JunctionBox: A Toolkit for Creating Multi-touch Sound Control Interfaces

Lawrence Fyfe
InnoVis Group
University of Calgary
2500 University Drive NW
Calgary, AB T2N 1N4
Canada

Adam Tindale
Alberta College of
Art + Design
1407 14 Avenue NW
Calgary, AB T2N 4R3
Canada

Sheelagh Carpendale
InnoVis Group
University of Calgary
2500 University Drive NW
Calgary, AB T2N 1N4
Canada

ABSTRACT

JunctionBox is a new software toolkit for creating multi-touch interfaces for controlling sound and music. More specifically, the toolkit has special features which make it easy to create TUIO-based touch interfaces for controlling sound engines via Open Sound Control. Programmers using the toolkit have a great deal of freedom to create highly customized interfaces that work on a variety of hardware.

Keywords

Multi-touch, Open Sound Control, Toolkit, TUIO

1. INTRODUCTION

JunctionBox is a new toolkit for building multi-touch interfaces for controlling sound and music that combines existing libraries while adding important new functionality. But why is a new multi-touch toolkit needed and what specifically do sound and music applications require in terms of functionality?

From DIY vision-tracking-based tables to commercially available tablet computers, multi-touch interfaces are becoming a pervasive interaction paradigm. As multi-touch interfaces become increasingly common, it is important for programmers to have high quality toolkits for developing applications that take full advantage of multi-touch hardware. Toolkits can provide the necessary building blocks that help programmers to focus on creative tasks by removing the burdens of low-level implementation, particularly for non-WIMP (window, icon, menu, pointing device) interfaces [2].

One approach to instrument design is to separate interface building from sound engine building (where a sound engine might be Pd [9], ChucK [12], SuperCollider [8] or a similar programmable development environment). In this scenario, information about the state of the interface must be sent to the sound engine. Since the most flexible way to handle messaging is to use Open Sound Control (OSC) [13], any toolkit for developing sound and music control interfaces should have the ability to handle OSC. In addition, a multi-touch sound control toolkit should provide an easy way to map actions on multi-touch hardware to OSC messages.

Because programmers, including the authors, use a variety of hardware and operating systems, toolkits should, whenever possible, be cross-platform.

The previous points lead to the following basic requirements:

1. Support multi-touch
2. Provide OSC messaging
3. Map multi-touch actions to OSC messages
4. Be cross-platform

Many toolkits exist for building multi-touch applications. The MoMu toolkit [1] maps many input parameters, including touch, on mobile phones (and tablets) to sound control. While MoMu offers a range of sound control possibilities, it is not cross-platform and so cannot be used by programmers who do not choose the hardware and software combination that MoMu requires.

The MT4J toolkit [6] is cross-platform and has multi-touch capability via TUIO [5]. However, it offers no ability to map multi-touch actions to messages. Other toolkits such as PyMT [3] and tuioZones [7] suffer from a similar lack of message mapping capabilities.

2. JUNCTIONBOX

JunctionBox was designed as a toolkit to meet the requirements from Section 1. This section will discuss the incorporated libraries and the classes provided by JunctionBox to programmers. Figure 1 shows the relationship of the incorporated libraries to JunctionBox.

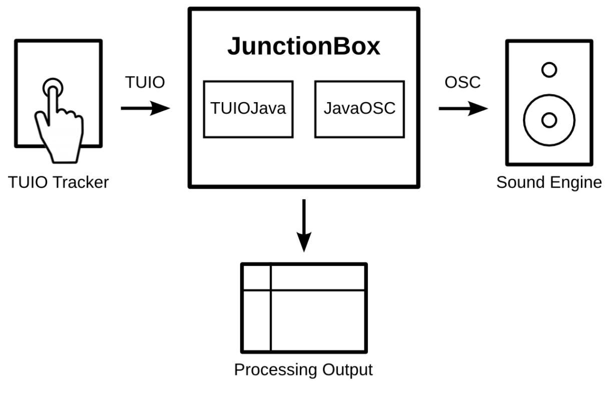

Figure 1: JunctionBox functionality and components.

To make JunctionBox cross-platform, it is written entirely in Java. TUIO was chosen for touch tracking since it has numerous implementations and is decidedly cross-platform, being based on Open Sound Control (OSC). A Java-based TUIO library called TUIOJava [4] provides basic TUIO client functionality. As a TUIO client, JunctionBox can work with any touch tracking systems that meets the TUIO specification. For OSC messaging, a slightly modified version of the JavaOSC [10] library included with TUIOJava is used. For visual output, JunctionBox uses the Processing [11] graphics engine.

The JunctionBox toolkit combines the basic components just described while providing unique functionality via classes described in the following subsections. Figure 2 shows the classes provided by JunctionBox, relating them to external functions like TUIO tracking and OSC messaging.

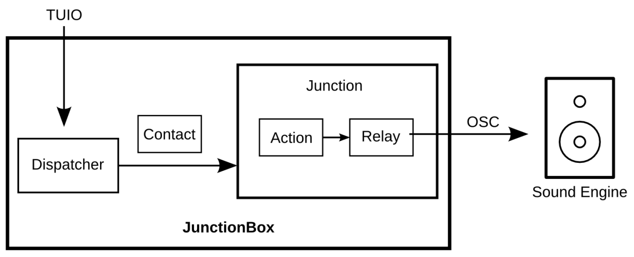

Figure 2: JunctionBox classes shown as boxes.

2.1 Dispatcher

All TUIO message handling in JunctionBox is done via the Dispatcher class. The Dispatcher is a TUIO client but only handles TUIO cursors (touches) and not TUIO objects (fiducial markers). Since not all hardware supports fiducial markers, to be more cross-platform, JunctionBox only handles touch interactions.

The "Box" part of JunctionBox defines the outer limits in width and height of the interactive touch area. This is generally mapped to the size of a touch surface like a video tracking table or a touch tablet. The following line of code creates a new Dispatcher object with a box width and height:

```
Dispatcher d = new Dispatcher(boxWidth, boxHeight,
"127.0.0.1", 6449);
```

The new Dispatcher code contains two additional arguments: a target IP address and port number. These arguments are inherited by Junctions (described below) for sending OSC messages to target sound engines.

2.2 Contacts

When TUIO messages are received by the Dispatcher, TUIO cursors (touches) are converted into Contact objects by the Dispatcher. The Contact class contains the same set of data provided via TUIO 1.1 including session ID, X and Y position, X and Y velocity vectors and acceleration. The Contacts are then dispatched to any Junction whose area coincides with the X,Y position of the TUIO cursor.

2.3 Junctions

The Junction class represents a defined interaction area that offers a set of actions be mapped to messages. Junctions can be created via the Dispatcher:

```
Junction j = d.createJunction(x, y, width,
height);
```

This allows Junctions to inherit values from the Dispatcher like box size and the IP address and port numbers of target sound engines.

A Junction is essentially defined by its area and can take on two shapes: rectangle and ellipse. That area and its location inside of the box determines whether a Junction receives Contacts based on whether touch events occur inside or outside of the area. This is shown in Figure 3.

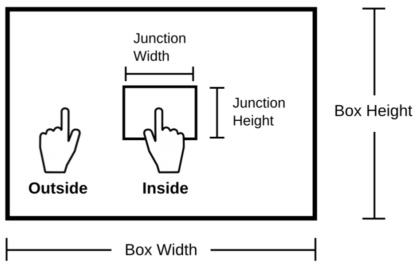

Figure 3: Touches that fall inside or outside of a Junction's area.

Junctions can be rotated, scaled and translated based on the movement of Contacts within a Junction's area. Rotation and translation can be done with a single touch while scaling requires two touches. Each of these actions can be turned on or off at the discretion of the programmer. Additionally, limits can set set on those actions. For example, a limit on Y translation can be coupled with the disabling of X translation to create something like a vertical slider. The following lines of code do this for a Junction j:

```
j.translateX = false;

j.limitTranslateY(100, 500);
```

The last line would limit translation of the Junction to a minimum of Y = 100 and a maximum of Y = 500. Note that these values come from the box size in pixels established when the Dispatcher is created which in turn is related to the canvas size of the Processing sketch containing the visual output code.

Junctions have no inherent visual output but are associated with rectangular and elliptical shapes in Processing. To draw a rectangle in Processing that inherits values from a Junction j:

```
rect(j.getCenterX(), j.getCenterY(), j.getWidth(),
j.getHeight());
```

When the rectangle is drawn in Processing, it will take the current values from the Junction j, so that a translation action will result in the rectangle moving across the screen as the translation occurs. Because Junctions move based on the location of their centers, rectangles and ellipses drawn in Processing must use the center mode to work correctly.

An unlimited number of Junctions can be defined with the Junctions being stackable. When multiple Junctions are created, the last one created receives Contacts where two or more overlap within the box.

A Junction can be added to another Junction. When this is done, the added Junction inherits actions performed on the parent Junction including rotation, scaling and translation.

2.4 Actions

Actions are a means to create mappings between touches and messages for a sound engine. To enable this, the Action class contains constants whose names correspond to actions built into Junctions.

To add a mapping between an Action and an OSC message requires just one line of code:

```
j.addMessage(Action.TRANSLATE_Y, "/osc/message");
```

Whenever the center Y value of the Junction j changes, that value will be sent as a float argument to ”/osc/message”. If the center Y value of the Junction j changes to 42, the message will be:

```
/osc/message,f 42
```

For a given Junction, any combination of Actions can be used from none to all. Figure 4 shows some example Actions.

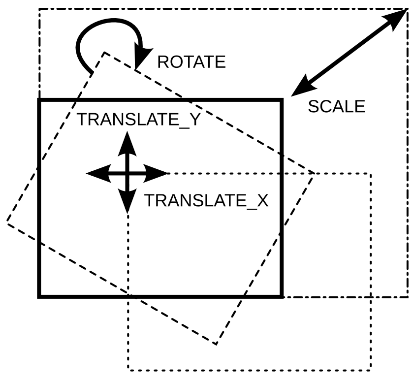

Figure 4: Some Junction actions that can be mapped to messages.

The following actions are available:

- ACTIVE

  If a Junction receives one initial Contact, it will send a message with a single integer value of 1. Messages are only sent when the active state of the Junction changes. So when all contacts have been removed from the Junction, it will send a message with an integer value of 0.

- TOGGLE

  When a Junction receives an initial Contact, it can send a message with its current toggle state. The first Contact sets the toggle state to 1 and triggers a message with that integer value. Any subsequent Contact after the first Contact is removed will trigger a change to the 0 toggle state and that value will be sent in an integer message. This action allows for the creation of simple touch switches.

- ROTATE

  Junction objects can be rotated a full 360 degrees (or less depending on custom limits). Each time the angle of the Junction changes, a float message will be sent out with the current angle normalized from 0 to 1 where 1 represents 360 degrees or 2 Pi radians.

- SCALE

  A two-Contact scaling gesture (where one Contact is held while the other is moved) when applied to a Junction will trigger a calculation of the ratio between the previous area and the current area after scaling. Whenever the scale value changes, a float message is sent with the normalized (0-1) value of the ratio. The normalization works because there is an absolute minimum value for both width and height of a Junction of 1 pixel. The maximum values for width and height of a Junction are the width and height of the box set in the creation of the Dispatcher.

- TRANSLATE_X

  Moving a Junction along the X axis (as defined in Processing) will trigger a float message with the current value of the center X point of the Junction. Messages are only sent when the center moves.

- TRANSLATE_Y

  As with X translation, moving a Junction along the Y axis can trigger a similar float message with the current center Y value of the Junction. Messages are only sent when the center moves.

- TRANSLATE_XY

  Like the above translate actions but with both float values of center X and center Y sent in the same message.

- CONTACT_COUNT

  Whenever the number of Contacts changes, an integer message with the current Contact count is sent.

- ROTATION_COUNT

  Each time a Junction is rotated more than 360 degrees, the current value of the angle is reset to between 0 and 360 degrees. When this is done, a counter for the number of rotations is incremented. This works for clockwise rotations. For counter-clockwise rotations, the angle is negative and the rotation counter is decremented. Any change in the rotation count will trigger an integer message with the current count.

There is no inherent mapping between the chosen actions and the messages sent other than the value associated with the action. While the arguments and their types are fixed, the messages themselves can be changed to any that suit the programmer.

2.5 Relays

The Relay class offers full-featured access to the OSC functionality available in the JavaOSC toolkit with some additional features. Junctions use Relays internally to send messages that are mapped to actions. Outside of Junctions, Relays are designed for occasions when more complicated OSC messaging is required.

A Relay object is created with a target IP address and port number. Then any number of messages can be associated with that target and referenced for later sending.

When using Relays, both the address and the argument number and types can be controlled explicitly. Any action that a Junction can perform can be emulated by getting the current state of Junctions and applying those values directly to messages via Relays.

For example, the following code will create a Relay that sends messages to localhost. Once the message is added to the Relay, the values obtained from a Junction j are added to the message and the message is sent containing the three arguments.

```
Relay r = new Relay(127.0.0.1, 6449);

r.addMessage("/relay/example");
```

```
r.addInteger("/relay/example", j.getToggle());

r.addFloat("/relay/example", j.getAngle());

r.addFloat("/relay/example", j.getCenterX());

r.send("/relay/example");
```

Relays can hold an unlimited number of messages for a given target. Each message can have its arguments set by referencing the message address pattern String as shown in the above example. Additionally, arguments can be added to all messages contained in a Relay with lines like the following that add a float value of 0.5 to all messages.

```
relay.addFloat(0.5);
```

All messages contained in a Relay can be sent by using:

```
relay.send();
```

Also, a list of message strings can be provided to send multiple specific messages.

```
relay.send(messageStrings[]);
```

Using Relays from within Junctions is an easy means of getting multi-touch actions to map to messages. By making the Relay class itself available to programmers, a new set of more complex options becomes available, leaving decisions about messaging and mapping up to the programmer designing the interface without interference from the design of the JunctionBox toolkit.

2.6 Simulator

The Simulator was created for situations where multi-touch hardware is not available and simulates TUIO tracking via the mouse. When used in Processing, the Simulator takes mouse data: whether a mouse button is currently pressed, which button is being pressed, the current X-Y position and the previous X-Y position. That data is then converted to TUIO messages that are received by the Dispatcher object as described above. For now, the Simulator can only simulate a single touch via the mouse.

3. SUMMARY

The JunctionBox toolkit both combines existing libraries for touch tracking and messaging with new features not offered by any existing toolkit. The most significant feature is the ability to easily map multi-touch actions to sound and music control messages.

4. ACKNOWLEDGEMENTS

We would like to thank the Alberta Association of Colleges and Technical Institutes, the Canada Council for the Arts, the Natural Science and Engineering Research Council of Canada, the Alberta Informatics Circle of Research Excellence, SMART Technologies, Alberta Ingenuity, and the Canadian Foundation for Innovation for research support. We would also like to thank the members of the Interactions Lab at the University of Calgary for feedback and support during the development of this project.

5. REFERENCES

[1] N. J. Bryan, J. Herrera, J. Oh, and G. Wang. Momu: A mobile music toolkit. In *Proceedings of the 2010 Conference on New Interfaces for Musical Expression*, pages 174–177, 2010.

[2] S. Greenberg. Promoting creative design through toolkits. In *Proceedings of the Latin-American Conference on Human-Computer Interaction*, CLIHC'09, pages 92–93, November 9-11 2009.

[3] T. E. Hansen, J. P. Hourcade, M. Virbel, S. Patali, and T. Serra. Pymt: a post-wimp multi-touch user interface toolkit. In *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces*, ITS '09, pages 17–24, New York, NY, USA, 2009. ACM.

[4] M. Kaltenbrunner. Tuiojava. `http://www.tuio.org/?java`, January 2011.

[5] M. Kaltenbrunner, T. Bovermann, R. Bencina, and E. Costanza. Tuio - a protocol for table-top tangible user interfaces. In *Proceedings of the 6th International Workshop on Gesture in Human-Computer Interaction and Simulation*, GW 2005, 2005.

[6] U. Laufs, C. Ruff, and J. Zibuschka. Mt4j - a cross-platform multi-touch development. In *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*, EICS '10, New York, NY, USA, 2010. ACM.

[7] J. Lyst. tuiozones. `http://jlyst.com/tz/`, January 2011.

[8] J. McCartney. Rethinking the computer music language: Supercollider. *Computer Music Journal*, 26(4):61–68, 2002.

[9] M. Puckette. Pure data: another integrated computer music environment. In *Proceedings of the International Computer Music Conference*, pages 37–41, 1996.

[10] C. Ramakrishnan. Javaosc. `http://www.illposed.com/software/javaoscdoc/`, January 2011.

[11] C. Reas and B. Fry. Processing: programming for the media arts. *AI & Society*, 20(4):526–538, 2006.

[12] G. Wang and P. Cook. Chuck: a programming language for on-the-fly, real-time audio synthesis and multimedia. In *Proceedings of the 12th annual ACM international conference on Multimedia*, MULTIMEDIA '04, pages 812–815, New York, NY, USA, 2004. ACM.

[13] M. Wright. Open sound control: an enabling technology for musical networking. *Organised Sound*, 10(3):193–200, 2005.