Mike Cottrell College of Business

Department of Computer Science & Information Systems

CSCI 4950 – Senior Project

Bryan Hutto

**Assignment 4**

(Proposed Design/Implementation)

The MusicScroller application is built to run on Windows 10. The Implementaions can be best described as three distinct actors, detailed below, which perform the key functions of image processing, data analysis, and scrolling.

Figure 1 Simple Visualization of Program Structure

**The Image Handler**

The image handler accepts three parameters from the user: the tempo, the time signature, and the path to the file. These are all that will be required to map the correct scroll rate to the piece of music.

The program utilizes Magick++, a somewhat arcane but extremely versatile API which allows C++ calls to the ImageMagick image-processing library. Both it and its dependencies are open-source. The API allows us to convert PDFs, otherwise nearly impossible to rasterize, into a manageable BMP format with uniform color depth and bytes per pixel. This is then saved to a temporary folder where it can be further operated on. If the PDF contains multiple pages, as will almost always be the case, they are saved as separate BMP files and will be the basis for our Page class.

The newly created BMP file is then read into the input buffer, after stripping and reserving the relevant metadata, and each is converted into an unsigned short matrix where, for every pixel in the original, a 0 represents opacity and a 1 the lack thereof.

A page object can then be instantiated and its member, EnumMeasures(), can be passed the matrix. The newly created Page object is then pushed into a vector and the next BMP file can be read.

try{

list<Image> imageList;

readImages(&imageList, location);

numImages = imageList.size();

for (int g = 0; g < numImages; g++) {

size\_t dep = 24;

imageList.front().depth(8);

imageList.front().alpha(false);

imageList.front().type(TrueColorType);

imageList.front().defineSet("bmp:bit-depth", "8");

imageList.front().type(Magick::GrayscaleType);

imageList.front().write(tmpFile.c\_str());

imageList.erase(imageList.begin());

}

*Converting a PDF to BMPs with Magick++*

**The Page Class**

The bulk of the Page Class is devoted to the EnumMeasures() method, which is responsible for enumerating and locating every line and measure in the page of sheet music, the protocol for which I will detail briefly.

The method implements nested for loops to iterate through each “pixel” in the matrix, checking for a zero (opacity). A Checker variable will preempt the next step if a measure divider was encountered recently on the same horizontal plan (representing a thick bar or a double/triple bar). Those conditions met, a while loop iterates down the y-axis checking for more zeroes, incrementing the height variable for the opaque segment. If the height of the traversed segment is near enough that of the rowFactor variable, it is presumed to be a bar.

[The rowFactor is initially calculated based on image height, but is replaced if a longer bar is found. Since a bar is unerringly encountered first, this prevents abnormally tall note stems from throwing off the algorithm.]

Once a bar is confirmed, the method checks if the bar is on a new line by comparing the difference of the previous height with the all-important rowFactor. If successful, a new row is recorded. Otherwise the current row’s measure count is incremented. In either case the coordinates are stored.

Despite attempts to provide tolerance for slanted lines and photocopy marks, this algorithm does not perform well for low quality scans, and can only be relied upon with perfect accuracy in the case of computer-generated sheet music. Fortunately, this is one of the most common forms, especially in the realm of sharing original arrangements.

if (prevMeasures.size() > 0 && abs(prevLevel - u) < barThreshold \* .5) {

for (int pr = 0; pr < prevMeasures.size(); pr++) {

if (abs(prevMeasures[pr] - y) < width / 20 || abs(y-width)<(width/100)) {

checker = false;

}

}

}

while ((data[y][tmpu] == 0) && checker) {

count++;

++tmpu;

if (count > barThreshold) {barThreshold=count;}

}

if (count >= barThreshold \*.95) {

if (prevLevel == 0 || abs(prevLevel - u) < barThreshold/2) {

prevMeasures.push\_back(y);

measures[rowCount-1] = ++currRowMeasures;

if (prevLevel == 0) { prevLevel = u; newRows.push\_back((tmpu+u)/2);}

totalMeasures++;

}

else {

newRows.push\_back(u);

prevMeasures.clear();

currRowMeasures = 0;

prevMeasures.push\_back(y);

measures[rowCount - 1] = measures[rowCount - 1] - 1;

measures.push\_back(++currRowMeasures);

rowCount++;

prevLevel = u;

totalMeasures++;

}

}

Figure 3 Bar-detecting while loop

**The Scroller**

The Scroller portion of program takes the data we’ve stored in Page objects and translates them into the correct scroll speed for each section of the piece.

First, it opens the original PDF file in Google Chrome, chosen for its fine scroll resolution, which avoids any jerkiness in the moving image, and it’s PDF capability.

The program then calculates the seconds per measure from the beats per minute (tempo) and beats per measure (time-signature) provided by the user. This represents the amount of time the musician will spend playing each measure. By accounting for the number of lines and the number of measures in each individual line, we can judge how long each row should be in the center of the screen.

By testing singular mouse wheel scrolls (WM\_MOUSEWHEEL = 120) over different PDF files, I’ve come up with a reliable factor which describes how much scrolling (we convert it to 1/30 the basic scroll for smoothness) is required to completely clear a PDF page when multiplied by the page height. This is represented by the ticksPerPage variable. Based on the seconds per measure and measures per row, we can calculate the seconds, or rather milliseconds at this point, each small tick should take.

To handle these many small delays the scroller calls a synchronous thread to sleep for the required time per tick.

However, the vertical distance between lines is not uniform, so the program must also account for the height of each new line, relative to the page height. This factor is multiplied by the ticksPerPage to determine how many ticks it will receive (a greater distance between lines means a smaller factor and a faster pace).

Testing revealed a factor accounting for the current measure was insufficient, resulting in too many sharp speed-ups and slow-downs when either the distance between lines or the measures per line (and therefore the scroll speed) changed. Instead it now implements a rather complicated algorithm which takes into account: the vertical distance from the previous, current, and next line and the measure count of the previous current and next line (including between pages). This also means the scroll will gradually speed up when approaching a line with fewer measures, giving the musician time to register the change.

for (int line = 0; line < pStor[y].getMeasures().size(); line++) {

mCount = pStor[y].getMeasures()[line];

cout << endl << mCount << endl;

if (line == pStor[y].getMeasures().size() - 1) {

if (y != pStor.size() - 1) {

nextmCount = pStor[y + 1].getMeasures()[0];

}

else {

nextmCount = mCount;

}

}

else {

nextmCount = pStor[y].getMeasures()[line + 1];

}

if (line == pStor[y].getMeasures().size() - 1) {

if (y != pStor.size() - 1) {

rowH = ((rowSpots[line] - rowSpots[line - 1]) / 2) + (((pStor[y].getHeight() - rowSpots[line]) + (nextRowSpots[0])/2));

}

else {

rowH = pStor[y].getHeight() - rowSpots[line];

}

}

else if (line == 0) {

if (y == 0) {

rowH = (rowSpots[line] / 2) + ((rowSpots[line + 1] - rowSpots[line]) / 2);

}

else {

rowH = ((rowSpots[line] + pStor[y - 1].getHeight() - prevRowSpots[pStor[y - 1].getMeasures().size() - 1]) / 2) + ((rowSpots[line + 1] - rowSpots[line]) / 2);

}

}

else {

rowH = ((rowSpots[line] - rowSpots[line - 1]) / 2) + ((rowSpots[line + 1] - rowSpots[line]) / 2);

}

double thisFactor = rowH / pStor[y].getHeight();

double spl = spm \* mCount;

double ticks = thisFactor \* ticksPerPage \* 30;

int pace = int(spl \* 1000);

double msPerTick = pace / ticks;

for (int r = 0; r < ticks; r++) {

if (!GetAsyncKeyState(VK\_ESCAPE))

{

Scroller scroll(msPerTick, &timer);

mouse\_event(MOUSEEVENTF\_WHEEL, 0, 0, -MINWHEELMOVEMENT / 30, 0);

}

else {

goto label;

}

}

Figure 4 Verbose scroll optimization