

G52GRP Interim Group Report

HEX - A Chain Reactive Music Generator

Group NHN2

7th December 2012

Supervisor: Dr. Henrik Nilsson

S.Cooke	-	skc01u
R. Fulton	-	rx01u
G. Hallam	-	goh01u
D. Huo	-	dxh02u
M. Tawafig	-	mxt41u
J. Sherry	-	jxs41u

Contents

I	Preliminary Text	3
1	Concept	3
1.1	ReacTogon Concept	3
1.2	Project Description	3
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	4
2.2	Systems Research Evaluation	5
2.3	Market Research	5
2.4	Music Research	6
2.5	Technical Research	6
2.5.1	MIDI	6
2.5.2	Working with Hexagons	7
2.5.3	Technical Research Evaluation	7
3	Requirements Specification	7
3.1	Functional Requirements	7
3.2	Non-Functional Requirements	8
II	Design	8
4	Software Design	8
4.1	The Grids	9
4.2	Pulse	9
4.3	Tiles	10
4.4	Keeping Time	10
4.5	Creation of Sound	11
5	User Interface Designs	11
III	Implementation	11
6	Key Control Elements	11
6.1	Removal of Duplicates	11
6.2	Timing and Tempo	11
6.2.1	<i>play()</i> method Implementation	12
6.3	Transposition	12
6.3.1	Transposition Formula	13
6.4	Sound Production	13
7	Implementation of Key Data Structures and Methods	13
8	Code Plan	14
9	Testing	15
IV	Project Meta-Comment	15
10	Updates since Interim Report	15
11	Time and Planning	15

12 Problems Encountered	15
V Conclusion	15
VI Appendices	16
13 References	16
14 Software Instruction Manual	16
15 Weekly Formal Meeting Minutes	16

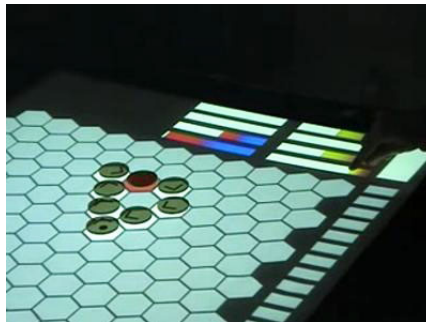
Part I

Preliminary Text

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 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.

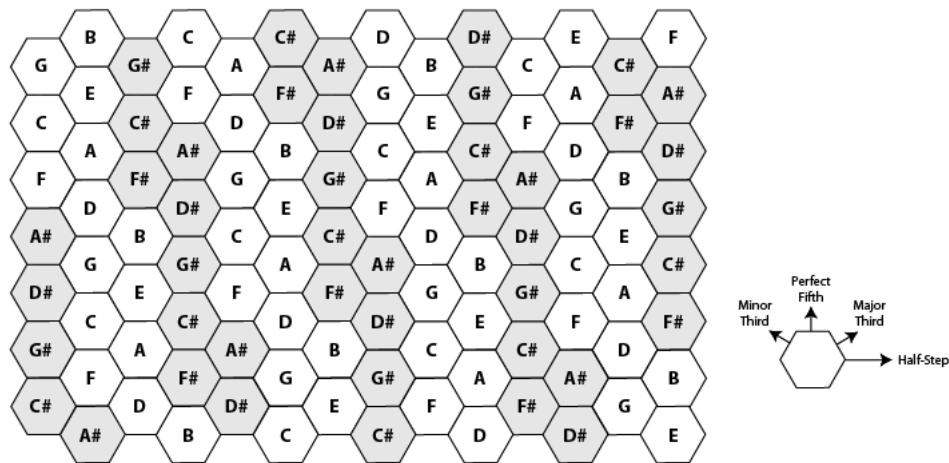
Among these common themes is the use of physical interaction as an interface between the system and the user. A considerable majority of the software systems available are for tablets or mobile devices using touch screen interfaces, with almost no standard desktop applications and only a few online hardware 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

TO BE COMPLETED

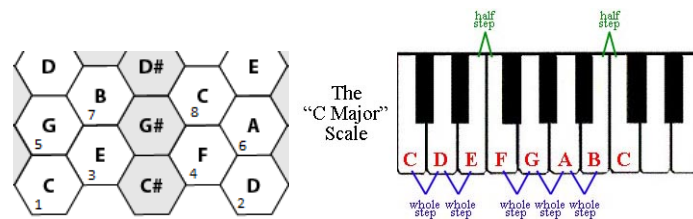
2.4 Music Research



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:

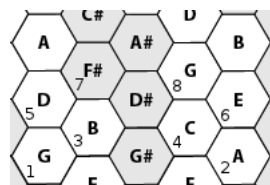
- Notes ascend by an interval of a fifth on the vertical axis.
- Notes ascend by an interval of a major third on one diagonal axis.
- Notes ascend by an interval of a minor third on the other diagonal axis.

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.

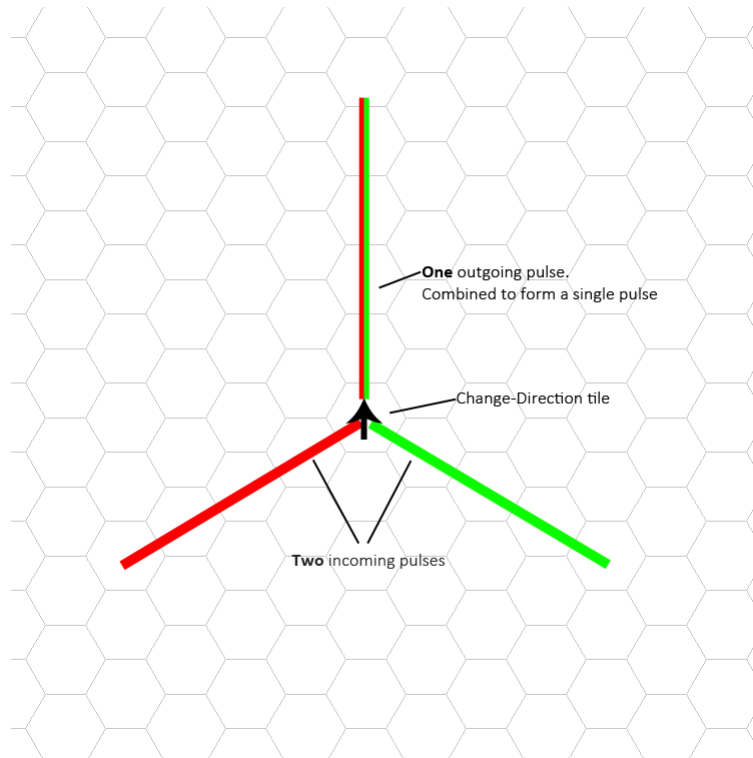


fig. 1

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 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.

4.4 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

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 grid (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.

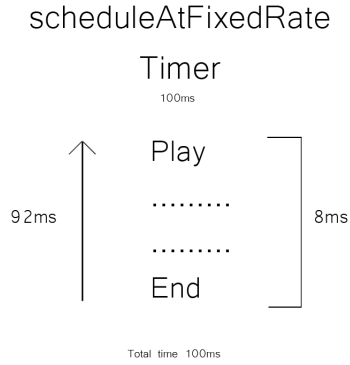
To keep track of the keys in the hash map there is an array list holding all the keys that are currently in use, to access the pulses it was necessary to traverse through this list and use the keys to get data from the hash map. Every loop, the array list was emptied and all keys were added again. By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

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

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.

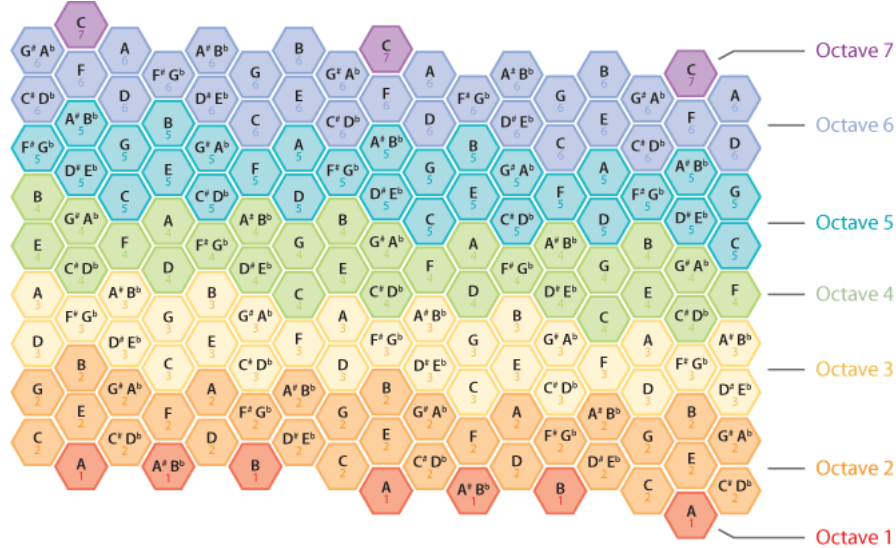
Then every loop of the base timer a value is incremented by 1 this value is then MODed against the tempo of each grid, if it is 0 then it will move the pulses inside this grid by one, if it is not it will skip this grid. This is explained clearly in the below code for the *play()* method. At the end 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 were stored in a 2D array 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

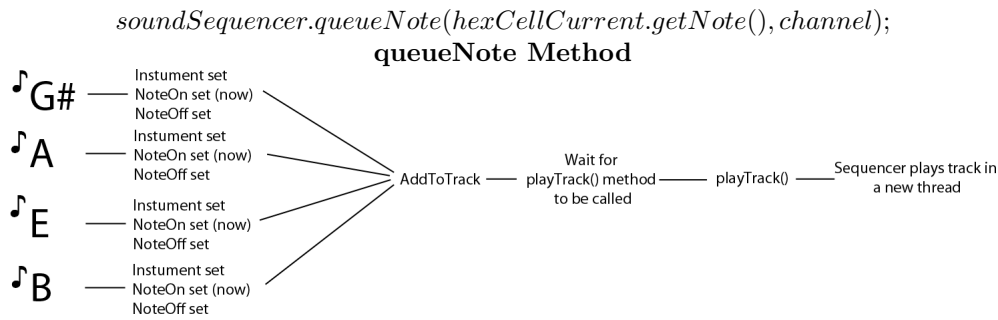
The note for any given i , and j cell on the grid can be given by:

$$n = \text{startNote} - (7 \times j)$$

$$\text{note} = ((n + 4) - (((i \% 2) \times 7)))$$

6.4 Sound Production

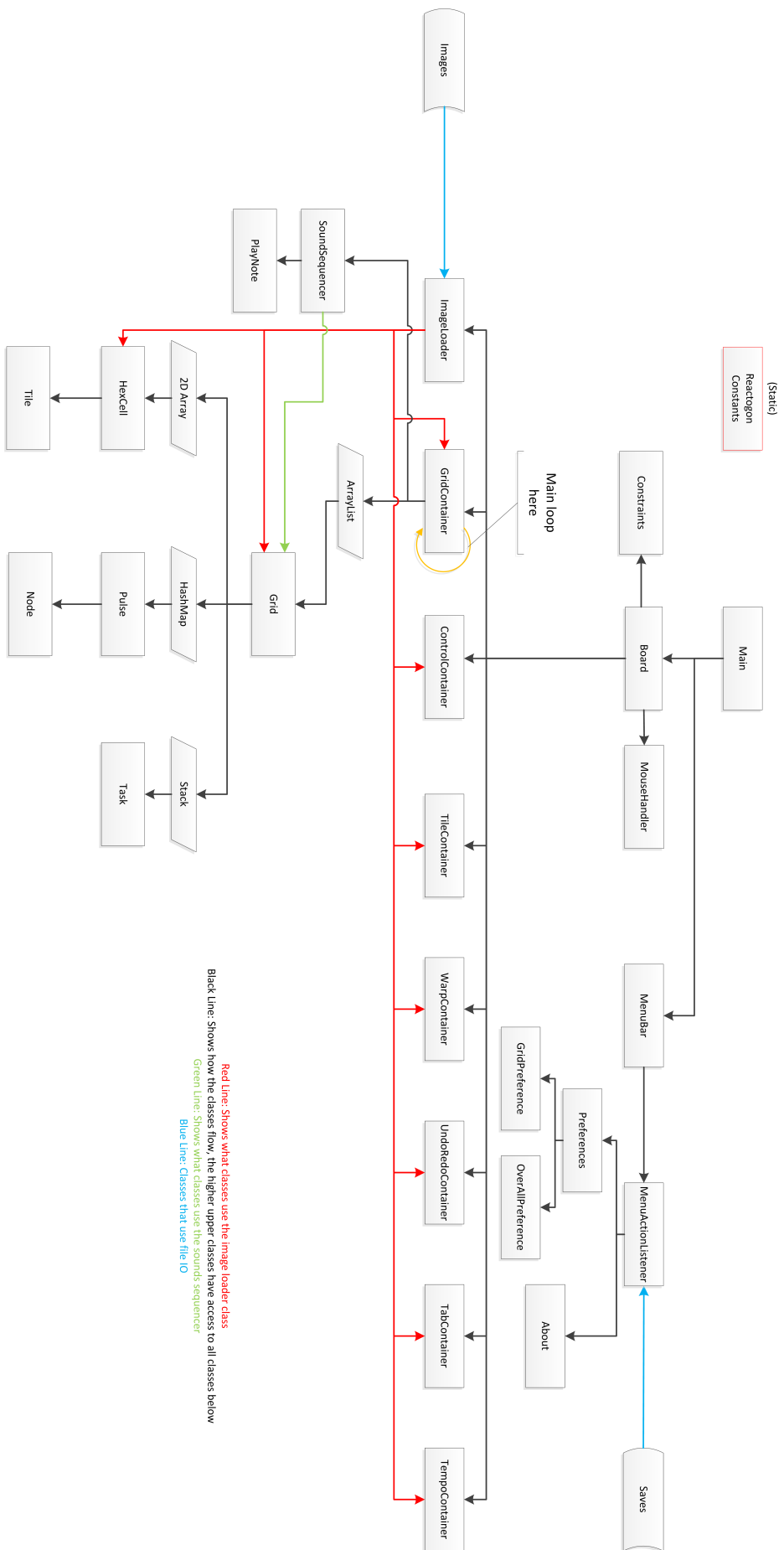
The sound creation went through many stages, and indeed, could still be seen as a work in progress. The most important of the system was keeping everything synchronised. If 20 notes were scheduled to play at the same time, then they all should be played at exactly the same time, not milliseconds apart. 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.



7 Implementation of Key Data Structures and Methods

????????????????

8 Code Plan



9 Testing

TO BE COMPLETED

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. TO BE COMPLETED

11 Time and Planning

TO BE COMPLETED

12 Problems Encountered

TO BE COMPLETED

Part V

Conclusion

TO BE COMPLETED

Part VI

Appendices

klujghf;kjghglkjghslkjh

13 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>

14 Software Instruction Manual

TO BE COMPLETED

15 Weekly Formal Meeting Minutes

TO BE COMPLETED