Visualizing MIDI Data in Real Time

I. Background

The use of computer-generated visuals to accompany music at concerts has surged in popularity in recent years, largely due to the rise of electronic music. While almost any concert by a major artist will feature such visuals, they are typically generated beforehand and simply played back along with the music. This works well with pre-recorded music, such as the background tracks that solo artists sing along to or the remixes that DJs play. However, this setup does not work as well with a live musician playing the piano or synthesizer. It also does not allow for interactive performances or shows featuring improvisation. The purpose of the system described in this paper, Spectral Keys, is to accommodate such performances. The system takes MIDI data gathered from a keyboard and uses that information to produce an image that changes based on what is being played. In this way, the visuals respond in real time to a live performance by a musician.

II. The Foundation: Processing

Spectral Keys was created using the Processing language and IDE. Processing is an open source software sketchbook created by Casey Reas and Benjamin Fry. It is currently distributed by the Processing Foundation, which aims to promote the visual arts through technology. Since Processing is free for anyone to use and build upon, it is a great alternative to expensive commercial software that often comes with limitations. The Processing environment was chosen for this project because its capabilities align with the goals of the system. Processing seamlessly combines the Java programming language and the OpenGL graphics library, facilitating the process of creating the visuals. Processing also allows for the use of a variety of libraries related to gathering MIDI data from a controller and playing back sound. Additionally, it makes it

possible to use Syphon and OSC to both send images to and control projection mapping software VPT 7.

III. Gathering MIDI data with MidiBus

The MidiBus is a library created for Processing that allows a program to collect MIDI data from a MIDI controller. It detects MIDI input devices and allows for the selection of a device. The MIDI device used during the creation of Spectral Keys is a CME M-Key. The M-Key is a 49-key USB keyboard. It connects to the computer using a USB type A to USB type B cable. This cable is used to both power the keyboard and send MIDI data to the computer. Once the desired device is selected, the MidiBus can listen for MIDI messages and send them to the program. When a MIDI Message is received, the midiMessage function is called. Inside this function, Spectral Keys extracts information from the MIDI message, such as the note number and velocity. It then uses this data to produce an image.

```
void midiMessage(MidiMessage message, long timestamp, String bus_name)
{
  int note= (int)(message.getMessage()[1] & 0xFF) ;
  int vel= (int)(message.getMessage()[2] & 0xFF);
  ...
}
```

IV. Keeping Track of Note Data

When Spectral Keys receives a MIDI Message, it first determines whether the key is being depressed or released. The keyboard used for testing the program sends similar MIDI Messages in both cases, making it important to distinguish between the events. Before this was accounted for, the program would always register the same note as being played twice in a row. Because of this, the program was designed to keep track of whether each note on the keyboard is currently being played or not. This information is stored in an array of booleans. When the

program receives a MIDI Message, it first checks to see if the corresponding note is currently listed as being on. If so, it indicates that it is now off. It then decrements the counter keeping track of the number of notes that are being played simultaneously. On the other hand, if the note is currently off, Spectral Keys indicates that it is now on and increments the note counter. It also carries out tasks related to playing the note and altering the visuals appropriately. This is described in further detail in the following sections.

V. Associating Notes With Colors

Once the relevant note data has been gathered, Spectral Keys uses it to visualize the music that is being played. It first determines the color corresponding to the note that has been played. Each note in the chromatic scale has been assigned a color. Starting with the note C, the colors chosen for the notes follow a rainbow pattern. That is, C is red, C#/Db is orange, D is yellow, and so on. This color scheme can be altered by playing a major or minor chord. If a major chord is detected, subsequent notes will be lightened. Similarly, if a minor chord is detected, any notes played after this will be darker in color. Spectral Keys detects chords by checking how many notes are currently being played. If the note counter indicates that three or more notes are being held down at once, the program will loop through the notes that are on and calculate the intervals between them. These intervals are then compared with intervals corresponding to standard major and minor chord positions. The system can detect major and minor chords played in root position, first inversion, and second inversion. Once the note color has been found, the image generated with these colors depends on the mode that Spectral Keys is in.

VI. Creating Images Using OpenGL Library for Processing: Cube Mode

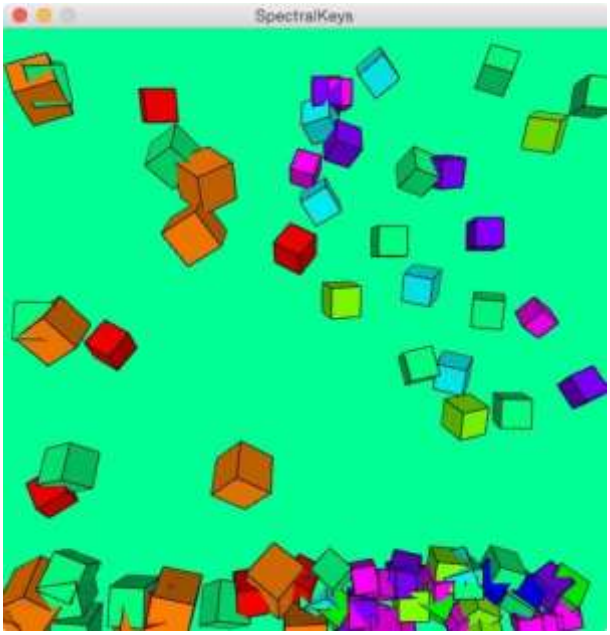
There are two modes to Spectral Keys, the default being Cube Mode. In Cube Mode, the current background color of the image will be transitioned to the color of the last note played. This is done by interpolating between the current background color and the new one. This interpolation results in a less jarring switch between colors. Not only does the note data affect the background color, but it is also used to generate cubes that appear to fall in front of the background. A new cube is created each time a note is played, and the cube's attributes are determined by the note name and velocity. The cube's color depends on the color assigned to that note. Cube color is also affected by the last chord that was played, just like the background color. In this way, each cube's color will match the color that the background was transitioning to at the time of its creation. Other attributes of the cubes are controlled by different aspects of the MIDI data. For example, each cube spins at a rate dependent on the velocity of the note that was played. A note that is played softly will generate a cube that spins slowly, while a note that is played harder will generate a faster spinning cube. Finally, the size and location of the cube depends on the pitch of the note. Higher notes generate smaller cubes, while lower notes create larger ones. The smaller cubes will appear on the right side of the display, while the bigger ones will fall on the left side. The cubes themselves are rendered using Processing's OpenGL library. I have created a class that controls the creation of the cubes. The class takes the relevant data, such as note number, color, and the number of cubes that have already been created, and uses this to properly generate, display and animate each cube.

```
/* The height is the size of the window, and cubes.length is the
number of cubes currently being displayed */

int stopHeight=height-cubes.length;
NoteCube cube1=new NoteCube(vel, color(newColor), stopHeight, note);
...
class NoteCube {
```

```
color cubeColor;
float xSpeed, ySpeed, cubeX, cubeY, xRot, yRot;
boolean frozen;
int stopHeight, cubeSize;
NoteCube(int vel, color c, int h, int note) {
  cubeColor=c;
  if(note < 127/2) {
    cubeX=random(10,250);
  }
  else {
    cubeX=random(250,495);
  }
  cubeY=-50;
  xSpeed=vel/1000.0;
  ySpeed=vel/1000.0;
  xRot=0.0;
  yRot=0.0;
  frozen=false;
  stopHeight=h;
  cubeSize=(int)((127-note)/127.0*30)+10;
}

void animate() {
  stroke(0);
  fill(cubeColor);
  if(!frozen) {
  xRot=xRot+xSpeed;
  yRot=yRot+ySpeed;
  cubeY++;
  }
  pushMatrix();
  if(cubeY>stopHeight) {
    frozen=true;
  }
  translate(cubeX,cubeY);
  rotateX(xRot);
  rotateY(yRot);
  box(cubeSize);
  popMatrix();
}

void drop() {
 cubeY=cubeY+5;
 fill(cubeColor);
 pushMatrix();
 translate(cubeX,cubeY);
 rotateX(xRot);
 rotateY(yRot);
 box(cubeSize);
 popMatrix();
}
```

}

An array is used to keep track of the cubes that have been generated so that they can all be displayed properly as the image is continually redrawn. When a cube reaches the bottom of the screen, it will stop moving. The location in which it lands is determined by the order in which it was created. Cubes that were created first will land closer to the bottom of the screen, while subsequent cubes will land higher up. There is the option to clear the cubes from the screen in case it gets filled. The user can simply press the 'C' key on their computer keyboard. This will trigger the drop function, which will cause all of the cubes to fall out of view. Once no more cubes are visible, the cube array will be cleared.



VII. Another Way to Generate Images: Sphere Mode

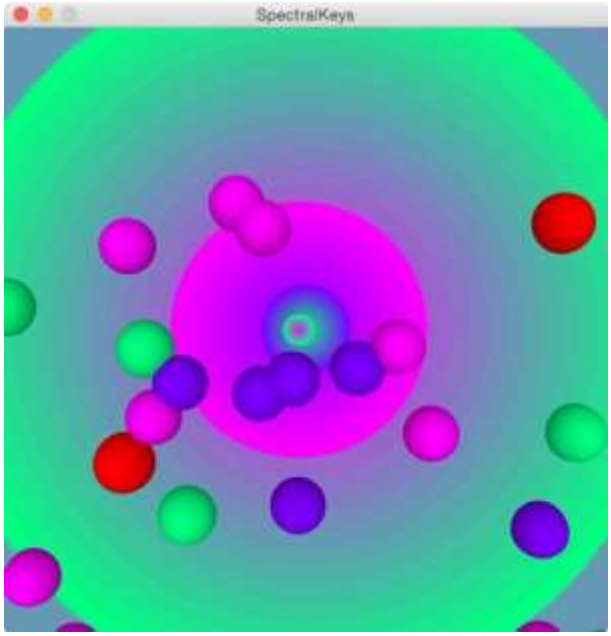
The second mode is Sphere Mode. The user can switch to Sphere Mode by pressing the 'S' key on their computer keyboard. When switching between modes, the status of the previous mode will be saved. This means that when the user switches back and forth between modes, the animations for each will pick up where they left off. If the user wants to reset all of the modes, he

or she can do so by pressing the ‘R’ key on their computer keyboard. This will return both modes to the entirely black screen with which they started out.

Sphere mode is similar to Cube Mode in that a three-dimensional object is falling across the screen. In this case, that object is a sphere. Just like in Cube Mode, the color of the sphere depends on the last note played. Larger spheres also correlate with lower notes and vice versa. However, there are a few differences when it comes to the behavior of the spheres. Once spheres reach the bottom of the screen, they disappear instead of stopping. The speed at which a sphere falls is determined by the velocity of the note played. As a result, louder notes will fall more quickly than quieter notes. Additionally, the x-coordinate of each sphere is not impacted by the pitch of the note. The portion of the display on which each sphere appears is purely random. The biggest difference between Sphere Mode and Cube Mode is that in Sphere Mode, the background does not fade between solid colors. The background consists of numerous circles which fade between the colors of notes that have been played. At any time, five nearly-concentric circles can be seen on the screen. These circles increase in size as more notes are played. A new circle is generated when the largest one comes out of view. This happens once every four notes are played. The circle’s color depends on the color of that note. The circles between the main circles are created by interpolating between the colors of each main circle. This is done to create a fade effect that looks as if the audience is peering down a colorful tunnel.

```
NoteSphere sphere1=new NoteSphere(color(newColor),note,vel);
...
class NoteSphere {
  color sphereColor;
  float sphereX, sphereY;
  int sphereSize;
  float sphereSpeed;
  NoteSphere(color c, int note, int vel) {
    sphereColor=c;
    sphereX=random(-250,250);
    sphereY=-300;
    sphereSize=(int)((127-note)/127.0*30)+10;
```

```
    sphereSpeed=vel/127.0*4.0+1.0;
  }

  void animate() {
    if(sphereY<600) {
      noStroke();
      fill(sphereColor);
      sphereY=sphereY+sphereSpeed;
      pushMatrix();
      translate(sphereX,sphereY);
      shininess(2.5);
      sphere(sphereSize);
      popMatrix();
    }
  }
}

/* x represents the x-coordinate of the center of the circle, and h is
the height (diameter) of the circle. Both parameters vary for added
effect */

Circle newCircle=new Circle(x,h,color(newColor));
...
class Circle {
  float x;
  float d;
  color c;

  Circle(float xLoc, float diameter, color circColor) {
   x=xLoc;
   d=diameter;
   c=circColor;
  }

  void display() {
    fill(c);
    ellipse(x,0,d,d);
  }

  void increaseSize() {
    d=d+d*0.3;
  }
}
```



VIII. Generating Sounds with SoundCipher

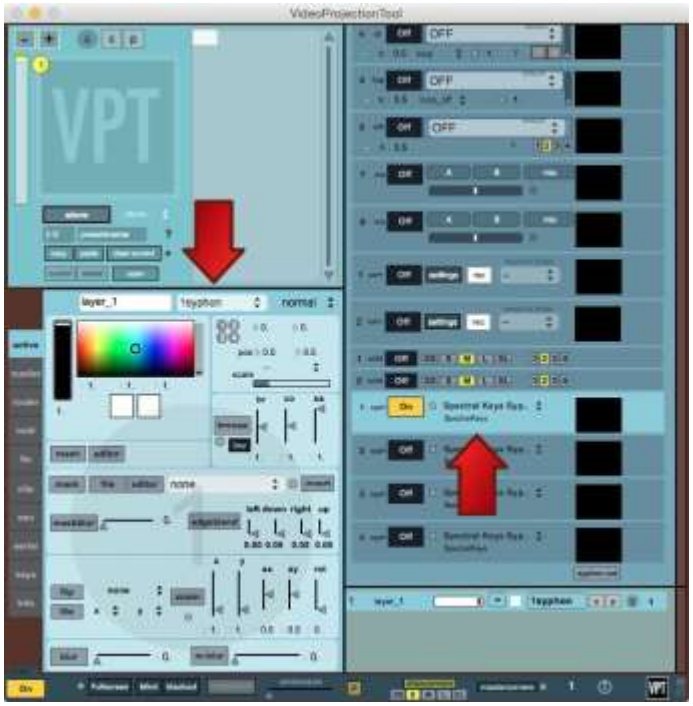
The SoundCipher library is used to generate the sounds heard when the keyboard is played. SoundCipher simply takes the note number and velocity of a MIDI message and uses it to play back the corresponding piano sample at the correct volume. The sample fades after half a beat, no matter how long the key is held down.

IX. Sending Images to VPT 7 Using Syphon

Like Processing, Syphon is an open-source technology. This Mac-specific framework was created to allow applications to share frames in realtime. The Syphon library for Processing is used to send the image generated by Spectral Keys to VPT 7. This library allows for the creation of a Syphon server which then communicates with VPT 7. After each frame is drawn in the Processing environment, it is sent to VPT 7 by the server.

X. Projecting the Image using VPT 7

Once the server is set up to send frames, VPT 7 has to be set up to receive them. On the right hand side of the VPT 7 window, there are numerous channels. Towards the bottom, there is a section for Syphon channels entitled “syph.” The first channel in this section must be turned on, and the server created by Spectral Keys must be selected in its drop-down menu. After this is done, “1syphon” must be selected from the menu next to where it says “layer_1” on the left side of the window.



Once these settings have been selected, the image sent from Spectral Keys appears on the preview and output windows in VPT 7. When the computer is hooked up to a projector, the output window becomes what is displayed by the projector. Using the input window, a mask and a mesh can be set to properly display the image on a three-dimensional surface. In this case, the three-dimensional surface is a box. The mesh allows the image to be wrapped around the box without any distortions, while the mask cuts off any part of the image that cannot be properly

wrapped onto the box. After the mask and mesh have been configured, the box looks as if it is being illuminated from three different sides.

XI. Future Work

Although Spectral Keys is a working system, there are several improvements that could be made in the future. Expanding the functionality of the system would make Spectral Keys more visually interesting as well as useful in a wider variety of situations. For example, additional modes could be added for greater options when performing. These modes would each consist of unique visuals. Perhaps the system could be adapted to keep track of how long the notes are played, and some modes could incorporate this information. Ideally, Spectral Keys would have a large number of modes in order to prevent the audience from becoming bored of seeing the same visuals. With a sizeable collection of modes, a performer could switch between them and complete a full-length performance without spending too much time on any single mode. Spectral Keys could also be improved by adding support for other MIDI instruments, such

as electronic drum kits and guitars. It could even be programmed to respond to multiple different instruments playing at once. This would make Spectral Keys useful for a larger number of musicians. Finally, support for more complex chords should be added in order to better accommodate different styles of music. This would also expand Spectral Keys' usefulness for many musicians. With these changes, Spectral Keys could be used at performances featuring a wide variety of genres of music.