A DISTRIBUTED MODEL-VIEW-CONTROLLER DESIGN PATTERN FOR A GRAPHICAL REMOTE CONTROL OF A MULTI-USER APPLICATION

Author Name(s)
Affiliation
Email addresses

ABSTRACT

The Model-View-Controller (MVC) is an object-oriented design pattern for architecting the interactions among human users of graphical computing systems with the software Controller that manages user input, the Model that houses system state, and the View that projects the Model state into intelligible graphical form. The present work examines extending MVC into a Distributed Model-View-Controller pattern, starting with a stand-alone MVC system that is amenable to distribution over a local area network (LAN). Distribution takes the form of cloning a stand-alone MVC application into distinct client and server programs, and then altering each for its purpose while maintaining the initial graphical compatibility of the starting, stand-alone system. A graphical client remote control running on a tablet computer sends its Model update events to the server's Controller via the LAN, acting as a remote input device for the server. The client uses simple, two-dimensional graphics for efficiency, while the server's graphical View can afford to use computationally expensive three-dimensional animations. Avoiding server-to-client synchronization avoids congestive LAN traffic and complicated interaction. The client acts as a one-way remote control, albeit a remote control with a display that is a simple version of the server's display.

KEY WORDS
Android tablet, distributed remote control, graphics.

1. Introduction

This report grows out of a course project in creating a graphical remote control (the client) on an Android tablet device [1] to manipulate a graphical application (the server) running on a PC or laptop, in the context of an undergraduate course in object-oriented multimedia programming. In the previous offering of this course, we used a text-based menu system on the client device to select and send commands to the server via the wireless local area network (LAN). While meeting minimum requirements for a remote control, the client user interface looked nothing like the graphical server. Its use was cumbersome and nonintuitive. This report explains how we distributed the well-known Model-View-Controller (MVC) design pattern [2] to convert a stand-alone, interactive graphical application into two programs, a graphical server running the main application, and multiple graphical clients whose user interface reflects the graphical nature of the server while avoiding excessive synchronization traffic on the LAN.

2. A Stand-alone Graphical Application

Figure 1 illustrates the graphical view of the handout code for the first, stand-alone assignment. Each rectangle represents a musical note, similar to a key on a piano keyboard. Color represents a note's position in its scale being played, where a musical *scale* is a cyclic sequence. Brightness represents a note's aural volume. The application was intended for a circular planetarium projector, hence the guide circle. The arrangement of notes was random in the handout code. Pitch and volume are ambiguous in Figure 1, because there are multiple notes of a given position in a scale, e.g., multiple "do", "re", "mi", etc. notes at distinct octaves, just as there are on a piano.

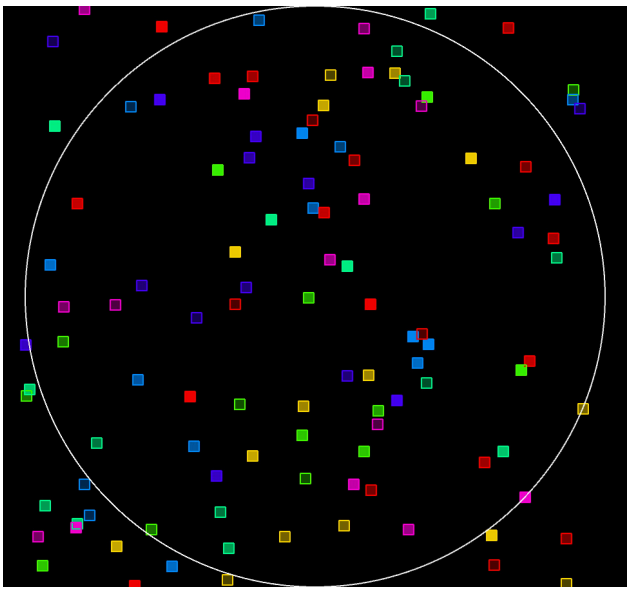

Figure 1: Randomly arranged notes in handout code

Graphical coding for this course is in Processing [3-6], which is a framework consisting of library classes, functions, coding conventions, and an integrated

development environment (IDE) in Java. A generated Java class encapsulates the programmer's *sketch* code. A timer-driven loop calls the programmer's draw() function periodically at the *frame rate*, by default every 60th of a second. By clearing the display and redrawing graphical shapes at slightly different locations on each invocation of draw(), the programmer can create animations.

Processing code can access the full Java class library, including the Musical Instrument Digital Interface (MIDI) [7-8] classes in package javax.sound.midi [9]. In this assignment project, pressing on a mouse button toggles the Processing *mousePressed* Boolean variable to true. A loop within the application's periodic draw() function finds the closest graphical note within Figure 1 to the pressed mouse and invokes its display() function. The note's display() function performs two actions: It displays a 3D graphical shape, centered at the 2D location of the closest note in Figure 1, and it sends a MIDI *noteon* message with the pitch and volume as parameters to a sound synthesizer selected at program start-up time. For the assignment we used software sound synthesizers built into the Java MIDI library. Releasing the mouse re-invokes the note's display() function, informing it to silence the sound with a MIDI *noteoff* message, and to return to displaying a 2D rectangle.

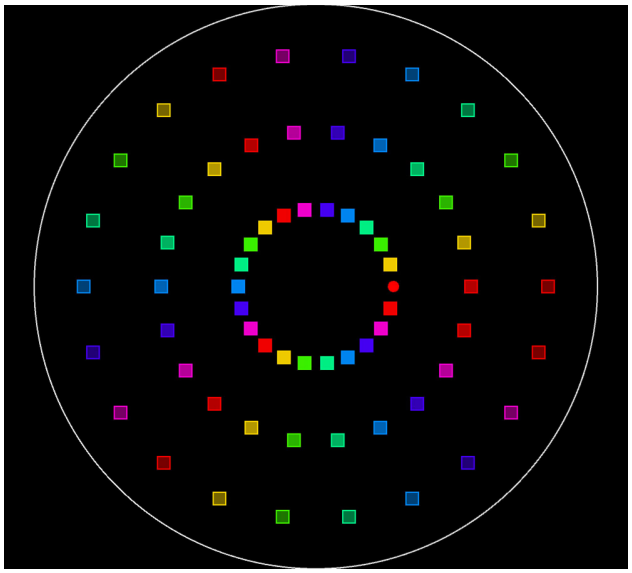

Figure 2: Notes arranged using polar geometry

The first assignment requirement was to arrange the notes in a logical order that would be apparent to a musician playing this virtual instrument. The author supplied pseudocode and a library of polar geometry functions for converting device coordinates to polar coordinates and vice versa. Polar geometry uses a normalized coordinate dimension of rotational amount in the radian range [0, 2π) for longitude around the circle from a reference ray (the line from the center to the rightmost edge of the circle), and a normalized dimension of [0.0, 1.0) for the latitude distance from the center at 0.0 to the perimeter. *Using normalized coordinates is a key design practice for distributing a graphical design across computer displays of varying pixel ratios*. The first step in interpreting a mouse location is converting from device coordinates to normalized polar coordinates; the last step in displaying a graphical object is converting from normalized coordinates to device coordinates. The author's polar geometry library functions convert using the pixel resolution of the device running the code. The library also supports normalized Cartesian geometry with X and Y coordinates in the range [-1.0, 1.0], and their conversion to device coordinates, for applications that bound the interaction area with rectangles.

In Figure 2, we mapped the pitch of each note to its longitudinal location around the circle, and its volume (MIDI *velocity*) to its distance from the center, with the loudest notes in the central ring. Three notes of the same color along a given ray going out from the center have the same pitch with decreasing volume. The most bass note is the red "do" note along the ray from the center to the rightmost edge of the circle; pitch increases counter clockwise, with the most treble note also being a "do" note just before the most bass note.

In this first assignment, students also selected a MIDI instrument and an audio effect (MIDI *controller*) to customize their instrument sounds [7], and they replaced the author's 3D box (cuboid) with a custom 3D shape to display when a note sounded due to a mouse press.

3. Applying Model-View-Controller

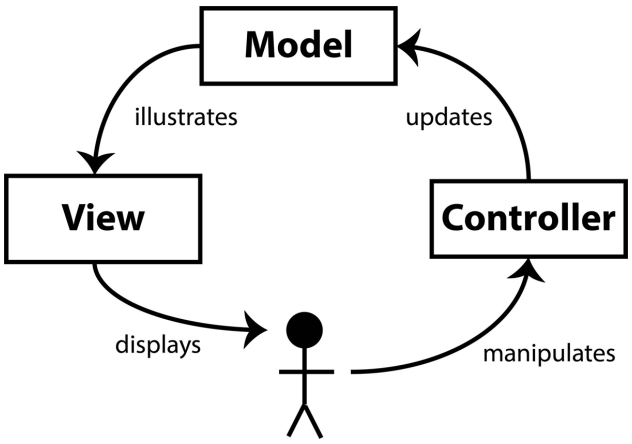

Figure 3: Model-View-Controller design pattern

Figure 3 illustrates the Model-View-Controller design pattern as it applies to the design of graphical user interface (GUI) based applications [2]. A user triggers input data events via sensors such as key presses and mouse manipulation, sending these events and their data to a code module comprising the *Controller*. The Controller updates the state of the application that resides in the *Model* module. The Model updates a sensory display that embodies the *View*, illustrating the updated state of the Model for the human user.

Figure 4 illustrates the Model-View-Controller design pattern as it applies to this application. Mouse movement updates Processing's *mouseX*, *mouseY* coordinate variables using device coordinates. Processing supplies *mousePressed* and *keyPressed* boolean variables that are true whenever these states are entered (the mouse or a keyboard key is pressed), and it invokes optional, programmer-supplied *mousePressed*() and *keyPressed*() functions whenever those events occur.

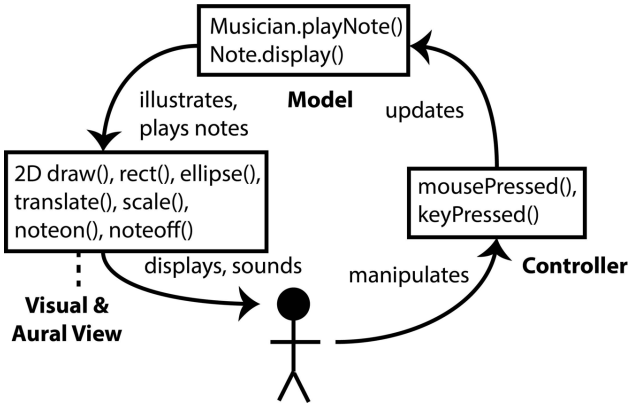

Figure 4: MVC for our stand-alone Processing sketch

The periodic draw() function of this single-process application finds the closest graphical note to mouseX, mouseY when mousePressed is true. Originally, we required a note's rectangular bounding box to enclose the mouse location before considering it a match. We later found that *computing the distance from the mouse to the nearest note using Processing's dist() function, and triggering that note when mousePressed is true, makes manipulation of an Android tablet much easier than requiring mouse containment in a note's bounding box*. On the Android tablet, pressing the display surface with a finger or stylus constitutes a mouse press. The periodic draw() code to locate and trigger note display when mousePressed is true is the primary implementation of Controller code. We also used code in the keyPressed() function to set global properties in the Model state such as display versus omission of the guide circle.

As implied by Figure 4, there are actual Musician and Note classes in this application. There are potentially up to 16 Musician objects, one for each of up to 16 MIDI channels, where a channel maps to a configurable, typically unique instrument sound. In the stand-alone program, Controller code invokes Musician[0].playNote() for every Note object, using a parameter to inform the Musician whether that Note object is being played. Musician.playNote() sends MIDI messages to the sound synthesizer for setting up its channel's instrument voice and audio effects, then invokes Note.display() for each Note object passed to it by the Controller. Note.display() draws a custom 3D shape and sends a *noteon* message to the sound synthesizer when the Note is first selected. It sends a *noteoff* message when the Note is first deselected, and it displays the default 2D square when the Note is not being played, colored according the Note's position in its musical scale, with brightness according to its MIDI velocity (loudness). The View in this sketch is both the graphical visual display and the MIDI sound synthesizer aural display.

3. Distributing Model-View-Controller

There were three very important design constraints for our distributed system: 1) Make this semester's Android remote control client graphical in a way that reflects the presentation of the server that it controls, 2) Avoid the need for server-to-client messages to synchronize visual state, in the interest of efficiency, and 3) Reuse as much of the code from the Figure 4 system as possible. Support for multiple client devices readily falls out of our architecture, as we will see.

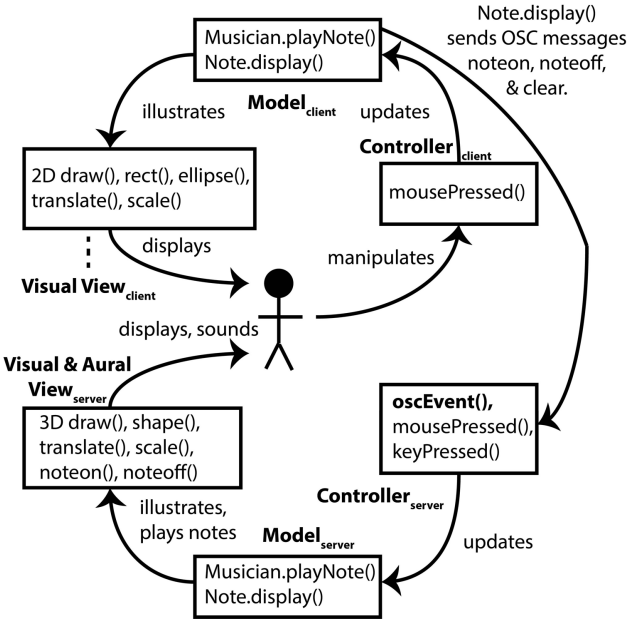

Figure 5: MVC for our distributed Processing system

3.1 The server-side design

In the second assignment for this application, the author supplied the server solution code appearing in the bottom half of Figure 5 (notice the **server** subscripts in the bottom half and the **client** subscripts in the top), and also supplied the solution to the previous, stand-alone sketch, along with instructions for the students to modify this sketch into the remote-control client.

Both the server and client sides of Figure 5 started out as identical code from the Figure 4 solution. The primary design enhancement in the server of Figure 5 is the addition of *oscEvent*() to Controller functions mousePressed() and keyPressed(). This new function is the receiver interface for Open Sound Control (OSC) [10], a distributed data transport mechanism that supports the passing of string

commands and an array of data strings, integers, and float numeric values between applications. Processing's oscP5 library [11] sends and receives User Datagram Protocol (UDP) datagrams [12] containing OSC messages with these data types. The Processing server starts an oscP5 listener thread that receives incoming OSC messages addressed to the server process; the oscP5 listener thread invokes *oscEvent*() when messages arrive. Given the fact that, like most GUI frameworks, Processing data structures are not safe for concurrent access, it is necessary for *oscEvent*() to insert incoming OSC messages into a thread-safe queue for later polling and processing from the main Processing thread's invocation of draw().

Initially, when a client starts execution, it sends its identifying OSC *client* message to the server, which responds with a *server* message, confirming receipt. Thereafter, all messages are remote control commands from the client to server, via this client-side function:

void sendOSCMessage(String command, int midiChannel, int pitch, int velocity)

Commands include *noteon*, *noteoff*, and a several *clear* variations for clearing so-called stuck notes. UDP does not guarantee message delivery, and very occasionally, the wireless LAN will drop a *noteoff* message, leaving a Note sounding indefinitely on the server. The clear messages can shut off all sounding Notes from a client, or all Notes on the server. Playing a Note from a client restarts it, so clearing is a transient action.

Parameters midiChannel, pitch, and velocity make it clear that OSC is serving as a proxy transport for messages to the server's MIDI synthesizer. An incoming server *oscEvent*() invocation supplies the arguments passed by a client via sendOSCMessage. The server's *oscEvent*() Controller logic identifies the Musician (via the midiChannel) and Note being played or silenced by an incoming *noteon* or *noteoff* message, and invokes Musician.playNote() with parameters supplied by the OSC message. Musician.playNote() deals with Notes played by the local mouse and by OSC messages identically.

A critical aspect of this distributed design is that *it must be possible to identify a Note object in the server's Model without reference to its graphical/geometric location*. The reason is that the server may apply creative graphical manipulations such as display rotation, scaling, animation, and 3D effects that are not sent back to the client. The client maintains a much simpler, 2D representation of the system, both for efficiency on an Android tablet, and to avoid congesting the wireless network with server-to-client state updates. The client sends proxy MIDI commands. The server does not update the client.

We achieved this critical design aspect by making each Musician object identifiable by its unique midiChannel in an array (there are only 16 channels), and each Note object identifiable by its composite pitch-velocity key in a Java HashMap. OSC messages trigger function calls in the appropriate Musician and Note objects, regardless of graphical locations, via these Musician array indices and Note HashMap keys. In general, in order to minimize the processing demand and synchronization overhead that would be required by server-to-client state updates for multiple client devices, it must be possible to address state-bearing objects within the server Model independently from their graphical locations and other properties in the server. Of course, Notes in the server still maintain location for display and spatial correlation with mouse presses in the server user interface, but these spatial attributes do not cross the OSC network.

The only other server enhancements beyond Controller OSC message handling are graphical and musical features that do not affect the clients. Figure 6 shows that the server constructs and displays Note objects for all possible scales – this is the chromatic scale in Western music, comparable to all the notes on a piano – and adds elaborate 3D graphics. The apparent pillars on the right side of Figure 6 are single notes being sustained. The Note.display() function stacks multiple visual note shapes vertically during play, and it plots a translucent copy of its note shape across the entire display area, overlaid with translucent copies of other sounding note shapes. This server architecture accepts concurrent messages from multiple clients via *oscEvent*(), calling Musician.playNote() indexed by each client's MIDI channel, and having those Musician objects concurrently display/play Note objects keyed by pitch and velocity. The single-Musician system of Section 2 has become an ensemble.

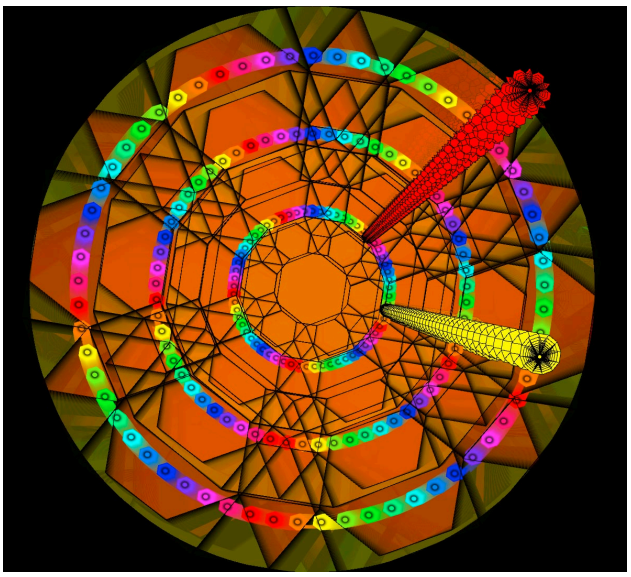

Figure 6: A three-dimensional server-side display

3.2 The client-side design

The top half of Figure 5 illustrates the client side of the design. We developed it on PCs and laptops, and then

cross-compiled for Galaxy S3 Android tablets for testing and deployment [13]. Processing has an Android mode built into its IDE that simplifies the job of building for Android.

Student instructions for porting the Section 2, stand-alone sketch to the client included the following main points.

A. Convert all 3D graphical display calls in the starting code to their 2D counterparts. The client uses strictly 2D graphics for efficiency on tablet computers.

B. Take out all of the MIDI library imports and function calls. The Android device does not support Java's javax.sound.midi library, and the goal in the architecture of Figure 5 is to use the tablets as remote controls for graphics and music generation on the server. We commented out MIDI calls in the client Musician and Note classes in order to document where *noteon* and *noteoff* calls occur.

C. Add a call within the programmer-supplied Processing setup() function to start an oscP5 listener thread, and add a call within draw() to manage menu-based user interfaces and the initial reply receipt from the server. The user interface steps through a series of menu states in a state machine, beginning with establishing contact with the server, through updating various client configuration parameters. Figure 7 shows the client GUI for setting the server's IP address and UDP port number. While most user interaction is via graphical manipulation of the Android display, client configuration parameters are set via menus, as they were in the previous offering of this course. By using Processing source code *tabs*, the author encapsulated all OSC and menu code in a way that eliminated the need for students to understand it. A Processing tab is simply a subset of a Processing sketch's source code. A sketch can have one or more tabs. When a programmer compiles and runs a sketch, the preprocessor combines all tabs into a single source class for the sketch, but programmers do not see that aggregation of source code. Processing's tab mechanism provides a means for hiding code that does not need to be visible to the students, simplifying their tasks. The single function calls added to student setup(), draw(), and Note.display() invoke functions defined within an author-supplied source tab.

D. Within Note.display(), replace commented MIDI *noteon* and *noteoff* function calls with calls to sendOSCMessage() as discussed in the previous section. OSC messages bear the same data as MIDI messages, from each client to the server. Musician.playNote() code for setting MIDI instrument types and effects is no longer used in the client. The server keyPressed() Controller interface supports configuring Musician objects.

E. Insert a few ancillary lines of code into draw(), for example "orientation(LANDSCAPE)" to guarantee stable orientation on a tablet (this runs but has no effect on a PC), and substitution of a client MIDI channel variable set by the user interface for the MIDI channel used as a Musician array index on the server. As in the previous assignment, there is only one Musician object on the client, and the user can change its MIDI channel via a Menu. The server supports up to 16 MIDI channels, and therefore 16 concurrent Musicians / instruments.

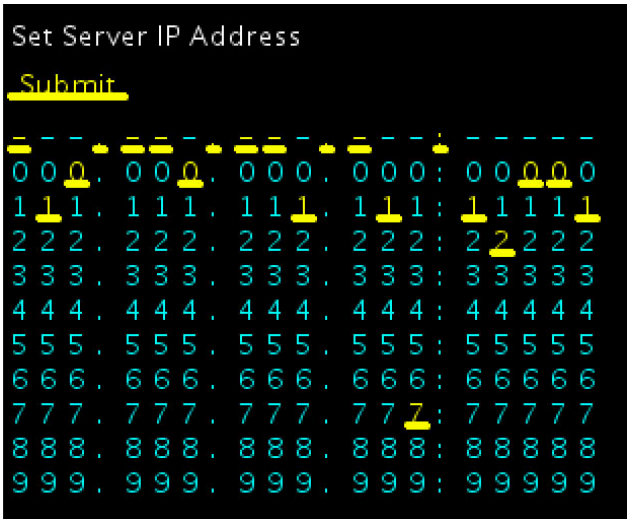

Figure 7: Client menu for setting server LAN address

To summarize the event and data flow in Figure 5, a user of a tablet client uses a finger or stylus to configure client parameters via menus, and then to manipulate Note objects via the interactive graphical display. The finger or stylus acts as the mouse. Note.display() invocations arrive at a client Note object, similarly to the Section 2 code and this section's server, but client Note.display() now acts differently. It turns its 2D rectangle into a circle, but more importantly, it invokes sendOSCMessage() to send MIDI *noteon* and *noteoff* data to the server. Thus, Note as part of the client's MVC Model sends visual function calls to the client View, and it sends musical function calls via OSC to the server's Controller. The server interprets incoming OSC messages similarly to mousePressed() events, locating the Musician object via the MIDI channel sent from the client, and locating the Note object by using the message's pitch and velocity as a compound key. The server invocation of Musician.playNote() and Note().display is identical for local mouse presses on the server and for OSC events arriving from one or more clients.

Figure 8 shows the client-side musical GUI. As in the server, Notes are arranged in a circular order by pitch, and outward from the center by velocity. There is one round note being sounded by a mouse or finger press in Figure 8. There are fewer Notes in the client's Figure 8 than there are

in the server's Figure 6 because the server must support all possible notes in the Western chromatic scale, i.e., the white and black piano keys within a five-octave range. The client must support only a three-octave range for the scale that it is playing. In performance pieces where all clients play exactly the same scale, we configure the server via keyPressed() commands to use exactly the same scale as the clients. This update to the server's Model propagates to the server's View, and there are then the same number of notes on the server as on the clients. The general design constraint is that *the server must support the union of all client configurations*, in this case musical scale and octave range.

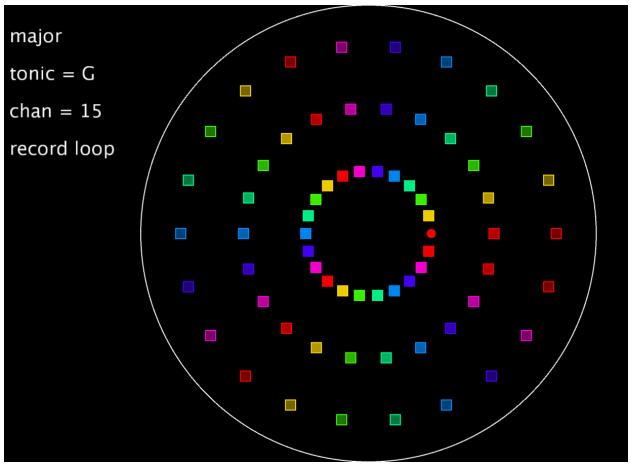

Figure 8: A two-dimensional client-side display

On the left side of Figure 8 are text boxes for setting scale (major here), the tonic "do" note (G), the MIDI channel (15), and the ability to record and play back loops of note presses. Mouse-clicking one of those text areas brings up its menu GUI for user reconfiguration of the client.

When in record mode, each sendOSCMessage() call saves an object containing its arguments into a timed sequence, in addition to sending an OSC message to the server. When in playback mode, a utility function invoked from draw() sends the saved sequence of notes to the server via sendOSCMessage() at the recorded times. The author added this looping enhancement after completion of the course. There are opportunities for additional enhancements to the client, such as playing chords (multiple notes) when a user triggers a note. The system as described supports multiple client devices used as distinct musicians, typically using unique MIDI channels mapped to unique instrument synthesizers on the server.

4. Other possible uses for distributed MVC

The general flow of events and data outlined for Figure 5 can apply to other applications. For example, the author and two students are in the midst of developing a graphical gravity simulator for a planetarium dome. A client user manipulating an Android tablet selects a new planet of a user-configured planetary type, size, and mass/density, and then swipes it into a planetary system display on a server. The server uses orbital mechanics to determine whether the incoming planetary mass is captured in an orbit by the server's star, or falls into the star, or escapes the system [14]. The final client GUI will include exemplary planets that are scaled and injected into the system by finger strokes. We are using OSC messages to send planet injection data to the server. As in the client of Figure 8, manipulation within the guide circle interacts with the server's graphical state (e.g., stellar system), and manipulation outside the guide selects configuration parameters (e.g., planetary properties). For Cartesian geometry systems, a guide square centered in the display replaces the guide circle of the current application.

The author also exhibits juried, computer-generated art videos, and is with working with digital art students. One problem with creating videos from interaction with stand-alone graphical applications is that, depending on the mode of video capture, the user's mouse cursor shows up. Distributed painting using tablet devices eliminates that problem. The painter controls the paintbrush configuration parameters, brush location, and brush application on a client tablet, which sends paintbrush data via OSC to a server that implements the canvas.

Efficient implementation requires that there not be a plethora of server-to-client messages clogging up the LAN. The present design avoids that potential problem. The only server-to-client message is confirmation of the initial client contact. This requirement is different from that of a networked graphical game, where distributed client GUIs must reflect the detailed state of the game server. In the present approach, clients and servers are all in the same room using a single LAN. Users can look at the server's display to perceive the server's View of its Model.

5. Conclusions

Lessons learned start with using a modular Model-View-Controller design as diagrammed in Figure 4, with the intent to clone-and-enhance it to the distributed MVC architecture of Figure 5.

Next, use normalized coordinates in the ranges [0, 2π) longitude and [0.0, 1.0) latitude for polar geometric arrangements, or the linear range [-1.0, 1.0] for Cartesian X, Y arrangements, converting them to device coordinates at the last step before graphical display. This normalization of geometry saves a lot of conversion coding when identical or isomorphic Views must appear on server and client displays with different width-to-height pixel ratios.

Minimize the number of server-to-client communications, ideally only to confirmation of initial contact. Doing so avoids congestion on the LAN and overly complex synchronization code.

In hand with the previous paragraph, keep graphics on a client tablet as simple and efficient as possible. Comparing Figure 6 to Figure 8 shows that it is possible to direct sophisticated graphics on a compute-intensive server from much simpler graphics on a client tablet.

Design a key-based mechanism for identifying server objects on the clients, such as the MIDI channel for identifying Musician objects, and the pitch-velocity compound key for identifying Note objects in this system. Doing so decouples the geometric locations of objects on the server from locations on clients. Given the fact that there may be multiple layers of nested geometric translations (moves), rotations, and scales on server and client graphical devices, communicating via location data would require multiple geometric transforms. It is much simpler to avoid the need for keeping track of geometric transforms at the system level of Figure 5.

6. Acknowledgements

The author would like to thank the Faculty Professional Development Grant Committee (FPDC) of the PA State System of Higher Education for funding summer work on this project, as well as funding the students working on the graphical gravity simulator outlined in Section 4.

References:

[1] A. Colubri, *Processing for Android* (New York, NY: Apress Media, 2017).

[2] G. Krasner & S. Pope, A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80, *Journal of Object-Oriented Programming, 1*(3), August/September 1988, 26-49.

[3] C. Reas & B. Fry, *Processing: a programming handbook for visual designers and artists*, Second Edition (Cambridge, MA: MIT Press, 2014).

[4] D. Shiffman, *Learning Processing, a beginner's guide to programming images, animation, and interaction*, Second Edition (Burlington, MA: Morgan Kaufmann, 2015).

[5] Processing home page, https://processing.org/, link tested January 2020.

[6] Anonymous reference by the present author.

[7] MIDI Technical Fanatic's Brainwashing Center, http://midi.teragonaudio.com/, link tested January 2020.

[8] MIDI Association, the official MIDI specifications, https://www.midi.org/specifications, link tested January 2020.

[9] Oracle Corporation, package javax.sound.midi, https://docs.oracle.com/javase/8/docs/api/javax/sound/midi/package-summary.html, link tested January 2020.

[10] Open Sound Control (OSC) documents and libraries, http://opensoundcontrol.org/, link tested January 2020.

[11]. A. Schlegel, The oscP5 Processing library, http://www.sojamo.de/libraries/oscP5/, link tested January 2020.

[12] E. Harold, *Java network programming*, 4th Edition (Sebastopol, CA: O'Reilly Media, 2013).

[13] Samsung, Galaxy Tab S3, https://www.samsung.com/global/galaxy/galaxy-tab-s3/, link tested January 2020.

[14] A. Milani, *Theory of orbit determination* (Cambridge, UK: Cambridge University Press, 2010).