

G52GRP Interim Group Report

HEX - A Chain Reactive Music Generator

Group NHN2

22nd March 2013

Supervisor: Dr. Henrik Nilsson

| | | |
|------------|---|--------|
| S. Cooke | - | skc01u |
| R. Fulton | - | rxr01u |
| G. Hallam | - | goh01u |
| D. Huo | - | dxh02u |
| M. Tawafiq | - | mxt41u |
| J. Sherry | - | jxs41u |

Contents

| | |
|--|-----------|
| I Preliminaries | 2 |
| 1 Concept | 2 |
| 1.1 ReacTogen Concept | 2 |
| 1.2 Project Description | 2 |
| 2 Background Information and Research | 3 |
| 2.1 Existing Systems | 3 |
| 2.1.1 Examples of Existing Software Systems | 3 |
| 2.1.2 Examples of Existing Hardware Systems | 3 |
| 2.2 Systems Research Evaluation | 4 |
| 2.3 Market Research | 5 |
| 2.4 Music Research | 5 |
| 2.5 Technical Research | 6 |
| 2.5.1 MIDI | 6 |
| 2.5.2 Hash Maps | 6 |
| 2.5.3 Working With Hexagons | 7 |
| 2.5.4 Identifying Hexagons | 8 |
| 3 Requirements Specification | 8 |
| 3.1 Functional Requirements | 8 |
| 3.2 Non-Functional Requirements | 10 |
| II Design | 11 |
| 4 Software Design | 11 |
| 4.1 The Grids | 11 |
| 4.2 Creation of Sound | 11 |
| 4.3 Pulse | 11 |
| 4.4 Tiles | 12 |
| 4.5 Keeping Time | 12 |
| 5 User Interface Designs | 13 |
| III Implementation | 17 |
| 6 Code Plan | 18 |
| 7 Central Control Algorithm | 19 |
| 8 Key Control Elements, Data Structures and Methods | 19 |
| 8.1 Removal of Duplicates | 19 |
| 8.2 Timing and Tempo | 21 |
| 8.2.1 Tempo Controller | 21 |
| 8.3 Sound Production | 21 |
| 8.3.1 queueNote Method | 21 |
| 8.4 Saving and Loading | 22 |
| 8.5 Transposition | 22 |
| 8.5.1 Transposition Formula | 22 |

| | |
|--|-----------|
| 9 Testing | 23 |
| 9.1 Unit Testing | 23 |
| 9.2 Functional Testing | 23 |
| 9.2.1 User Interface Testing Example | 24 |
| 9.3 Performance Testing | 24 |
| 9.3.1 Stress Testing | 24 |
| 9.3.2 Compatability Testing | 25 |
| 9.3.3 Usability Testing | 25 |
| 9.3.4 Acceptance Testing | 25 |
| IV Project Meta-Comment | 26 |
| 10 Updates since Interim Report | 26 |
| 11 Time and Planning | 26 |
| 12 Problems Encountered | 29 |
| V Conclusion | 30 |
| 13 References | 30 |
| VI Appendices | 31 |
| 14 Markey Research Survey | 31 |
| 15 Software Instruction Manual | 32 |
| 16 Weekly Formal Meeting Minutes | 33 |

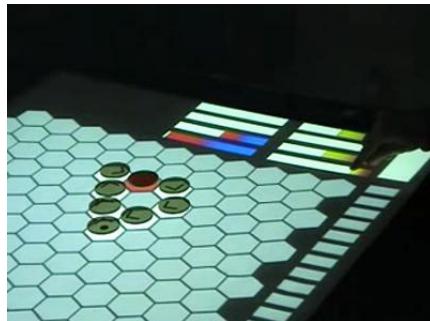
Part I

Preliminaries

1 Concept

1.1 ReacTogon Concept

The ‘*ReacTogon*’[1] is a concept instrument produced by Mark Burton in 2007. Described as a ‘chain-reactive performance arpeggiator’, it acts as a physical user interface to a synthesiser through the placement of tokens on an interactive surface. The surface itself is a grid of hexagons, where each hexagon represents a musical note on the Harmonic table[2] — a novel method of placing musically complementary notes together, providing an easy and intuitive way to create chords and melodic sequences.



The ReacTogon

1.2 Project Description

Our project, branded ‘*HEX*’, aims to emulate and extend the ReacTogon concept. Through the production of a software system that implements the harmonic table within the framework of a pattern based sequencer, we aim to provide a completely novel musical experience.

To this end, we have written a Java application that implements the harmonic table in this way. Control is provided through the use of five counters or ‘operator tiles’ that are placed onto the grid, namely Play, Stop, Change direction, Explode and Warp. The latter four have active and inactive states that dictate whether they play a note or not during an interaction. In addition to this, there are buttons and sliders to change tempo and instrument. There are also multiple layers of grids, set out in tabs, for the creation of harmony and a multi textured, polyphonic composition.

Our software is designed with a PC and touch screen in mind, as the tactile nature of interaction greatly lends itself to the interface. This said, the application will still run with a standard mouse and indeed, the nature of a Java application allows portability and support for other operating systems.

2 Background Information and Research

2.1 Existing Systems

2.1.1 Examples of Existing Software Systems

1. JR Hexatone Pro



The Hexatone Pro;
the main board of which is a harmonic table.

The JR Hexatone Pro is an application developed by Amidio for iOS that implements the harmonic table [3]. In a similar manner to the ReacTogon, tiles or 'cell commands' are placed on a circular grid of hexagons and control the nature of the sound produced. Its unique selling point is derived from the fact that it uses '*artificial intelligence and advanced randomisation algorithms*' [4] to randomly alter the sound as the loop progresses to create a constantly changing sound. In addition to this, in place of a 'start' tile, sound is propagated from six 'oscillator' tiles at the centre of the grid and the playhead moves to one of three adjacent tiles. [5]

2. TonePad / TonePad Pro

The TonePad is another application for the iOS platform, developed by LoftLab.[6]

While it does not implement the harmonic table, it is based on a 16x16 matrix of notes in a pentatonic scale. To this end, it has a similar musical effect to notes played on a harmonic table. While the TonePad is simply controlled by selecting notes on the grid and has no placeable operation tiles, the ideas of playhead propagation and chain reactions are exemplified in this application. The playhead moves left to right across the grid and when a selected cell in the grid is encountered, it triggers others within a certain radius.

A key feature of the TonePad application is its ability to import and export songs. Tracks can be exported as ringtones on iPhones or to m4r files. It is also possible to export song 'code' for sharing with others; this is integrated into the app with an 'upload' button.

2.1.2 Examples of Existing Hardware Systems

1. Tenori-On



Developed in 2005 by Japanese artist Toshio Iwai and Yu Nishibori at Yamaha [7], the Tenori-On is a hand-held music sequencer. Fundamentally, it is conceptually similar to the TonePad, but in hardware form and with a variety of different modes. The hardware itself consists of a 16 x 16 matrix of pressable buttons which light when activated. These buttons can act in a similar manner to the cells in the TonePad in the main sequencing mode, where highlighted cells are played when the playhead encounters them.

In addition to this, five other modes exist such as 'push mode', which produces a continuous sound when a cell is pressed; 'bounce mode' which causes the playhead to oscillate between the selected cell and the edge of the matrix, producing sound when 'bouncing' off the side and 'grouping' which is used to sequence patterns together.

Each loop is composed within a layer and each layer can be thought of as 'performance parts' of which there can be a total of sixteen. Different notes and instruments can be assigned to each layer and all layers can be played together in synchronisation. Each set of sixteen layers is called a block; a total of sixteen blocks can be stored and dynamically switched between during a performance. In this way, musical loops and motifs can be generated and played sequentially to create a complete piece of music [8].

2. **AXiS-64** The AXiS-64 is possibly the most prevalent MIDI controller and piece of hardware that utilises the harmonic table. The equipment consists of the a harmonic keyboard, composed of 192 hexagonal keys; 8 preset keys for storage of user defined keyboard configurations; 4 cursor keys, for navigating between banks; a pitch bend wheel; a modulation wheel and two rotary dials [9]. The four analogue controllers can be easily reprogrammed and used for any MIDI controller change.



The fourth revision of the AXiS-64 firmware introduced three different keyboard modes; 'single', 'split' and 'layer'. In 'split' mode the keyboard is split into three 64 note keyboards; in layer it becomes one keyboard sending a signal on up to three MIDI channels when a note is played and in 'single' it acts as a single keyboard.

2.2 Systems Research Evaluation

The products listed above are but a few among the systems available on the market, but each one exhibits interesting or unique characteristics that can be taken as inspiration for our project. In addition to this, there are persistent themes among all products that too will influence our designs.

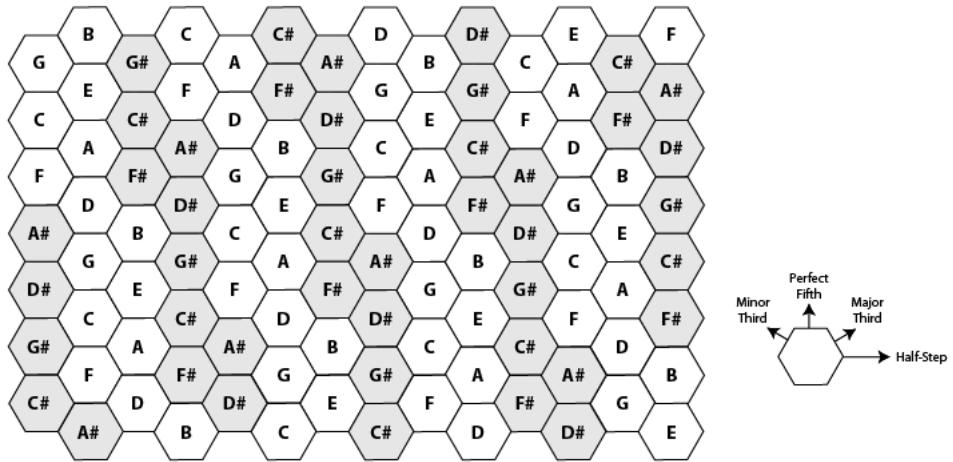
Among these common themes is the use of physical interaction as an interface between the system and the user. A considerable majority of the software systems available are for tablets or mobile devices using touch screen interfaces, with almost no standard desktop applications and only a few online hardware simulations or alternatives. This information justifies our choice of using a touch screen interface with our software. In addition to this, most products act, in some propensity, as a single touch harmonic table keyboard. As such, this is an important feature to implement.

In addition to this, particularly interesting features of researched products that we may wish to take forward into our project include the multiple 'layers' featured in the Tenori-On and the TonePad Pro's ability to export and share music and created projects.

2.3 Market Research

We conducted a survey (attached in appendices) targeting general music enthusiasts as well as complete beginners in the music field to find out whether our software would be desirable. Very few of the people we asked had heard of the ReacTogon, the concept instrument, but most had some sort of interactive music experience where you would either use a mouse for a desktop application or a touch screen app to create some sort of music. We received contrasting opinions on the pre-existing systems, some of them liked the idea of the program being a portable touch screen app so they can continue working on their music on the go but they also pointed out that some of the features were not as advanced as the desktop applications. The desktop applications that were reviewed were not targeted at beginner users as some of the features were too complicated and functionality was not as easy as the touch screen applications. The suggestion of a desktop application with touch screen capabilities was received well. Most said they would be interested in using a program which combined the desirable features of both a touch screen and desktop applications.

2.4 Music Research

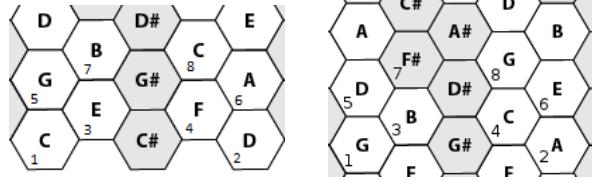


In musical theory, an interval is the distance between two notes or pitches, the degree of highness or lowness of a note — based on its frequency. Typically, these are represented with tones and semitones, where tones are the base unit of an interval. Certain intervals appear particularly often within music composition as they can provide key melodies. Such intervals include the perfect fifth, and major and minor thirds. These intervals, which are 7, 4 and 3 semitones above the base note respectively, make up the notes used in major and minor triads and are the intervals used in the harmonic table based on the following set of rules:

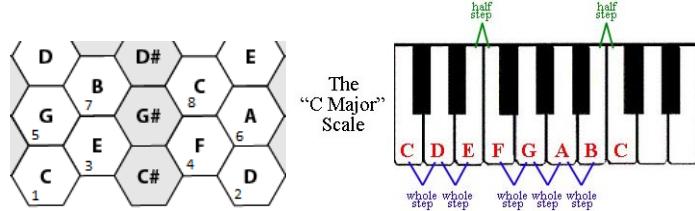
- Notes ascend by an interval of a fifth on the vertical axis.
- Notes ascend by an interval of a major third on one diagonal axis (north easterly).
- Notes ascend by an interval of a minor third on the other diagonal axis (north westerly).

This layout clusters complementary notes together, for example, all the notes in the C Major chord, namely C, E and G are adjoining. Scales are played by using simple, repeated patterns that can be transposed for playing in different keys.

Transposition, *the shifting of a melody, a harmonic progression or entire musical piece to another key, while maintaining the same tone structure* [10], can be done far more easily on a harmonic keyboard than on a conventional keyboard as the basic pattern is always retained. This is shown in a comparison between the C and G major scale below:



The C Major scale (left) cf. the G Major scale (right).
Note the same progression.



Above is shown a comparison of the C major scale played on a harmonic table based keyboard (left) and on a conventional piano keyboard (right). The progression is enumerated from 1 to 8.

2.5 Technical Research

2.5.1 MIDI

MIDI, short for Musical Instrument Digital Interface is a technical standard that describes a protocol, digital interface and connectors and allows a wide variety of electronic musical instruments, computers and other related devices to connect and communicate with one another [11]. A link in MIDI can handle 16 channels, routes of communication, to transmit information.

MIDI uses messages containing instructions which controls a device. A MIDI message consists of a status byte, which indicates the type of the message, followed by up to two data bytes that contain the parameters [12]. Using MIDI can you send channel messages which are heard only by the channel they are sent to, or system messages which are heard by all devices on all channels.

The main types of messages sent, in real time, over a single channel include [13]:

- Note on messages: Specify a notes, volume (velocity), note, pitch and the channel it should be played on.
- Note off messages: Same of note on, but turns off a note.
- Control change messages: Changes instrument settings.

This research gave us a grounding in MIDI; learning that it was possible to set each grid layer to be a channel, and each hexagon cell in the grid could send MIDI messages specifying note-on, with the note number, "velocity" "program" change [instrument] and note-off messages. These factors can help in emulating interesting features of existing products. One slight disappointment we found when researching MIDI was (at the time of the research) Android devices did not fully support MIDI [14].

2.5.2 Hash Maps

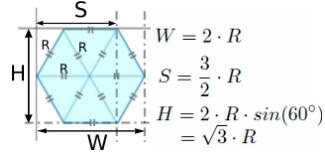
Hash maps are a data structure that use a hash function to generate unique keys for identifying values within a table. A key is made using a hash function that will index directly into the correct position, where the requested value will be found [15].

The hash function will assign each possible key to a unique location. This property of hash maps ensures that no two pulses occupy the same space on the grid. Our method of collision resolution is to simply overwrite keys in the event of a collision. This removes the need for advanced collision resolution algorithms such as *Separate chaining* or *Open addressing*. As such, the access time of the hash map will always be constant: affording an access time of $O(1)$ for insert, search and delete methods [16]. Of course we will need an array to keep track of the keys in use.

To create the perfect hash function it is necessary to ensure there are no *unwanted* collisions; collisions that occur with pulses in different locations on the grid. Some sort of hashing function in base 6 to take advantage of the senary properties of hexagons.

2.5.3 Working With Hexagons

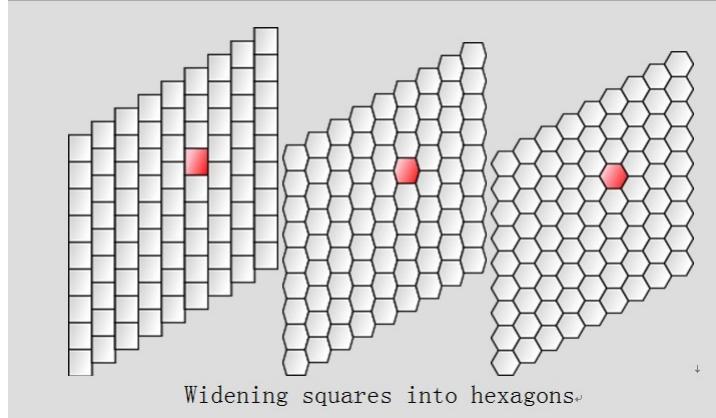
There are several operations one might need to perform when working with hexagons: finding a hexagon's position by its index in the grid; picking a hexagon by mouse; finding neighbour cells; finding hexagon's corner coordinates etc.



Using simple properties of hexagons, as shown in *Fig. 1*, we are able to derive useful methods of manipulating and arranging them, and produce relevant algorithms for use in our code.

We define the hexagon by its radius R , and find some other parameters based on it, such as W ("width"), S ("side") and H ("height"). We can calculate W, S, H easily with a known R .

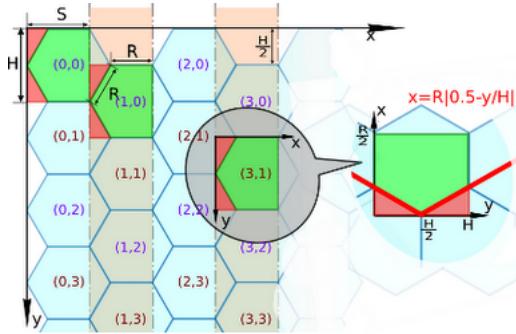
Hexagon grids can be derived from square grids. Since coordinate systems for squares are much easier, the derivation will guide us in designing coordinate systems for hexagons [17]. To turn a square grid into a hexagon grid, there are two steps needed. First, we must offset the columns (or rows). Second, we should split half the square edges and bend them in the middle.



The most common way [18] to offset columns is to offset every other column. Code using this approach looks at whether it is an odd or even column and chooses whether to offset it. A simpler approach is to offset each column by half a height more than the previous column. Code using this approach is more uniform, but the map shape is no longer rectangular, which can be inconvenient.

With either offset approach, the next step is to split the vertical edges of the squares and bend them. When the bend is reduced from 180 degrees to 120 degrees, we get regular hexagons.

2.5.4 Identifying Hexagons



From the picture above, we can figure out the formula for finding position of the top left corner of a hexagon cell by its array index (i, j):

$$y = \begin{cases} j \cdot H & , i \text{ is odd} \\ j \cdot H + \frac{H}{2} & , i \text{ is even} \end{cases}$$

8.

With this knowledge we are now able to perform basic the operations and manipulations with hexagons necessary for use in our software.

3 Requirements Specification

Details of key features to be required can be described in terms of the functional requirements, in other words, definitions of what the system must do; and non-functional requirements, definitions of how the system is supposed to be. These are detailed henceforth:

3.1 Functional Requirements

1. The application must be able to play a note at the press of a hexagonal tile on the grid.
 - (a) It must be possible to manipulate the sound through variation of voice/MIDI instrument.
2. The application must implement the Harmonic Table (see musical research for more) in the layout of playable notes.
 - (a) The notes must be set out on a grid of contiguous hexagons.
 - (b) Moving immediately up a hexagon will increase the pitch by an interval of a 5th and vice versa.
 - (c) Moving immediately right a hexagon will increase the pitch by a semitone and vice versa.
 - (d) Moving in a north easterly direction on the grid will increase the pitch by a major third, i.e. 4 semitones.
 - (e) Moving an a north westerly direction on the grid will increase the pitch by a minor third, i.e. 3 semitones.
3. ‘Operator tiles’ should be placed on individual hexagons within the grid to manipulate sound in various ways.
4. The following operator tiles should be implemented:
 - (a) *Play* - Begins the sequencer, sending an initial pulse in a straight line in a specified direction.
 - (b) *Stop* - Stops and ‘absorbs’ the pulse at that position.
 - (c) *Change* - Changes the direction of the pulse.

- (d) *Explode* - Causes the pulse to 'split', propagating new playheads from each edge of the hexagon it is placed on, but not going back on itself.
 - (e) *Warp* - When the playhead meets a warp tile, it will spawn new playheads at each other warp tile placed on the grid, moving in the direction of entry.
5. Tempo must be controlled in two ways:
- (a) A global base tempo must be set.
 - (b) Tempos on each grid can be set independently.
 - i. Individual grid tempos must be a multiple of two, relative to one another (i.e. grids can be twice, four times, eight or sixteen times faster than the base).
6. A moving 'pulse' or 'playhead' should be used to trigger tiles placed on hexagons.
- (a) The playhead must move faster or slower based on the tempo specified.
 - (b) The playhead must move consistently with the tempo specified.
7. Each operator tile should have an off or an on status, which dictates whether it will play a note when encountered by a pulse.
- (a) The sound produced is based on the position of the tile on the harmonic table grid, rather than the tile itself.
8. Operator tiles should be removed with a delete mechanism.
- (a) When a 'delete' key is depressed, a click must remove a tile from the grid.
 - (b) There must be a 'clear' button to remove all tiles from an individual grid.
9. The following operator tiles could be rotatable when placed such that they will send the playhead in different directions:
- (a) *Play*
 - (b) *Change*
10. Six layers should be implemented for use in harmonic, polyphonic projects.
- (a) Layers must be independent.
 - (b) It must be possible to place different tiles on different grids (i.e. all grids can be unique).
 - (c) Each grid must be able to have its own tempo.
 - (d) Layers must be able to be played simultaneously.
11. Global 'Start' and 'Stop' buttons should be implemented to start and stop playheads over all grids.
- (a) Individual grids must be able to be selected and deselected for play.
12. Saving and loading must be implemented for ease of producing and sharing projects.
- (a) Individual grids must be able to be saved independently
 - (b) Projects must be able to be saved as a whole.
13. The application must implement a control system suitable for a touch screen and also mouse interface.
- (a) The application must be controlled through use of single touch sliders and buttons.
 - (b) It must have either 'tap to scroll' and/or 'drag and drop' selection.
 - (c) Menus must be available through a single click mechanism.
14. The menu must contain at least the following key features:
- (a) Create new project.
 - (b) Save/Load.
 - (c) Select Grid.
 - (d) Change instrument on a certain grid.
 - (e) Select grids to play.
 - (f) Open help screen.

3.2 Non-Functional Requirements

Our application is aimed to be used by a wide variety of people, with varying levels of musical capability. With this in mind we must place much effort into making the application as accessible and easy to use as possible. As such, it is important that our project conforms to the following principles:

1. Extensibility - The application should be fully extensible, i.e. adding new features, and carry-forward of upgrades should be viable while maintaining the core mechanic.
2. Maintainability - The application should be easy to maintain.
3. Performance - The application should run comfortably on a wide range of computers.
4. Usability - the application should be easy to use and have simple, intuitive user input.

Part II

Design

4 Software Design

We looked at the Reactogon and decided to expand upon it, please look at the functional requirements for this.

4.1 The Grids

Our software will comprise of 6 layers; each layer being a grid. Each grid would have to be unique yet have different properties; this was the first indication that the object-oriented paradigm was suitable for our codebase. Each grid would have a set of properties, these properties being:

- Each grid will be assigned a unique instrument, but needs to support changing dynamically, mid play. This suggested that a grid should be assigned its own channel in a MIDI sequencer [?].
- Each grid can be dynamically transposed — supporting different octaves, through the research we found that MIDI can play 127 different notes ranging from octaves 0 to 10 [11]. This suggests that each grid should have a starting note and some sort of algorithm then fills the grids with correct notes.

The tempo of the grid will be changeable, but all grids must have synchronisable tempos based on a global base tempo. Grids can then play at $1\times$, $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$ or $\frac{1}{16}$ times the base pace, these values were chosen due to the research in music theory. This implies that a grid can be halted for set amount of time/tick.

The grid itself would be made up of hexagonal 'cells', these should hold information about what note each cell is, this could simply be a number. From this it would be sensible to store the cells as a 2D array. As discussed above this would then make it quite simple to transpose notes with an algorithm. Information about where it should be placed in the grid itself. Some of the properties of hexagons are outlined in the research and will help for placement and using hexagon properties to our advantage. Because this is quite a visual piece of software some information needs to be held about its state, as we wanted to show if a cell is active or not.

4.2 Creation of Sound

If 4 separate pulses hit 4 different tiles on the same tick, they all should be played at the exactly the same time, and all 6 grids must follow this rule. For this to happen, there needs to be some sort of queue, where all notes that should be played at a set point in time are grouped together, then sent off to be played all at once. This could happen multiple times per second, so it looks like some sort of thread will be needed to play multiple sequences at the same time and will not halt the program.

4.3 Pulse

The vision of a pulse is some sort of object moving through the grid, from cell to cell. The pulse itself should be quite simple as all it must do is move from one cell to another. The tempo of the grid should manipulate how fast the pulse moves from one to another. The grid must be able to support multiple pulses, all moving in different directions at the same time. As well as this, pulses must not affect one and other, crossing/colliding pulses shouldn't be a problem and the software must be able to handle this. But the software must be able to handle pulses that occupy the same space, as not to create exponential loops see figure 1. When a pulse comes into contact with a tile it should affect the pulse in some way.

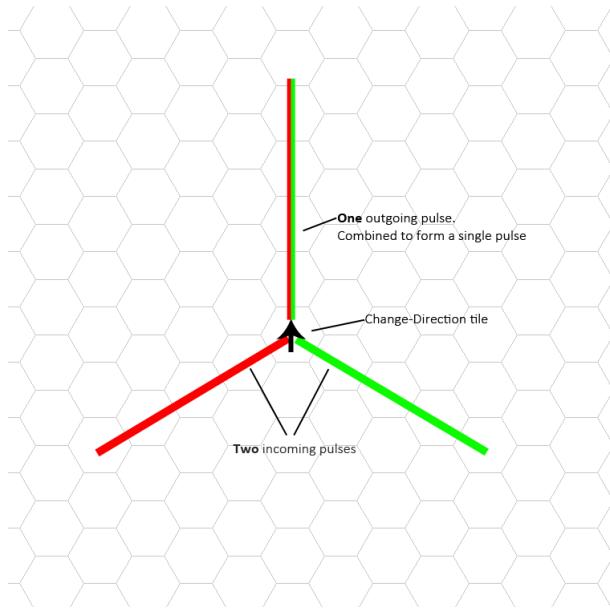


fig. 1

4.4 Tiles

The main conceit of the software is the placement of tiles to manipulate music. A tile can be placed in a cell on the grid and used to alter a pulse when encountered. Different tiles have different effects. The tiles are listed as follows:

- Stop: Should stop and destroy the pulse.
- Change direction: This tile should hold a direction and pass it to an incoming pulse, redirecting it. Graphics will need to be updated when the tile is clicked to rotate the tile and the direction changed.
- Play: Will work in much the same way as 'change direction' as it should be able to change a pulses direction. As well as this, it will be the starting position of a pulse, creating it.
- Explode: Will destroy an incoming pulse in much the same way as stop does, but in turn create 5 new pulses sending them in every direction bar the input direction. It would be sensible to assume that a pulse should hold information about its previous position in the grid.
- Warp: Will warp a pulse from one place to another. If multiple warps are places then the pulse should come out of those tiles also. A pulse will be destroyed and new pulses created in all output locations. There will need to be some way of storing locations of all warp tiles in a grid, all with their own unique ID.

The tile object should therefore have a type and a direction at minimum. When a pulse comes in contact with a tile it should play a note, but there are also tiles that should not play a note, this means there must be some data variables stored in each tile if it should play a note or not when hit by a pulse.

4.5 Keeping Time

The whole system needs to produce music at a reliable, constant rhythm, therefore there must be some timer that tells each grid to increment all pulses by one step. The timer must always have the same interval.

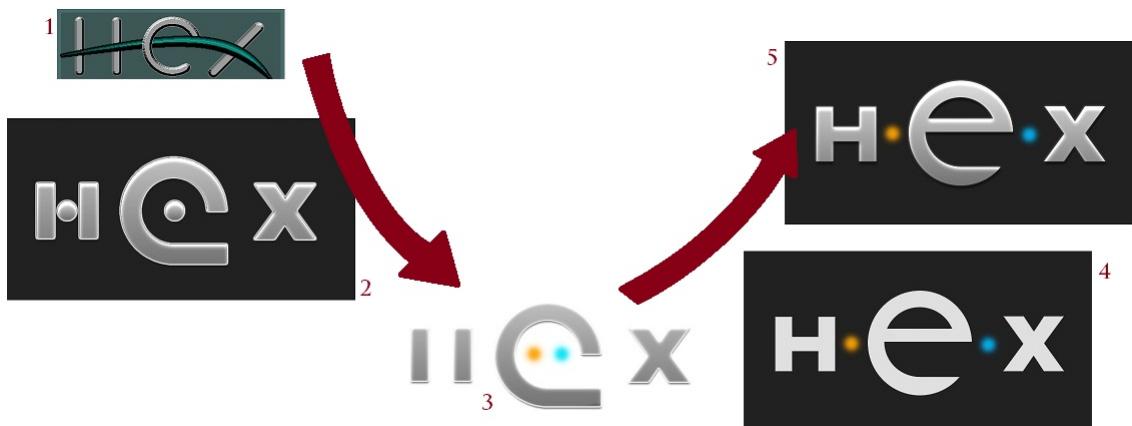
5 User Interface Designs

The colour scheme of our software is grey-scale, with blue and orange highlights. This is comparable to the popular video game portal 2, and allows for easy differentials to be made between different functions of each tile, as well as allowing for an attractive overall look. The design of the tiles has been through several iterations as can be seen below:

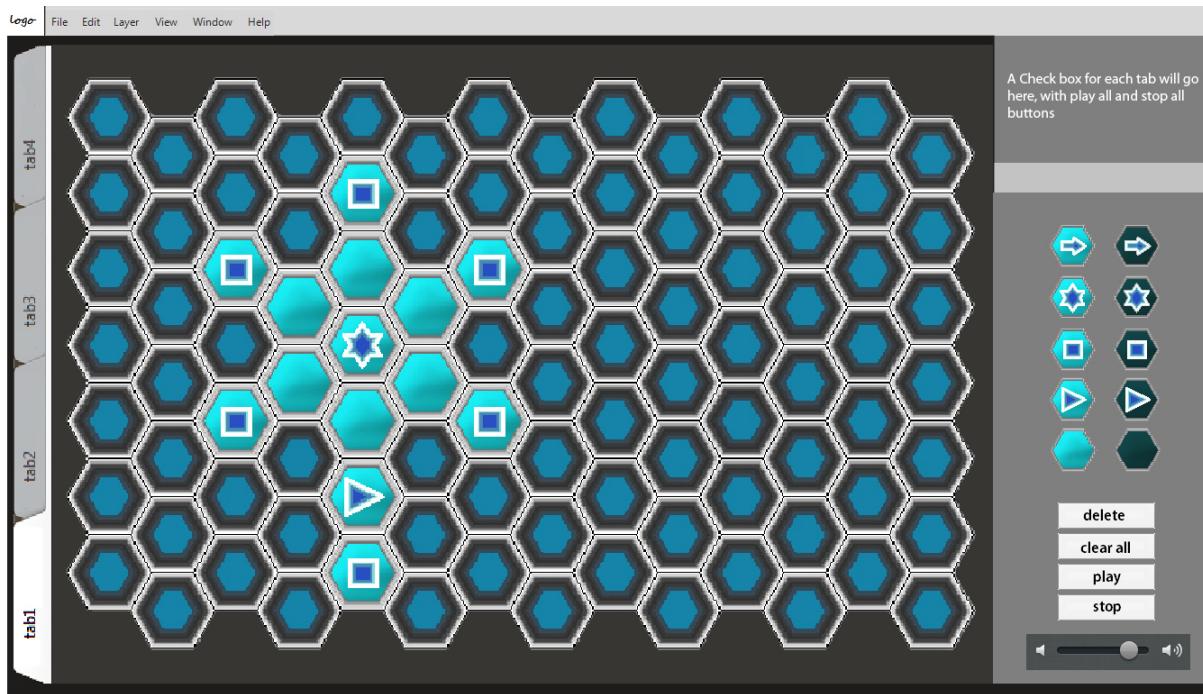


In the end, we decided on a drop shadow chrome look, accented with each colour depending on the corresponding tile's purpose. The progression is from left to right, and illustrates the process in which we went through designing the tiles. We started with a functional place holder, before investing more time into the appearance of the tile for the second iteration, and finally made a high resolution version for implementation. Each area of the graphics underwent a similar design progression.

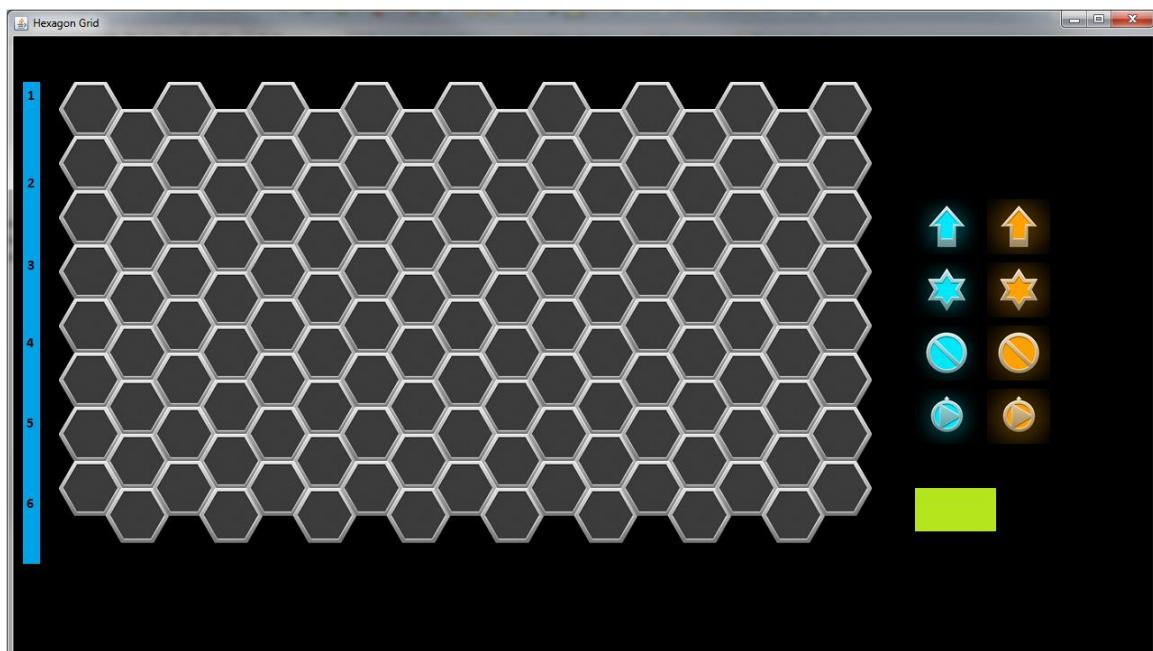
The changes in the logo design can be seen below, and illustrates a process of refinement which involved critique from group members.



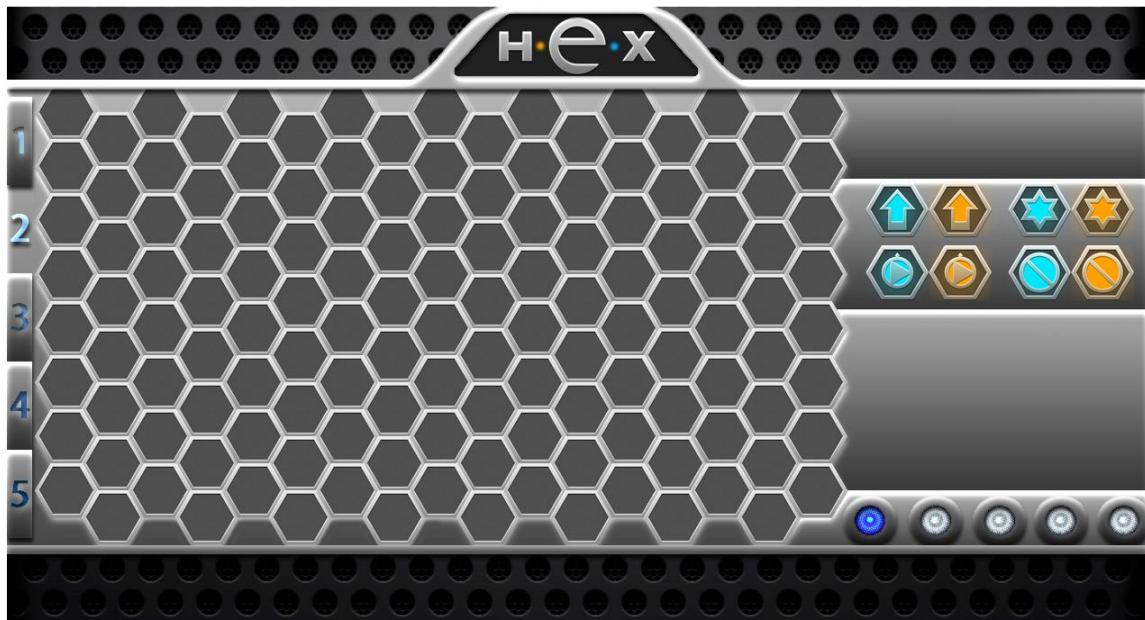
The control section of the window changed over time, as we added new features to the project and developed the design theme and layout. The first version we put serious time into introduced the metallic theme to compliment the logo and tiles. There were a few issues with the chunkiness and curviness of some aspects and in order to improve beyond this, we decided upon a new look. We split the window into two halves; moved the tabs below the grid and introduced a more simplistic control panel look. These improvements were very well received, so the rest of the graphics were updated to match, yielding our final design as can be seen below:



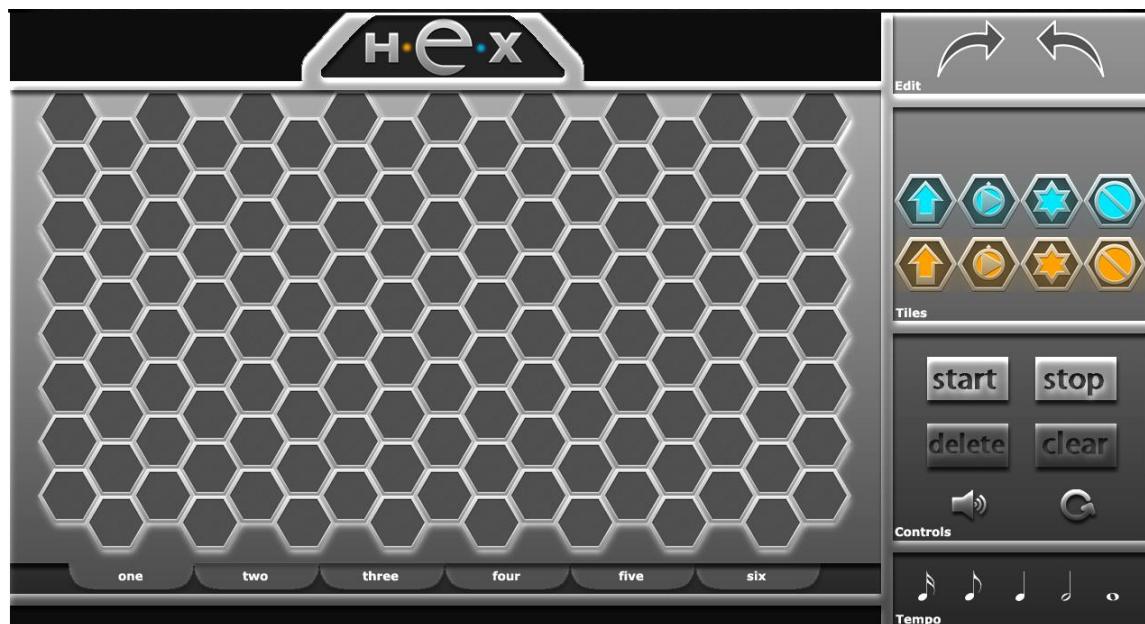
This was the first design we came up with, and we decided early on where we wanted to initially put the controls, and the basic layout.



We then implemented the high definition tiles, and started to attempt to place the grid at x and y coordinates which would work with our idea for the finished design.



Following this, an update design proposal was made, with a background and more fancy integrated controls.



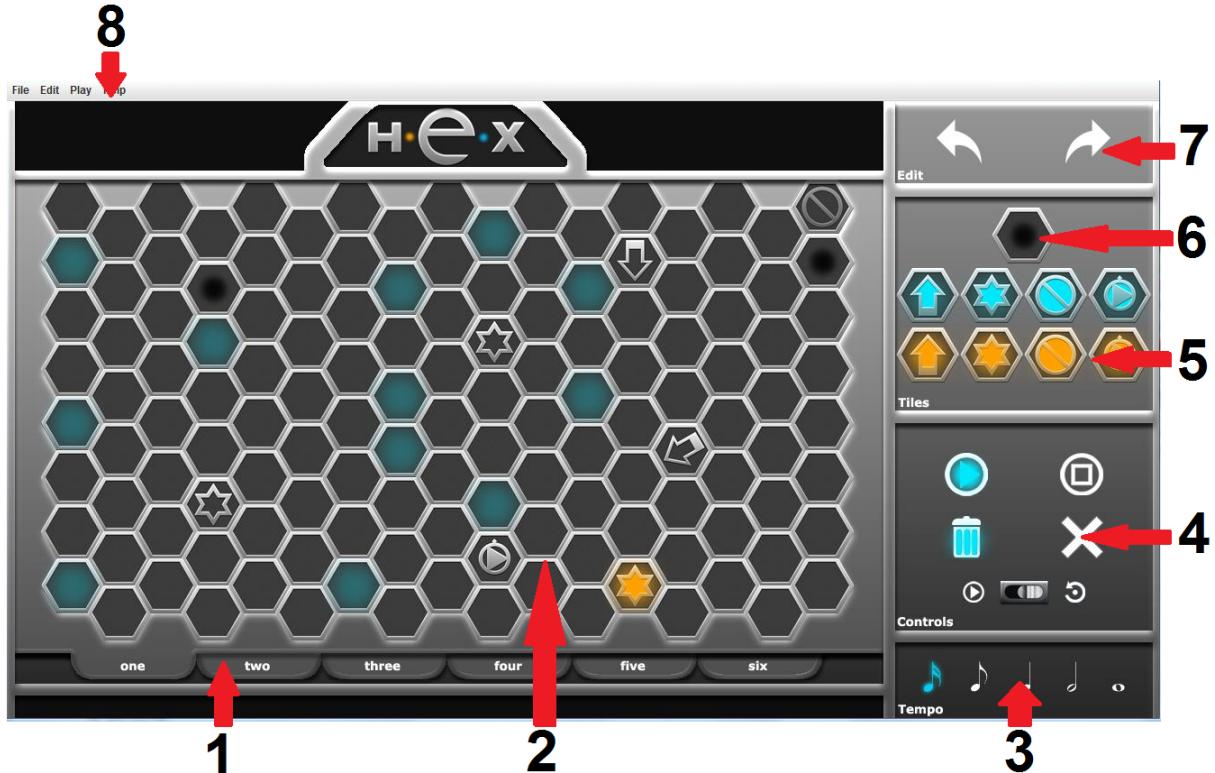
Following from a critique of the updated design, we made several refinements, and completed some updated controls. We also relocated the tabs to the bottom of the grid, as we decided it looked better.



We then made our final changes, consolidating the design into a simpler and more clean look. A ridge was also added to the tabs to make them look better, and the rest of the graphics were finished and integrated.

As you can see, we have a consistent colour scheme, with elements of each part of the static and interactive design visibly tied together to give the impression of quality and completeness.

Below is a description of the various features of our software's layout as labelled here:



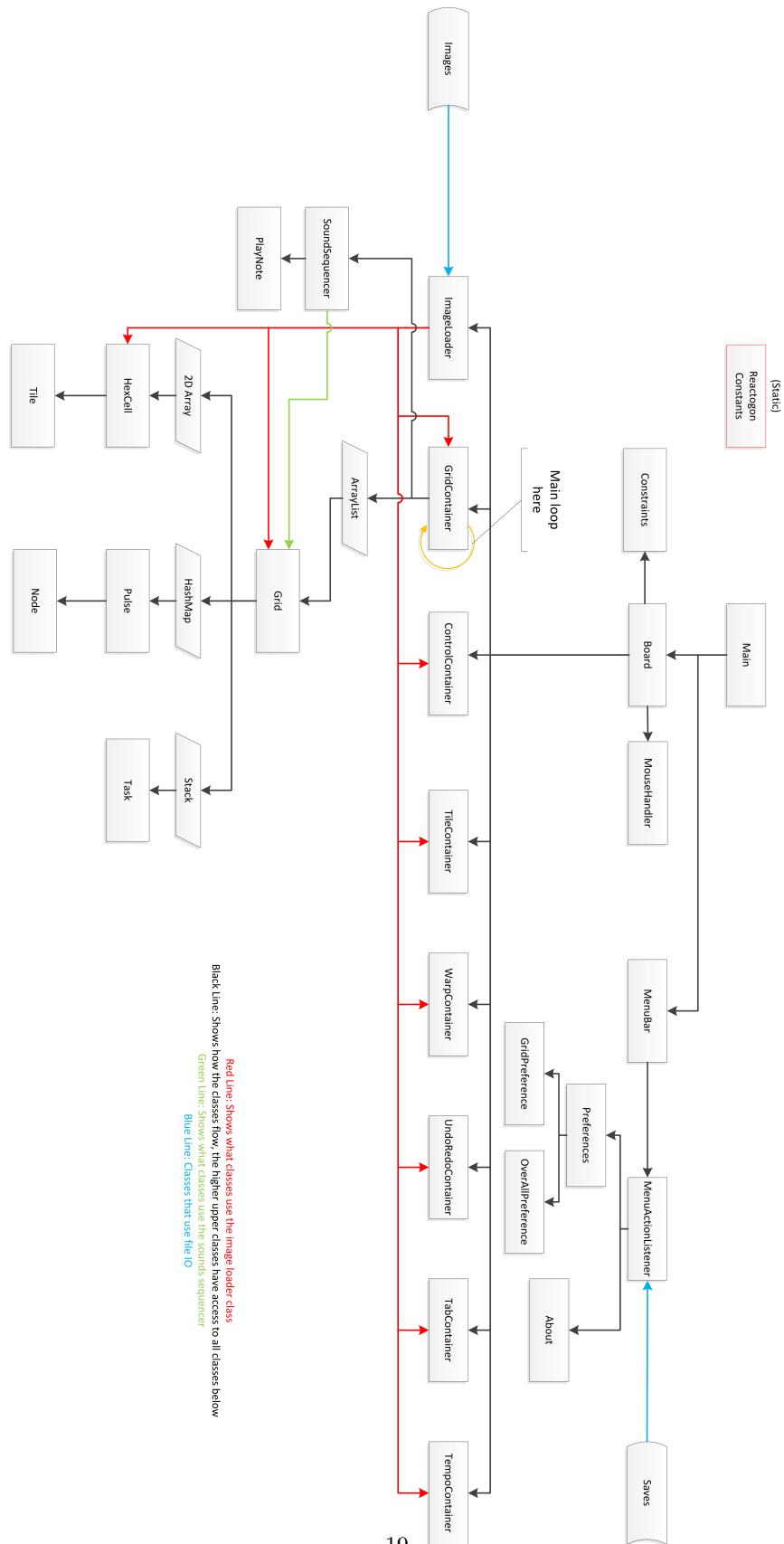
1. This is where the user changes the tab they are editing. The design of this tries to make it obvious that the tabs are above one another and how to change between them.

2. The grid makes up for the bulk of the window real-estate, and rightly so as it is the main interactive element of the program. This is situated as can be seen, in the centre of the GUI, to the left of the controls.
3. The user can change the tempo of the current tab here. The symbols of semiquaver, quaver, crotchet, minim and semibreve represent 16x, 8x, 4x, 2x, 1x, from left to right, respectively. This is to allow for an intuitive control of the current tab's tempo.
4. This is the controls section of the control bar. It includes buttons which start/stop the pulses, enable deletion of individual tiles or clear the entire grid. Below these controls is a switch which changes what happens when the user clicks on the grid. The right hand selection will rotate a tile clicked on, and the left hand option will play the note corresponding with where the grid was clicked.
5. This is where the tiles are placed. The blue tiles play a note when hit by a pulse, and then carry out their function, whereas the orange tiles carry out their function silently. A user clicks on a tile before choosing where to place it in the grid.
6. This is the latest tile addition and is different to all the others. It allows the user to transport the pulse and its direction across the grid, without having to travel there. It is called the warp tile and is selected and placed like the other tiles.
7. This is the edit section of the control bar. It allows the user to undo or redo any actions on the grid. The selection of grids to be played is now located in the menu.
8. The menu bar is situated at the top left for conformity reasons. On a windows application, the menu is regularly here, and this is what our software is designed for.

Part III

Implementation

6 Code Plan



The code plan shows how classes interact with each other. Classes higher in the tree create the classes below them, and therefore can access their methods. The red lines show which classes use the image loader, and in turn draw graphics to the screen. The green line shows which classes can talk to the sound sequencer and play or create notes. The blue line shows file input and output.

Important classes:

- *Board*

This manages mouse clicks, and contains the timer which in turn starts the loop inside the grid container. It is also where all container classes are made and then stored in an array list. *Board* calls the paint method for all other classes.

- All container classes

There are many container classes, they are used to make mouse clicking more efficient. They are basically classes containing hit boxes, and they are also used to draw the images contained within that hit box. The one exception to this is *GridContainer* which holds more information.

- *GridContainer*

This contains an array list of all grids. Within this class is the main loop which controls how often pulses are incremented, and items discussed below in the timing and tempo section. It includes methods to put tiles into grids, as well as starting and stopping grids.

- *Grid*

This contains the *play()* method which is the central control algorithm. This tells the pulse how it should move through the grid. It has information on what to do when a pulse comes in contact with a tile. It holds all of the hexagon cells for that grid, as well as a hash map of pulses. Each grid has its own undo and redo stack — allowing items specific to that grid to be undone and redone.

7 Central Control Algorithm

The following pseudocode details the main algorithm which iterates over all pulses on each tick, updating the grid. See source code for full implementation.

```

loop{
    hashTable.removePulse(currentKey)
    location = pulse.getLocation()
    nextLocation = getMove(pulse.direction, location)
    if( nextLocation = blank)
        pulse.nextLocation() = nextLocation
    else
        getMove(hashTable, nextLocation.getTile)
        soundSequencer.enqueueNotes()
    update()
}

```

8 Key Control Elements, Data Structures and Methods

8.1 Removal of Duplicates

Removal of duplicates,—figure 1— in the design section shows a problem of pulses merging together. Two or more pulses could occupy the same space and this would cause the number of pulses to grow exponentially and in turn cause the program to slow down, eventually crashing due to large array sizes. The method for removing duplicates from this array was an algorithm similar to the bubblesort mechanism - continually comparing elements. As such this algorithm would take $O(n^2)$ time; where n is the number of elements in the array.

To solve this problem, a hash map was used to store the current location of every pulse in each grid.

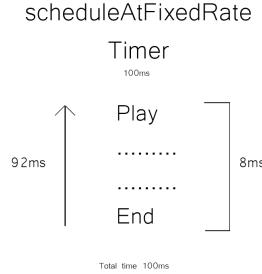
The hashing algorithm took the pulses current location and direction to generate a unique key to place it in the hash map. The hashing algorithm is shown below (equation 1). The hashing algorithm was relatively simple as, by virtue of using hexagons, it was possible to work in base 6. By doing this, if two pulses occupied the same space on the grid (going the same direction etc.) they would be placed in the same position in the hash map. This would cause a collision, but as a hash map only accepts unique values, the second value (and any after that) would simply write over the old value. See the Hash Maps section of Technical Research for more information.

To keep track of the keys in the hash map there is an array list holding all keys currently in use. To access the pulses it was necessary to traverse through this list and use the keys to get data from the hash map. Every loop, the array list was emptied and all keys were added again. By using a hash map access time was reduced to $O(1)$ and removal was never a problem, as a hash map with no linear probing just writes over a value. It also meant there was no adding or removing of pulses from an array.

$$Key = ((Pulse.iLocation \times 36) + (Pulse.jLocation \times 6) + direction) \quad (1)$$

8.2 Timing and Tempo

Timing the whole system was very important. To do this we used a Timer that was scheduled at a fixed rate. This meant that it would call the run method every set amount of time irrespective of if it had finished the methods or not. The timer controlled how often the pulses moved one space in the grid. We needed to keep all grids synchronised while still having different tempos. To achieve this, each grid is assigned a value of 1, 2, 4, 8, or 16 to represent its speed/different tempos. See the Grids Section in Design for more information.



Every loop of the base timer a value is incremented by 1. The modulus of this value is then taken against the tempo of each grid, if it is 0 then it will move the pulses inside this grid by one, otherwise it will skip this grid. This is explained clearly in the code below for the *play()* method. At the end of every loop queues of notes are played. This mechanism keeps everything in sync.

8.2.1 Tempo Controller

```
function play()
    if (tempoControl > 16)
        tempoControl = 1
    for all grids
        if (tempoControl % grid.getTempo == 0)
            grid.play()
    tempoControl++
    playNotes()
```

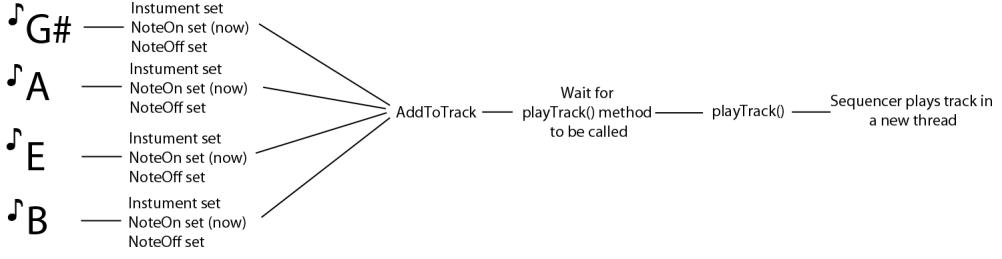
8.3 Sound Production

Sound creation went through many stages, and is arguably still a work in progress. The most important part of the system was keeping everything synchronised. If twenty notes were scheduled to play, then they should all be played at exactly the same time, not milliseconds apart. Figure 3 details the *playNotes()* method. This plays a track containing all notes that should be played on that tick. The pulse moves through the grid, step by step, reacting when encountering a tile.

If that tile is set to play a note, then the *queueNote* method (figure 8) is called and the note for that hexagon is passed into the sound sequencer. Once a loop is completed, the *playNotes()* method sends this track to a sequencer in a new thread and plays all notes simultaneously. This occurs for each iteration. This process is detailed in figure 9.

8.3.1 queueNote Method

```
soundSequencer.queueNote(hexCellCurrent.getNote(), channel);
```

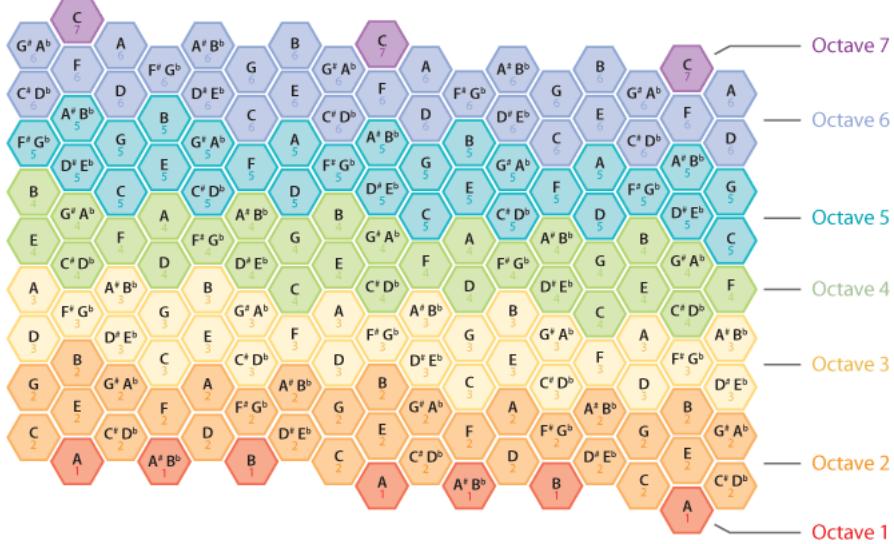


8.4 Saving and Loading

Saving and loading is useful for a program like HEX as it allows the user to share content and resume a project they were unable to complete in one session. The saved file is based on a CSV (comma separated values) with our own suffix '.hex'. Upon the user clicking save they are presented with a dialog to choose which grids they want to save. Each grid will initially save the instrument, start note and tempo. A loop then runs for all cells in the grid that have a tile present, storing the x and y coordinates along with the type and direction of the tile.

Loading is essentially the reverse of saving in the sense that saving gets information whereas loading sets the information. When the user clicks load they are presented with a dialog box to select their previously saved file. At this point another dialog box is presented stating the next grid that will be loaded from the saved file. The user can choose to either not load this grid or to store it in any of the 6 grids on the board. If the user stores 2 grids within the same grid, only the latest selection will be stored. When a grid is loaded, the grid's instrument, tempo and start note are restored along with all the tiles coordinates, type and direction.

8.5 Transposition



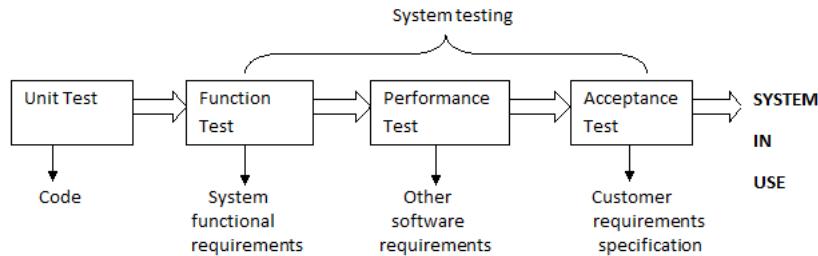
The transposition of grids, changing the start note of each grid on the fly, was much easier to implement than expected. As the grid is a 2D array of hexagons, MIDI notes can be stored as numbers. Having a numerical representation of notes within a 2D array, and having knowledge of the harmonic table (above) allows a simple but powerful formula to be produced for transposition (below). Using this function within a loop over the array means it can be called at any time — even during play. This formula works for any size of grid, so if we wanted to extend the table at some point that would be possible.

8.5.1 Transposition Formula

The note for any given i, and j cell on the grid can be given by:

$$\begin{aligned}
 n &= \text{startNote} - (7 \times j) \\
 \text{note} &= ((n + 4) - (((i \% 2) \times 7)))
 \end{aligned}$$

9 Testing



Planned Testing Phases

The objective of testing is to find 'faults, defects in design requirements and in code'[19]. With the help of testing, the defects can be fixed and the program can be improved to better meet customers needs. In addition, if tests find few faults, it can lend confidence in the quality of software. Our tests are planned to be completed in four phases which are unit, function, performance and acceptance testing. Once some components of our program are coded, we can begin testing. The sections below only provide an overview of the testing of the system. Detail test cases are given in the appendix.

9.1 Unit Testing

Unit tests are written with JUnit. Some code is written to provide a predetermined set of data to the components which are being tested. The results are observed and the integrity of the internal data, logic, boundary conditions for input and output data [19] are checked. To write a unit test, the first step is to read through the code and figure out exactly what the functions and classes do. The code must be compiled, then using some test data as input, the output is compared with the expected result. In JUnit, the function `assertEquals()` is usually used for this.

For example, the class HexCell creates a hexagon, calculates its dimensions, location in the grid, MIDI note number and image assigned to it. Functions within the class include `getImage()`, `flipState()`, `setNote()` etc.

The class HexCellTest is created as a unit test. In this test class, some initial values are set up.

Example code:

```
assertEquals(il.getImage(tile, state), hc.getImage());
```

The code above is used to test the `getImage()` method. The first parameter in `assertEquals()` is the expected output and the second one is the actual output. If they are the same, when running JUnit, the test of this method will pass. This means the components which are being tested meet the requirements.

There are other unit tests like PulseTest and TileTest. Some example data is provided to test the methods. Further test code is provided in the project file.

When finishing testing the code, the system testing can be initiated.

9.2 Functional Testing

Function tests are run to check whether the program is satisfying the customer's needs, they are based on the functional requirements. Our functional tests work at two levels:

1. Code level: Add `System.out.println()` in the code to print some example output data and check if it is processing right.

For example, code `System.out.println("Note =" + note);` is added in method `playNote()` and `queueNote()` to show the MIDI note numbers.

2. User interface level. Observe the results in the user interface level. For example, observe the image, the lights and sounds which the application actually produces. Compare these with the result that the users are expecting. If they are exactly the same, the test is passed which means it meets the requirement. Otherwise, the test is failed and debugging is needed to make the test pass. An example test is shown in the following section.

9.2.1 User Interface Testing Example

Functional requirement to test: Explode operator tile which can split a pulse into 5 different directions should be implemented.

Test steps:

1. Click and place the blue play tile in the first empty space within the grid.
2. Click and place the blue explode tile on the second empty space which is in the path of the first.
3. Click the start button.

Expected result in steps:

1. Should display a grey 'play' tile on the first hexagonal tile.
2. Should display a grey 'explode' tile on the second hexagonal tile.
3. Should play a note and display blue light which represents the pulse from the first tile to the second tile. Then it should play a different note at the second tile and display blue lights from the second tile to all directions except the direction of input.

Actual result in steps:

1. A grey play tile displays on the first hexagonal tile. Example output data: The node is (10,6)
2. A grey explode tile displays on the second hexagonal tile. Example output data: The node is (10,3)
3. The application plays a note and a blue light displays from the first tile to the second tile. Then the application plays a different note at the second tile and displays blue lights from the second tile to all directions except the direction of input.

Example output data:

Note =67. The direction is 0.

The node is (10,6). Now the direction is 0.

The node is (10,5)

This output data is very long and the complete data is shown in the appendix. The actual results and the accepted ones are the same after comparing. Hence this test is passed. There are a substantial number of functional tests needed to make sure that the system operates as required. The detailed test cases are provided in the appendix.

9.3 Performance Testing

Performance tests are run to check whether the program meets the nonfunctional requirements. We decided to cover three types of performance tests which are stress, compatibility and usability tests.

9.3.1 Stress Testing

This test is to ensure the software does not crash through creating conditions of insufficient computational resources such as low memory or lack of disk space.

The CPU usage rate is used in this test, although this may not be very accurate for the reason that the CPU usage rate is always changing in a small range.

System Spec:

- Intel®Core™i5-2450M CPU @ 2.5GHz (4 CPUs)
- 6GB DDR3 RAM
- Windows 7 Home Premium x64

Test:

Before running the application, the CPU usage rate is 0-1%. When opening the application it rises to 38% at most and stabilized around 8%. When put the operator tiles on the grid and start to play note, the CPU usage rate is around 7% to 12%. There is no obvious changes when increase the number of tiles on the grid. When use more grids to play note concurrently, the CPU usage rate increase to around 22 at most and stabilized around 9. The average CPU utilization rate of the application is around 5.7. Hence, our applications CPU usage is not very high and it is not easy to crash even in conditions of insufficient computational resources.

9.3.2 Compatability Testing

The compatibility testing is based on the non-functional requirement that the application should run comfortably on a wide range of computers. It is done to check if our application can run on different configurations of hardware. The application is tested to run on my laptop at home, my classmates computer, computers in school lab and library computer. It can run successfully on all. Other tests have been run on various GNU plus Linux distributions and have all been successful. Therefore it can be concluded that the application does run on Windows 7 and the sampled linux distributions.

9.3.3 Usability Testing

Usability testing, which is also called human factors testing, is based on the non-functional requirement that the application should be easy to use. It is concerned mainly with the use of the application.

The test is divided into two aspects. The first one is to test the efficiency of using the application, calculate the time and the steps to complete basic tasks such as load a hex file, save a hex file and stop a beat. The application can respond immediately to complete tasks. And the steps are designed to be minimized. When start runs the application, the time to create layout is 505.0 ms(in some computers, the time may be longer).When using the application, for example, to play note through operator tiles, the basic steps are clicking the operator tile you want to add, clicking on the place on the grid and finally clicking the start button. And it only cost several seconds.

The second one is to test the emotional response. It is to check if the user interface is easy to use and understand. Our application contains a clear background, clear layout and meaningful images for the control buttons. There are few words on the board to make things simpler. When the user does not know the meaning of the icons, the help menu can be used. In addition, the users may make mistakes and the edit button undo, redo can be useful. Furthermore, Keyboard Shortcuts are enable to make sure the ease of use. The customers feelings are asked in the acceptable test.

9.3.4 Acceptance Testing

The acceptable tests are conducted, evaluated by users. So we ask some other people to use our application, ask them to check if our application can meet their needs. This can also test the installation of the software when we run the software on the other computers. This is an ongoing test with each release of our software.

Part IV

Project Meta-Comment

10 Updates since Interim Report

There have been numerous updates to the project as a whole since the last report. The most obvious change is the overhaul to the software frontend and graphical user interface. Placeholder graphics have been replaced with finalised, heavily improved designs for a more immersive and polished feel, this is described in detail in the User Interface Designs section. In addition to this, a menu system has been added, where most major functionality can be accessed.

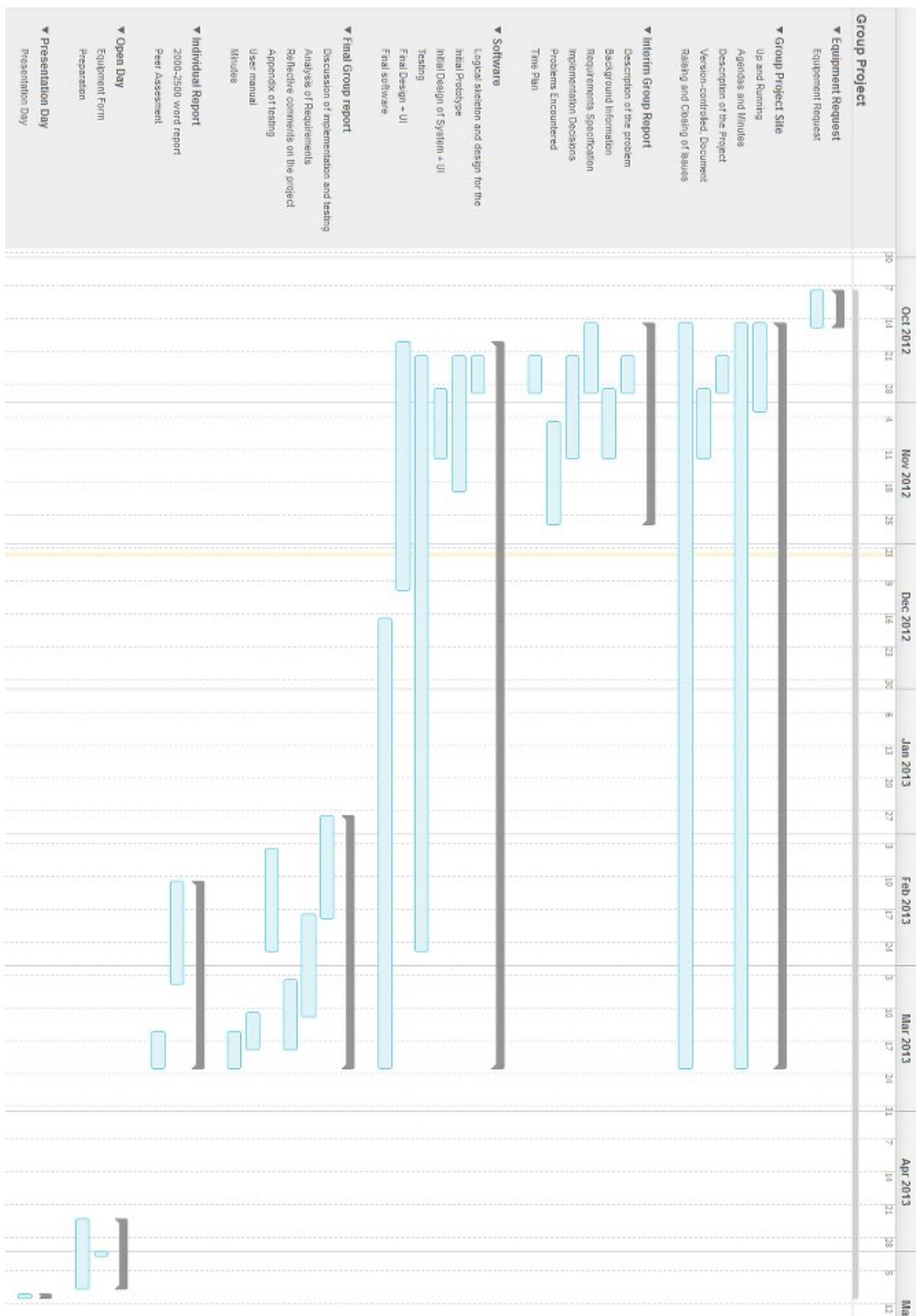
In terms of changes to functionality of the software, since the interim report we have implemented multiple grids; the *Warp* tile; the ability to change grid tempo and instrument on the fly and saving and loading have been finished and added.

The code has also been vastly improved in terms of efficiency, having completely rewritten the sound sequencer, decreasing memory usage and required CPU time.

Overall, in terms of improvements and updates, we have attempted to turn what was effectively a prototype into a finished, complete product.

11 Time and Planning

At the start of the project we immediately contacted each other via email to meet in order to organise meeting times and to assign roles within the project to each team member. After comparing each other's timetables we agreed to meet 4 times in a week; one 30 minute formal meeting with our supervisor, one 1 hour informal meeting at the beginning of each week to separate tasks for that week and review tasks completed the previous weeks, a 4 hour group working session and a quick 20 minute meeting before our formal meeting with our supervisor. We also agreed early on that the best way to monitor our progress and organise our time would be to create a Gantt chart (See below).

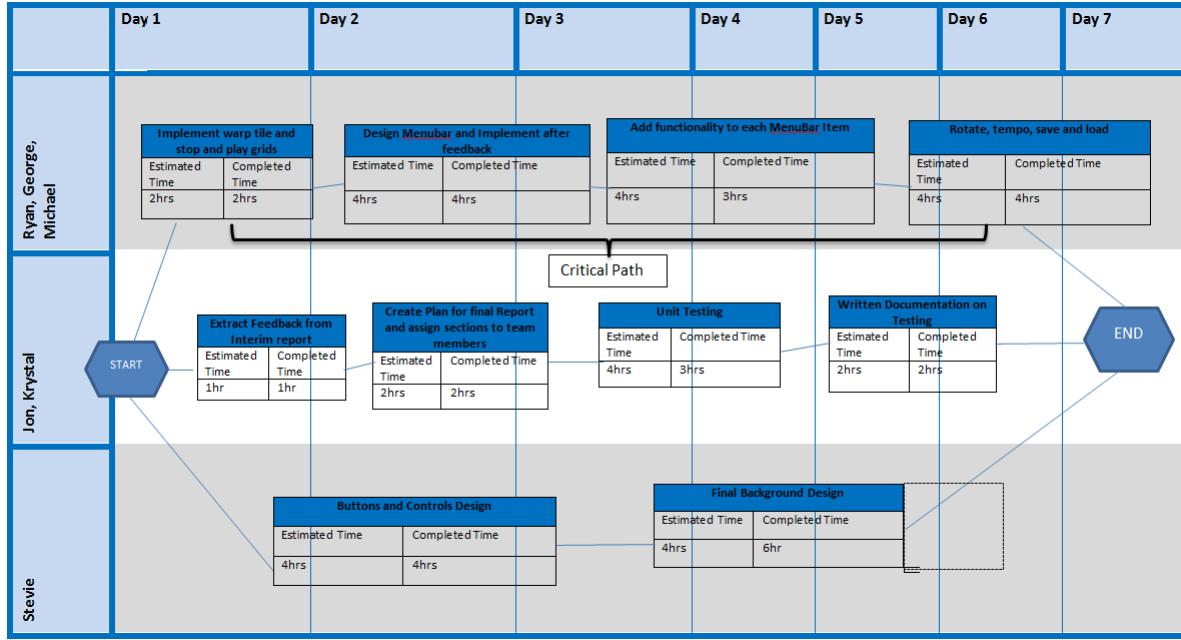


The first thing we needed to do was create work breakdown structure so we separated tasks into different topics and then created sub-topics for each. Using deadline dates as guidance we made a rough estimate of how long each task will take then broke that down further for each sub topic and finally determined the order of the tasks. There is also a percentage complete column inserted, which was updated regularly so we can view progress on that particular section. We can also see if we are on track to complete each section. Using the Gantt chart we could view the critical path of the project as a whole. Critical path analysis shows an overview of the time-scale of the project - showing us if we are on track to complete the project on time. The critical path allows for slack time, so if a task is taking longer than necessary then it can be compensated as tasks overlap and eventually the critical path is not affected. For each subtopic, tasks were created and placed onto an online Kanban board.

KanBanFlow Main Board

In our 1 hour informal meetings we were all assigned tasks for that week; these tasks were placed on a Kanban board using [20]. A Kanban board is an electronic board, which contains all the tasks in the product backlog (tasks that need to be completed); each task is separated into different columns, tasks that need to be done, tasks being analysed, tasks in development, tasks being tested or awaiting feedback and finally tasks completed. Each task has a name and description, each type of task is colour coded to correspond to a topic in the Gantt chart and also is an effective way of differentiating between the types of tasks. A team member is assigned to each task and a time estimate is made, when a task is completed the team member responsible fills in the time it took them to complete it. While a task is being done each member can show their progress by moving their task to the appropriate column. Each task has a due date and as it is an interactive board, it sends reminders to your email to tell you a task is due soon. The Kanban board was a very effective way of assigning tasks to each other and also a brilliant overview of the immediate tasks that needed to be completed. By using the Kanban board we were able to create a pert chart using the tasks.

A pert chart is similar to a Gantt chart but for a shorter timescale and each task in it is more detailed and specific. By using the tasks on the kanban board, we created a weekly Pert Chart to ensure the weekly tasks were completed and if any task is taking longer than usual then we were able to compensate as a critical path was determined first. As tasks weren't completed sequentially but in parallel we could allow for slack time as well. Below is an example of the weekly charts we created.



We used the tasks that were created at the beginning of the week. The critical path was then identified. The tasks that were on the critical path had to be monitored closely to ensure the successful completion of tasks that week, we did this by keeping in contact daily via Facebook or email or during our weekly group work session. Also during our Monday meetings we reviewed both the pert chart and Kanban boards and checked if we had successfully completed our assigned tasks for that week after this we updated our Gantt to view the overall progress of the project and check if we were on schedule to meet deadlines.

12 Problems Encountered

Fear of conflict remains to be a constant problem in group working and in our team it was apparent in the early stages of the project. As we were working with each other for the first time nobody wanted to offend each other so opinions on certain matters weren't challenged and constructive criticism was taken in the wrong way which in turn affected the quality of our initial work. Fear of conflict led to the avoidance of accountability this was because at times the team would refuse to confront difficult issues within the project by avoiding responsibility with the issue at hand and just focus on their given roles. These issues were eradicated with time as the more we worked together the better understanding we had of each other so we were regularly able to provide each other feedback with tasks and also raise any issues if we had any and admit responsibility to some issues. This led to the team being more productive as the focus was working together to produce a high standard of work instead of everyone for their own.

Working in a team meant that competency levels within the team differed. For example none of the team had considerable experience in testing. As a result some team members were asked to learn more about J-unit testing and begin writing test cases. Our lack of experience was evident as we weren't sure which parts of the program needed testing, therefore we didn't have sufficient test cases for our initial prototypes. This made it difficult for our developers to see which part of the code had bugs and needed changes, but after advice from our supervisor and increased proficiency in testing our latest prototype had sufficient test cases. Another issue with different competency levels was that some of the team were working considerably more than others, this was apparent when we were assigning tasks at the beginning of the week, lack of commitment was evident as some would avoid taking responsibility for the harder tasks and just take the smaller ones or even take responsibility for the tasks and not produce their work within the given time. By organising a group working session we were able to help each other stick to task deadlines and motivate each other.

Part V

Conclusion

It is our conviction that we have produced a project of a high quality and high standard. 'HEX' conforms to, fulfills and arguably exceeds the specification drawn up at the beginning of the project. Not only have we successfully emulated the reacTogen hardware, but we have also extended it: affording a truly novel musical experience. The success of the software not only lies in the unique, neoteric functionality and experience; but also in the usability and contemporary design of the user interface. These factors coupled with the quality of the codebase, which employs Java best practice and object-oriented design yields a culmination of our efforts and an ultimately successful project.

13 References

References

- [1] M. Kaltenbrunner, “Tangible musical interfaces,” 2013. [Online]. Available: <http://blog.makezine.com/2008/04/17/reactogen-arpeggiator-tab/>
- [2] [Online]. Available: http://en.wikipedia.org/wiki/Harmonic_table_note_layout/
- [3] [Online]. Available: <https://itunes.apple.com/app/hexatone-pro-idm-rhythm-generator/id324435715?mt=8>
- [4] [Online]. Available: <http://www.blabbermouth.net/news.aspx?mode=ArticlenewsitemID=124671>
- [5] [Online]. Available: <http://amidio.com/manuals/JR-Hexatone-Pro-1.0-User-Manual.pdf>
- [6] [Online]. Available: <https://itunes.apple.com/gb/app/tonepad/id315980301?mt=8>
- [7] [Online]. Available: <http://en.wikipedia.org/wiki/Tenori-on>
- [8] [Online]. Available: <http://uk.yamaha.com/en/products/musical-instruments/entertainment/tenori-on/tnr-o/?mode=model>
- [9] [Online]. Available: <http://www.c-thru-music.com/cgi/?page=spec-64>
- [10] H. C. Koch, *Musikalischs Lexicon*, 1865.
- [11] A. Swift, “An introduction to midi,” 1997. [Online]. Available: http://www.doc.ic.ac.uk/nd/surprise_97/journal/vol1/aps2/
- [12] S. Brewster, “Nonspeech auditory output. in: The human-computer interaction handbook: Fundamentals, evolving technologies, and emerging applications.” 2003.
- [13] D. M. Huber, “The midi manual,” 1991.
- [14] [Online]. Available: <http://blog.umito.nl/2010/05/07/midi-on-android.html>
- [15] [Online]. Available: http://en.wikipedia.org/wiki/Hash_Maps
- [16] C. Stein, “Introduction to algorithms,” 2001.
- [17] [Online]. Available: <http://www-cs-students.stanford.edu/~amitp/game-programming/grids/>
- [18] [Online]. Available: <http://www.gdreflections.com/2011/02/hexagonal-grid-math.html>
- [19] J. M. A. S. L. Pfleeger, “Software engineering 4th edition,” 2010.
- [20] [Online]. Available: <http://www.kanbanflow.com>

Part VI

Appendices

14 Markey Research Survey

To be insterted.

15 Software Instruction Manual

TO BE INSERTED SEPARATELY

16 Weekly Formal Meeting Minutes

TO BE INSERTED