
hex

G52GRP Interim Group Report
HEX - A Chain Reactive Music Generator

Group NHN2

22nd March 2013

Supervisor: Dr. Henrik Nilsson

S.Cooke	-	skc01u
R. Fulton	-	rxf01u
G. Hallam	-	goh01u
D. Huo	-	dxh02u
M. Tawafig	-	mxt41u
J. Sherry	-	jxs41u

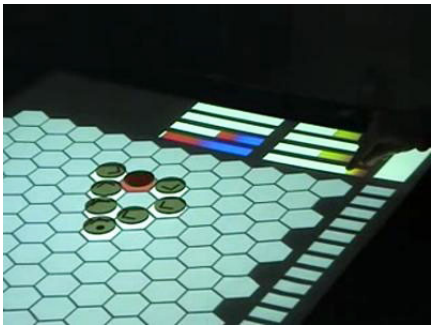
Contents

Part I
Preliminary Text

1 Concept

1.1 ReacTogon Concept

test 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 applet 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, Explode and Warp. The latter four have active and inactive states which 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 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 applet 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 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' 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. [4]

2. **TonePad / TonePad Pro**
   The TonePad is another application for the iOS platform, developed by LoftLab.[5]

   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 it's 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[6], 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 a 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 in performance. In this way, musical loops and motifs can be generated and played sequentially to create a complete piece of music[7].

2. **AXiS-64** The AXiS-64 is possibly the most prevalent MIDI controller and piece of hardware that utilises the harmonic table. The equipment itself consists of the keyboard itself, 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[8]. The four analogue interfaces 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.

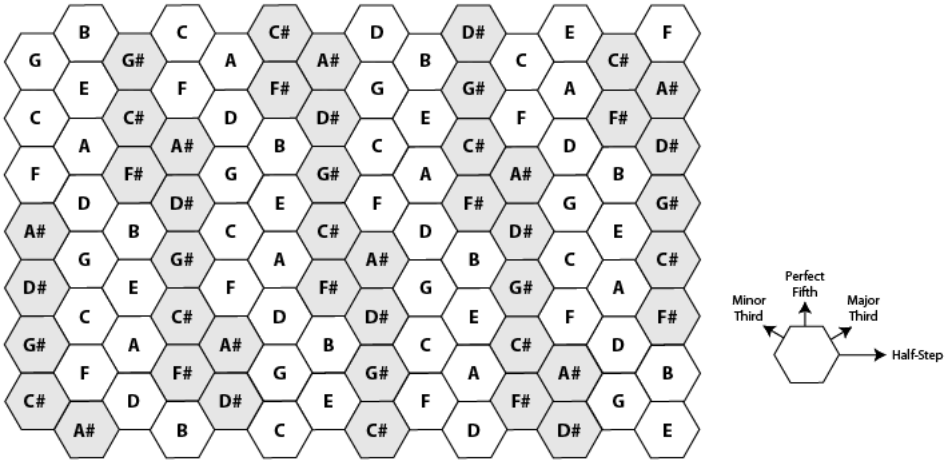
In addition to this, particularly interesting features of researched products that we may wish to take forward into our project include the multiple 'layers' featured in the Tenori-On and the TonePad Pro's ability to export and share music and created projects.

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 werent as advanced as the desktop applications. The desktop applications that were reviewed werent targeted at beginner users as some of the features were too complicated and functionality wasnt as easy as the touch screen apps. The suggestion of a desktop application with touch screen capabilities was received well and most said they would be interested in using a program which combined the desirable features of both a touch screen app and a desktop application.

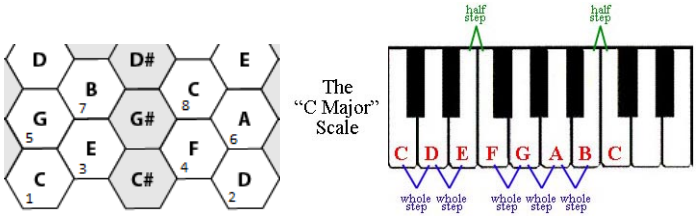
2.4 Music Research



In musical theory, an interval is the distance between two notes or pitches. Typically, these are represented with tones and semitones, where tones are the base unit of an interval. Certain intervals appear particularly often within music and 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:

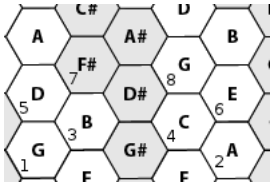
- Notes ascend by an interval of a fifth on the vertical axis.
- Notes ascend by an interval of a major third on one diagonal axis.
- Notes ascend by an interval of a minor third on the other diagonal axis.

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. In this way, music can be easily and quickly created.



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.

Scales and triads can be transposed far more easily on a harmonic keyboard than on a conventional keyboard as the basic pattern is always retained. This is shown in the G major scale below:



2.5 Technical Research

2.5.1 MIDI

TO BE COMPLETED

2.5.2 Working with Hexagons

TO BE COMPLETED

2.5.3 Technical Research Evaluation

TO BE COMPLETED

3 Requirements Specification

Our project has several important key features which need to be followed and implemented; they can be described in the functional, 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 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.
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.
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 must be able to rotate when placed in order to send the playhead in different directions:

   (a) *Play*

   (b) *Change*

10. Six grids should be implemented for use in harmonic, polyphonic projects.

    (a) Grids must be independent.

    (b) Different tiles must be able to be placed on different grids.

    (c) Each grid must be able to have it's own tempo.

    (d) Grids 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 would comprise of 6 layers; each layer being a grid. Each grid would have to be unique yet have different properties; this was the first clue to using OO design. Each grid would have a set of properties, these properties being:

- Each grid could be assigned a unique instrument, but needs to support changing it on the go during mid play. This suggested that a grid could be assigned its own channel in a MIDI sequencer. (Rees, 2001)
- Each grid could support different octaves, through the research we found that MIDI can play 127 different notes ranging from octaves 0 to 10, our grids therefore need to be able to be transposed mid session. This suggests that each grid have a starting note and some sort of algorithm then fills the grids with correct notes.

  The tempo of the grid should be changeable, but all grids needs to have some sort of relation to each other, so some sort of base tempo needs to be applied, then each grid can then play at $1\times, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}$ or $\frac{1}{16}$ times that 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 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.

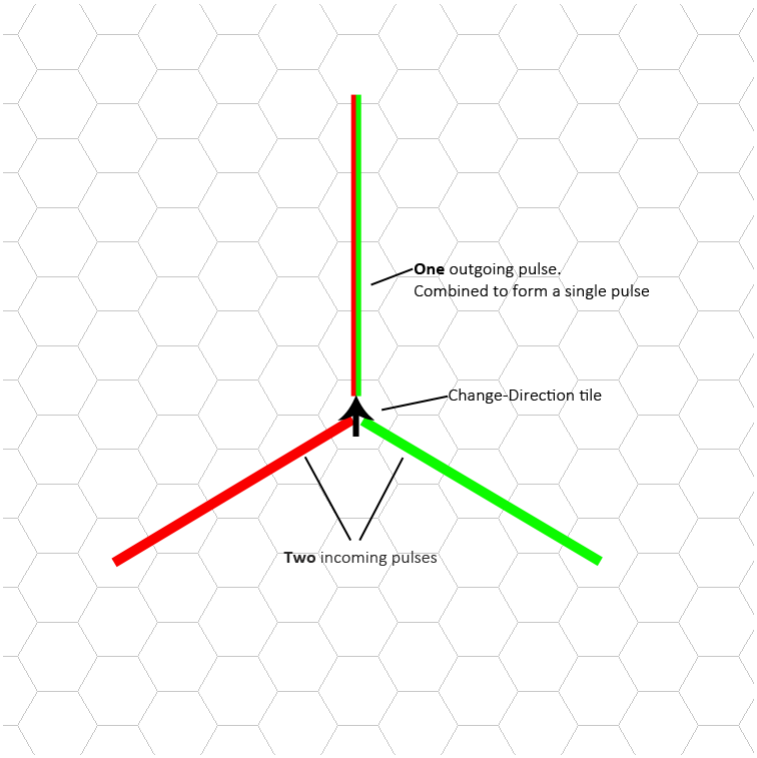


fig. 1

4.3 Tiles

The main idea of this software is the use of tiles to manipulate the music. A tile can be placed in a cell on the grid and used to manipulate an incoming pulse. Different tiles have different effect. The tiles are listed as follows:

- Stop: Should destroy the pulse
- Change direction: This implies that the tile should hold a direction and then be able to pass this information onto an incoming pulse. Because the tile will have different arrows for different locations will mean that the graphics will have to be redrawn with the corresponding arrow.
- 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, maybe creating it.
- Explode: Will destroy and incoming pulse in much the same way as stop does, but in turn create 5 new ones and send them in every direction apart from the input. From this knowledge 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. From this I will make a warp tile destroy a pulse, and create new ones 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 I 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 time if it should play a note or not when hit by a pulse.

4.4 Keeping Time

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

4.5 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.

5 User Interface Designs

TO BE COMPLETED

Part III
Implementation

6 Key Control Elements, Data Structures and Methods

6.1 Removal of Duplicates

Removal of duplicates, figure 1 in the design shows a problem of pulses merging together. Two or more pulses could occupy the same space and this would cause the amount of pulses to exponentially grow and in turn cause the program to slow down and eventually crash.

This was due to pulses being held in an array, to removing duplicates from this array would take $O(n^2)$ time, and even then it would still have problems detecting them. To solve this problem, a hash map was used to store the location of the pulse in the grid. The hashing algorithm took the pulses current location, and direction to place it in the hash map, the hashing algorithm is shown in below (equation 1). The hashing algorithm was relatively simple because 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 gird (going the same direction etc.) they would be placed in the same position in the hash table; this would cause a collision, but as a hash table 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.

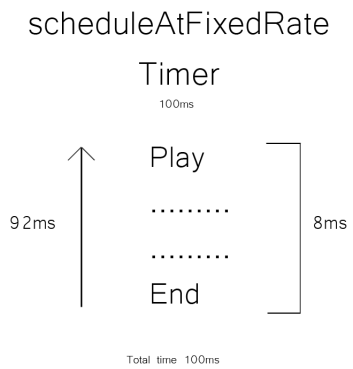
To keep track of the keys in the hash map there is an array list holding all the keys that are currently in use, to access the pulses it was necessary to traverse through this list and use the keys to get data from the hash map. Every loop, the array list was emptied and all keys where added again. By 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 I was not adding and removing lots of pulses from an array list all the time.

$$Key = ((Pulse.tLocation \times 36) + (Pulse.fLocation \times 6) + direction) \tag{1}$$

6.2 Timing and Tempo

Timing the whole system was very important. To do this I used Timer that was scheduled at a fixed rate. This meant that it would call the run method every set amount of time irrespectively if it had finished the methods or not. The timer controlled how often the pulses moved one space in the grid. Because each grid could have a different tempo I needed to keep all the grids synchronised but still have different tempos. To do this in grid a tempo is assigned with a value of 1, 2, 4, 8, or 16.

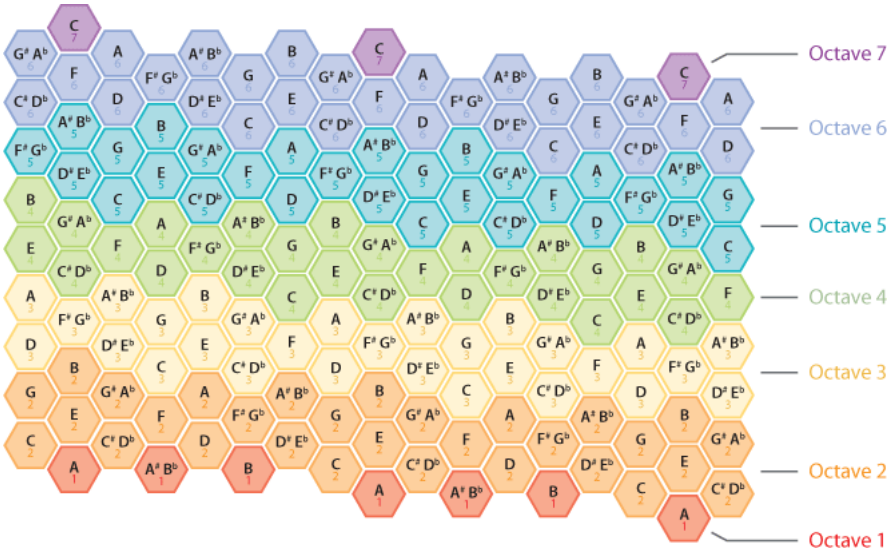
Then every loop of the base timer a value is incremented by 1 this value is then MODed against the tempo of each grid, if it is 0 then it will move the pulses inside this grid by one, it if it not it will skip this grid. This is explained clearly in the below code for the *play()* method. At the end of every loop queues of notes are played, this keeps everything in sync.



6.2.1 *play()* method Implementation

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

6.3 Transposition



One feature of the program that was much easier to implement than expected was being able to transpose the whole grid and changing the start note of each grid on the fly. You could use the fact hexagons where stored in a 2nd to your advantage and that MIDI notes could be stored as numbers. Studying how the harmonic table works (above), and finding patterns how the notes were placed we were able to write a simple formula (below), that selecting a start node of the grid, it could then place the correct notes in the correct place. Putting this formula then in a loop that goes through the array meant it could be called at any time even during play, as hexagon cells themselves held the note numbers. This formula would work for any size grid, so if we wanted to extend the table at some point that would be possible.

6.3.1 Transposition Formula

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

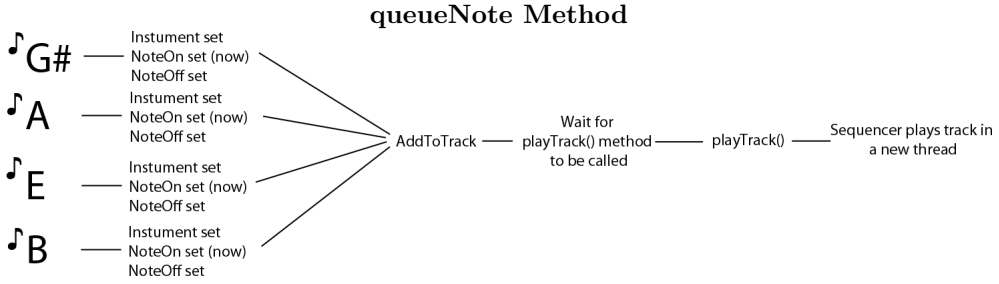
$$n = startNote - (7 \times j)$$
$$note = ((n + 4) - (((i\%2) \times 7)))$$

6.4 Sound Production

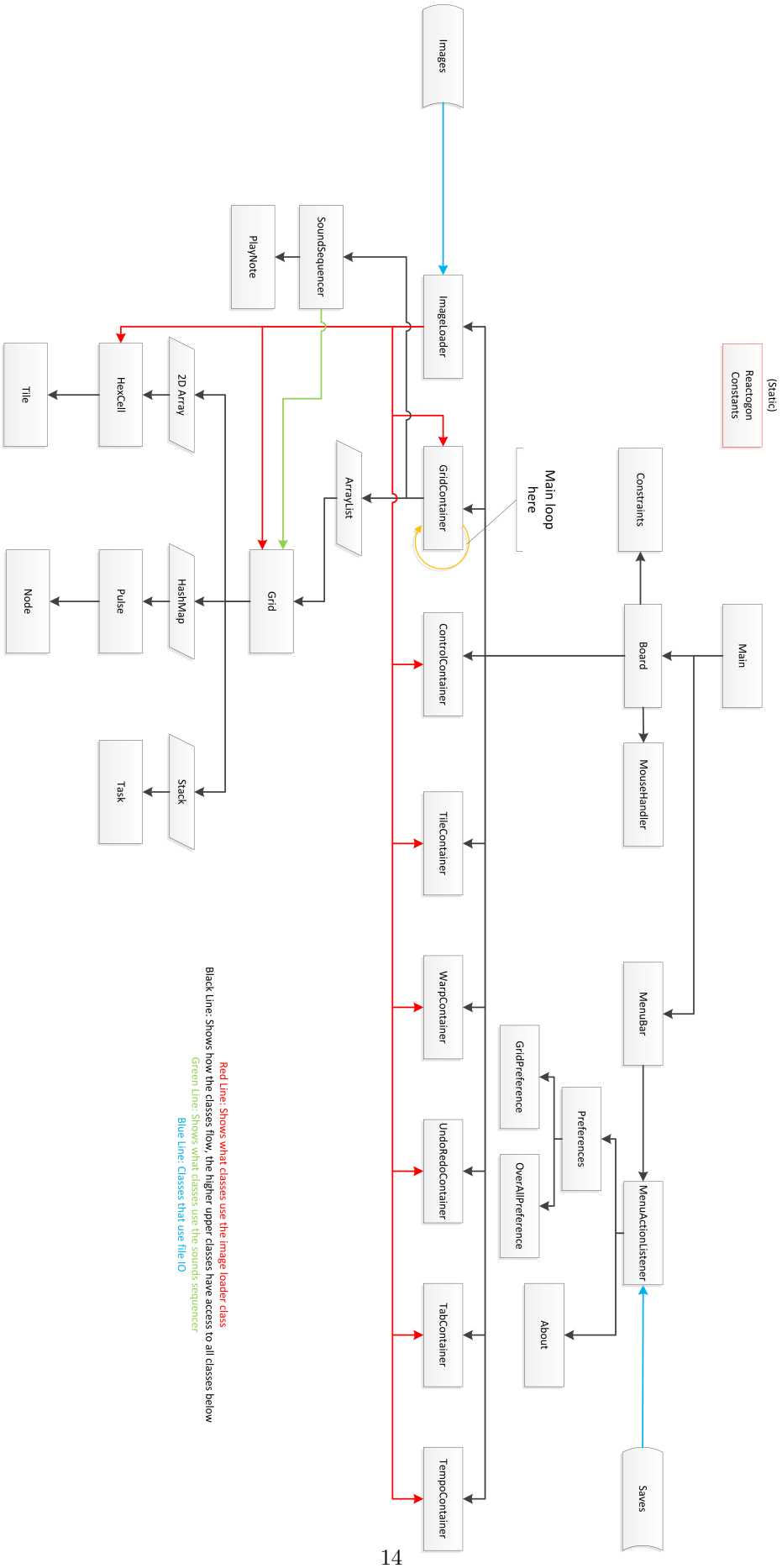
The sound creation went through many stages, and indeed, could still be seen as a work in progress. The most important of the system was keeping everything synchronised. If 20 notes were scheduled to play at the same time, then they all should be played at exactly the same time, not milliseconds apart. To do this you can see in figure 3 there is a method call playNotes() this would play a track containing all that notes that had to be played at that moment in time. The way the notes were added to this track was; As you now know, the pulse move through the grid, step by step, if it came into contact with a tile it would have an effect. If that tiles properties were set for it to play a note, then a method was called (figure 8), and the note for that hexagon was passed into the sound sequencer, each note had an on, off and instrument set to it. Once one whole loop had gone through, the playNotes() method would send this track off to a sequencer in a new thread and play them all at once, this would happen repeatedly every loop. Figure 9 explains this in picture form.

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

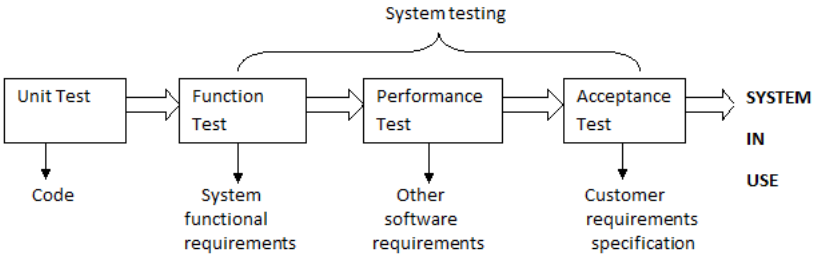
queueNote Method



7 Code Plan


Images
SoundSequencer
PlayNote
ImageLoader
(Static)
Reactogon Constants
Tile
HexCell
2D Array
ArrayList
GridContainer
Main loop here
Constraints
Node
Pulse
HashMap
Grid
ControlContainer
Board
Main
MouseHandler
Task
Stack
TileContainer
WarpContainer
MenuBar
GridPreference
Preferences
OverAllPreference
UndoRedoContainer
MenuActionListener
About
TabContainer
Saves
TempoContainer
Red Line: Shows what classes use the image loader class
Black Line: Shows how the classes flow, the higher upper classes have access to all classes below
Green Line: Shows what classes use the sounds sequencer
Blue Line: Classes that use file IO

8 Testing



Planned Testing Phases

The objective of testing is to find faults, defects in design, requirements and in code[**?**]. With the help of testing, the defects would be fixed and the program can be improved to better meet customers needs. In addition, if tests find few faults, it can give confidence in the quality of our software. Our tests are planned to be completed in 4 phases which is unit test, function test, performance test and acceptance test. Once some components of our program are coded, we begin to test. This part only provides a discussion on the testing of the system, the detail test cases are given in Appendix 3.

8.1 Unit Testing

Unit tests are written with JUnit. Some codes are written to provide a predetermined set of data to the components which are being tested and observe the results and check the internal data, logic, boundary conditions for input and output data [**?**]. To write unit test, the first step is to read through the code and figure out what exactly the functions, the classes do. Make sure the code can be compiled and use some example data as input and compare the output with the expectation result. In JUnit, the function assertEquals() is usually used.

For example, the class HexCell create a hexagon, calculates dimension, note and image of it. There are some functions like getImage(),flipState(),setNote() etc in it.

A 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 to test method getImage(). The first parameter in assertEquals() is the expected output and the second one is the actual output, if they are the same, when running the JUnit, the test of this method can pass which means the components which are being tested meet the requirement.

There are other unit tests like PulseTest, TileTest. Some example data is provided to test the methods. Other test codes are provided in the project file.

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

8.2 Functional Testing

Function tests are run to check whether the program is satisfying the customers' 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 light and the sound which the application actually presents. 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 below:

8.2.1 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 button on the first hexagon tile.
2. Click and place the blue explode button on the second hexagon tile which is in the direction of the first one.
3. Click start button.

Expected result in steps:

1. Should display a grey 'play' tile on the first hexagonal tile
2. Should display a grey 'explode' button 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 should play a different note at the second tile and display blue lights from the second tile to all directions except the opposite direction of the first tiles.

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 display blue lights from the second tile to all directions except the opposite direction of the first tiles.

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 be a quite a lot of functional testing to make sure that the system operates as required and the detailed test cases are provided in appendix 3.

8.3 Performance Testing

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

8.3.1 Stress Testing

This test is to ensure the software does not crash in conditions of insufficient computational resources such as low memory or running out of disk space.

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

Test: Before run the application, the CPU usage rate is 0 1. When open the application it rise 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.

Part IV
Project Meta-Comment

9 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.

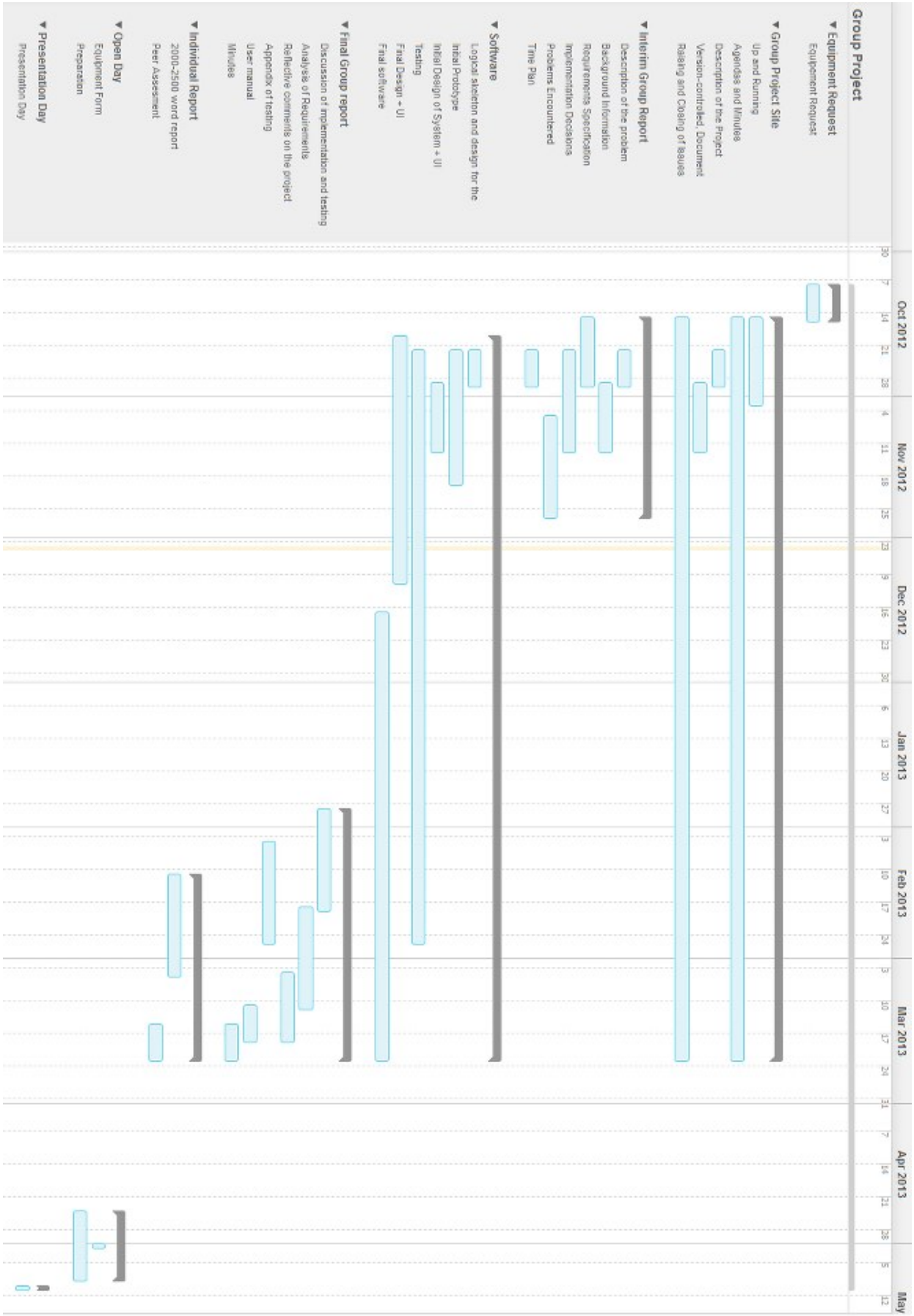
In terms of changes to functionality of the software, since the interim report we have implemented multiple grids; the *Warp* tile; the ability to change grid tempo and instrument on the fly and saving and loading have been finished and added.

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.

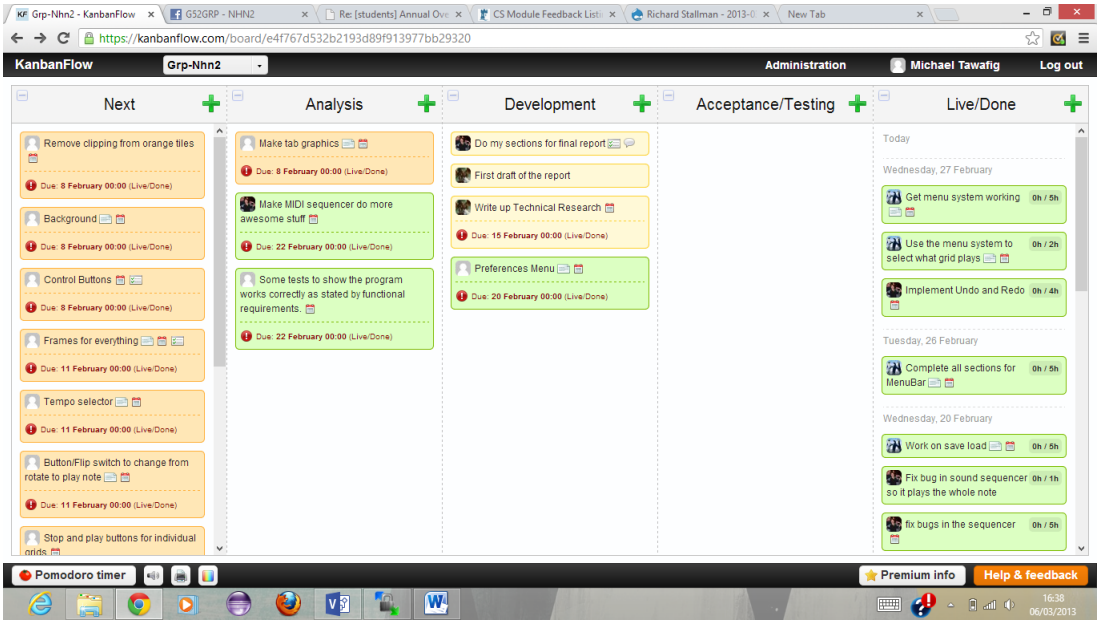
10 Time and Planning

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

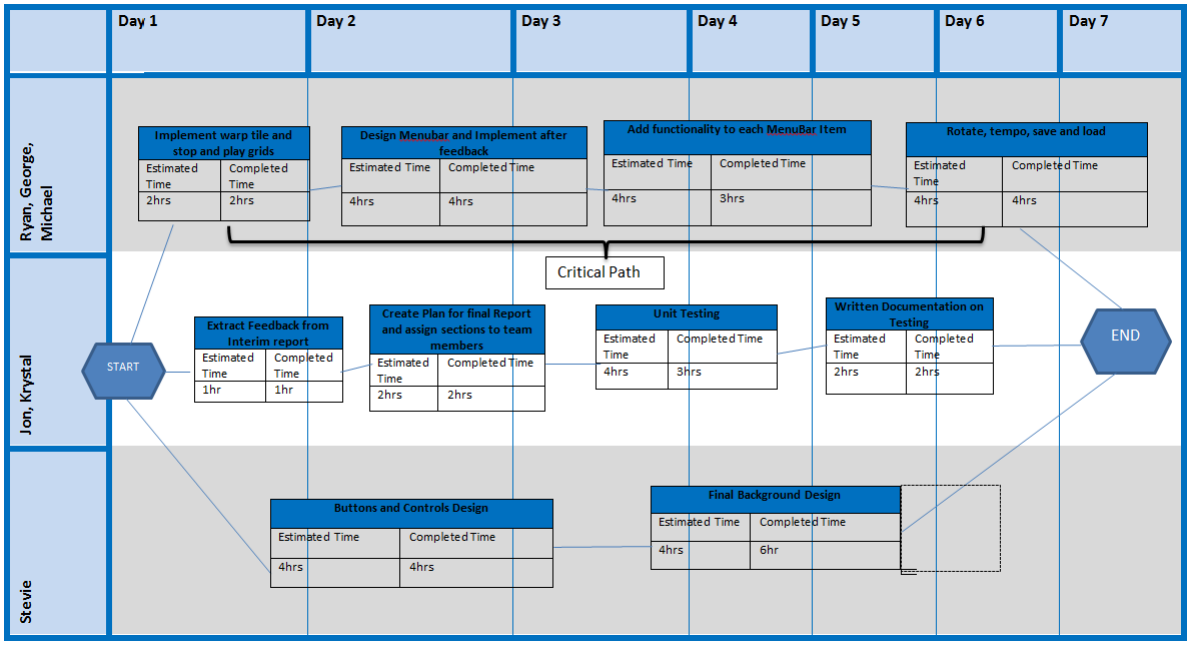
topics and then created sub-topics for each. Using deadline dates as guidance we made a rough estimate of how long each task will take then broke that down further for each sub topic and finally determined the order of the tasks. There is also a percentage complete column inserted, which was updated 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 [**?**]. 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 werent completed sequentially but in parallel we could allow for slack time as well. Below is a 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.

11 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 werent 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 werent sure which parts of the program needed testing, therefore we didnt 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 succesfully emulated the reacTogon 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.

Part VI
Appendices

12 References

References

[1] M. Kaltenbrunner, "Tangible musical interfaces," 2013. [Online]. Available: http://blog.makezine.com/2008/04/17/reactogon-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://amidio.com/manuals/JR-Hexatone-Pro-1.0-User-Manual.pdf

[5] [Online]. Available: https://itunes.apple.com/gb/app/tonepad/id315980301?mt=8

[6] [Online]. Available: http://en.wikipedia.org/wiki/Tenori-on

[7] [Online]. Available: http://uk.yamaha.com/en/products/musical-instruments/entertainment/tenori-on/tnr-o/?mode=model

[8] [Online]. Available: http://www.c-thru-music.com/cgi/?page=spec-64

13 Software Instruction Manual

TO BE COMPLETED

14 Weekly Formal Meeting Minutes

TO BE COMPLETED