AlloSphere Summer Workshop #5B:
The Device Server

September 8, 2010

Media Arts and Technology and the AlloSphere Research Facility

Instructor: Charlie Roberts

Email: charlie@charlie-roberts.com

Location: 1601 Elings Hall, California Nanosystems Institute, UCSB

Contents

1 Overview

The DeviceServer is both a framework and application driving interaction in the AlloSphere. The motivation and development of the Device Server stems from the practical concerns of managing multi-user interactivity with a variety of physical devices for disparate performance and virtual reality environments housed in the same physical location.

The interface of the Device Server allows users to see how devices are assigned to application functionalities, alter these assignments and save them into configuration files for later use. Configurations defining how applications use devices can be changed on the fly without recompiling or relaunching applications. Multiple applications can be connected to the Device Server concurrently.

The Device Server provides several conveniences for performance environments. It can process control data efficiently using Just-In-Time compiled Lua expressions; in doing so it frees processing cycles on audio and video rendering computers. All control signals entering the Device Server can be recorded, saved, and played back allowing performances based on control data to be recreated in their entirety. The Device Server attempts to homogenize the appearance of different control signals to applications so that users can assign any interface element they choose to application functionalities and easily experiment with different control configurations.

1.1 Related Work

VRPN is an excellent library used in Virtual Reality systems. The Digital Orchestra Toolbox Mapping tools are Max/MSP applications geared towards musical performance and experimentation.

- VRPN, http://www.cs.unc.edu/Research/vrpn/
- Digital Orchestra Toolbox Mapping Tools, http://www.idmil.org/software/mappingtools

2 Obtaining and Building

The DeviceServer source code can be downloaded from:

`http://www.allosphere.ucsb.edu/DeviceServer/`

There is an alias to a precompiled application in the download found above. It should run without problems on OS X 10.5 and higher and comes with a number of plugins (HID, Wiimote, MIDI, Audio) precompiled.

To build from source, first run the main DeviceServer.xcodeproj file and build the DeviceServer application target. Then, in the DeviceServerBundle folder build whichever plugins you're interested in receiving messages from. The plugins are also built through XCode and will be installed into the "plugins" folder that is in the same directory as the DeviceServer project file.

Once you have built the appropriate plugins open the DeviceServer application. You should now be able to connect devices and monitor signals.

3 Communicating with the Device Server

All DeviceServer communication currently occurs through OSC. Below is a list of the steps required for an application begin receiving signals from the Device Server; all steps will be explained in the coming subsections

- Send an OSC /handshake message from your application to the Device Server. This provides the Device Server with the application name, ip address and port.
- Create an **Interface** file for your application. This file contains the names of all functionalities you would like to expose for control along with the min and max values the functionalities expect to receive. It also contains the OSC addresses for the functionalities.
- Create an **Implementation** file for your application. Each application can have multiple implementations; these files reference the single Interface file and define controls on particular devices to assign to application functionality. You can also write expressions for signal processing here.
- Applications should send a /disconnectApplication message to the Device Server when they quit.

3.1 Handshaking

Handshake messages take the following form:

```
/handshake applicationName portWhereApplicatoinWillReceiveData
```

If sending an OSC message from Lua using the osc library built into LuaAV, the command might look like this:

```
oscout:send('/handshake', 'ApplicationOfGOZBALL', 10000)
```

From Processing, using the oscP5 library, it would be similar to the following:

```
msg = new OscMessage("/handshake");
msg.add("ApplicationOfGOZBALL");
msg.add(10000);
oscP5.send(msg, deviceServerIP, deviceServerPort);
```

The DeviceServer automatically determines the IP address of the application by looking at the packet headers of the handshake message. You can also add a specific IP address before the port if you want to override this default behavior.

3.2 Other Messages Your Application Should Send And Respond To

In addition to the previously mentioned /handshake message, there are a number of other important messages used to communicate with the Device Server; some are specific to the AlloSphere.

First, when your application quits it should send a /disconnectApplication message to the Device Server with the application name as the sole argument. This removes the application from the Device Server GUI and frees any memory associated with the application object.

Second, all applications being deployed in the AlloSphere should respond to a /volume message. The volume message will have one float as an argument between 0-1. This float should be used to set the level of your application's sound output.

Finally, all applications should respond to a /quit message. Many times in fullscreen applications the GUI thread becomes locked and it is impossible to quit using the mouse and keyboard. Meanwhile, the OSC thread continues running in the background giving a backdoor to force the application to quit. Please implement this if you are planning on having your app displayed in the AlloSphere.

4 The Interface File

There is one Interface file per application. It describes the functionalities that each device exposes in the following terms:

- Functionality Name - A plain jane English name for the functionality that will be displayed in the Device Server GUI. For example, we might name a function that moves a camera on the X Axis "Move Camera X", or "Translate Camera X".
- OSC Address - The OSC address pattern where the application will accept values for the functionality
- Min - The minimum value the application expects to receive for this functionality
- Max - The maximum value the application expects to receive for this functionality

In addition to the above items, each Interface file can define a default Implementation file that the DeviceServer will read immediately after receiving a handshake message from an application. A sample Interface file is given below:

```
properties = {
  defaultImplementation = "WirelessGamepad.lua",
}

functionalities = {
      { name = "Circle␣X␣Position",   destination = "/X", min = -1,  max = 1 },
      { name = "Circle␣Y␣Position",   destination = "/Y", min = -1,  max = 1 },

}
```

You can also place Lua functions inside of the Interface file. This allows these scripts to be accessed regardless of which Implementation file is currently in use by the Device Server; scripting the Device Server will be covered more in Section 6

5 The Implementation File

Implementation files map particular controls on particular devices to particular application functionalities (the application functionalities themselves are defined in the Interface file). For applications running in the AlloSphere there are usually multiple Interface files describing how different devices can interact with the application. A very simple example is to have Implementation files for both Wired and Wireless devices. An implementation file is loaded when an application handshakes with the DeviceServer; this implementation file is specified in the Interface file. Once an application has shook hands with the Device Server users can easily select a different Implementation file to use via the DeviceServer GUI.

Each implementation file consists of a collection of mappings. Lua functions for signal processing can also be placed in Implementation files. In general, if functions are specific to particular devices it's probably best to place them in the Implementation file that utilizes the device. If the functions are more general purpose it's best to place them in then Interface file so that they can be used regardless of which Implementation is currently selected.

At a bare minimum mappings must contain three pieces of information:

- Functionality Name - this specifies which functionality (as defined in the Interface file) you are mapping a device and control to.
- Device - The device you want to map to the functionality. You can specify the Device by name in string format, or by looking up the Devices assigned number in the Master Device List.
- Control - The specific control on the device you want to map to the functionality (for example, one button on a joystick). You can specify the Control by name in string format or by looking up the id number in the Lua file describing the device.

There are also two optional parameters.

- Expression - A Lua expression that will be applied to the values generated by the control. Expressions are covered in Section 6
- ControlID - An integer that will be added to the end of every message generated by this mapping. As one example, instead of having separate addresses in different mappings for "/button1" and "/button2" and "/button3" you could have one address "/buttons" and specify different controlID numbers to differentiate which button was being pressed.

A sample implementation file that matches the interface file from earlier in the handout is given below.

```
mappings = {
  { name = "Circle␣X",  device = "SpaceNavigator", control = "X", expression="y␣=␣x␣*␣2" },
  { name = "Circle␣Y",  device = "SpaceNavigator", control = "Y", },
}
```

6 Scripting the DeviceServer

All expressions are introduced in one of two places: the implementation files of an application or the device configuration files. An expression normally takes the following form:

```
y = x + (x / 2)
or
y = someFunctionName(x)
```

where someFunctionName is the name of a Lua function that has been defined in the implementation file. In these expressions, y represents the output value passed to the application (hence it appears on the left side of the equal sign) while x is the value originally generated by the control. If we call a Lua function in the expression, it needs to return a value for Y to be passed to the application. An example of a function that will pass the highest value stored is given below:

```
highestValue = -10000
function getHighestValue(x)
    if x > highestValue then
        highestValue = x
    end
    return highestValue
end
```

In the implementation file, this function would be called assigned in a mapping as follows:

```
expression = "y␣=␣getHighestValue(x)"
```

Console Object
The console object can be used to print to the console view in the Device Server Gui Methods: print(aString) Example script:

```
console:print("The␣value␣is␣" .. x)
```

Mapping Object
The Mapping object represents the mapping that the Lua expression currently being evaluated belongs to. It contains a number of properties and access to the Application object. rawX() is a particularly useful method that gets the value of the signal feeding into the mapping before any offsetting, scaling or expressions have been performed. preExpressionValue() is another useful method used to get the value after offsetting and scaling but before the expression has been evaluated

Methods: min(), max(), scalar(), rawX(), preExpressionValue(), application(), deviceID(), controlID()

- min() - the low end of the device signal range max() - the high end of the device signal range

- scalar() - a scalar is calculated when the mapping is created in order to match the signals to the range expected by the mapping's application
- offset() - an offset is used in conjunction with the scalar for matching signal range to expected application range
- rawX() - the value of the signal feeding into the mapping before any offsetting, scaling or expressions have been performed.
- preExpressionValue() - the value of the signal after offsetting and scaling but before the expression has been evaluated
- application() - returns the application object that the mapping belongs to
- deviceID() - the ID number of the device feeding the mapping
- controlID() - the ID number of the control feeding the mapping

The example script below feeds the raw value (in this case of a DPad on Logitech joystick) and creates a value in the range -1, 1 using a lookup table.

```
expression = "y␣=␣haty(mapping:rawX())"

function haty(x)
    if x == 1 or x == 2 or x == 3 then
        return 1
    elseif x == 5 or x == 6 or x == 7 then
        return -1
    else
        return 0
    end
end
```

Application Object The Application object represents the application the mapping belongs to. It is primarily used to get access to other mappings using the mappingForName(mappingName) function. The mappingForName function returns the applicationÊŒs mapping that matches the name provided. This allows one mapping to base its output off of the value of another mapping. One case where this could be used is in with coarse and fine pitch control. When the coarse control is changed, you want to output the value of the coarse signal plus whatever the last value of the fine signal was and vice versa. An example script for this is given below:

```
function coarse(x)
    -- we use the preExpressionValue because the final output of fine will
    -- have added the coarse value to it...
    f = application:mappingForName("fine"):preExpressionValue()
    y= x+ f
    return y
end

function fine(x)
    f = application:mappingForName("coarse"):preExpressionValue()
    y = x + f
    return y
end
```

Constants

DNR - When a function returns DNR (Do Not Return) no value is sent out to applications. This can be used to create momentary triggers from continuous signals or to only send values under special circumstances. For example, if we only wanted to send a message when the volume of an audio signal was above a particular value we could do so as follows:

```
function audioThreshold(x)
    if x > .75 then
        return x
    else
        return DNR
    end
end
```

gPwd - gPwd is a global that refers to the present working directory when used with application Implementation files; it points to the directory that scripts are stored in for the running application. Example:

```
dofile(gPwd .. "Wireless␣No␣Wiimotes.lua")
```

dofile simply runs the script at the provided location.

7 Adding Devices to the DeviceServer

Devices in the DeviceServer are run via a plugin system. Each category of Device has its own dedicated plug-in. Examples of plugins include MIDI, VRPN, HID etc. If the device you wish to utilize uses a communication protocol for which a plugin has already been made then all you need to do is create a device description file in Lua and edit the Master Device List so that the DeviceServer knows to look for your device. If your device communicates via a protocol not currently supported by the DeviceServer than you need to either make a plugin or use OSC to send values from your device to the DeviceServer.

7.1 Master Device List

The Master Device List is a Lua file that contains a list of all the devices that can be connected to the Device Server. ItÊŒs important to note that being on this list is not a guarantee that a device will be connected, only that it is usually available in whatever VRE the Device Server is running. Each device is enumerated with its human readable name (in most cases this is the name reported by whatever driver the device uses) and unique id number. The id number is arbitrary (as long as each is unique). In implementation files you can refer to devices using either their names or their id numbers.

Below is an example of a very simple Master Device List document.

```
devices = {
    { id = 1, name = "Wiimote␣1" },
    { id = 11, name = "PPTTracker␣1" },
    { id = 21, name = "Logitech␣Dual␣Action" },
    { id = 22, name = "Logitech␣Cordless␣RumblePad␣2" },
    { id = 23, name = "Logitech␣RumblePad␣2␣USB" },
    { id = 24, name = "USB␣2-Axis␣8-Button␣Gamepad" },
}
```

7.2 Device Description Files

Every device listed in the Master Device List has its own Lua document that enumerates each control in the device. The enumerations include a unique id for the control (arbitrary), HID usage page and usage numbers (if applicable), a human readable name (usually whatever name is reported by the device drivers), the range of values the control generates and an expression that can be applied to every message the control generates. This expression is important in assuring that devices output a range of value useful to developers. The Musical Instrument Digital Interface (MIDI) protocol, for example, specifies that most output values are seven bit numbers in the range of 0. 127. With the exception of controlling other MIDI devices, this range of values has very little practical usage. Changing these values to the range 0, 1 with an expression at the device level removes the need for the expression to be scattered throughout multiple application implementation files, or, even worse, for an application developer to handle the scaling and offsetting in their own code.

The Device Descriptions are all located within the deviceDescriptions folder, located in the same directory as the XCode project. They are copied into the application bundle when the application is compiled.

Below is the Device Description for the Trackpad device found on Apple laptops.

```
atrributes = {
  name = "Trackpad"
}

controls = {
  { id = 0, usagePage = 1, usage = 48, name = "X", minimum = -50, maximum = 50, expression = "" },
  { id = 1, usagePage = 1, usage = 49, name = "Y", minimum = -50, maximum = 50, expression = "" },
  { id = 2, usagePage = 9, usage = 1, name = "Button␣1", minimum = 0, maximum = 1, expression = "" },
}
```

7.3 OSC Devices

Software and hardware can register as Devices over OSC. There are two steps in addition to creating a Device Description File and adding the device to the Master Device List. The first is to send an OSC registration message. The registration message takes the form:

```
/registerDevice deviceName deviceIPAddress devicePortForReceivingMessages
```

The next step is simply to send values over OSC. These messages are formatted as follows:

```
\OSCDeviceMsg deviceName controlName controlValue
```

These messages will be distributed to interested applications.

7.4 Device Server Plugins

You can create your own Device Server plugin using the Xcode template project included in the Device Server download. Detailed instructions for using the template project are included in the README file included in the template Xcode project.