Introduction

Hi. This project is an intro into visualizing music. The main goals of this project were to sync up musical events with graphic events and to use what I have learned to make those graphics more interesting/artistic.

Previous Works

Below are 4 previous works that I had in mind while implementing my algorithms. I took some of their ideas and tried to make them as original as possible.

https://www.shawnlawson.com/portfolio/kessel-run/ [Law14]

https://www.clatcraive.net/2017/11/20/string/ [Lei17]

https://lukasbiewald.com/2015/06/11/why-do-certain-musical-notes-sound-good-together/ [Knu15]

https://www.youtube.com/watch?v=A6s49OKp6aE [Mal74]

The first paper is an example of Shawn Lawson using code to both make music, and display images related to the noises being made. He has it setup to make new images appear as he types in code, like a live interpreter. I think this project has a very good grasp on brightness and I would like to incorporate that into my project. The second paper is from Carla Leitao (an RPI student/alumni)who displays images of what is being played on the instruments live. This animation style is more closely linked to the actual soundwaves than the notes. Even though this isn't the direction I want to go (I want to be midi focused) I still would like to be able to pull off a live element in the project (Most likely extra). The third paper talks about which notes sound well together based off of the relations between their respective waves (C and G sound good because their frequencies make a nice ratio). The fourth link is to the YouTube video by Stephen Malinowski which is the main source that this project is based off of. His work has both of the goals that I set out in the beginning along with some textural aspects to the graphics. His work almost looks how it sounds.

Implementation

Creating the song

Creating the song is probably the hardest thing to do because unless you can find midi files of all the parts of your favorite songs, you have to make them. Luckily for me, I have 15 years of experience in piano and have taken classes on music production/learning how to use programs like Ableton. I recorded 4 separate tracks for 4 separate instruments and then exported those tracks as midi files. I also included an optional 5þtrack for the background.

MidiFile

Midifile is the main library that was used to translate midi data into c++ readable objects. It was made by Craig Sapp and is the whole reason this project is actually feasible.

OpenGL

We've been using it all semester, it is the best way to do graphics in c++.

Data Storage

Note

A Note object stores 3 pieces of data: Whether the note was pressed or released, when the note happened in the song and how hard the note was pressed. This is a simple enough object and it makes sense not to store the note value because we will use it later in the Shape Data object.

Shape Data

The ShapeData class stores all of the data for each graphical entity, each note on the keyboard or each part of the drum kit. The data that we store are, the pitch value, the mesh of which shape it represents, the color of the mesh when off, the color of the mesh when on, and a list of Note objects. There are also OpenGl objects that draw the image which are the draw type (GL_TRIANGLES, GL_LINE_LOOP), the vertex and color data, and vertex and color buffers. As well there is one extra enum called instrument type which tells us which instrument this data is for (Solo, harmony, bass, drums, background). This data structure's job is to update the color using a decay function of some sort. In my example I chose linear, inverse and hyperbolic tangent based decays. The decay would happen while the note was pressed and then go to a sharp linear decay after the note is released. This class also deals

with displaying the object on the screen. More details on the implementation are discussed later.

Instrument

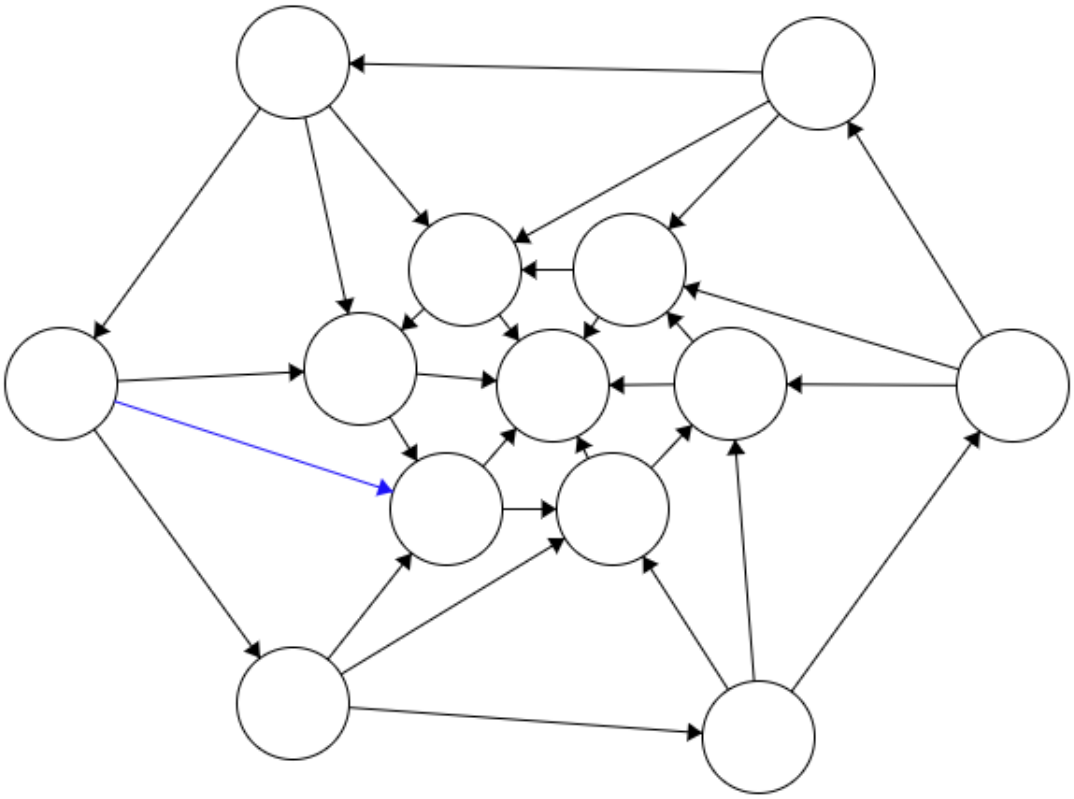
The instrument class is where most of the program happens. This class stores a list of shape data objects, a map from pitch values to note objects and another list of notes that represent the pedal. The main algorithm to read in notes from a file is seen here,

```
file.read(a.pianoPath);
for (int i = 3; i < file.getEventCount(0); i++) {
    notes[file.getEvent(0, i).getP1()].push_back(
        Note(file.getEvent(0, i).getP0() == 144,
             file.getTimeInSeconds(0, i) + a.pianoOffset * 2,
             file.getEvent(0, i).getP2()));
  }
```

We start at 3 because the first three notes in a midi file are used to store data about that file. Then, we take each midi input and extract the 3 pieces of data for a note. file.getEvent(0, i).getP0() obtains whether the note was pressed or released (144 is pressed, 128 is released), file.getTimeInSeconds(0, i) obtains the time that the event happened and file.getEvent(0, i).getP2() obtains how hard the note was pressed, the velocity. The pitches are simply the midi value of each note on the keyboard. These are 21 to 108 from a low A0 to a high C8 going chromatically up the keyboard. There is another check to see if the note is a pedal and some more checks for finding the minimum and maximum notes. In addition to loading and storing this data, this object also creates and organizes the visual meshes.

The Mesh

I used one mesh for every object in the scene. The point of this object is that it would look as if the notes fade into the background. The way I went about designing this mesh was by first taking the mesh I was working with, a circle, and creating a smaller circle of points inside that circle. Then, I color each of the outer points the same as the background and the inner ones the color I am trying to display. This way, when all the triangles are filled in between the points, the natural color interpolation of the triangles will create the illusion of fading. As for connecting up the triangles, here is a diagram that shows which triangles are made,



Updating the Mesh

OpenGL

There are a set of OpenGL commands that need to be run to change the position or color of any object. The full draw function of commands is below,

```
glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(float) * mesh.size() * DIM, vertices,
             GL_STATIC_DRAW);
glVertexAttribPointer(0, DIM, GL_FLOAT, GL_FALSE, 0, (void *)0);

glEnableVertexAttribArray(1);
glBindBuffer(GL_ARRAY_BUFFER, colorbuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(float) * mesh.size() * DIM, colors,
```

```
            GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, colorbuffer);
    glVertexAttribPointer(1, DIM, GL_FLOAT, GL_FALSE, 0, (void *)0);

    // Draw the triangle !
    glDrawArrays(
        drawtype, 0,
        mesh.size()); // Starting from vertex 0; 3 vertices total -> 1 triangle
    glDisableVertexAttribArray(0);
    glDisableVertexAttribArray(1);
```

The important parts here are that we bind our buffer with our vertex data and set the buffer to be the correct size, every time we draw the object. Normally we do this once at the beginning and use shaders or some other transformations to change things but this is a nice hack to not make the OpenGL sections too complicated.

The visuals

The visuals are updated differently depending on which instrument it is but the general idea is the same. Whenever a note is pressed it will decay in brightness according to a decay function and when released it will much more quickly decay according to another function. The three I picked for this example were linear decay, inverse decay $\left(\frac{1}{x}\right)$ and a hyperbolic decay (sigmoid function). I used the inverse decay for the piano drums and bass, the linear decay for the solo and the hyperbolic decay for the background. I found that these choices made the most sense as the instrument I used for the solo did not decay in volume over time. By contrast, held down piano notes get softer because the string vibrates less and less over time.

Organization of the instruments

Keyboard

The keyboard is made with a generalized function. This function takes the note value, the minimum and maximum notes on the keyboard, the center of this graphical shape, the left and right borders and the shape of the note (I only used circles for my example). Making the keyboard is a bit tricky considering that all of the black keys need to be elevated and spaced correctly between the white keys. I implemented a brute force solution where you search to see if a key is white and space them out evenly. If the key is black, you go to the previous white key and place it up and to the right. This involves having a complete list of all of the white keys which looks like this,

```
vector<int> whiteKeys = {21, 23, 24,  26,  28,  29,  31,  33, 35, 36, 38,
```

```
40, 41, 43,  45,  47,  48,  50,  52, 53, 55, 57,
59, 60, 62,  64,  65,  67,  69,  71, 72, 74, 76,
77, 79, 81,  83,  84,  86,  88,  89, 91, 93, 95,
96, 98, 100, 101, 103, 105, 107, 108};
```

Drums

The drums were made manually by spacing the snare toms and cymbals to be about the place that they would be on an actual drum kit. This was mostly artistic guesswork.

Background

The background is just one big copy of the mesh from before, centered at the origin, displayed with GL_LINE_LOOP. I kinda did it by accident, liked the way it looked and kept it :D.

Timing the instruments

The most frustrating issue by far was timing the instruments so that they would be in time with eachother. Each midi file begins during the first measure that notes were played. This means that the two solutions to the problem are to add a dummy note to the beginning of each instrument or to add in offsets for each instrument. This is where we got that a.pianoOffset * 2 factor from before (there are also offsets for bass, drums etc but the code snippet was specifically from the piano section).

Recording the output

Taking the screenshots

Thank you submitty!!! The library used for screenshotting the output was scrot. We can invoke the screenshot command in c++ by using the system function in c++. This function executes a command as if it was the terminal. Therefore with the code,

```
std::string command = "scrot " + screenshot_name + " -u";
std::system(command.c_str());
```

We can take multiple screenshots of the program at specific times. These times are calculated by taking into account the beats per minute of the song, and the fps of the resulting video. Therefore the time step between each frame should be, $\frac{bpm}{120*fps}$. The extra factor of 2 comes from the fact that 1 measure goes from 0 to 0.5 instead of 0 to 1.

FFMPEG

Now that we have our possibly tens of thousands of screenshots, we need to compile them into one video. Luckily for us, FFMPEG can do just that in one command. Similarly to the previous section, we can again use the system function to run the command,

```
command = "ffmpeg -framerate " + to_string(fps) +
          " -i /home/oem/Documents/Code/Music-Visualization-CS4530/src/"
          "screenshots/img-%d.png -vb 20M video.avi";
std::system(command.c_str());
```

which makes a video with the correct fps and saves it to the running location.

Finalizing the video

The final part to do is add in the audio to the video portion. I did this using Filmora but if there is another video editing software that is more convenient, use that. The audio export should be perfectly in sync with the visual if both start at the same time. After that, I personally used OBS to record my output but exporting the video through your software would also work.

Results

The first iteration and the final product can both be seen at the two respective links,
https://youtu.be/7cPKPaWNnkg https://www.youtube.com/watch?v=53PX2VOUbGE

Future Work

Motion

One of the things that was lacking in my implementation was the element of motion. This was apparent in every other reference I gave and mainly represented time or sound waves (more agressive sounding music would move more). Any and all of the transformations would be good for this project. The ones I thought of were slight nudges whenever a note is played, slow rotations for the background and maybe some zooming for the drum kit.

Opacity

This is something that would have made more colors/overlapping elements much easier with the current data structures I have. With more than an off and on color in mind, I would have

organized my data better such that I would be able to better accommodate these features (I did actually try and program this part but It was too buggy before I had to make the final video so I scrapped it)

General color

General color is another way of saying I wanted to be able to tie the color of a note to each note press, not to the note as a whole. This would make it much easier to have more than one color on each element.

More Music

I do wish I had more time before the due date to make more music to visualize. The reason I didn't is mostly because I need to record a whole new song with 4 parts in order to visualize it which takes days, sometimes weeks to get it to a point where I am happy with releasing it. As well, there is the added hour or so of screenshot taking where I cannot focus on any other window on my computer (or else scrot will take a screenshot of that screen).

Conclusion

This project took me 4 coding sessions worth of work, roughly 10 hours each. That added with the additional time from making the song and getting the cmake file working puts this project at around the 50 hour mark. I learned how to use Midifile, more about OpenGL, using multiple libraries in cmake, the system command, scrot, ffmpeg and even more about the midi file format. It was super fun to do and I am very glad I had the chance to learn so much!

Cited Sources

References

[Mal74] Stephen Malinowski. *Music Animation Machine*. 1974. URL: https://www.musanim.com/. (accessed: 04.06.2021).

[Law14] Shawn Lawson. *Kessel Run*. 2014. URL: https://www.shawnlawson.com/portfolio/kessel-run/. (accessed: 04.06.2021).

[Knu15] Donald Knuth. *Why Do Certain Musical Notes Sound 'Good' Together*. 2015. URL: http://www-cs-faculty.stanford.edu/~uno/abcde.html. (accessed: 04.06.2021).

[Lei17] Carla Leitao. *STRING*. 2017. URL: www.clatcraive.net/2017/11/20/string/. (accessed: 04.06.2021).