April 2006

HyperScore MQP

Benjamin Alexander Payne
Worcester Polytechnic Institute

Gregory C. Tomek
Worcester Polytechnic Institute

Matthew V. McGonigle
Worcester Polytechnic Institute

Project Number: CS-GTH-0510

Harmony Line MQP

A Major Qualifying Project Report

Submitted to the Faculty

Of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

By

_____

Matthew V. McGonigle

_____

Benjamin A. Payne

_____

Gregory Tomek

Thursday, April 27th, 2006

Approved:

_____

Professor George T. Heineman, Major Advisor

Abstract:

Flash, JavaScript, and Java are powerful tools that can be used together to make web applications of high quality. Our group has created a prototype to show the proof of concept of whether these technologies can be used together to recreate the experience of the standalone application HyperScore. This proof of concept will serve to show the feasibility of using this model as the primary distribution of the application.

Table of Contents:

1 INTRODUCTION

Harmony Line, Inc., located in Cambridge, Massachusetts, commercializes music technologies that unlock the expressive musician and composer in everyone [1]. Their flagship product, HyperScore, breaks down barriers that prevent people – both young and old – from realizing their innate musical talents. HyperScore succeeds because it does not require that the user have any formal musical training, and thus they can immediately begin creating music. However, even though HyperScore only requires a few minutes to install and begin using, many users are reluctant to install applications (even free ones) and so there is an opportunity to expand the community of users by recreating (as much as possible) the HyperScore experience within a Flash application.

Macromedia Flash™ (or Flash for short) refers to both the multimedia authoring program and the Macromedia Flash Player™ that can present multimedia content within a web browser [2]. Macromedia dominates the commercial market for multimedia content on web pages and has had a major impact in the way that users experience commercial web sites. Internet users are aware of Flash applications and are also aware that there are few malicious applications that run as multimedia content through the Flash Player. We have developed an approach that should enable Internet users to experience a near-faithful rendition of HyperScore without having to incur the overhead of installing HyperScore (it will also enable access for those users who do not have administrator privileges to install software). However, it must be noted that our target responsibility is to create a prototype as a proof of concept that can be used to determine the feasibility of this approach as a new business model.

In Section 1.2 we present a reduced set of requirements, derived from the original HyperScore application, that document exactly the goals that we hope to accomplish in this project. Section 1.3 describes the status of our current prototype that we have constructed to verify that we will be able to complete this project as planned.

1.1 Project Goals

Figure 1 contains a snapshot of the HyperScore application executing on Windows XP. The only feature omitted from the display is a percussion motive window.

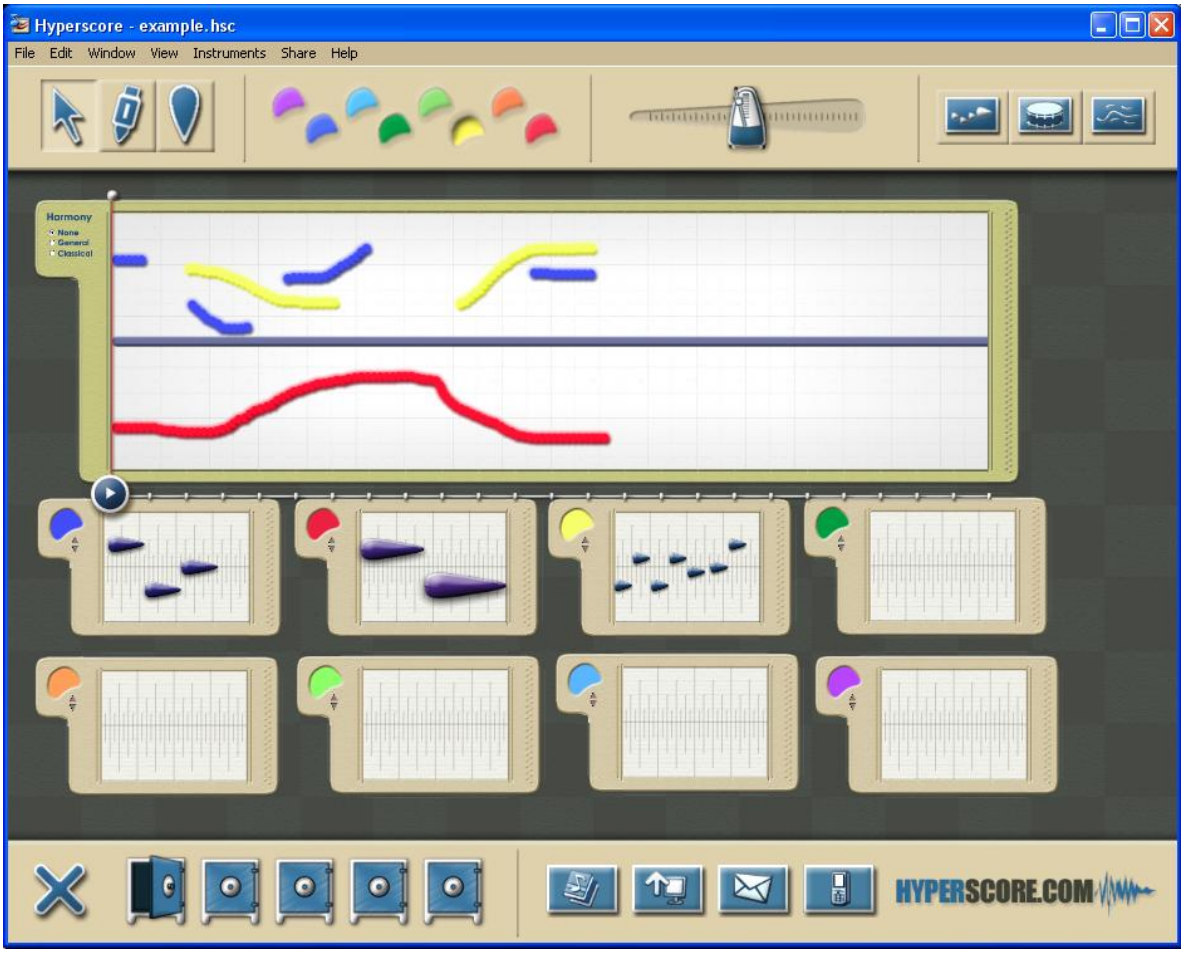

Figure 1. Original HyperScore Application

We intend to deliver a Flash application that recreates this visual experience as much as possible while retaining the same auditory and creative experience.

1.1.1 Visual Experience

Since our goal is to recreate the primary functionality of the program and provide a proof of concept, certain elements are not top priorities. One of the more visible restrictions we adopt is the removal of the menus. The feature of having multiple compositions open simultaneously will not be included. This means the X and the five safes used for storing other compositions will not

be in our application. We also remove the other bottom buttons, which are respectively “Open Music Library”, “Upload Music”, “Email Music”, “Send Music to Cell Phone” and a link to Hyperscore.com.

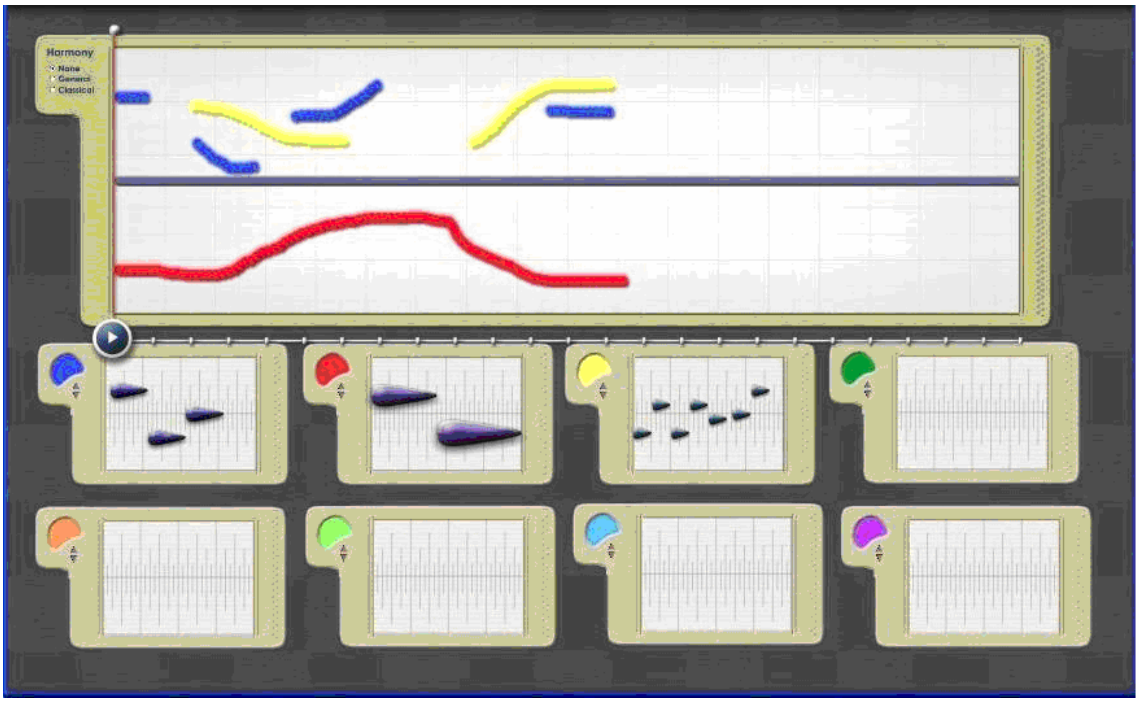

Figure 2. Proposed Flash Application GUI

The top menus will also be removed. File is not necessary, as we are not planning to implement a file server from which to load and save files. Edit is also not necessary, as all the editing we support will be available through mouse clicks alone. Window, **View**, **Instruments**, and **Share** are all superfluous and will not be included in our initial application. They, like **Edit**, all contain extra ways to access features already accessible, such as by keyboard shortcuts. **Window** is just used to create motives, which will already exist in our version. **View** does nothing but control zoom levels. **Instruments** holds a list of usable instruments, but this would be easier accomplished by allowing users to right click on a motive and select an instrument from the entire MIDI set, a feature we will not implement but could be included in future versions It seems as though the goal of the available instrument set is to simplify the selection of instruments to the user and allow them to see one of every major type of instrument from the start. This can still be accomplished by listing those instruments first, if this effect is desired. The **Share** menu will not be needed as sharing will not be a part of our prototype of the application. The help menu has also been removed.

We will not attempt to incorporate percussive motives into our initial application. There will be no button to create a percussion motive. Certainly future work should consider integrating Percussion motives into the Flash application.

Zooming will be accomplished by simple button presses on the keyboard. It may also be a good idea to add buttons to each window (sketch and motive) that allow the user to minimize, which would return to the normal view shown below, and maximize, which would zoom in fully on any sketch or motive window; however these features are beyond the scope of this proof of concept.

The HyperScore application uses a sophisticated representation for the "brush strokes" of music within a Sketch window, using a physics-based string model. We will not attempt to recreate this feature. Instead, our prototype will allow the user to draw a sketch as a solitary curve that is static – no part of the curve will have effect on drawing the rest of the curve. This means the user simply draws a line with his cursor, like in MS Paint. The line itself will not be draggable, nor will the user be able to grab parts of it to manipulate it.

Users shall be able to edit motive windows to add or insert notes; clicking on a note shall play its sound. We shall support moving of notes (and the ensuing sounds as the note is changed) as well as the shrinking and expanding of notes within a motive.

As the primary goal is to create a proof of concept, the main HyperScore features we will be recreating are the basic functionalities of the motive and sketch windows. All the motives and sketch windows will automatically be visible in our version. Motives will not be resizable initially.

We intended to support the ability to highlight either motive notes or sketch lines when the music is played. A moving bar shall move left to right, in pace with the existing tempo, to track the progress of a sketch or a motive, but this proved quite difficult given the object detection provided in Flash, and a working version was not included in our final product. We do not intend to support the playing of an individual sketch brush stroke.

1.1.2 Auditory Experience

Our proposed Flash application will integrate with a custom Java applet by passing the user's created music to the Applet for processing and playback. We do not expect to completely reproduce the harmonic integration of instrumental motives (using General or Classical modes of harmonization). Our aim is initially to recreate the "None" harmonization option and investigate the possibility of including some harmonization ability.

1.1.3 Environment

As a Flash/Java application, the user is no longer concerned with the underlying operating system. There will be no ability to print documents or access the network. Long term future work for this project (that we envision, but will not attempt to implement) includes the ability to remotely access all music which is persistently stored elsewhere.

1.2 Architecture

We envision an integrated solution that relies on Flash Player for the graphical presentation, Java technology for playing MIDI data streams natively to the computer's audio hardware, and JavaScript as the "glue". Future work will connect the Java Applet to a remote database that stores the "*.hsc" files being manipulated by the client, and then pass the data to Flash so that the user can load stored songs and edit them using the Flash UI.

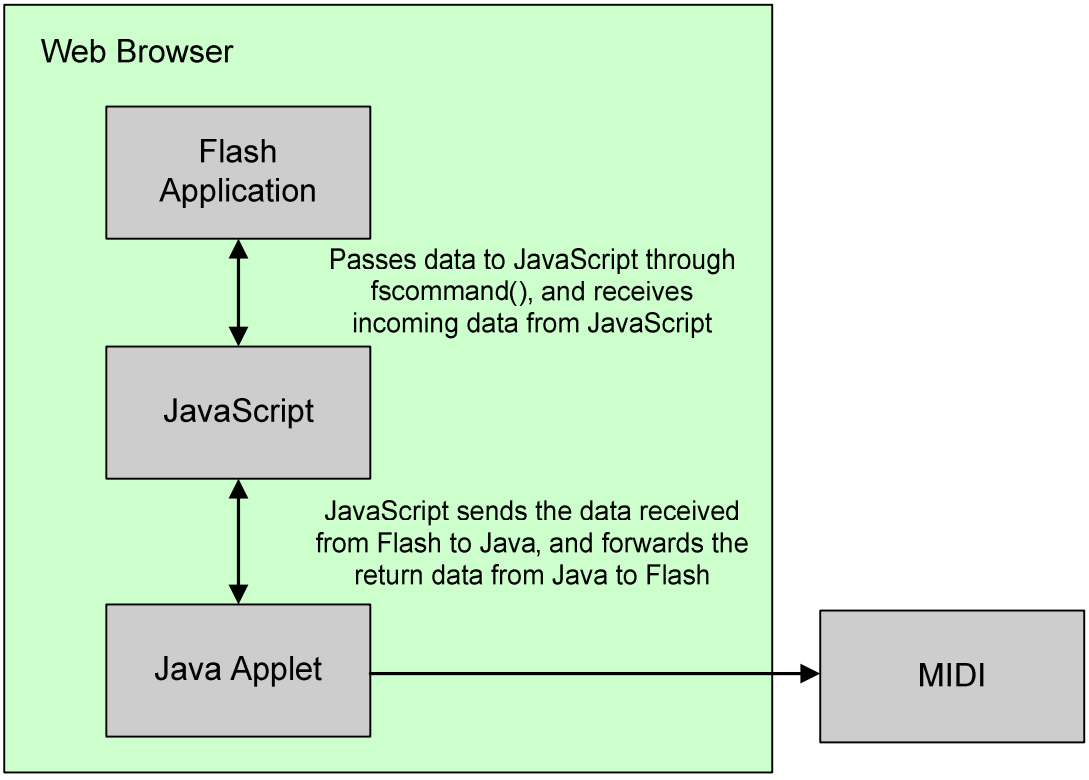

Figure 3. Prototype of Integrated Solution

The sequence of interaction is as follows:

1. **Flash** creates a user interface, where the user may interact with the windows and create their music
2. When the play button is pressed, **Flash** generates a string based on the motive window or sketch window. (String format described in Section 1.3.4)
3. **Flash** passes this information to **JavaScript** using standard **Flash** API method calls.

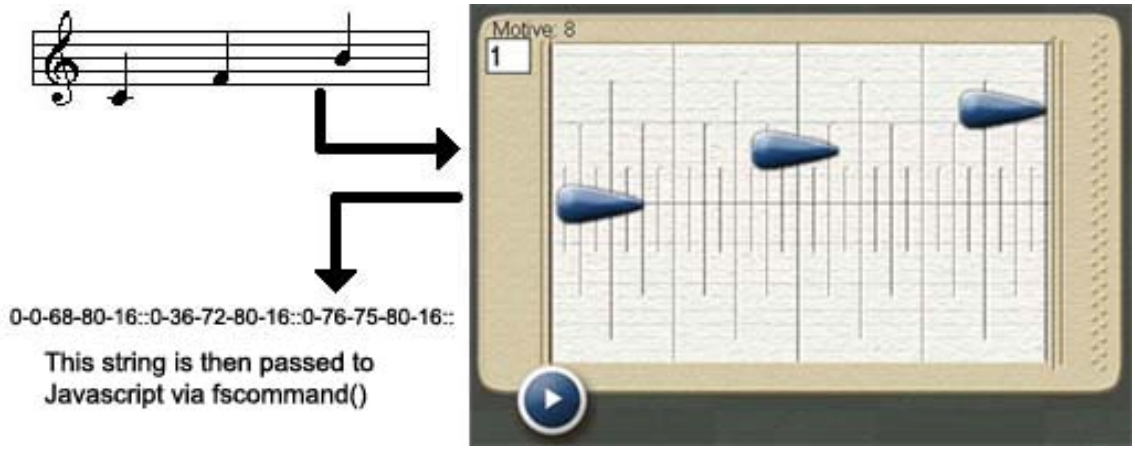

Figure 4: The flow of music from sheet music, to Flash, to JavaScript

4. **Javascript** passes the string to **Java** to parse and tells **Java** to play the string.

```
function playJS(fMovie, mystring) {
        document.HyperScore.parseString(mystring);
        document.HyperScore.play(); }
```

5. **Java** parses the string and creates a MIDI sequence to play the corresponding notes
6. **Java** plays the MIDI, and sends calls **JavaScript** code to communicate with a **Flash** text box.

Our proof of concept clearly shows that these interactions are possible, as the prototype uses all 4 methods of communication (Flash à JavaScript, JavaScript à Java, Java à JavaScript, and JavaScript à Flash). Thus the first goal of our proof is complete – showing that the integration of the 3 technologies is possible.

1.2.1 Prototype

As stated in our architecture, our prototype will be employing 3 technologies to provide the end user with the HyperScore experience. The programming needs to be separated between Java and Flash upon several distinct boundaries. Flash provides the user interface, and therefore must perform any calculations based on the user's input into the application before sending it to Java. However, ActionScript (the programming language native to Flash) is interpreted, and thus we must avoid doing any complex or time-consuming work in Flash as it will introduce large latencies.

The Java applet is needed for two main reasons. The first, as mentioned above, is because doing anything overly complex in ActionScript will introduce a large latency and cause problems in recreating the HyperScore experience accurately. Java will allow us to handle this problem by being far more efficient as it is not strictly an interpreted language (it executes as compiled byte code within a virtual machine). The other reason is that Flash has no innate support for sending MIDI data directly to the client's hardware while Java does. All of these areas need to be coded in Java.

1.2.2 Flash

Flash covers the visual and interactive parts of the prototype. There are currently 2 frames animated, effectively eliminating any tween animations that were used in previous versions. All of the movie clip elements of the Flash are handled in an object oriented manner. There is a window class and a note class. Although they are not written in a normal class manner, in use, they behave like any other class. All of the code in this prototype is embedded into the flash movie clip objects, and written in Action Script 2.0.

The window class is broken up into a few different parts. These include the left bar, right bar, middle bars, play button, and background. Currently, the windows are fixed width. However there is base code to allow variable a window width, but this feature is not currently usable. Windows can be selected, and the play button becomes visible, allowing for a visual confirmation of which window you have selected. Windows can also be moved around by clicking on the outer edges of the window.

Extending the window class is a motive window and a sketch window. Both are based off the same basic code, however, the background element is different in each. Also, the play function is different for each window.

In the motive window, the background consists of a motiveCenter object, which is an image of HyperScore's motive background. When the motive window is clicked, a check is made to see if any notes are present where clicked. If none are found, a new note object is attached. The positioning of the note is relative to the motive window, so when the motive is dragged around, the notes stay in the same relative position. (Effectively, the notes are part of the same movie_clip) Since these notes are attached directly to the background, they inherit its event code, and therefore cannot have any event code on themselves. This is why the background code does the check to see if a note is already placed when clicked. The background loops through each known note for the given motive and checks the bounds of the note to see if a note is already placed where clicking. If one is found, Flash tells JavaScript (which in turn tells the Java Applet) to play the current note, and then sets the note for dragging. When the note is placed, it snaps to a grid to ensure that two notes of different y-position cannot have the same pitch. When the mouse is released, Flash once again tells JavaScript to play the new note. Lastly, when the play button is pressed, Action Script calculates the notes of the entire motive window, and generates a string which is then passed to JavaScript and then to Java, which the applet is then able to parse.

The sketch background works in a very different manner. Our construction of the sketch window functions drastically different from HyperScore's for two reasons: (1) the project is a mere proof of concept, so we did not burden ourselves with attempting to replicate the complex math involved in the sketch, and (2) the code to do so correctly is proprietary to Harmony Line, and thus we did not have access to it. When the play button of the sketch window is pressed, the note data from the motive windows will be processed to create a string representation of what is to be played based on the sketch that has been drawn. This string (following a format described in section 1.3.4) is passed to Java via the JavaScript bridge for processing and playback.

There is a considerable amount of code that is being unused right now in the Flash. Much of it is to lay the groundwork for many of the features that we were unable to implement stably enough to release. Some of these features include: note duration, tempo, instrument selection, and note removal. Our feature list has changed a bit from the original conception of this project as design decisions were made along the way. Some features did not make it into our final release due to underestimation of the number of dependencies that those features hinged on. Others were not implemented due to unforeseen defects. Still other features were dropped as design decisions

were made that ruled them out of being necessary to show proof of concept, and thus we would only try to implement them if we had leftover time at the end of the project. We chose not to include any features that were not working as intended, but left the code in as a framework for a continuation of the project.

In programming in Action Script, there were many obstacles. The main obstacle was learning a new language that does not follow all the same conventions as a standard object oriented language. Our group was inexperienced with Flash, and Action Script was completely new to us. Also, the interaction between movie clips and action script is unique to the Flash environment. Code is attached to certain clips, and those clips are in turn embedded in other clips. This is what creates the object oriented environment to program in. While it is not the same as in C++ or Java with defined classes and public and private variables, the end result are movie clips that act like classes. They are defined visually, and the variables associated with the class are stored locally in each instance of the class. This led to some interesting coding practices. Classes (movies) were created in parts (clips), and combinations of certain parts made up the entire class object. One drawback of this is that any clips that were attached to specific clips inherited the parent code. This means that if a note is placed, it either has to attach to the _root object and move independently of the motive, or it must embed it directly into the motive and thus lose any code on the note.

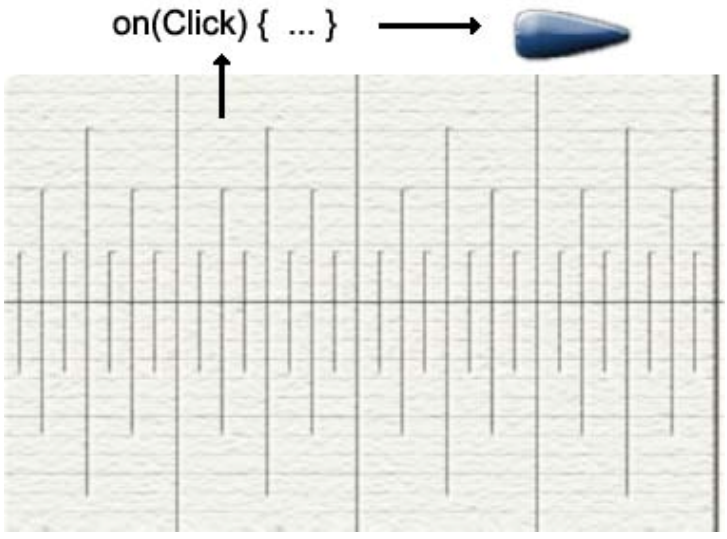

Figure 5. Code Inheritance

The final obstacle is that Flash lacks a robust development environment. Much of the debugging was done through trace() events. This proved to be very frustrating due to the fact that

it made it very difficult to step through the program and track variables. All the code is embedded into many different flash movie clips made this an even harder task. While it made it easier to organize and program, it made it very difficult to debug. At any point in the movie, code is being executed in many different movie clips simultaneously. Many of the clips were even accessing the same information. Even with Flash Professional 8's built in debugger, stepping through a program of this complexity required a lot of manual debugging.

While Flash was moderately complicated to learn, it worked wonders for this type of application. It is very easy to make a stimulating user interface while still offering the power of an object oriented language. The further ease of communication between JavaScript and Flash, and vice versa, allowed for a very easy round trip communication between Flash and Java.

1.2.3 JavaScript

The JavaScript portion of our code serves primarily to facilitate communication between the Flash, which handles all interaction with the user, and the Java, which performs the core functions of the software. This is achieved through the use of an `fscommand()` call, which allows data to be passed quickly from Flash to Java. The command, when executed in the Flash ActionScript, activates a function in JavaScript and passes it any variables it may need.

1.2.4 Java Applet

While Flash is a great tool for creating a user-friendly and visually pleasing interface for the user, it currently lacks the ability to actually synthesize any music using MIDI (The recently released version 8 of Flash supports a fraction of MIDI only for generating keypad tones). Java, on the other hand, can easily and efficiently synthesize a sequence, but is unable to produce the quality user interfaces created effortlessly in Flash.

The Java applet that we have created for this project is thus used simply to process and play the sequence that the user defines in the user interface created in Flash. Flash will pass the data of the sequence to the Applet via the JavaScript bridge that we have created for the transfer of this data in String format.

Once the Applet has the string that the user has created via the Flash interface, it can begin processing it. It will first parse the string using the following schema:

> For each note in the sequence, a String will be appended to any existing String containing data. The notes will be in the format of: [Channel]-[Start Time]-[Pitch]-[Volume]-[Duration]::

where each bracketed item in the string is a sole variable (the brackets are not part of the string, just used here to enclose the variables the string represents). The Applet will then parse the string out to collect all of the data and create the appropriate MIDI events. Once these have been created, the events will be added to the sequencer to prepare them for playing. Once the entire string has been processed in this manner, the Applet will play the string using the currently installed synthesizer.

While the sequence is playing, Flash will require some feedback from the Java Applet in order to correctly process its visual feedback to the user. Java will thus use MetaMessages to store the data that Flash will need to receive upon the playback of a note.

Our original Java Applet implementation did not use the built in `javax.sound.midi` package, in favor of manual control of the notes. However, it proved to be too cumbersome and less efficient to create a custom sequencer to hold and process the data, and in the end the design was switched to purely use the `javax.sound.midi` package to handle all of the MIDI input and sound output. Because of this change in design decision, it is now required that the client have the Java Runtime Environment version 1.5 to function correctly, as many of the functions used from the midi package have been modified in 1.5.

There are no defects in the Java code that accompanies this project – for every feature that was implemented, the function that drives it in the source code works without any unexpected behaviors. There are, however, some features that were not fully implemented that someone unfamiliar with the usage of the Applet may interpret as a defect.

The first said un-implemented feature is the persistence of instrument assignment. Since the changing of instruments was never implemented in the Flash user interface, persistence of instrument assignment was never a necessity in the Java. However, when using the debug-mode of the Applet (and thus calling a Java interface for the user), the code is enabled for changing the instrument for one playback of the sequence string. To change this to a persistent system, all that would be required is to add an array to the Applet that will not get erased every time the sequence is erased after playback, which is trivial.

The other un-implemented feature is changing the tempo of the sequence. In the final version of our software, the ActionScript inside the Flash interface handles all of the duration changes based on the tempo, making handling this in Java obsolete. Java does, however, have

built in support for setting the tempo each time you create the sequence. Adding a function into the Java source to allow for the Flash interface to "outsource" the tempo change would only take one function call to the built in javax.sound.midi package, and would thus take some stress off of the Flash interface in processing data.

The current schema of the string to be passed from the Flash interface to the Java will need to be revised as more features are added into the Flash interface, since Flash will need to inform the Applet of more data that it will need to pass back to Flash upon the playing of each individual note. For example, when Flash is ready to support flashing the notes upon playback, a NoteID variable will need to get be sent from the Applet to the Flash interface to inform Flash which note is currently playing, and thus which note to flash. There are several other examples of changes that will need to be made for the schema depending on the added functionality that one would like to implement.

1.2.5 MIDI

MIDI stands for Musical Instrument Digital Interface. MIDI is used by many people to be able to create music without requiring that the actual instruments themselves to be played and recorded, and instead attempt to synthesize the sounds that the instruments would have created.

MIDI is composed of a series of events that instruct the synthesizer on what to output. Each event is a singular instruction accompanied by the data to appropriately carry out the instruction. For example, one instruction may be to play a note at a certain velocity. The instruction would be NoteOn and it would need to be accompanied by the data for the pitch and velocity of the note. There are many instructions that can be used to control the output of the synthesizer [4]. Each of these instructions is musical in nature, as they correspond to all the quantifiable aspects of music creation. In addition to turning notes on and off through the synthesizer, different events can cause things such as pitch bend, volume swells, and many other nuances that make music sound pleasing to people.

The synthesizer will ultimately create output based on the sequence it receives. A sequence is a collection of all the music to be synthesized, collected in a set of channels. Each channel will have an instrument associated with it, and will have a track of events that it must play.

Thus, the architecture here is as follows: A **synthesizer** is a device that takes in one or more **sequences**, each of which contain a set of **channels** to operate on, each of which contains a

track which is made up of a series of **events**, each of which instruct the synthesizer to take a particular action.

1.2.6 Key Issues to Face

There are several key issues which we considered as we built our prototype. Among the most prevalent of these is the issue of latency. Since HyperScore displays visual feedback during the playback of music, it is imperative that a final version of this application be capable of correctly synchronizing the progress bar in motive and sketch windows during playback with the actual music that is being played. This is one of the key items we considered when determining the feasibility of this approach. Since the visual feedback will be provided by Flash, and the MIDI is being played by Java, this requires that the Java code and Flash animation be linked in such a way so as to produce a negligible amount of lag due to the communication. Our plan was to require Java to inform Flash when a note is being played so the notes can be flashed at the correct times, and so the progress bar can move accurately. However, this data transfer takes time, and additional notes and motives require even more time.

Another key issue is security. Aside from the usual measure taken to protect the copying of code, special measures must be taken in this case to insure that the music created cannot be downloaded directly from the application in anyway. If users were able to download the MIDI versions of the music they created, Harmony Line would be unable to sell the music as ring tones and be unable to profit. This issue is a bit beyond the scope of our current goals, but must be dealt with eventually. The likely solution would be to save the files exclusively as .hsc files, and never allow users access to any MIDI files.

It is important to note here that our architecture discussed in this project could be greatly simplified if security was not a large concern. Were Harmony Line not interested in the security of this product, then mp3 files could be generated to play the result of the user's input. This would lead to no longer needing Java or JavaScript, as Flash does have innate support for the handling of mp3s. This approach is not satisfactory, however, because a knowledgeable person would be able to extract this mp3s upon download to their PC and thus bypass the need to pay for a copy of the song they have created.

2 PROCEDURES AND ACHIEVEMENTS

2.1 Proof of Concept

The first phase of our project was to examine the functionality of HyperScore and determine if it could be realistically and effectively recreated using Flash and HTML. After conducting some preliminary research, we were unable to find any instances of this particular kind of interactivity taking place anywhere on the web, so it was necessary to create a simple prototype to display the basic groundwork. The prototype needed to be capable of taking input in Flash, sending it to Java via JavaScript, having Java play some MIDI, and then returning confirmation of the playback back to Flash through JavaScript.

The goal of proving that our architectural design was able to handle our goals was accomplished. All of the communications between Java and Flash were implemented and used seamlessly.

2.2 Additional Features

After we completed the initial prototype proving the viability of our architecture, we began to incorporate more of the features of HyperScore into our software. Movable notes were added next. A key feature added at this phase for debugging was lag detection. The ability to detect lag allowed us to plan for future issues with lag and begin avoiding them ahead of time. We found that while playing, our software exhibited an average lag of 20 milliseconds on each note. This is less then the amount of discrepancy that humans can detect (about 50 milliseconds), so this is an acceptable amount of lag [3]. However, the first time the play button was pressed in the Flash environment, there was an additional lag of several second to over a minute depending on the system. This was a potentially large problem, but was easily averted by sending a dummy note through before the user actually presses play to initialize the MIDI output and prepare the system for actual use.

However, as time went on, our latency issue grew into a large problem. The more data that needed to be passed back and forth between Flash and Java introduced large amounts of lag, and the more data that the interpreted language ActionScript needed to process slowed things down a significant amount. As it stands now, the latency has already reached a point at which it is noticeable and quite undesirable. As more functionality is added, the latency will only increase.

2.3 Obstacles

We encountered numerous obstacles in our attempt to get our motive window working correctly. The first was that sound would not play on all computers we ran the prototype on. We

tested various different operating systems and internet browsers, and also experimented with different versions of Flash installed. In the end, the reason playback did not work on all setups turned out to be an issue with Java. The standard latest JRE release (1.5.0_06) simply does not include the necessary MIDI data file (soundbank.gm) to allow MIDI playback in Java. This is a standard file, and fixed the issue once copied off of our schools network to the correct location. Users installing the JRE have the option to customize the installation, at which time they can select to install the requisite audio soundbank file. This issue will not necessarily be a problem for any potential users of our software, but those who keep up to date with the latest JRE releases will need to address it. The problem is easily solved by provided the soundbank file for download with directions on where to place it.

Another key issue plaguing the playback of our prototype is the hit detection used in Flash to determine when the notes in the motive window have been reached. Flash inherently uses a collision model which detects bounding boxes. Due to this, we could not fire events based upon when the progress bar struck a note. This began to plague other general areas of the program as well. Because the notes have bounding boxes, one could click on what should have been whitespace in the motive window, but still unwittingly select a note due to the bounding box discrepancy. This will cause further problems when trying to select a sketch that has been created in the sketch window. Presumably all of the collision detection mechanisms can be modified, but doing so was beyond the scope of this project.

The differences between recent versions of Flash have also proven quite problematic in the creation of our prototype. The newest version of Flash is Flash Professional 8, and as such, we chose to use it to do our development. However, Flash 7 is still in wide usage, and Henry Kaufman, our contact at Harmony Line Inc. has chosen to do most of his recent work in Flash 7. This presented a problem, as everything written in Flash 8 is not necessarily backwards compatible with users running Flash 7. In addition, files being developed in Flash 8 must be exported in Flash 7 format to allow developers to view them correctly, but exporting alone is not always enough. Certain elements must be further modified specifically to work in Flash 7.

2.4 Achievements

The chief goal for this project was to create a proof of concept that Flash could be integrated with a Java applet to accurately recreate the HyperScore experience and evaluate the feasibility of this approach. As such, we attempted to recreate as many of the features of the product as our time would allow, while not expending too much time implementing non-“core” features – these could be easily implemented by Harmony Line if the project was deemed

feasible. Some of the features needed to be changed to allow for the online environment, others due to limitations of either the development tools or time.

As such, we have succeeded in recreating many of the key features that make HyperScore what it is. The motive windows have been recreated in our Flash environment to function with most of the same behavior as in the desktop application. When the user clicks inside the stage area of a motive window, a new note is placed where the user clicks. This note can then be further manipulated to be larger or smaller in size. Although this is supposed to extend or shorten the duration of the note, this feature was not reproduced at the current time. In addition, although the support for the notes to be increased/decreased in size is implemented, the windows themselves are not resizable. The code for this has been started, but had too many defects to be enabled in the final release. The note can also be dragged to a new place within the motive window, and is bounded by the edges of the stage of that window, as one would expect. If a user clicks to select a note, while there is no *visual* confirmation that the given note has been selected, the current pitch of the note is played.

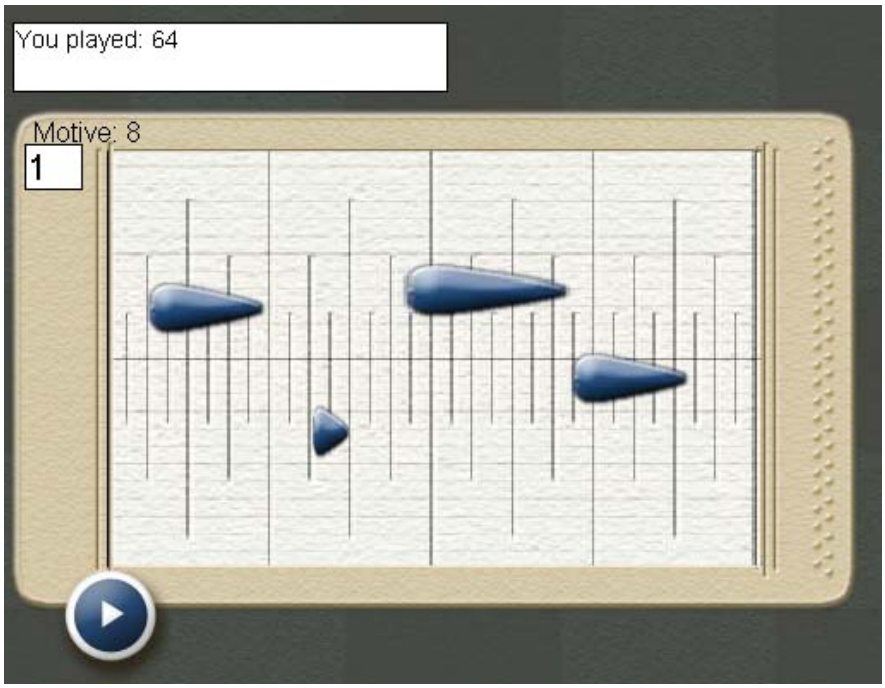

Figure 6: Motive Window and Notes

There is a key feature of the motive window that has not been reproduced, however. The notes are never flashed when the playback of the appropriate note occurs. This occurred as an underestimation on our part of the amount of effort this functionality would require. However, although the end functionality has not been achieved, the groundwork has been laid for this to be implemented by another team that may choose to continue this project. The reason that the feature

was not included in our final release is because the other features that it depends on are not currently in a releasable state due to defects. In this case, although the concept of the communication existing in all 4 ways is proved, the processing of the data by Flash was not finished without defect. In our final version, a visible textbox appears in the top left of the application showing whatever data is being passed back from the Java, but this data is not parsed and appropriated to the correct functions at the current time. Another key piece that was not overcome was the ability to have a progress bar scroll across the screen as the notes are playing. The reason for this was a combination of problems. Our first attempt at creating the progress bar began to fail due to an error where the bounding box on the actual moving bar was larger than the size of the bar, which caused errors in timing. As a result, the bar was separated from the play button, but this caused a new error in which we could not get the bar to play based on moving its x-position forward by a set value every repetition of the loop of frames. Again – the code and logic for this feature are currently included in our release, but they are not currently working. Thus, a group choosing to continue our work would want to get this feature working quickly. Eight of these motive windows appear on the stage of our prototype, and all are able to be dragged anywhere on the stage.

In addition to the motive window, we have a sketch window that does show the proof that our model could be used to accurately reproduce HyperScore. The sketch window is currently hard coded to only allow 2 sketches with a length of repeating the appropriate motive windows (windows 7 and 8) twice each. The decision for this was made because the algorithm for determining what to do with the sketch is proprietary information that we do not have access to. We have proved that math can be performed based on a curve drawn on the stage of the sketch window, and thus Harmony Line's proprietary algorithm could be used to mix the data appropriately if placed here.

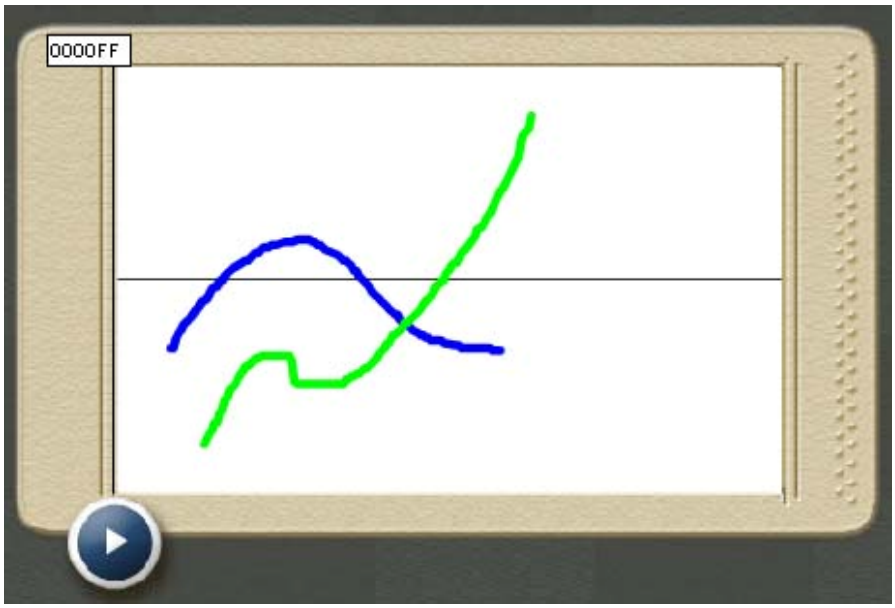

Figure 7: Sketch Window

Other than these features, the sketch window shares many of the same functionality and shortcomings that the motive windows have. The sketches can be drawn and played, but they will not show the user any visual confirmation that they are playing. The progress bar suffers the same issues as with the motive windows, and the sketches do not flash when they are being played either. It is important to note here that getting the sketches to flash will require somewhat more effort than getting notes in a motive window to flash, as it will take more data being processed based on our current structuring of the data.

There is a textbox in the upper left corner of every motive window that is the beginning of a basic interface to change instruments in the application. While the changing of the value in that box does not currently get forwarded to Java, due to our already shown communication it is proven that this data transfer is possible. The Java Applet included in our final release does contain support for the changing of instruments, as is clear when running the applet on its own rather than as support for a Flash interface. Thus, implementing this feature fully is left to any group choosing to further this research. A key difference between our sketch window and the one provided in the desktop installation of HyperScore is the look and feel of the brushstrokes. HyperScore allows users to create sketches that can move backwards on themselves and even produce loops. In addition, they operate on a string based model where moving one part of the sketch will cause effects in many other areas. We did not burden ourselves with the physics of trying to implement a string based model for our sketches, thus they will look different to the user than the desktop version. In our prototype, since we were steering away from the complex math involved in HyperScore's complex proprietary algorithm, we restricted the user to only be able to

create sketches from left to right (which implies that there can be no 2 points with the same x-value).The sketches, once drawn, are final – they can not be dragged, edited, or deleted. While this is a hefty restriction, it presumably would not be continued to the end-user; our sketch window stands to prove that the concept can be extended to include the full functionality of the original product.

In our proposal, it was decided that we would not necessarily implement a percussive motive window. While we indeed do not have the percussive window implemented in our user interface, it can effectively be reproduced using the Java that we have created. To do this requires the user to be in "debug" mode, where we use the Java interface rather than the Flash interface for the application. When creating the string to send to the parsing and playing functions, one must simply set the instrument of the channel that contains the note data to a drum set from the list of instruments that the Java MIDI synthesizer provides.

Several of the goals of the project as described in our initial proposal were soon re-evaluated. As what we were making a proof of concept that at least the core features that make the HyperScore experience what it is can be reproduced online, we needed to focus on the areas that were considered essential. For example - a design decision was made fairly early on in the project to drop the menu system using File and Edit to a lower priority. These features would be included after all of the required functionality of the application was implemented and debugged to be working properly.

The next feature that is discussed in the initial proposal for the project – including Help information for many pieces of the project – was dropped due to being out of scope. The project that we have designed and implemented is intended to stand on its own as proof that the selected technologies can function together to recreate HyperScore in a web browser rather than on the desktop. This goal will be achieved without the addition of many links to pop up new windows with help information on how to use the project – that is a feature of a finished product, not a proof of concept.

A key feature that a user of this product would need in order to efficiently create music is the ability to zoom in and out on the stage. This feature is included in our final release. The simple pressing of the Page Up or Page Down buttons will zoom the camera in or out of the stage, respectively. The camera will zoom into the center of the stage, regardless of what is selected. However, the stage can be moved simply by holding the Ctrl button while dragging anywhere within the Flash interface. Using these two key pieces of functionality, the user can thus zoom far in or out of where they need to see in order to create the music that they desire.

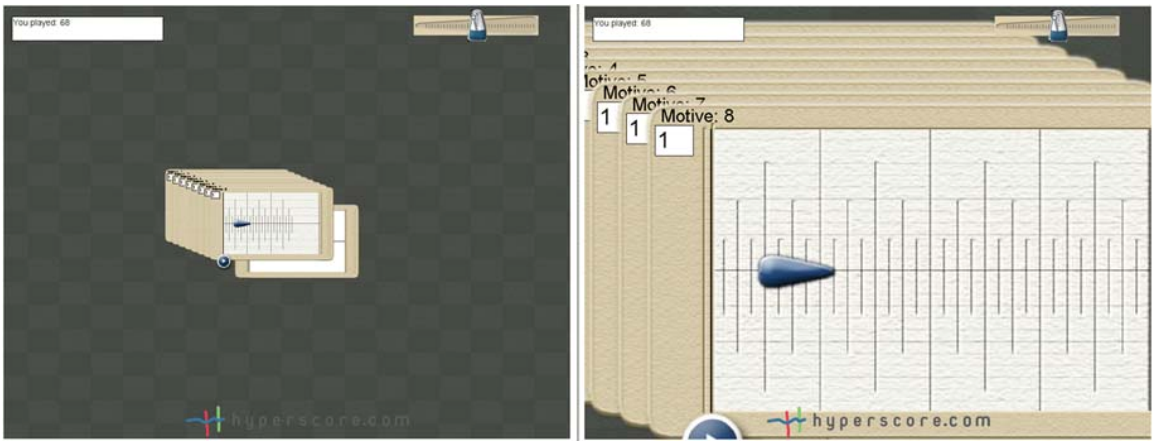

Figure 8: Zooming

When all of these features are included together, our prototype stands on its own both as a proof that HyperScore can be recreated in an online environment, and that this is an effective test of the feasibility of such an approach to a new business model for Harmony Line. Many of the features have been implemented in a basic way to show that a Flash/Java architecture can in fact be used to produce these features, but were not implemented to such a point that they are ready to place in a final product. As such, we are able to draw the conclusions stated in Section 4 of this document.

3 BACKGROUND RESEARCH

3.1 MIDI

Files in MIDI format were the first widely shared music files on the internet. MIDI versions of just about every famous song are available. A wide variety of original compositions by desktop composers are also available on the web. MIDI is a great format for sharing across the internet, primarily because of their incredibly compact size. An average MIDI file is typically about 50 KB. The drawback to MIDI is that it is not an actual recording, and therefore can not contain vocals or complex instrument sounds. With the unavoidable technological advancements of faster internet connections and larger hard drives, music enthusiasts have focused more on MP3s.

MIDI is still a useful format for storing music because of its many distinct features. First, because MIDI files contain actual note information instead of recorded sounds, they are very easy to edit. With proper editing software, any imaginable change can be made. The instrument used on any track, the volume, duration, or pitch of any note, or the tempo of any part of a composition can be edited with ease. Every minute detail is readily accessible. This is in stark contrast to WAV and MP3 files, which can be spliced and edited for speed, as well as have effects such as echo or distortion applied to them, but are impossible to edit in terms of details like instruments and pitches. Second, due to the large amount of actual viewable data contained in MIDI files, they are useful for composing music and printing sheet music for others to play. Third, due to their incredibly small size, MIDI files are a great way of transferring music to musicians. The score of an entire musical for every part of an orchestra would take up less room then a single MP3.

The basic information contained in a MIDI file is as follows:

- Channel: each of the 16 channels typically plays one track of the file, using a specific instrument, or patch
- Pitch: each note has a pitch (C, C#, D) indicated by a number (60, 61, 62)
- Duration: how long each note lasts
- Velocity: also called attack volume, the volume at which a note is originally sounded. The decay of the volume of the note can also be modified

3.2 MP3

MP3 is the predominant music file sharing format in use today. MP3 files are compressed versions of the actual recordings. They discard the less important information, similar to image compression using the JPEG format. MP3 files are typically compressed at a bit rate of 128 – 256, and can go as high as 320 kbps. Most people can not distinguish the difference between the actual recording and an MP3, especially files compressed at the upper range of bit rates (256-320.) The best feature of MP3 files is their small size. Granted, they are much larger than MIDI files, but MP3s are the smallest file that actually sound like the original recording.

3.3 Examples of Interactive Music on the Web

Guitar Shred Show- Mika Tyyskä [5]

This interactive Flash game allows the user to play an electric guitar solo over a backing track by holding down various keys on the keyboard. The sound bytes are stored in MP3 format, and will play only while the user is holding down a key. There are 36 sound clips, all of which are in the same key and will fit well with the backing track. There is also a training section which shows the user what is being played by each key. This is shown in tablature, a type of written music popular on the internet for transcribing guitar parts. The tab shows up in the bottom portion of the Flash window when each key is pressed, and is also printable.

The user is not allowed to create any truly unique music with this software, but it is an interesting step in the direction of actual composition. With 36 separate sound clips, each of which can be played any number of times in part or in whole, and several minutes to work with, it is unlikely that any two users will ever create the same solo. It is also interesting to note the precision with which the Flash animation was created. Many of the sound clips feature very fast picking and fretting (10+ notes per second), and the animation is precisely on cue with the sound files. The choice to make all the sound files in the same musical key is also important. This limits the choices available to the user, but insures some level of harmony in the final product and removes all dissonance. Both the precise animation and option to remove dissonance are available in HyperScore.

Punk-o-Matic - “Evil-Dog” (contains some coarse language) [6]

This piece of Flash software allows the user to compose punk rock songs using some basic looping. Sound bytes are stored in MP3 format, and each is 1 – 4 measures long. Again, all the sound clips have been recorded in the same key, so they all sound good together. There are 9 drum clips, 12 rhythm guitar clips, and 9 lead guitar clips, for a total of 33 clips. This is less then the number of guitar clips from the previous example, but since the three parts play simultaneously the number of unique songs grows considerably faster. It is again true that no actual unique music can be composed, but just listening to a few random compositions gives the user a general idea of the variety of sounds that can be created with a relatively simple tool. It becomes clear that even without the ability to compose new tracks, much can be done by simply looping previously recorded tracks.

A novel feature of this environment is the ability to easily save your work in an incredibly simple format. Since very little information is required to specify a composition, work can be saved by merely creating a text file with numbers and dashes. Numbers indicate one of the clips, and dashes indicate rests. The entire file containing the song information takes up less then 1 KB.

Musical Sketch Pads- Morton Subotnick [7]

This page is implemented in Java and has many similarities to HyperScore. It combines some of the functionalities of HyperScore motive and sketch windows into one all-purpose window. Users can create their own music by drawing pixels in window. Multiple instruments can play at the same time, and a 3-stage tempo changer is also available. Under the "Sketch pad with variations" link, your compositions can be altered in a variety of ways, including volume changes, stretching and squashing the length of the notes, horizontal and vertical inversions, and transpositions in all directions. This piece of software also keeps all music created in the same key by only allowing users to place notes in the diatonic scale. This insures a certain level of harmony, but also restricts the user from creating music with different sounds, barring notes from the blues scale and the harmonic minor scale, among others.

When the user hits the play button, all the information currently in the window is exported to a MIDI file, which plays in QuickTime in a separate frame. This exporting to MIDI is very similar to the way our prototype works, but this implementation suffers from considerable lag every time the user presses play.

4. Conclusion

It is our opinion that, at the present time, the approach presented in this project would be unsuccessful at recreating the HyperScore experience. The main problem is that Flash lacks some key features that would be necessary in order for an application like HyperScore to operate. Another important issue is the lag associated with the transfer of information from Flash to Java and back. The recent versions of Flash and Java may also hinder the ease of use of such an online application and

Flash lacks the ability to select objects onscreen easily, a feature which is very important to the operation of HyperScore. Each object is surrounded by an invisible rectangle determined by the maximum x and y values the object occupies, and clicking anywhere within that rectangle selects the object. This makes the selection of notes in close proximity to each other very hard. It also makes the editing of multiple lines in the sketch window almost impossible. In order to reach some level of usability, a very unintuitive layer system would have to be created in order to access the various lines, and keeping the operation of the software intuitive is very important to the makers of HyperScore.

Flash also lacks MIDI support for web browsers. There are some MIDI capabilities for use on cell phones, which at first glance seems like it would be a good fit, since the goal of HyperScore is to get users to download their ring tones to cell phones. However, the interface of a standard cell phone is not anywhere near adequate for using software like HyperScore. If MIDI support for web browsers were added to Flash, along with the changes to the bounds of objects, it may be possible to recreate HyperScore in its entirety in Flash. This would probably be the optimal solution, as it would also remove the complicated transfer of data and potentially eliminate the issue of lag.

The lag between the user clicking the play button, the music playing back, and the visual confirmation (blinking of notes) has been slowly increasing throughout our entire time working on this project. Our initial tests placed the value at around 20 milliseconds, an acceptable number. However, these tests were very simple, just sending the pitches and durations of a couple notes. As the application became more complex, more and more information was being passed and more calculations being made. Our most recent tests have a noticeable amount of lag, and there are still many features unimplemented. If all the functionality of HyperScore was implemented, there would no doubt be a considerable increase in latency. This would be unacceptable.

It is also necessary to have the correct versions of Flash and the Java Runtime Environment (JRE) in order for the software to work correctly. Forcing users to have the newest

version of Flash isn't that bad, but people may be more reluctant to download and install Java updates.

For all these reasons, it is our recommendation that Harmony Line not attempt to switch their primary focus of development to this paradigm. Flash could perhaps be used to create a simplified demo similar to what we have created, or a game introducing users to HyperScore. It is not, however, a viable means with which to recreate the entire HyperScore experience.

Appendix A – Full Defect List

Defect	Reason
1. Notes placed in motive window may sound multiple times	This appears to be a defect when using FireFox, and the location of the code causing the defect is unknown.
2. Some clicks produce multiple notes (after moving mouse, another note or more may appear under the cursor)	Again seems to be an issue inherent to using FireFox, and the location of the code causing this defect is also unknown.
3. Entire application stops producing any sound after numerous reloads	Possibly a resource allocation error.
4. Sketch window will not play the correct number of times (2) unless two sketches are present.	This functions as it is coded; the sketch window is hard coded to only play when both motives 7 and 8 contain notes and 2 sketches are drawn, otherwise math errors occur
5. Sketch window will repeat motive 3 times instead of the intended 2 if only one motive contains notes	Seems to happen only in IE and not FireFox. Seems to be again an issue of browser compatibility.

Appendix B – Unimplemented Feature List

- Notes not being allowed to be on the same X plane as another note
  - Many defects were found attempting to implement this feature. While it can do the required feature, it removed other notes. Due to the feature never working properly, the attempted code was removed.

- Visual confirmation of notes being played
  - While the base code is in place for this feature, it would require more graphical work to create the flash effect.
  - There is no slider bar, as the visual synchronization is easier to program once the other visual confirmations were programmed

- Visual confirmation of sketches being played
  - Much like the notes, the base code is in place, but graphical work is needed to fully implement

- Tempo, instrument selection
  - These features were simple to add. However, due to time constraints, base code was put in as a stepping stone for further work on the project, but the features were not fully implemented.

- Selecting and deleting already placed notes
  - While notes can be selected, there is no persistence in the selection
  - Implementing a persistent selection would require a minor overhaul of the way notes are handled.

- Selecting, editing and deleted already drawn sketches
  - The bounds of a sketch are rectangular, therefore determining between overlapping sketches was very hard
  - Currently, due to the hard coded nature of the sketch window, once a sketch is drawn, the movie clip is closed. To enable editing would require a non-hard coded in sketch.
  - Deleting sketches was possible, and at one point implemented. However, since the sketch was redone in hard coding, deleting the sketch was not necessary, and therefore removed.

- Drawing of new sketches in window to be defined in non-Cartesian function format
  - The sketches must be drawn left to right (they cannot create loops).
  - Only 2 sketches can be drawn. The first (Blue) corresponds to motive window #8, and the second (Green) corresponds to motive window #7.

- Drum kit
  - The drum kit was only to be implemented after all other features were implemented.
  - The base of the window is the same as the motive and sketch windows, so that class can be extended to create a drum kit window at a later date.

- Variable window sizes
  - There is base code in for variable window sizes. However, the hard coded sketch window became infinitely more difficult to deal with when this feature was implemented. So, the window size was decided to be hard coded as well until the sketch was also no longer hard coded.

- Menu Bar
    - Since this feature does not effect the basic use of the prototype, it was determined to be a minor feature that could be implemented later

- Saving/Loading
    - This feature was not necessary in a proof of concept.

- Harmonization
    - Being proprietary information, we did not program in any sensitive formulas into the prototype. If this were needed, Harmony Line could incorporate it themselves.

References

[1] Harmony Line, Inc. http://harmonylinemusic.com
[2] Flash Player, http://www.macromedia.com
[3] Speed of Human Sight, http://www.wonderquest.com/sight-whale-tern.htm
[4] MIDI messages table, http://www.midi.org/about-midi/table1.shtml
[5] http://www.guitarshredshow.com/
[6] http://www.newsandentertainment.com/zfpunkomatic.html
[7] http://www.creatingmusic.com/mmm/mmm.html