
      <section>
        <ul class="text-x-small">
          <li>Gingerbread</li>
          <li>Chocolate Chip Cookies</li>
          <li>Tiramisu</li>
          <li>Donuts</li>
          <li>Sugar Plum Gummies</li>
        </ul>
      </section>
      <footer>
        <p>Grocery list</p>
      </footer>
    </article>
</script>
```

Then refer to it in javascript using WS.addHTML

```
var tree = new WS.Cards();
tree.addHTML('tpl_card0');
tree.add('Body 1', 'Footer 1');
WS.cardTree(tree);
WS.displayCardTree();
```

## 3.2 API

**WS.Cards** [Tree] Empty card tree configuration, create with "new WS.Cards()".

**Tree.add(String body, String footer, [Function selected, Function tapped], [String menuTitle, Function menuAction]..)** [Tree] Add a card to a tree. Can be chained for fluency

**Tree.addHtml(String htmlID, [Function selected, Function tapped], [String menuTitle, Function menuAction]..)** [Tree] Add a full HTML card to the tree. Can be chained for fluency.

**WS.cardTree(tree)** [void] Generates the full card tree in native memory.

**WS.displayCardTree()** [void] Shows the CardScrollView

CHAPTER 4

# User Input

**gestureCallback(String event, Function callback)** [void] Register to get gesture events using the string of one of the events below (following GDK names, see below).

**Event Types**

**onGesture(String gesture)** The gestures that can be returned are listed here: LONG_PRESS, SWIPE_DOWN, SWIPE_LEFT, SWIPE_RIGHT, TAP, THREE_LONG_PRESS, THREE_TAP, TWO_LONG_PRESS, TWO_SWIPE_RIGHT, TWO_SWIPE_UP, TWO_TAP

**onGesture<GESTURE>()** Shorthand for a specific gesture (e.g., onGestureTAP).

**onFingerCountChanged(int previousCount, int currentCount)** see FingerListener in GDK

**onScroll(float displacement, float delta, float velocity)** see ScrollListener in GDK

**onTwoFingerScroll(float displacement, float delta, float velocity)** see TwoFingerScrollListener in GDK

**onEyeGesture(String gesture)** One of WINK, DOUBLE_WINK, DOUBLE_BLINK, DON, DOFF

**onEyeGesture<GESTURE>** Shorthand for a specific gesture (e.g., onEyeGestureWINK)

```
WS.gestureCallback('onGesture', function (gesture) {
    WS.say(gesture);
});
WS.gestureCallback('onFingerCountChanged', function (i, i2) {
    WS.log('onFingerCountChanged: ' + i + ', ' + i2);
});
WS.gestureCallback('onScroll', function (v, v2, v3) {
    WS.log('onScroll: ' + v + ', ' + v2 + ', ' + v3);
});
WS.gestureCallback('onTwoFingerScroll', function (v, v2, v3) {
    WS.log('onTwoFingerScroll: ' + v + ', ' + v2 + ', ' + v3);
});

WS.gestureCallback('onGestureTAP', function () {
    WS.say('tapped');
});

WS.gestureCallback('onEyeGesture', function (gesture) {
    WS.say(gesture);
});
```

```
WS.gestureCallback('onEyeGestureDOUBLE_BLINK', function () {
    WS.say('double blink');
});
```

**speechRecognize(String prompt, Function callback)** [void] Displays the prompt and calls your callback with the recognized speech as a string

Callback of the form *function mycallback(String recognizedText)*

```
WS.speechRecognize('Say Something', function speech(data) {
    WS.say('you said ' + data);
});
```

**qr(Function callback)** [void] Open a QR scanner, return scan results via a callback from zxing

**Callback of the form *function mycallback(data, format)***

**String data** The scanned data (e.g., http://wearscript.com) is returned

**String format** The format of the data (e.g., QR_CODE)

```
WS.qr(function (data) {
    WS.say(data);
});
```

CHAPTER 5

# Sensors

**sensorOff(int type)** [void] Turns off sensor

**sensorOn(String type, double period, [Function callback])** [void] Turn on the sensor and produce data no faster than the specific period. Optional callback name that is called at most once per period of the form *function callback(data)* with data being an object with the properties:

**name(string)** Unique sensor name (uses Android name if one exists)

**type(int)** Unique sensor type (uses Android type if one exists), convert between them using WS.sensor(name) -> type

**timestamp(double)** Epoch seconds from when we get the sensor sample (use this instead of Raw unless you know better)

**timestampRaw(long)** Potentially differs per sensor (we use what they give us if available), but currently all but the light sensor are nanosec from device uptime

**values(double[])** Array of float values (see WS.sensor docs for description)

**For the Android built in sensors see the Android docs for their values, custom values are:**

- battery: Values [battery_percentage] (same as displayed in the Glass settings)
- pupil: Values [pupil_y, pupil_x, radius]
- gps: Values [lat, lon]

## 5.1 iBeacons

**WS.beacon(fn range, fn enter, fn exit)** [void] Starts scanning for beacons and performs callbacks. The fields in the callback are channel_name, UUID, powerLevel, majorNum, and minorNum.

## 5.2 Sensor Types

Sensors have unique names and integer types that are used internally and can be used as WS.sensor('light') which returns 5. The standard Android sensor types are positive and custom types are given negative numbers.

| Type | Value |
|---|---|
| ibeacon | -8 |
| pupil | -2 |
| gps | -1 |
| accelerometer | 1 |
| magneticField | 2 |
| orientation | 3 |
| gyroscope | 4 |
| light | 5 |
| gravity | 9 |
| linearAcceleration | 10 |
| rotationVector | 11 |

CHAPTER 6

# Camera

**cameraOn(double imagePeriod, [int maxHeight, int maxWidth, Function callback])** [void] Continuously capture camera frames. If a callback is specified it is given the data base64 encoded.

```
<html style="width:100%; height:100%; overflow:hidden">
<body style="width:100%; height:100%; overflow:hidden; margin:0">
<img width="640" height="360" id="image" />
<script>
function main() {
    if (WS.scriptVersion(1)) return;
    WS.cameraOn(.5, 180, 320, function (x) {
        document.getElementById('image').setAttribute('src', '' + x);
    });
}
window.onload = main;
</script></body></html>
```

**cameraOff()** [void] Turns off camera

```
<html style="width:100%; height:100%; overflow:hidden">
<body style="width:100%; height:100%; overflow:hidden; margin:0">
<img width="640" height="360" id="image" />
<script>
function main() {
    if (WS.scriptVersion(1)) return;
    WS.cameraOn(.5, 180, 320, function (x) {
        document.getElementById('image').setAttribute('src', '' + x);
    });
    setTimeout(function () {
        WS.cameraOff();
    }, 5000);
}
window.onload = main;
</script></body></html>
```

**cameraPhoto([Function callback])** [void] Take a picture and save it to the SD card and callback to *callback(String path)*. Can be used as src attribute for an image.

```
<html style="width:100%; height:100%; overflow:hidden">
<body style="width:100%; height:100%; overflow:hidden; margin:0">
<img width="640" height="360" id="image" />
<script>
function main() {
    if (WS.scriptVersion(1)) return;
```

```
    WS.cameraPhoto(function (x) {
        document.getElementById('image').setAttribute('src', 'file://' + x);
    });
}
window.onload = main;
</script></body></html>
```

**cameraPhotoData([Function callback])** [void] Take a picture and callback to *callback(String imageb64)* with a base64 encoded jpeg.

```
<html style="width:100%; height:100%; overflow:hidden">
<body style="width:100%; height:100%; overflow:hidden; margin:0">
<img width="640" height="360" id="image" />
<script>
function main() {
    if (WS.scriptVersion(1)) return;
    WS.cameraPhotoData(function (x) {
        document.getElementById('image').setAttribute('src', '' + x);
    });
}
window.onload = main;
</script></body></html>
```

**cameraVideo([Function callback])** [void] Record a video and save to the SD card.

```
<html style="width:100%; height:100%; overflow:hidden">
<body style="width:100%; height:100%; overflow:hidden; margin:0">
<script>
function main() {
    if (WS.scriptVersion(1)) return;
    WS.cameraVideo(function (x) {
        WS.log(x);
    });
}
window.onload = main;
</script></body></html>
```

CHAPTER 7

# Publish/Subscribe

**publish(String channel, args...)** [void] Sends PubSub messages to other devices

**subscribe(String channel, Function callback)** [void] Receives PubSub messages from other devices. Callback is provided the data expanded (e.g., if ['testchan', 1] is received then callback('testchan', 1) is called). Using javascript's 'arguments' functionality to get variable length arguments easily.

## 7.1 Example: Ping/Pong

```
<!-- WearScript on Glass/Android -->
<html style="width:100%; height:100%; overflow:hidden">
<body style="width:100%; height:100%; overflow:hidden; margin:0">
<script>
function main() {
    if (WS.scriptVersion(1)) return;
    WS.serverConnect('{{WSUrl}}', function () {
        WS.subscribe('pong', function (chan, timestamp0, timestamp1, groupDevice) {
            WS.log('Pong: ' + groupDevice + ': Remote - Glass0: ' + (timestamp1 - timestamp0) + ' Gla
        });
        setInterval(function () {
            WS.publish('ping', 'pong', (new Date).getTime() / 1000);
        }, 250);
    });
}
window.onload = main;
</script></body></html>
```

To start the manager in client mode (i.e., it connects to the Go server) use the following. All messages are sent example_ping <-internet-> go server <-internet-> glass which means you will have higher latency than connecting directly but you can stil use the Playground fully. This is normally recommended for development.

```
python example_ping.py client <client endpoint here>
```

To start the manager in server mode (e.g., Glass connect directly to this script, bypassing the Go server) use the following. Note that in your script you have to change WS.serverConnect to use your server's ip and port (e.g., ws://localip:localport) which mean you won't get playground connectivity (e.g., no logs, can't resend code) until you stop the script ("Ok Glass" -> "Wear a Script" -> Scroll to Stop). The benefit is that you can get substantially lower latency connecting directly to your computer.

```
python example_ping.py server <server port here>
```

```
# Python: Client or Server
import wearscript
import argparse
import time

def callback(ws, **kw):

    def get_ping(chan, resultChan, timestamp):
        ws.publish(resultChan, timestamp, time.time(), ws.group_device)

    ws.subscribe('ping', get_ping)
    ws.handler_loop()

wearscript.parse(callback, argparse.ArgumentParser())

// Go: Server
package main

import (
    "code.google.com/p/go.net/websocket"
    "github.com/OpenShades/wearscript-go/wearscript"
    "fmt"
    "net/http"
    "time"
)
func wshandler(ws *websocket.Conn) {
        // Single user mode, see wearscript-server for multi-user/device example
        Manager, err := wearscript.ConnectionManagerFactory("server", "demo")
        if err != nil {
            return
        }
        Manager.Subscribe("ping", func (c string, dataBin []byte, data []interface{}) {
            resultChan, ok := data[1].(string)
            if !ok {return}
            Manager.Publish(resultChan, data[2], time.Now().UnixNano() / 1000000000., Manager.GroupDe
        })
        conn, _ := Manager.NewConnection(ws)
        Manager.HandlerLoop(conn)
}
func main() {
    http.Handle("/", websocket.Handler(wshandler))
    err := http.ListenAndServe(":8081", nil)
    if err != nil {
        fmt.Println("Serve error")
    }
}
```

https://raw.github.com/wearscript/msgpack-javascript/master/msgpack.js https://raw.github.com/wearscript/wearscript-js/master/wearscript-client.js

```
<!-- JavaScript client in a webpage -->
<html><head><script src="msgpack.js"></script><script src="wearscript-client.js"></script></head>
<body><script>
var ws = new WearScriptConnection(new WebSocket('CLIENT ENDPOINT HERE'), "client", "demo");
ws.subscribe('ping', function (chan, resultChan, timestamp) {
    ws.publish(resultChan, timestamp, (new Date).getTime() / 1000, ws.groupDevice);
});
</script></body></html>
```

## 7.2 Example: Image/Sensor Stream

```
<!-- WearScript on Glass/Android -->
<html style="width:100%; height:100%; overflow:hidden">
<body style="width:100%; height:100%; overflow:hidden; margin:0">
<script>
function main() {
    if (WS.scriptVersion(1)) return;
    WS.serverConnect('{{WSUrl}}', function () {
        WS.sensorOn('accelerometer', .25);
        WS.cameraOn(1);
        WS.dataLog(false, true, .15);
    });
}
window.onload = main;
</script></body></html>

# Python: Client or Server
import wearscript
import argparse


def callback(ws, **kw):

    def get_image(chan, timestamp, image):
        print('Image[%s] Time[%f] Bytes[%d]' % (chan, timestamp, len(image)))

    def get_sensors(chan, names, samples):
        print('Sensors[%s] Names[%r] Samples[%r]' % (chan, names, samples))

    ws.subscribe('image', get_image)
    ws.subscribe('sensors', get_sensors)
    ws.handler_loop()

wearscript.parse(callback, argparse.ArgumentParser())

// Go: Server
package main

import (
       "code.google.com/p/go.net/websocket"
       "github.com/OpenShades/wearscript-go/wearscript"
       "fmt"
       "net/http"
)

func wshandler(ws *websocket.Conn) {
     // Single user mode, see wearscript-server for multi-user/device example
     Manager, err := wearscript.ConnectionManagerFactory("server", "demo")
     if err != nil {
        return
     }

     Manager.Subscribe("image", func (c string, dataBin []byte, data []interface{}) {
         timestamp := data[1].(float64)
         image := data[2].(string)
         fmt.Println(fmt.Sprintf("Image[%s] Time[%f] Bytes[%d]", c, timestamp, len(image)))
     })
```

```
    Manager.Subscribe("sensors", func (c string, dataBin []byte, data []interface{}) {
        names := data[1]
        samples := data[2]
        fmt.Println(fmt.Sprintf("Sensors[%s] Names[%v] Samples[%v]", c, names, samples))
    })

    conn, _ := Manager.NewConnection(ws)
    Manager.HandlerLoop(conn)
}

func main() {
    http.Handle("/", websocket.Handler(wshandler))
    err := http.ListenAndServe(":8081", nil)
    if err != nil {
       fmt.Println("Serve error")
    }
}
```

```
<!-- JavaScript in a webpage -->
<html><head><script src="wearscript-client.js"></script></head>
<body><script>
var ws = new WearScriptConnection(new WebSocket(URL), "client", "demo");
ws.subscribe('image', function (chan, timestamp, image) {
    console.log(JSON.stringify({chan: chan, timestamp: timestamp,
                                                        image: btoa(image)}));
});

ws.subscribe('sensors', function (chan, names, samples) {
    console.log(JSON.stringify({chan: chan, names: names,
                                                      samples: samples}));
});
</script></body></html>
```

CHAPTER 8

# Arduino

Using an Arduino with WearScript you can integrate all varieties of sensors, input devices, LEDs, etc. In the wearscript-arduino repo we have an example manager.py that connects to the Arduino over a serial connection. There are folders of examples (described below) that contain both a wearscript and an arduino file (you flash the arduino yourself).

To start the manager in client mode (i.e., it connects to the Go server, see PubSub for details) use the following.

```
python manager.py --serialport /dev/ttyACM0 client <client endpoint here>
```

To start the manager in server mode (e.g., Glass connect directly to this script, bypassing the Go server) use the following. See PubSub for details on how to get Glass to connect locally and caveats of doing so.

```
python manager.py --serialport /dev/ttyACM0 server <server port here>
```

## 8.1 Basic

Connected to an Arduino (tested with Uno and Due), the following uses the scroll gesture to control a servo on Pin 9 and an LED on pin 13 (both specified in the .ino file). We access them by sending data of the form [”arduinobasic”, device, value] where devices index contiguously into servo pins then LEDs (i.e., binary pin, need not control an LED) specified in the .ino file. For Servos the value is written directly to the servo and for LEDs any non-zero value is “on” and zero is “off”.

```
<html style="width:100%; height:100%; overflow:hidden">
<body style="width:100%; height:100%; overflow:hidden; margin:0" bgcolor="#000000"><script>
function main() {
    if (WS.scriptVersion(1)) return;
    WS.serverConnect('{{WSUrl}}', function () {
        WS.gestureCallback('onScroll', function (v, v2, v3) {
            var v = Math.min(180, Math.max(0, v / 6));
            WS.publish('arduinobasic', 0, Math.round(v));
        });
        WS.gestureCallback('onGesture', function (v) {
            if (v == 'TAP')
                WS.publish('arduinobasic', 3, 0);
            else if (v == 'TWO_TAP')
                WS.publish('arduinobasic', 3, 1);
        });
    });
}
window.onload = main;
</script></body></html>
```

## 8.2 Neopixel

Neopixels are independently addressable LED strips that can display multiple colors, change brightness, and display patterns over time. Using the included neopixel .ino file and sample script https://api.wearscript.com/#/gist/9390329/glass.html complex patterns can be programmed and executed on the Arduino including loops.

## 8.3 Powertail

The model used is https://www.adafruit.com/products/268#Description with an Arduino Uno

| Arduino Pin | Connected to |
|---|---|
| 3.3v | power tail 1: +in |
| pin #4 | power tail 2: -in |
| ground | power tail 3: Ground |

Bluetooth version See the *bluetooth* page for setup details (same pins).

```
#include <SoftwareSerial.h>

SoftwareSerial mySerial(3, 2); // RX, TX

void setup()
{
  Serial.begin(57600);
  while (!Serial) {
  }
  mySerial.begin(9600);
  pinMode(4, OUTPUT);
}

void loop()
{
  if (mySerial.available()) {
    char c = mySerial.read();
    if (c == '0')
      digitalWrite(4, 0);
    else
      digitalWrite(4, 1);
    Serial.write(c);
  }
  if (Serial.available()) {
    delay(10); // HACK
    mySerial.write(Serial.read());
  }
}
```

```
<html style="width:100%; height:100%; overflow:hidden">
<body style="width:100%; height:100%; overflow:hidden; margin:0">
<script>
function main() {
    if (WS.scriptVersion(1)) return;
    WS.serverConnect('{{WSUrl}}', function () {
        WS.gestureCallback('onGestureSWIPE_RIGHT', function () {
            WS.bluetoothWrite('20:13:12:05:04:11', '1');
        });
```

```
        WS.gestureCallback('onGestureSWIPE_LEFT', function () {
            WS.bluetoothWrite('20:13:12:05:04:11', '0');
        });
        WS.bluetoothList(function (devices) {
            WS.log(JSON.stringify(devices));
        });
    });
}
window.onload = main;
</script>
</body>
</html>
```

CHAPTER 9

# Bluetooth

## 9.1 Any -> Glass

If you are running a WearScript on a device that is paired to Glass, you are able to send gestures to Glass just like the MyGlass app. If you want to setup the socket before you actually send a command, use INIT.

```
WS.control("TAP");
WS.control("SWIPE_LEFT");
WS.control("SWIPE_RIGHT");
WS.control("SWIPE_DOWN");
WS.control("INIT");
```

## 9.2 Keyboard -> Glass

By connecting a BT keyboard to Glass you can control the timeline and use the keypresses inside WearScript like you would in a normal webpage. Keyboards like this http://www.amazon.com/dp/B005C6CVAE/ work well with Glass. First we have to install the Settings.apk so that we can access the pairing options. Install the Settings.apk from here http://www.glassxe.com/2013/05/23/settings-apk-and-launcher2-apk-from-the-hacking-glass-session-at-google-io/ and start the settings using

```
adb shell am start -a android.intent.action.MAIN -n com.android.settings/.Settings
```

You may want to remove the settings apk after you are done as it cause annoying crashes on startup

```
adb uninstall com.android.settings
```

Put your device into a bluetooth pairing mode. Select Bluetooth, then use the touchpad to select your device. You'll be asked to type a code using the keyboard and then you are all done.

## 9.3 Arduino <-> Glass

Bluetooth module: http://www.amazon.com/dp/B0093XAV4U

Arduino code. Make sure to plug the RX from BT into TX of arduino which is pin 2 below.

```
#include <SoftwareSerial.h>
SoftwareSerial mySerial(3, 2); // RX, TX
void setup() {
  Serial.begin(57600);
```

```
  while (!Serial) {
  }
  mySerial.begin(9600);
}

void loop() {
  if (mySerial.available())
      Serial.write(mySerial.read());
  if (Serial.available()) {
      delay(10); // HACK
      mySerial.write(Serial.read());
  }
}
```

```
<html style="width:100%; height:100%; overflow:hidden">
<body style="width:100%; height:100%; overflow:hidden; margin:0">
<script>
function main() {
      if (WS.scriptVersion(1)) return;
      WS.serverConnect('{{WSUrl}}', function () {
          WS.bluetoothWrite('20:13:12:05:04:11', 'bluetooth message from glass to arduino');
          WS.bluetoothRead('20:13:12:05:04:11', function (c) {
                WS.log('BT:' + c)
          });
          WS.bluetoothList(function (devices) {
            WS.log(JSON.stringify(devices));
          });
            });
}
window.onload = main;
</script>
</body>
</html>
```

CHAPTER 10

# Augmented Reality

While Glass has a relatively small display and won't be able to create full immersive overlays, there are several interesting applications of AR that have been under-explored including short term "micro-interactions". We have put together the building blocks to get you started, this will require understanding how they work to use them properly and this module is highly experimental.

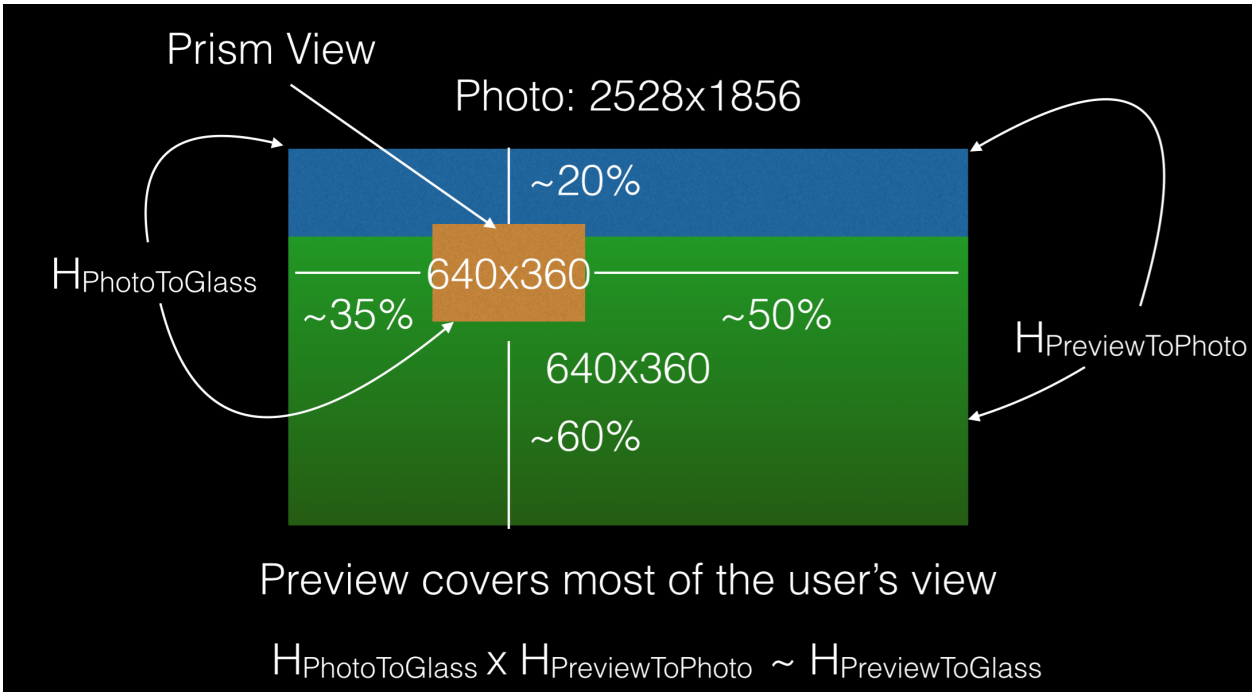


Figure 10.1: This shows the relationship between the Prism, Photo, and Preview (640x360) images.

hPhotoToGlass: Slightly different for each Glass and good results require calibation (see the wearscript-ar repo). hPreviewToPhoto: Each preview image has a different area (not all sizes are supported, see below) however they are constant across devices; however, changes in the underlying Glass camera code have caused changes in the past.

**displayWarpView([Array hPhotoToGlass])** [void] Warps each preview image to the display such that it overlaps with what the user sees (works for objects > 7ft away, currently supported resolutions are 640x360 and 1280x720). If the hPhotoToGlass homography is not provided a default is used; however, it won't match perfectly, each Glass is slightly different and they only need to be calibrated once.

**warpPreviewSampleGlass([Function callback])** [void] Publishes the next preview image it gets, it is used to match subsequent images to, a local copy is stored as the overlay and can be replaced using WS.warpSetOverlay.

- Callback has parameters of the form function *callback(Homography array)*

**warpSetOverlay(String imageb64)** [void] Sets the overlay being warped that corresponds with the last sample selected.

CHAPTER 11

# Eye Tracking

We developed a custom eye tracker for Glass so that we can experiment with using it as an input device. It is for research and development purposes, there is a lot of potential for this sub-project but it is still early so bare with us. See the video below for details of this project. Code is available at https://github.com/wearscript/wearscript-eyetracking

**https://github.com/wearscript/wearscript-eyetracking/blob/master/hardware/eye_tracker4.stl** <script src="https://embed.github.com/view/3d/wearscript/wearscript-eyetracking/master/hardware/eye_tracker4.stl"></script>

## 11.1 Getting Started

- The only OS recommended is Ubuntu Linux, it would be possible (though not easy because of opencv's lacking camera support on osx) to get it working on OSX and if you only have Windows consider using an Ubuntu virtual machine.
- Install the dependencies: gevent, wearscript-python, numpy, opencv (need a recent version! if you have a brand ubuntu 13.10+ you can use apt-get, else build from source).
- Acquire/build an eye tracker (best to contact brandyn in IRC)
- Run python track_debug.py debug, that will show you the eye camera and it should show a ring around your pupil
- We have several things you can do with it after this, but we are tidying things up (more info will be posted here after)

## 11.2 Building an Eye Tracker

- http://www.amazon.com/Microsoft-LifeCam-HD-6000-Webcam-Notebooks/dp/B00372567A
- 2 LEDs (recommend you get 4+ as you may break/lose some while soldering): http://www.digikey.com/product-detail/en/SFH%204050-Z/475-2864-1-ND/2207282 (shipping was $2.50 and each LED is ~$.75-$1 depending on amount)
- Access to a 3d printer (print the .stl file, currently #2 is the best design)
- Skills: Need to teardown a camera (requires patience), surface mount (de)solder LEDs (is doable with normal soldering equipment), need to take off/break the IR filter on the optics (requires the care)
- Tools (see build video): Side cutter, thin phillips screwdriver, spudger/x-acto knife (remove IR filter), soldering iron, solder, prybar, wrench (for cracking open case)

## 11.3 Tips

- The webcam must be manually focused, if it appears blurry twist the lens until the image is in focus. It may help to take the webcam out of the plastic mount and hold it roughly where it would be while doing this.

CHAPTER 12

# Myo

The Myo uses bluetooth-le and can only be connected to Android Kit Kat devices (e.g., Nexus 5).

This demonstrates pairing the Myo, saying out loud which gesture is performed. Remember, the setup gesture must be performed every time the device is paired before it will return data. NONE is produced after a gesture is stopped (e.g., a clenched fist will stay FIST until it is released and then will be NONE). It also streams the accelerometer, orientation, and gyro to the sensors tab (see WS.dataLog).

```
  <html style="width:100%; height:100%; overflow:hidden">
  <body style="width:100%; height:100%; overflow:hidden; margin:0">
  <script>
  function server() {
      WS.myoPair(function () {
    WS.say("paired myo");
});
      // Currently one of {NONE, FIST, FINGERS_SPREAD, WAVE_IN, WAVE_OUT, THUMB_TO_PINKY, ARM_RECOGN
      WS.gestureCallback('onMyo', function (x) {
          WS.say(x);
      });
      // Value is a quaternion (w, x, y, z) and (pitch, yaw, roll) for convenience
      WS.sensorOn('myoOrientation', .15, function (data) {
          WS.log('got data: ' + JSON.stringify(data));
      });
      // 3 axis accel (x, y, z)
      WS.sensorOn('myoAccelerometer', .15);
      // 3 axis gyro (x, y, z)
      WS.sensorOn('myoGyroscope', .15);
      WS.dataLog(false, true, .15);
  }
  function main() {
      if (WS.scriptVersion(1)) return;
      WS.serverConnect('{{WSUrl}}', server);
  }
  window.onload = main;
  </script></body></html>
```

CHAPTER 13

# Pebble

The Pebble connects to an android device over bluetooth, and requires the Pebble Application on the phone. The app can be downloaded from the play store.

To use Pebble with Wearscript, the Wearscript pebble client must be installed on the Pebble. Installation instructions are included in the repository.

This script demonstrates pebble click events, setting text in the view, vibration, and using accelerometer data.

```
<html style="width:100%; height:100%; overflow:hidden">
<body style="width:100%; height:100%; overflow:hidden; margin:0">
<script>
function main() {
    if (WS.scriptVersion(1)) return;
    WS.pebbleSetBody('connected!', false);

    WS.sensorOn(WS.sensor("pebbleAccelerometer"), .25, function (data) {
        WS.log('x: ' + data['values'][0] + ' y: ' + data['values'][1] + ' z: ' + data['values'][2]);
        WS.pebbleSetBody('x: ' + data['values'][0] + ' y: '+ data['values'][1] + ' z: ' + data['value
    });

    WS.pebbleSetSubtitle('Accel Data', false);

    WS.gestureCallback('onPebbleSingleClick', function (name) {
        WS.log('onPebbleSingleClick: ' + name);
        WS.pebbleSetTitle(name + ' clicked', false);
        WS.say(name);
    });

    WS.gestureCallback('onPebbleLongClick', function (name) {
        WS.log('onPebbleLongClick: ' + name);
        WS.pebbleVibe(2);
    });

    WS.gestureCallback('onPebbleAccelTap', function (axis, direction) {
        WS.log('onPebbleAccelTap: ' + axis + ' ' + direction);
    });
}
window.onload = main;
</script>
</body>
</html>
```

CHAPTER 14

# Tips/Tricks

- If you swipe down the script will continue to run in the background
- To turn off WearScript start it and select “Stop” (swipe one option to the right)
- When calling WS.serverConnect, if the argument passed is exactly ‘{{WSUrl}}’ then it will be replaced with the websocket url corresponding to the server the playground is running on and the last QR code generated.
- Unless you use a script that makes a server connection (i.e., WS.serverConnect(‘{{WSUrl}}’), ‘callback’) you will not be able to control WearScript from the webapp
- More interesting uses of WS.serverConnect include making a custom server for your application and then Glass will connect to it and stream data while your app can continue to control Glass.
- Every time you press the QR button on the webapp you get a unique auth key which replaces the previous.
- Multiple Glass devices can use the same QR code
- You only need to auth your Glass once and then again anytime you want to change servers (using the adb command provided when you press the QR button).
- When using scripts in the Playground editor, make sure to specify http:// or https:// and NOT use refer to links like <script type=”text/javascript” src=”//example.com/test.js”></script>. The script you put in the editor will be saved locally on Glass, and links of that form will not work.
- If you are connected to a server and use WS.log(‘Hi!’), that message will show up in the Android logs and the javascript console in the Playground.

CHAPTER 15

# Websocket Wire Format

All encoding is done using msgpack with lists as the base structure that have a string type as the first element. Websockets with a binary message payload a re used. Message delivery follows a pub/sub pattern.

## 15.1 Motivation and Goals

- Keep local data local if possible (e.g., glass -> phone shouldn't need to go through a server if they are connected)
- Pub/Sub over Point-to-Point: Focusing on channels which represent data types naturally handles 0-to-many devices reacting to it.
- Minimize data transfer, if nothing is listening on a channel then sending to it is a null-op
- Support a directed-acyclic-graph topology, where local devices can act as hubs (e.g., a phone is a hub for a watch, Glass, and arduino) and connect to other hubs (e.g., a remote server)
- Instead of having strict guarantees about delivery, provide a set of constraints that can be met given the fickle nature of mobile connectivity to eliminate edge cases

## 15.2 Protocol Rules

- Messages are delivered with "best effort" but not guaranteed: devices can pop in and out of existance and buffers are not infinite
- If a sender's messages are delivered they are in order
- A message SHOULD only be sent to a client that is subscribed to it or is connected to clients that are
- A message sent to a client that neither it or its clients are subscribed to MUST ignore it
- The "subscriptions" channel is special in that it MUST be sent to all connected clients
- When a client connects to a server it MUST be send the subscriptions for itself and all other clients
- If multiple channels match for a client the most specific MUST be called and no others
- When a client subscribes to a channel a single callback MUST be provided

The following table gives examples for when data will be sent given that there is a listener for the specified channel.

| sub | send | sent |
|---|---|---|
| “” | a | false |
| “” | “” | true |
| a | a | true |
| a | a:b | true |
| b | a:b | false |
| a: | a | false |
| a: | a:b | false |
| a: | a::b | true |
| a:b | a | false |
| a:b | a:b | true |
| a:b | a:bc | false |
| a:b | a:b:c | true |

CHAPTER 16

# Contributing

- All patches must be provided under the Apache 2.0 license
- Please use the -s option in git to "sign off" that the commit is your work and you are providing it under the Apache 2.0 license
- Submit a Github pull request to the "dev" branch and ideally discuss the changes with us in IRC
- We will look at the patch, test it out, and give you feedback
- New features should generally be put in feature branches
- Avoid doing minor whitespace changes, renamings, etc. along with merged content. These will be done by the maintainers from time to time but they can complicate merges and should be done seperately.
- All pull requests should be "fast forward"
  - If there are commits after yours use "git rebase -i <new_head_branch>"
  - If you have local changes you may need to use "git stash"
  - For git help see progit which is an awesome (and free) book on git

CHAPTER 17

# About

- OpenShades (the new OpenGlass) is our community
- IRC freenode #wearscript and #openshades (if you want to collaborate or chat that's the place to be)
- Project Lead: Brandyn White (bwhite dappervision com)
- G+ Community (we post work in progress here)
- Youtube (all demo videos)
- Dapper Vision, Inc. (by Brandyn and Andrew) is the sponsor of this project
- Code is licensed under Apache 2.0 unless otherwise specified

CHAPTER 18

# Contributors

See contributors for details. Name (irc nick)

- Brandyn White (brandyn)
- Andrew Miller (amiller)
- Scott Greenwald (scottgwald)
- Kurtis Nelson (kurtisnelson)
- Conner Brooks (connerb)
- Lance Vick (lrvick)
- Daniel Grove (iShortBus)
- Justin Chase (jujuman)
- Alexander Conroy (geilt)