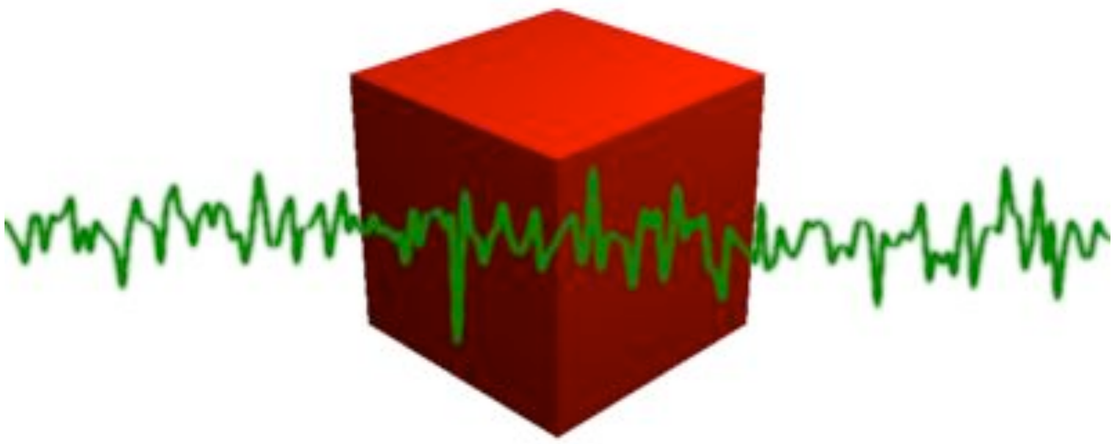
Music Visualisation Program

Project Report



Author	Nicholas Martin
Candidate Number	XXXX
Submission Year	2004
Degree Programme	BSc Multimedia & Digital Systems
Department	Informatics
Project Supervisor	Dr P. Newbury
Word Count	7565

Summary

This project shows the development of an application which produces a graphical representation of sound. Implemented using Java and Java 3D, an object-orientated approach using detailed design diagrams. The application works using a live audio input. The frequency spectrum of the audio data is calculated using a Fast Fourier Transform, forming the source of control for each of the four visualisation themes. Graphical content is 3D and utilises mechanical equations for motion characteristics by implementing custom interpolator classes.

The graphical content and audio input correlate successfully, outputting between 90 and 210 frames per second with a latency figure of approximately 80ms. Issues were found within the Java 3D API when implementing advanced transparency.

The application confirms that Java is capable of carrying out an FFT in real time from a live audio input with minimal latency. Transparency with Java 3D is possible but its usage must be limited. Resource leaks may occur within the scenegraph if its elements are not manually removed.

Contents

1 Introduction

In a live music event the presence of a graphical accompaniment can add great impact. The idea of combining live visuals and music is not new; even before the times of computers, bands such as Pink Floyd have had live visuals at their concerts. Currently there are many software packages which can create real time graphics in time with music on a standard desktop PC. This includes both mp3 players and software dedicated to VJing, the term used for creating dynamic live visuals. All perform a frequency analysis of sound and manipulate graphical content accordingly.

The accommodation of a live audio input is restricted to select packages which often require a high level of expertise to use. The range of "plug and play" software available is narrow and follows standard themes. Using objects to represent elements of musical structure has been attempted but based on beats per minute calculations. However, the implementation of a live form of frequency abstraction using 3D objects has been limited.

This report investigates the development of such a visualisation program. It is known that an audio signal's frequency structure can act as the stimulus for graphical content, so this will form the primary line of investigation. Multi-track audio is possible but requires hardware beyond the resources of this project. Visual content will be confined to 3D graphics only.

The report is Structured as follows. Initial research of existing applications, implementation technologies and target user requirements are discussed. A set of specifications are laid out from this research and the development framework realised. The high-level system design is documented, with more detailed work in the appendix. The underlying principals within the application are explained, ending in example screenshots of the final release. Results of the final testing are shown and their significance discussed so that an evaluation may be made. The appendix contains documentation of every iteration, including methodology and iteration testing.

2 Current Visualisation Implementations

The following section introduces some current applications which express sound in graphical form.

2.1 iTunes by Apple®

Apple's *iTunes* [1] is an mp3 player with a visualisation feature (see Figure 2.1). The graphics are implemented using OpenGL, although the visualisation does not have a 3D style. The strongest feature of its visualisations is the fact that the structure and content of the graphics are constantly changing so that the same thing is never seen twice; themes and devices are used more than once, but its creation is spontaneous and the result is always unique. The sound's spectral waveform is the basis for most of the graphical content, with everything else reacting to it.

This is a very advanced visualisation program and consequently requires a high level of computational power to run in full screen mode. The ability to implement visuals which develop so freely and independently are beyond the scope of this project, but inspiration can be gained from some of the techniques used.

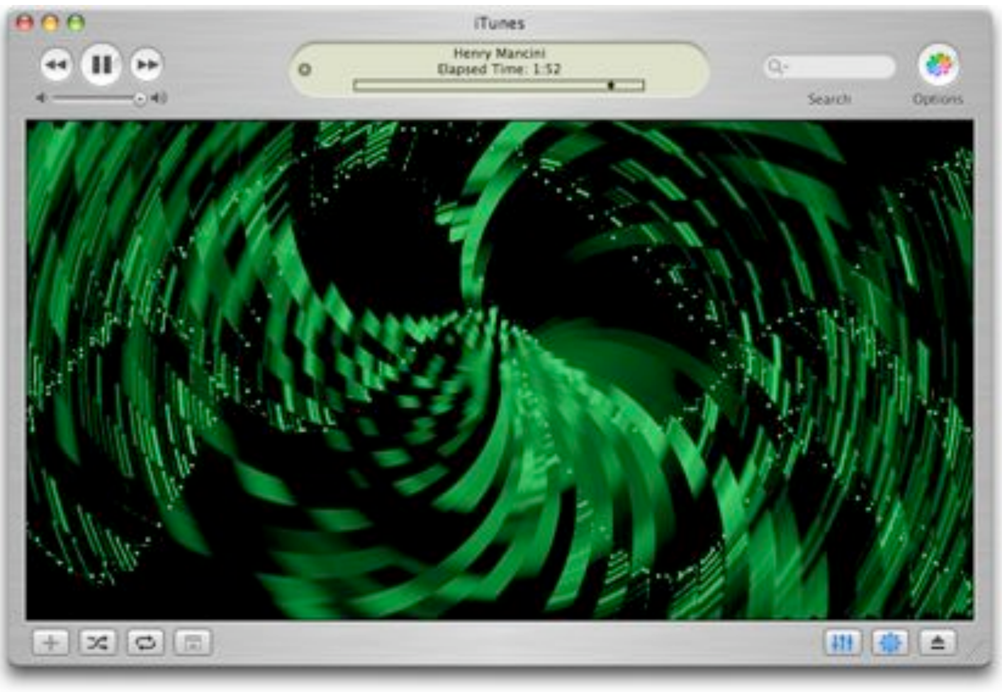


Figure 2.1 a sample of iTunes in action

2.2 Windows Media Player by Microsoft®

Another media player [2] which features visualisation capabilities, here different styles may be chosen, each with a different technique for reacting to music. However the styles are very prescriptive, preventing the visualisation from developing. Again much of the visualisation is based on the spectral waveform with swirls of colour being effected by it. As with *iTunes*, the visualisations may only be based on music played from file.

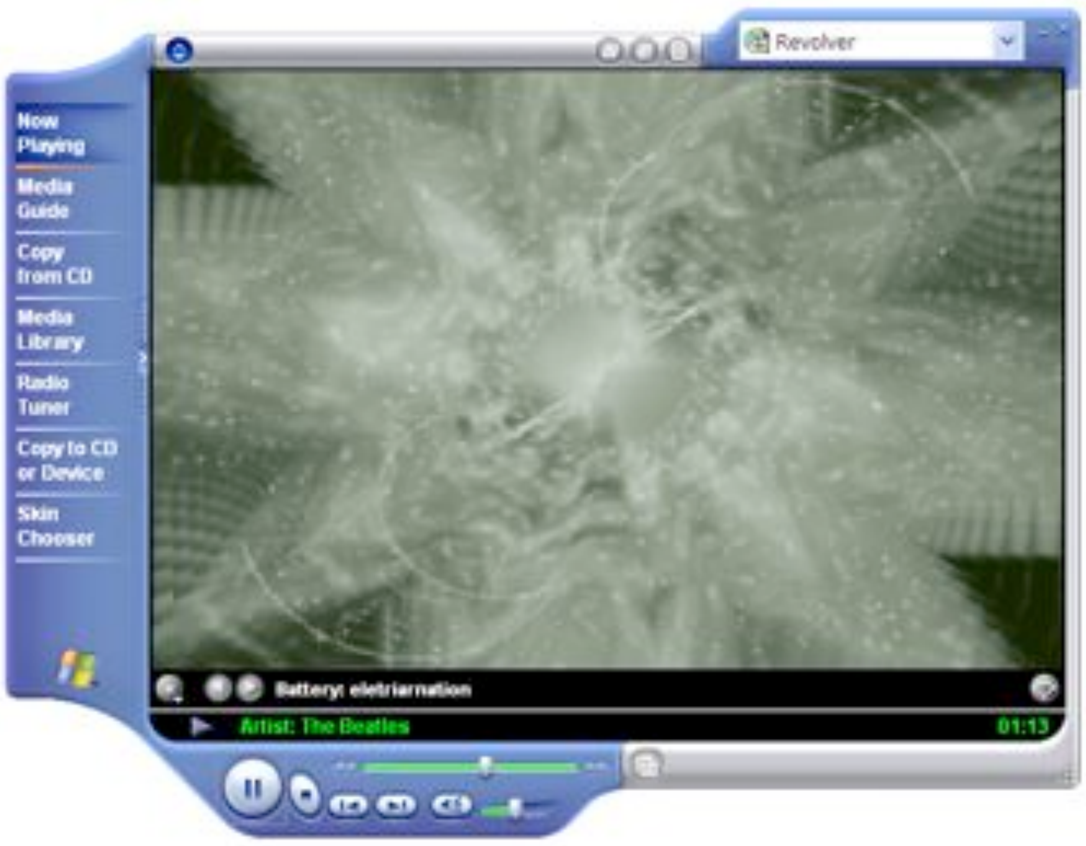


Figure 2.2 Windows Media Player

2.3 Audition by Adobe®

Formerly known as *Cooledit* this is a digital audio editing environment which allows for audio recording, mixing, editing and effects processing [3]. Although not directly linked to music visualisation it does in fact utilise the most fundamental graphical representation of sound - the audio waveform (Figure 2.3). Not only does it create a visual representation of sound in the time domain, but it can also perform a frequency analysis, spectral analysis or spectral view of a wave in real time; a technique used by both *iTunes* and *Windows Media Player*.

Aside from its visual techniques, *Audition* can also find and mark musical beats automatically; something which could prove powerful in trying to create synchronised graphics.

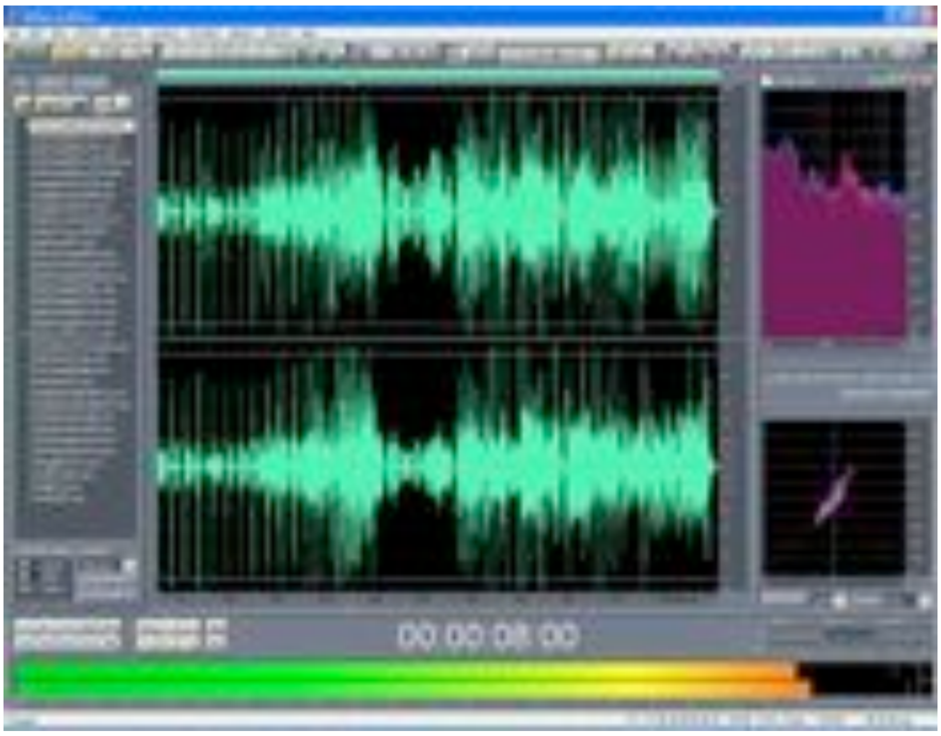


Figure 2.3 Audition's main editing window

3 Requirements Analysis

3.1 Target Users and Deployment

The program is intended to be used for real time visuals at a concert either by a live band or DJ. The master out from a mixer which the audience hears is the source for the visuals and the resulting graphics would be projected onto a screen. As the venue will have its own PA system there is no need for any sound to be output from the program.

Most visualisation implementations currently used by performers are projections of pre-produced video and graphical content; there is no direct correlation between what is seen and heard, other than possibly style or mood. A more elaborate and effective technique used is Vjing; the art of mixing together visual content in real time, much like a DJ mixes together records. Here the VJ may react to the tempo and mood of the music in a more complementary fashion. Although this does create a more interesting visual experience to a live event, the performer is faced with the problem of finding someone who possesses what is an eclectic skill.

3.2 Functional Requirements

Real time execution of the software is essential to the visualisations being effective, but whether this may be achieved in the timescale and using the available technologies is not certain. With testing the degree of acceptable latency can be assessed.

The software will need to run “straight out of the box” without the need for any composition beforehand. It is certainly possible to create a program which is executed and left to produce a graphical output. However, the issue of independent development is more difficult to address, and any audience is going to get tired of seeing the same thing for even 10 minutes. The level of independence seen in *iTunes* is beyond the scope of this project, but the merging of different themes into each other is something which may be achievable.

A target system for the software is a normal desktop PC without any specialist hardware other than a reasonable soundcard which allows for analogue input and a graphics card with 3D capabilities. It must be seen to what degree full screen graphics can be executed in real time.

4 Specifications

A set of specifications acts as both a blueprint and reference point for the project. Derived from the requirements analysis, they set out the goals of the development like a set of guidelines. By sticking to them the progress of the project may be kept in check. However, this is not a rigid set of rules which indicate the success or failure of the application.

4.1 Execution Speed

- Graphical content must correlate with the audio input in real time
- A frame rate of 30 fps should be aimed for as *iTunes* (see Section 2.1) allows for its visualisations to be capped at this value without any detriment to quality

4.2 Graphical Content

- The nature of the graphics should allow an obvious reaction to the music
- There must be some variation in the visual content in the form of different visualisation techniques
- The user should not be required to compose any graphical content

4.3 Deployment

- Maximum hardware requirements
    - Intel® Pentium® II or higher, AMD Athlon™ processor, Macintosh® G4 or higher
    - Hardware-Accelerated OpenGL® graphics card
    - 16-bit soundcard capable of analogue input
- Maximum software requirements
    - Microsoft® Windows XP
    - Apple® Mac®OS X (10.3.*)
- Additional software requirements may only include freely available packages
- Once set up the software should require minimal operation to change themes
- No technical knowledge of the software should be expected from the user

5 Implementation Technologies

The technology chosen to implement the project is extremely important; it has a bearing on the quality, complexity and viability of the final result. Here the main available choices are investigated; the two languages C++ and Java and their respective 3D graphics APIs, OpenGL and Java 3D.

5.1 C++

Speed of execution is the major factor in this development as the graphics must react so closely to the music. This makes C++ the ideal language of choice due to its speed relative to other languages. C++ also has the most extensive amount of third-party libraries of extension code for routines such as the FFT. To its disadvantage C++ does lack the portability of Java, so compatibility with the different operating systems in the specification (see Section 4.3) would require separate implementations. Also, the extra load required to learn C++ would impact on the possibility to extend the functionality of the application.

5.2 OpenGL

For the graphics side of the application OpenGL would offer the highest level of quality and content. Being one of the most advanced real time graphics libraries available, it offers the programmer a large amount of freedom and power. OpenGL is not a language but a set of graphics libraries which may be used in order to produce graphical content. It is possible to implement OpenGL in C, C++ and Java using the appropriate wrapping APIs. However, regardless of the language used, it is a complex language to implement with many low level elements to accommodate.

5.3 Java

There are many speed issues associated with Java which make it a less popular choice for high performance programs. However, recent releases of the API have shown great improvements, so much so that some newsgroups claim it is as fast as, if not faster than C++ for many applications [4]. Regardless of the accuracy of this claim, Java is a well established programming language which I have tested using Sun's own Java Sound Demo [5] and found to perform without any problems. Java has a dedicated sound API which provides methods for the input of audio data through a computer's ports.

5.4 Java 3D

Although implemented entirely in Java, Java 3D is not part of the Core API but rather a supplementary package which may be used to implement interactive 3D graphics. Java 3D uses a scenegraph, which is a higher level screen descriptor than that used for OpenGL, allowing for easier manipulation of a scene's elements [6]. It should be noted that Java 3D still uses OpenGL for the rendering of a scene but the programmer need never know as this is handled by the API; instead a more object-orientated approach to programming may be taken, rather than dealing with the low-level parameters of directly accessing OpenGL itself. Being 100% Java allows it to be seamlessly integrated with core Java classes within an application.

5.4 Choice of Technology

The primary factor in making a choice of implementation technology is the amount of functionality that may be achieved. C++ and OpenGL offer the most in terms of power, but their low level complexities would hinder the progress of the application. Although Java and Java 3D are possibly lesser programming languages in terms of power, the conceptual approach to programming they offer would allow the functionality of the application to be pushed further. Therefore, the application will be implemented in Java and Java 3D

6 Application Development

6.1 The Development Framework

A development framework will define the tasks and objectives for the life of a project [7]. Spending time on planning the project lifecycle saves time in the long run through properly directed development.

6.1.1 The Overall Plan

The project was undergone using an iterative, incremental lifecycle. The development process is carried out in stages instead of a single large release; evidence of functionality can be achieved quicker, allowing for baseline code (see Section 6.1.2) and the fixing of bugs as soon as they happen [8].

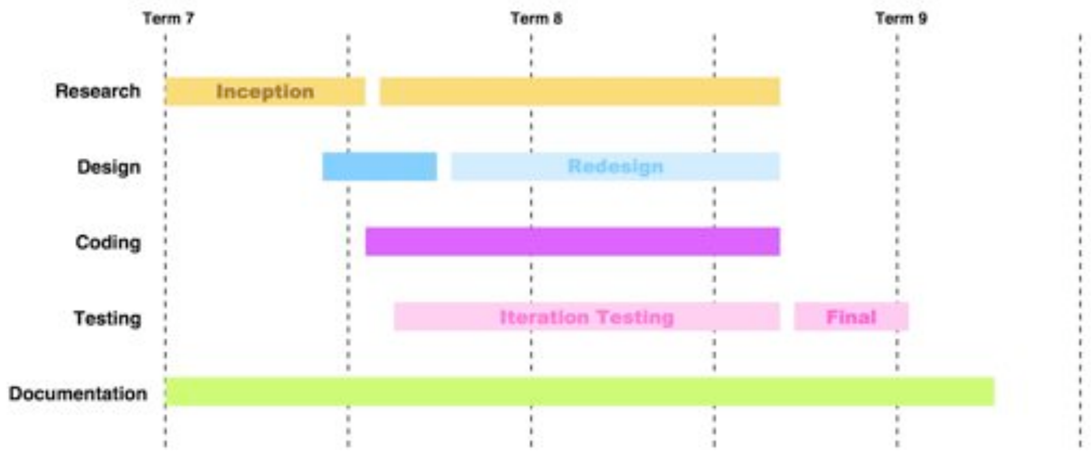


Figure 6.1 The project time plan

The main constraint upon the project is time, as the completion date is unalterable. Figure 6.1 shows the duration of each stage in the project. Although from the time plan it appears that research, design, coding and testing are all being carried out at the same time, Figure 6.2 illustrates how they are broken down into separate iterative stages.

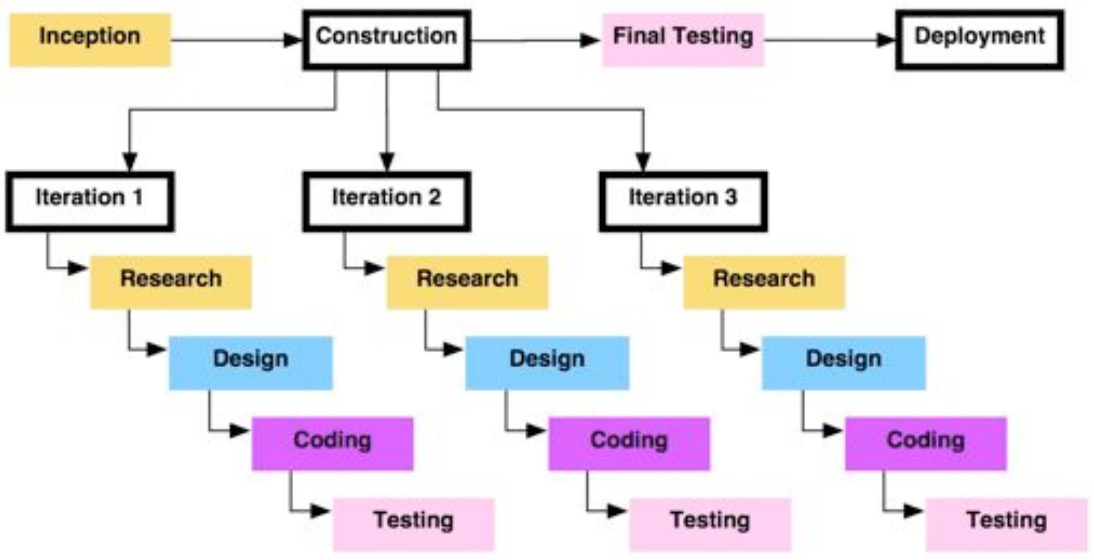


Figure 6.2 The project's iterative, incremental lifecycle

6.1.2 Baseline Points

The risk of serious errors late in the development of the application could jeopardise the success of the project. Having a fully functioning and tested section of code that represents a conceptual milestone ensures that this risk of errors is reduced [9]. The baseline points are ordered such that the highest risks are tackled first. The application may be divided into two main risk areas – audio processing and graphic generation. The point of the graphical content is to react to the audio input signal, making the audio processing the higher risk area. The ordered baseline points are based on the success of,

1. Audio input
2. Frequency spectrum calculation
3. Generation of the "pulsing boxes" 3D scene
4. Coefficient calculation (this comes later as it is a low risk audio process)
5. Updating the visualisation using coefficients derived from the audio data
6. Implementation of a GUI
7. Export of application into an executable file

Baseline point 5 is repeated for each of the visualisation themes developed. The iterations where the points were signed off may be seen in Appendix C.

6.2 Development Techniques

In order to develop the application's functionality various techniques were used; mainly based on visualising the execution and dataflow of the system, their aim is to achieve more effective development, rather than simply relying on trial and error coding.

6.2.1 Design Diagrams

By coding from designs time is saved by not encountering deep running errors; the combination of class and interaction diagrams enables the architecture of the system to be laid out before development becomes too advanced for any changes to be made.

The goal of creating design diagrams is to maintain good software engineering principals; an advanced level of abstraction, encapsulation and information hiding will lead to a system which executes to its full potential. Just as the application's aim is to create an abstraction of sound, so an abstraction must be made of the application's architecture.

6.2.2 Graphing Mathematical Expressions

A graph gives a clear view of the nature of a mathematical function; when dealing with complex expressions it is not possible to know its detailed shape without first plotting its values. This is especially true if a defined set of values must be interpreted – simply using trial and error at the coding stage is not sufficient.

An example of this is the creation of the coefficient scaling function (details in Appendix C.12.2). The aim was to find an expression which would generate a coefficient output between 0 and 1 from a set of decibel values between $-\infty$ and 0. Through the use of *Curvus Pro X*, parameters in the expression could quickly be changed and the effects seen until the desired characteristics were achieved. At this point the function may be tested in the code.

6.2.3 Analysis of Music

To be able to visualise music effectively its properties must be known. There are two main attributes which are of interest – frequency content and amplitude. The way in which an audio signal is handled must be based what is required from it; the purpose of the application is not to faithfully represent every nuance of sound in visual form, but for this visual form to be influenced by the sound's overall makeup.

The frequency range which a musical instrument functions at corresponds to the fundamental tones it can generate [10]. Figure 6.3 shows the ranges for common acoustic instruments, along with the musical notes it represents. However, the fundamental frequencies an instrument produces span further into harmonics. These overtones give an instrument its own unique timbre – an identity that distinguishes it from similar sounding instruments. This identity is of no interest to the application, which is only concerned with what is being played. The harmonic content of music is important in terms of fidelity, but is always based on the fundamental frequencies present.

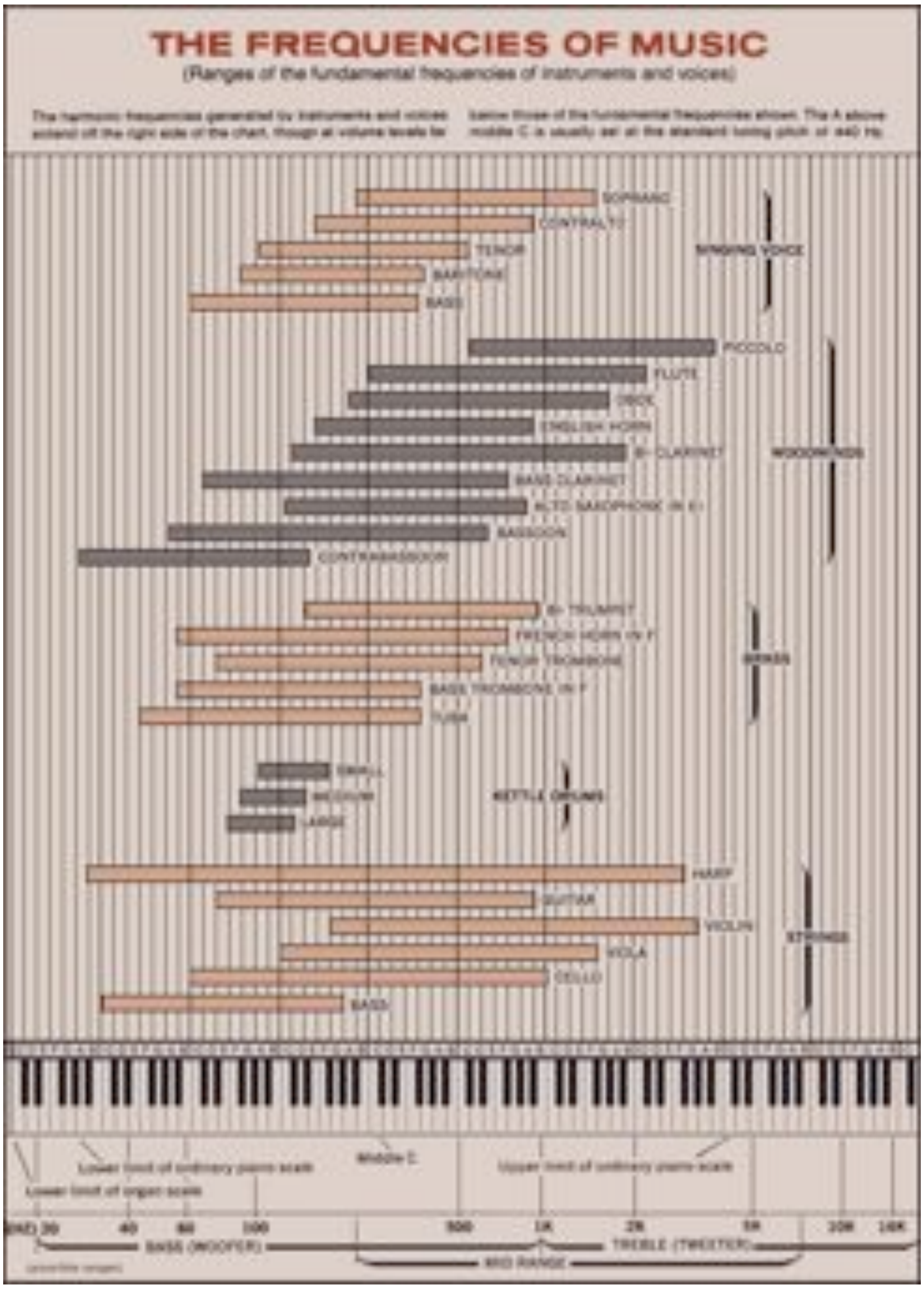


Figure 6.3 A chart showing the fundamental frequencies of common musical instruments [11]

The amplitude of an audio signal determines how loud it is perceived to be. In terms of visualisation, the greater the amplitude of a frequency range, the greater its impact should be upon the scene. To find the amplitude range which is to be considered for the visualisation tests must be run using the application itself. Details of these tests may be seen in Appendix C.12.2

7 High Level System Design

7.1 Execution Cycle Flowchart

The basic execution of the application may be broken down into three stages, once the system resources have been initialised (Figure 7.1). Firstly, the audio data is fetched from the line in port and saved in a buffer. The fast Fourier transform routine then analyses the frequency content of the signal and using the data a graphical output may be displayed.

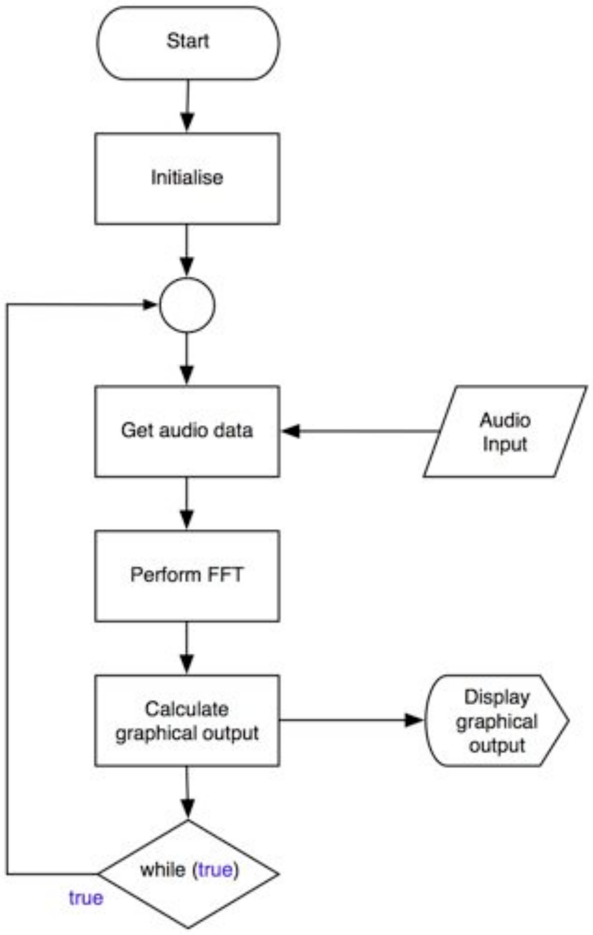


Figure 7.1 The execution cycle

It is here that the importance of speed can be seen; performing the FFT and calculating the graphical output must take less time than it takes to fill the audio buffer. The speed of the whole cycle may be impeded should a single stage be too slow. For the detailed set of low level designs see Appendix A.1.

7.2 System Interaction

There are two system interaction scenarios; audio input (Figure 7.2) and the user interfacing with the GUI (Figure 7.3). An actor is generally restricted to human users or other systems [12]. However, as this application contains only one human interaction the effects of the audio input have been considered a valid actor upon the system.

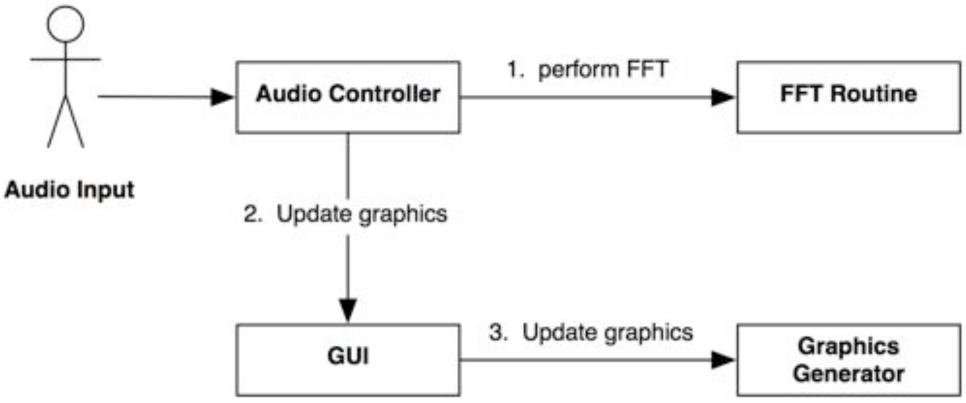


Figure 7.2 Interaction diagram for audio input

Low coupling exists between the classes as method calls are kept to a minimum. There is tight cohesion, maintaining low class responsibilities which are tightly related and sharply focused on their abstracted roles; each class is only involved to an appropriate level in an activity, not taking on technical responsibility for actions which belong to other classes. For example, the GUI may receive the final frequency data results from the Audio Controller, but cohesion would be lost if the GUI had to call the FFT routine itself.

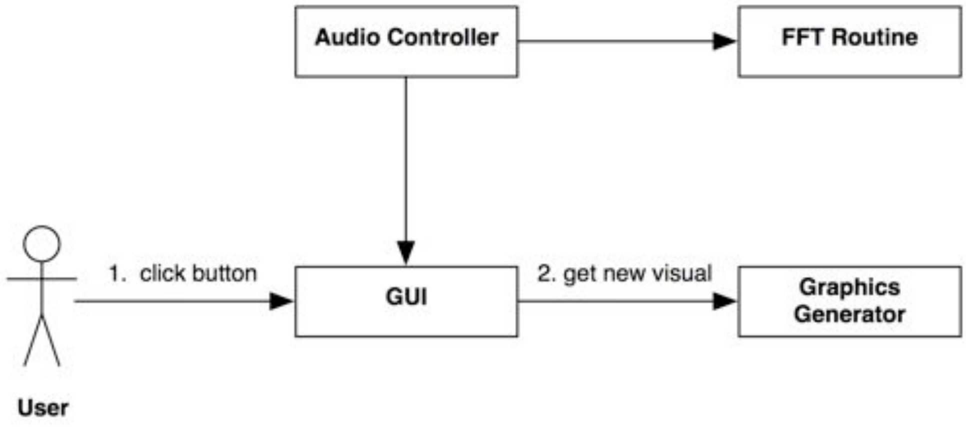


Figure 7.3 Interaction diagram for a button click

7.3 Design Class Diagram

The application is made up of 8 main classes (Figure 7.4) which may be abstracted into three sections; the classes `AudioController` and `FFT` deal with audio input and manipulation, outputting the data to be visualised; `MainWindow` is the front-end to the application, housing the controls and graphical content; the remaining classes generate the 3D graphics content. Aside from low-level system issues the conceptual flow of data is from the top downwards.

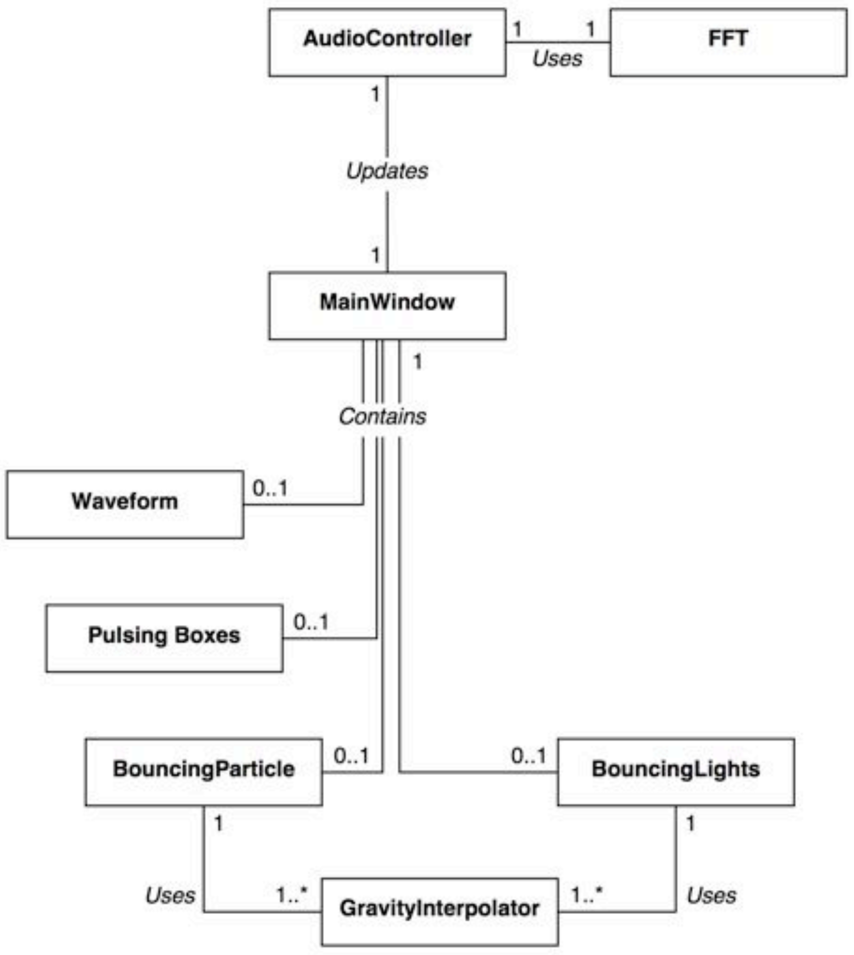


Figure 7.4 High level system design class diagram with association multiplicity and naming

The class structure is an abstraction of the flowchart (Figure 7.5), where each of the stages is dealt with individually with an object-orientated approach. There are five classes concerned with calculating the graphical output, but as they are not used simultaneously may be considered a single entity. Each generates a different type of visualisation with only one being displayed at a time.

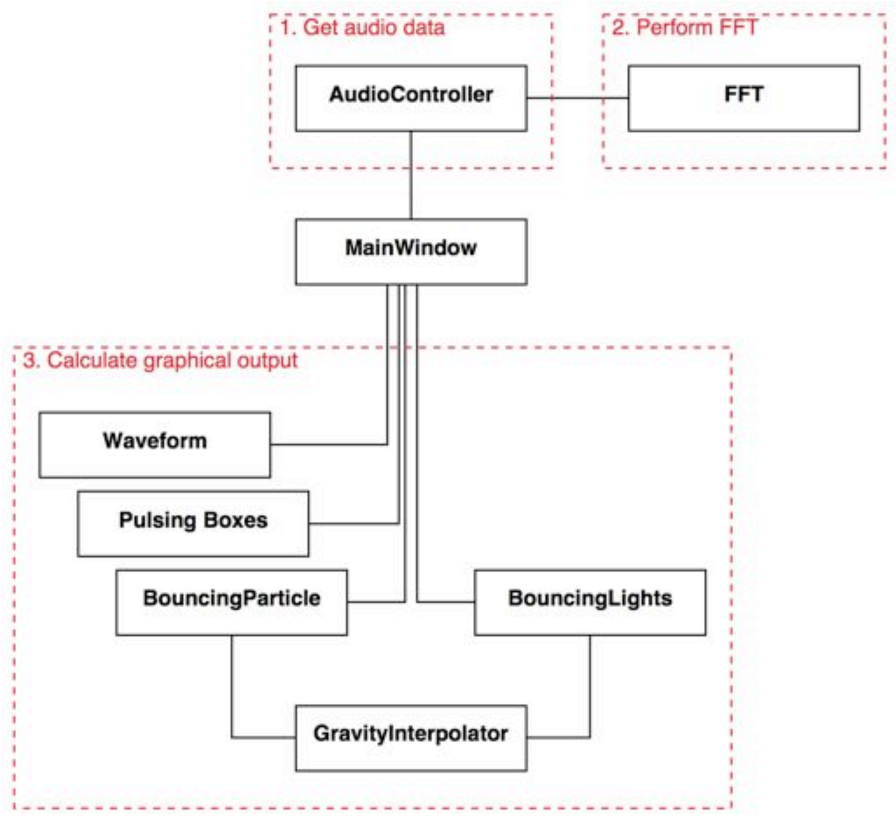


Figure 7.5 Correlation between the flowchart and classes

8 Principals of the System

8.1 Audio Data Structure

The main input to the system is analogue (or possibly digital if an optical or S/PDIF line in were used) audio data from the computer's soundcard. This information must be read, analysed and the results passed on (Figure 7.1). There are many concepts and considerations involved in this process.

8.1.1 The nature of digital audio

Once the audio signal has travelled through the soundcard it is represented in digital form. Although made up of a series of bits, it is organised into a particular audio format definition. Briefly, there are two major factors in a digital audio format:

- **Sampling Rate**, measured in Hz, expresses the number of samples recorded per second. The maximum frequency which can be represented is double the sampling rate [13].
- **Sample Size**, indicates how many bits are used to store each sample's amplitude. This determines both the maximum dynamic range and the signal to noise ratio.

The audio format chosen for the application effects both the speed and results; the higher the quality of format used, the more accurate the results, at the cost of execution speed. As the application is predominantly concerned with the frequency content of the input, sampling rate is kept at the CD quality of 44100Hz. Initially a sample size of 16 bits was used, but extra implementation is required for handling the data, as is discussed in Section 8.1.2. As sample size bears its greatest impact on the quality of sound, a factor that does not affect this application, 8 bits was seen as sufficient to express the amplitude of the signal. This also reduces the computational load.

8.1.2 Audio input using the Java Sound API

There are many ways of implementing an audio input routine using the variety of classes provided. Sun's Sound Programmer Guide [14] provides an example of how this may be achieved (Figure 8.1).

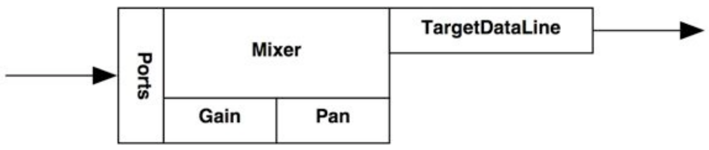


Figure 8.1 Sun's suggested design for an audio input system

It is the `Port` object which represents the hardware audio line in, fetching audio data when the `TargetDataLine` calls its `read` method. However, after implementing this design (see Appendix C.1.1) it was found that the `Port` class is not scheduled for implementation until version 1.5.0 [15]. Instead, the `TargetDataLine` was requested directly from the `AudioSystem` class without any further implementation required for audio input to be achieved.

Using a 16 bit audio format for the `TargetDataLine` requires extra implementation. As is illustrated in Figure 8.2, when a 16 bit sample is placed into a byte buffer it is split in two. A 16 bit sample's contents is not linear, such that when split, only the combination of both 8 bits has meaning. If a sample size of 16 bits is to be used the 8 bit pairs must be combined before processing.

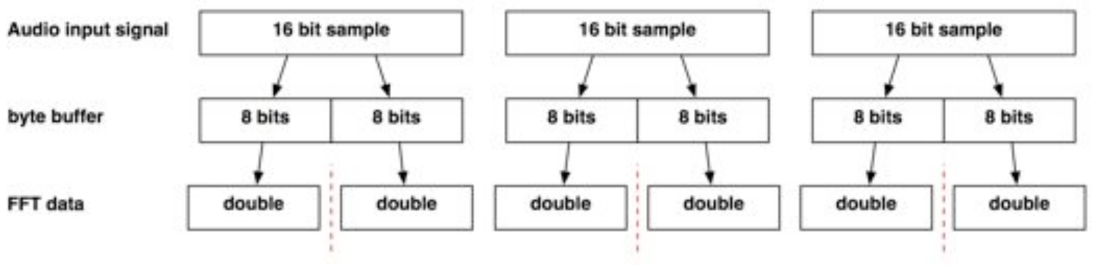


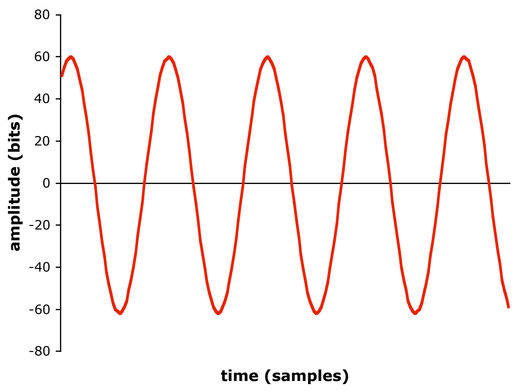
Figure 8.2 Sample splitting

By using an 8 bit sample size, bytes may be read straight out of the buffer, reducing the computation load. This reduction is twofold as both half the amount of bytes are passing through the system and the additional implementation is not required.

8.1.3 Fast Fourier Transform

The graphical visualisation uses the audio signal's frequency content as its source. In order to extract this frequency data from the linear byte stream a Fast Fourier Transform (FFT) routine is used. A specified number of samples are taken by the routine and their overall frequency spectrum returned. Figure 8.3 shows the correlation between the sample amplitude and frequency data, the details of which may be seen in Appendix C.3.4.

The implementation of an FFT routine's code is beyond the scope of this project, but as it forms such a fundamental element to the operation of the program, a conceptual understanding of its operation is still necessary. There are two directions in which an FFT may be performed; a forward transform takes the time dependant amplitudes of a waveform and converts them into frequency dependant amplitudes, as used in this application; the backward transform works in exactly the opposite direction, going from a frequency to time dependant waveform.



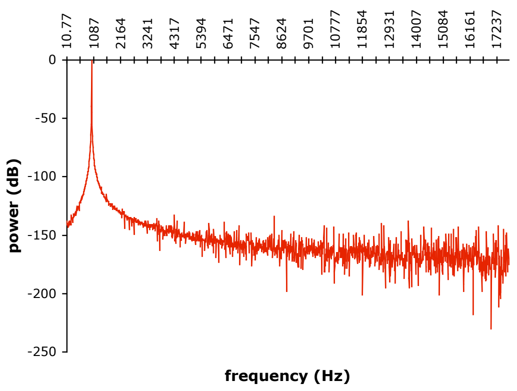


Figure 8.3 The waveform and frequency content graphs for a 1000Hz sine wave extracted from the application

As frequency is an expression based on time, a series of samples must be used by the FFT. The input consists of both the real set of numbers prescribed by the audio signal and a set of imaginary numbers. As only real numbers are being used the resulting output from the FFT routine will only contain frequency data in the first half of the sample set. The output must be interpreted into decibel values for a correlation to be found with the audio input signal.

In order to determine the power spectrum of the wave in decibels at a frequency *x* the squared values of the real and imaginary components are added together:

$$\mathbf{Power}(x) = \mathbf{Real}(x)^2 + \mathbf{Imag}(x)^2 \qquad (8.1)$$

To translate the power spectrum into decibels 10 is multiplied by log base 10 of *Power(x)*. Normalising the power spectrum will give a more even range of values and is achieved by dividing by the maximum power value first:

$$\textit{Normalised dB}(x) = 10\log_{10}\left(\frac{Power(x)}{Power(x)_{MAX}}\right) \qquad (8.2)$$

In order to plot the results of Equation 8.2 the frequency values for which each of the entries in the series corresponds must be found. The frequency multiplier (8.3) specifies the frequency difference between individuals in the series.

$$\textit{frequency multiplier} = \frac{f_{MAX}}{(\textit{number of points})} \qquad (8.3)$$

8.1.4 Calculating Frequency Band Coefficients

In order to maintain tight cohesion within the system, the frequency spectrum must be translated into a set of coefficients by the `AudioController` in a way that does not require any technical knowledge from the graphics classes.

Frequency is an exponential unit and so must be divided accordingly; the octave is a unit used to define a frequency range, specified by a doubling in frequency. Hardware devices such as graphic equalisers split an audio signal into octave bands (Figure 8.4), or fractions thereof [16].

31	63	125	250	500	1000	2000	4000	8000	16000

Figure 8.4 Centre frequency values often used by hardware devices

Such divisions are generic and may be adapted for the task at hand. It was decided that 5 frequency bands (Figure 8.5) would give the best range of values, whilst not being too computationally expensive. An increase in the number of bands creates extra computation not only in the `AudioController` class but also in the extra visualisation required. These values were chosen as they cover the frequency range specified by the table in Figure 6.3.

Band 1	Band 2	Band 3	Band 4	Band 5
31 → 125	125 → 250	250 → 500	500 → 1000	1000 → 2000

Figure 8.5 The 5 frequency bands used

The average value for the results of Equation 8.2 are found for each of the frequency bands, giving a set of five decibel values. These values must be converted into a coefficient ranging between 0 and 1 for use by the graphics classes; this both scales the decibel values more expressively and maintains good encapsulation. Much investigation went into finding the right mathematical function to perform the translation, the details of which can be found in Appendix C.12.2. The final scaling function chosen for calculating the coefficients is shown in Figure 8.6.

$$k = 1.5e^{0.02p} \qquad k < 1.0$$

Figure 8.6 The scaling function for calculating the coefficient k, where p is the normalised decibel value from equation 7.2. Values of k are limited to no more than 1.0

8.2 Scenegraph Elements in Java 3D

The scenegraph is a hierarchical data structure which specifies the relationships between instances within a 3D scene; an element in the scenegraph may have any number of children and a single parent. In the application there are two main scenegraph branches – the view and content `BranchGroups`.

8.2.1 View Side of the SceneGraph

All elements that are responsible for rendering to the screen are contained within the view side of the scenegraph (Figure 8.7) [17]. The class `SimpleUniverse` defined in the `com.sun.j3d.utils` package contains all the view elements within itself, but at the cost of easy access to control. In order to be able to have freer control over the view elements the `VirtualUniverse` class was used instead and the hierarchy built manually.

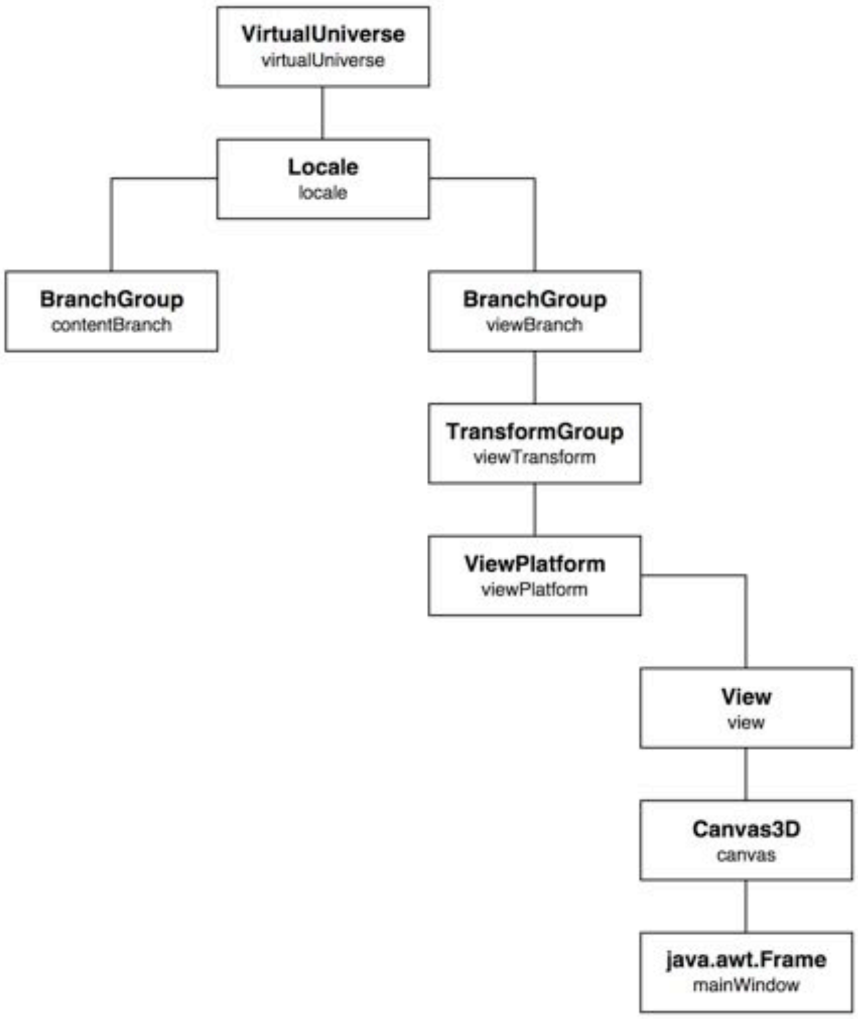


Figure 8.7 The view side of the scenegraph

8.2.2 Content Side of the SceneGraph

The physical objects within a scene and the elements that control and define them are contained under the content side of the scenegraph. Each visualisation's scenegraph is shown in Appendix A.3, as their details are different. However, the techniques used are similar, in particular the use of hierarchy when manipulating objects. Figure 8.8 shows a typical structure used in the application to apply a transformation to an object.

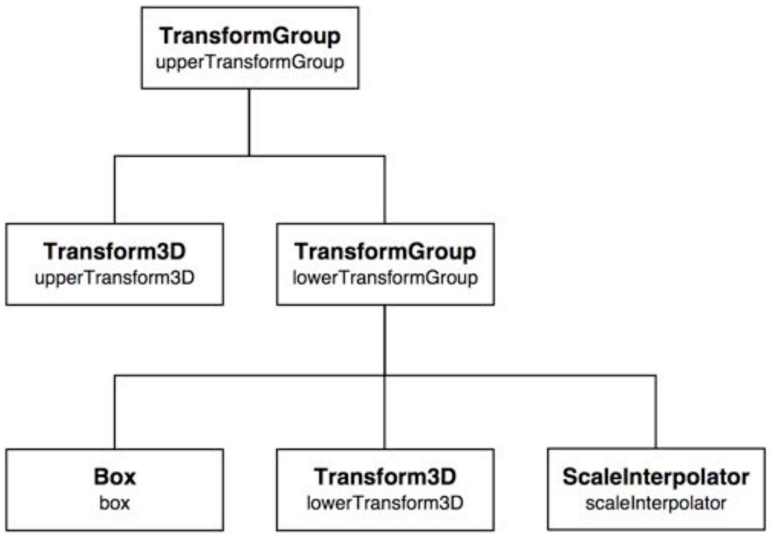


Figure 8.8 A simple scenegraph structure

The structure above specifies a `Box` object, its position, and the `ScaleInterpolator` attached to it. A `TransformGroup` is manipulated by a single `Transform3D` object contained directly below it, the effects of which have a knock-on effect upon every child element. Therefore, in the scenegraph above, `upperTransform3D` can be used to position the box and will not be effected by any further manipulation using `lowerTransform3D`.

The `Interpolator` class contains an array of subclasses for interpolating an object between states; a gradient of values is applied to a target element associated with a physical object, over a period of time. This time period is specified to the `Alpha` class, of which a parameterised version between 0 and 1 is returned to the interpolator.

8.3 Simulation of Mechanics

Although the visualisation does not bear any resemblance to reality, the simulation of real-world motion using mechanical functions gives greater effect to the graphics. By extending the abstract `TransformInterpolator` class and overriding the `computeTransform` method, custom interpolator classes could be implemented.

8.3.1 Implementing Custom Interpolators

The Alpha class provides parameters for the attenuation of increments between 0 and 1. However, as can be seen in Figure 8.9, the acceleration and deceleration parameters are paired up for increasing and decreasing phases, limiting the motion which can be simulated. Therefore, a custom interpolator must be implemented in order to interpret a linear set of Alpha values into a more advanced function.

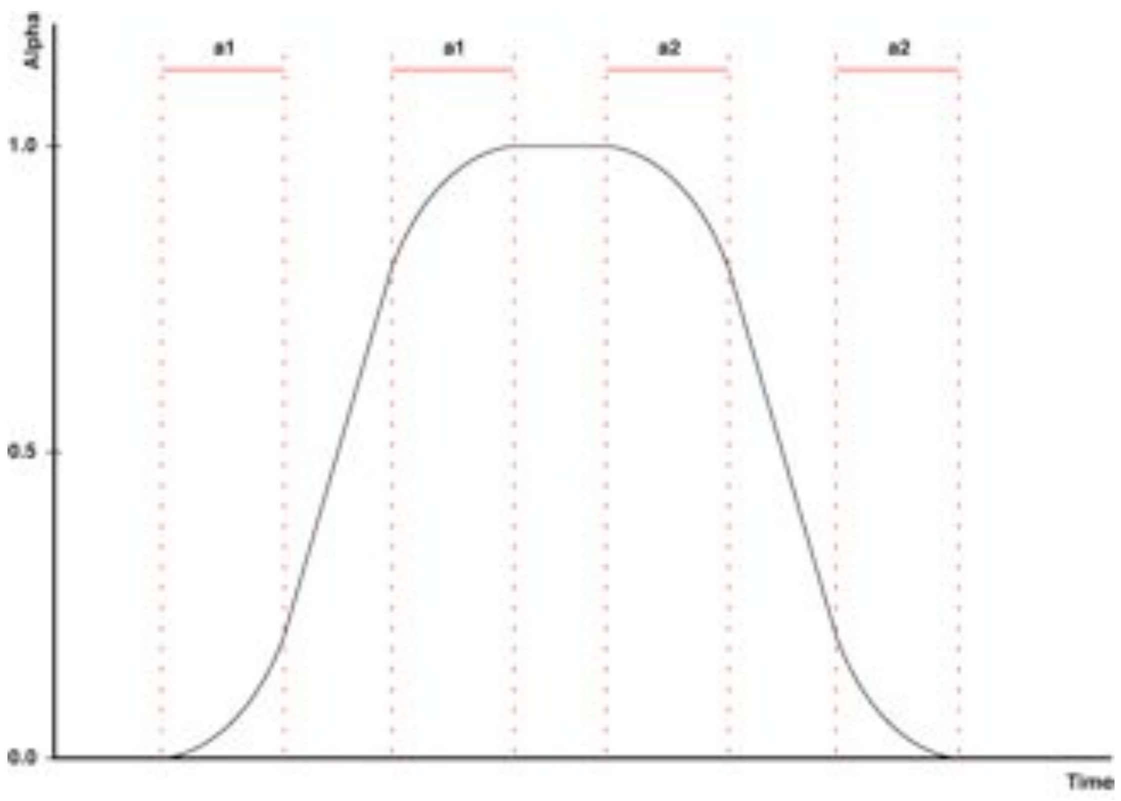


Figure 8.9 Attenuation regions within the Alpha class

The `computeTransform` method in a `TransformInterpolator` subclass gets called every frame for an update to the accompanying `Transform3D` object. The method still gets values between 0 and 1 from an Alpha object to give a time parameter for the transformation, but this only specifies the overall pace of the motion.

8.3.2 Modelling a Bouncing Ball

Two visualisations in the final release of the application utilise the `GravityInterpolator` class, which models the motion of a sphere being struck vertically up into the air and moving under the influence of gravity. The displacement of an object is defined by the equation:

$$s = ut + \frac{1}{2}at^2 \qquad (8.4)$$

u = initial velocity
t = time
a = acceleration

The only acceleration acting on the sphere is gravity, thus making a = -9.8 m/s^2. The initial velocity directly influences not only the height at which the ball will peak but also the duration of elevation. As the `Alpha` values run between 0 and 1, a value of 4.9 is used for the initial velocity, giving a base relationship between time and displacement (Figure 8.10).

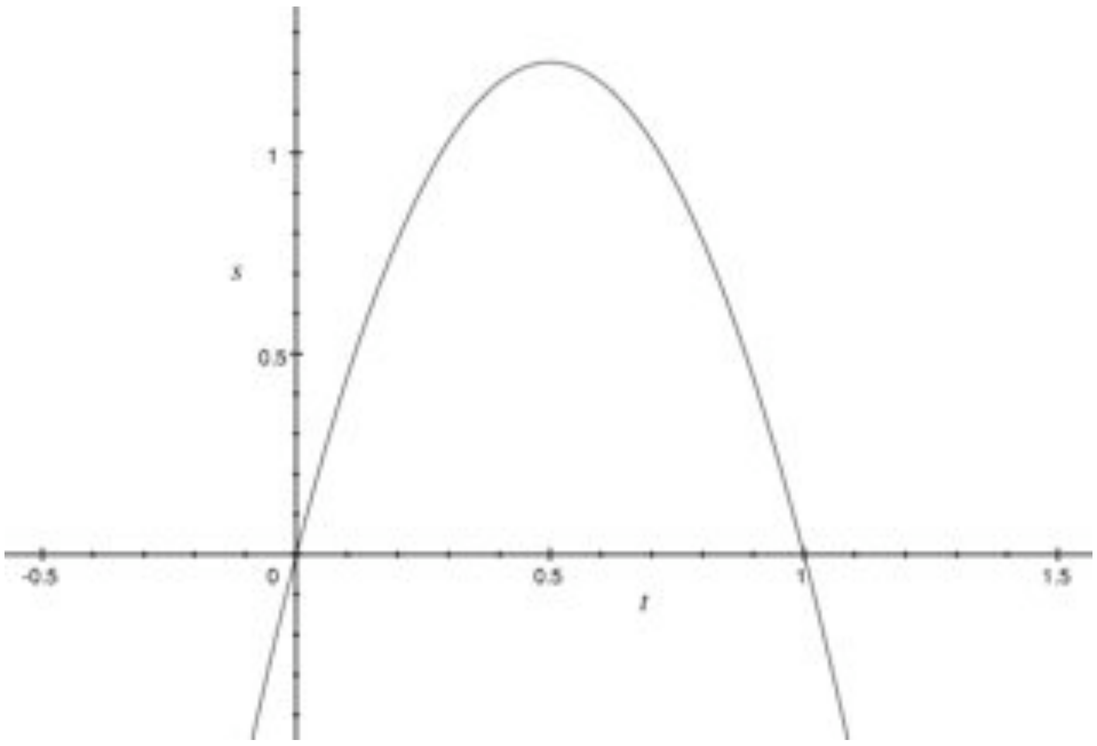


Figure 8.10 The function s = ut + $^1/_2$ at^2

The most mechanically correct way to change the peak height would be to alter the initial velocity at which the ball is projected, with the relationship shown in Figure 8.10 being the maximum achievable height. This leads to some issues concerning updating the graphics; calculating the initial velocity required to reach a certain peak height is not possible within the application as the dynamic manipulation of polynomials would be necessary. A linear relationship between peak height and initial velocity is possible but, as half the initial velocity results in a quarter of the peak height, a visual imbalance would be created.

The more computationally efficient method of changing the peak height is to apply a multiplier to the calculated displacement. This is not mechanically correct as the total time in elevation is the same, regardless of the peak height. However, the regularity of motion between all five spheres lends itself well to visualising music and the idea of it being a "glorified graphic equaliser". The intention of using mechanical equations was to add an element of realism to the motion.

When the flight of a ball is reset by the `update` method whilst in descent, seemingly being struck from below, the reset state must be changed. The `Alpha` object is reset back to zero, but the initial displacement value must not be set to zero as well. By offsetting the `Alpha` value the initial displacement can be changed whilst maintaining the correct motion characteristics, as can be seen in Figure 8.11. A single check must be made to ensure that the displacement does not fall below zero.

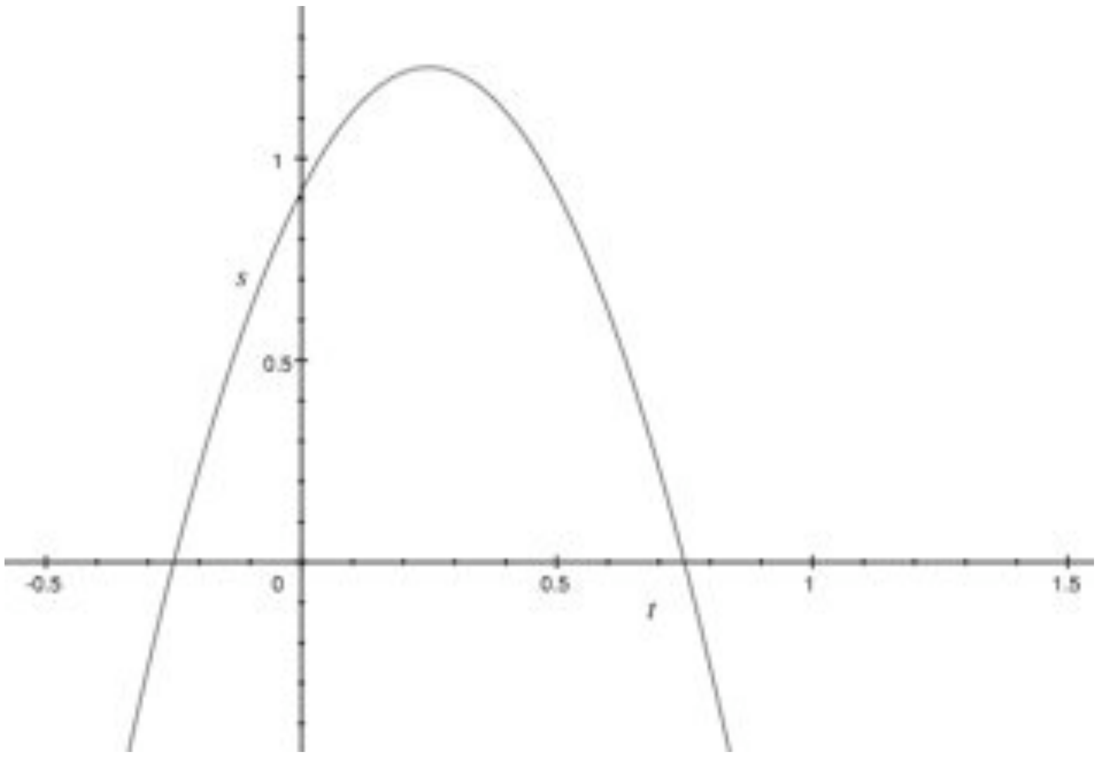


Figure 8.11 The function s = u(t+0.25) + $^1/_2$ a(t+0.25)^2

If the offset value is to be calculated then Equation 8.4 must be re-arranged. The working out is included in Appendix B.2, but the result can be seen here:

$$k = \frac{a + \sqrt{a^2 + 4as}}{2a} \qquad (8.5)$$

a = -4.9
s = displacement

8.3.3 Modelling a Flocking Particle

The aim of the `ParticleInterpolator` class was to create a flocking particle which would follow a ball, similar to if it were attached to be piece of elastic. Applying an acceleration proportional to the distance away from its natural position – the point of attraction – would create such an effect.

The use of Equation 8.4 with an ever incrementing time value, fuelled by an `Alpha` object, is not possible as it only applies to constant acceleration. In such cases where acceleration is not constant, calculations using this equation must be carried out using differentiation, which is not possible in software.

Simple harmonic motion describes the motion of a particle towards a fixed point, where its acceleration is proportional to the distance away from that point [18] (Equation 8.6). However, this is dependant on the point of oscillation remaining stationary.

$$\ddot{x} = -w^2 x \qquad (8.6)$$

where w is a real number

To overcome these issues the motion of the particle is calculated incrementally on every call to the `computeTransform` method, using the final velocity and displacement values from the last call to find the new position. The acceleration of the particle is changed, based on a set of conditional statements. More than 1.5 units below the particle's natural position counts as under the influence of elasticity, otherwise moving freely under the force of gravity. To prevent the particle from oscillating uncontrollably its deceleration value when within the elastic region is much greater than both its acceleration due to elasticity and the force of gravity.

Appendix B.1 illustrates the flow of execution for the `computeTransform` method. There are 6 values which must be calculated:

1. The "natural position" of the particle
2. The time value since the last method
3. The displacement from the natural position
4. The acceleration
5. The new displacement value
6. The final velocity, for use as next call's initial velocity

This is computationally expensive, such that a mass of particles would not be possible to generate in real time. A more sophisticated flocking algorithm would be required to allow for a large scale particle system.

8.3.4 Modelling Spiral Motion

The `SpiralInterpolator` class utilises Equation 8.7 to specify a spiral path, where t is time. The x, y and z parameters correlate to the fields within the `Transform3D` class, such that the transform may be applied directly. By updating the field t the path is followed by the object. Plotting the equation gives the graph shown in Figure 8.12.

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} Lt \\ -\sin(Dt) \\ -\cos(Dt) \end{bmatrix} \tag{8.7}$$

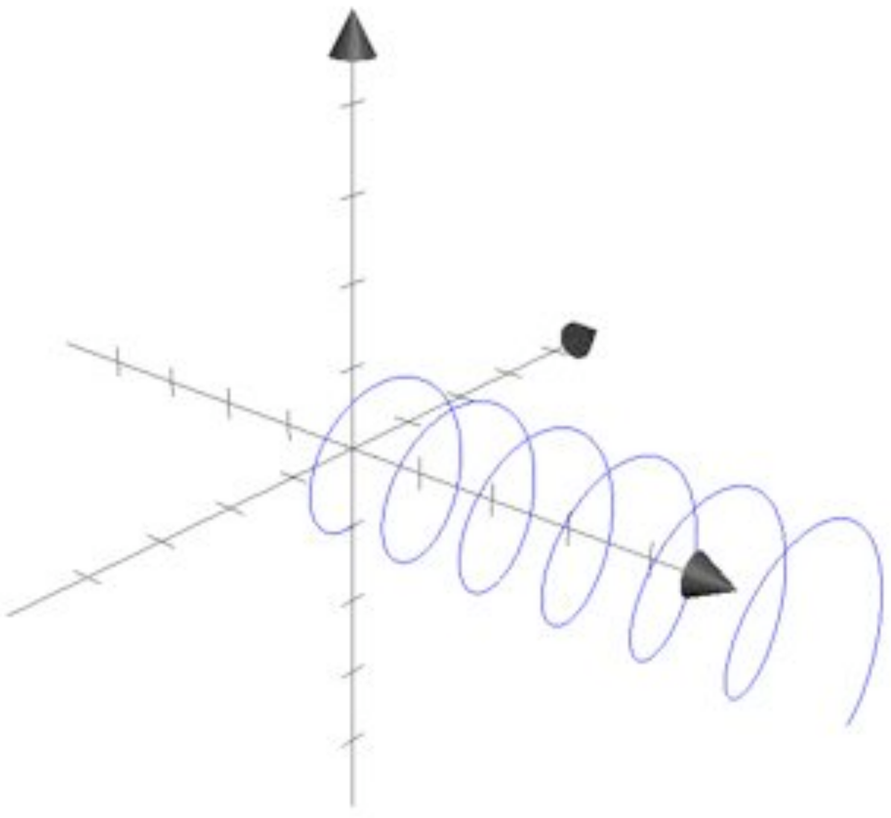


Figure 8.12 The spiral function, where *D* = 6

The graph above specifies the range $t = 0 \ldots 2\pi$, creating 6 full revolutions. However, `Alpha` values range from 0 to 1, providing less revolutions. The spirals are created by the oscillations of the sine and cosine functions in the y and z coordinates. The variable *D* increases the frequency of oscillations, thus specifying the density of spirals so that when increased more revolutions are added to the path. The length of traversal is determined by the x coordinate. The variable L magnifies the motion across the x-axis, hence controlling the length of traversal.

8.4 Resource Handling

The Java runtime system carries out memory management tasks asynchronously. When an object no longer has a reference the garbage collector picks it up and frees the memory and process time it occupies [19]. Sun suggest that the object be dropped by assigning it a `null` value [20]. When changing between visualisation themes this method was attempted but, as may be seen from Figure 9.1, the garbage collector failed to pick up the object. An attempt was made to manually invoke the garbage collector to clean up any unused objects, as follows:

```
pulsingBoxes = null;

// assign a new object to pulsing boxes
```

```
pulsingBoxes = new PulsingBoxes();
add("Center", pulsingBoxes.getCanvas3D());

// force the garbage collector to run
System.gc();
```

However, the same resource leak occurred, causing the application's performance to decline as more and more objects execute concurrently. To solve this issue a new method `kill` was added to each of the graphics classes:

```
public void kill() {
        universe.removeAllLocales();
}
```

This tells the `VirtualUniverse` object to remove all of the `Locale` objects attached to it. As can be seen in Figure 8.7, the effect of this clears away the entire contents of the scene, clearing up the resource leak.

8.5 Final Release

The following screen shots show the deployed application visualising a piece of music. Each visualisations appeared as the result of pressing the highlighted button. The first screenshot shows the application when first launched.

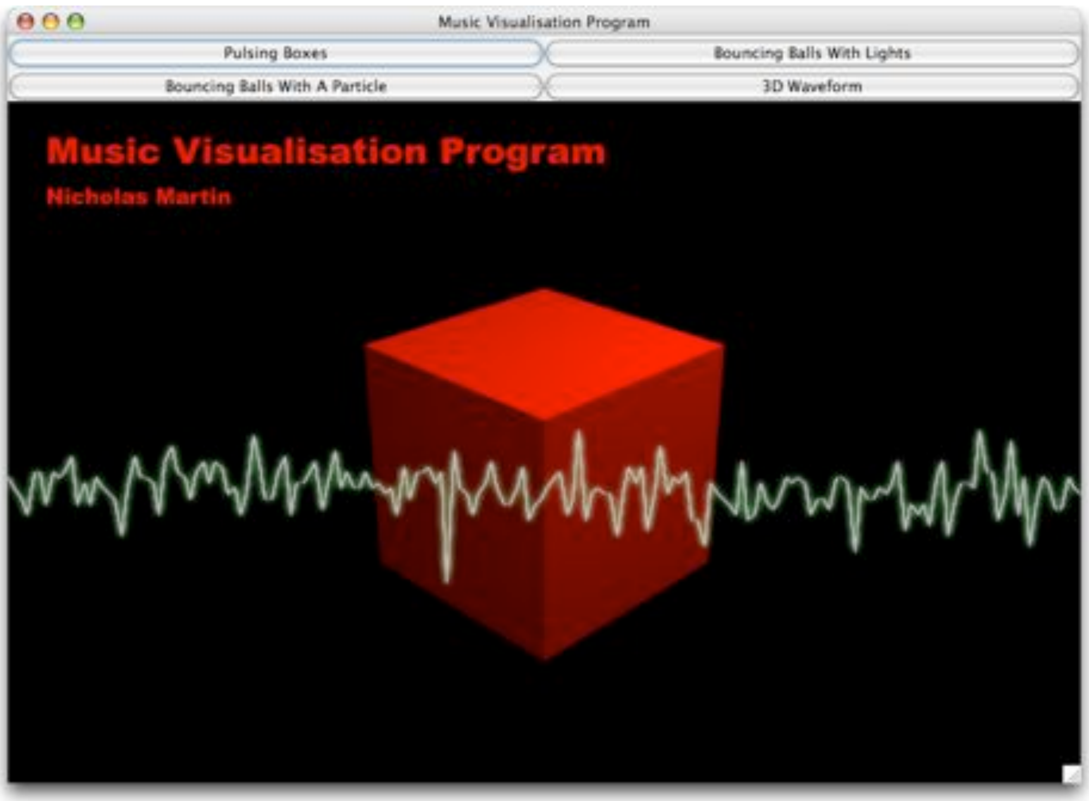

Music Visualisation Program
Pulsing Boxes
Bouncing Balls With Lights
Bouncing Balls With A Particle
3D Waveform
Music Visualisation Program
Nicholas Martin

Figure 8.13 The introductory screen

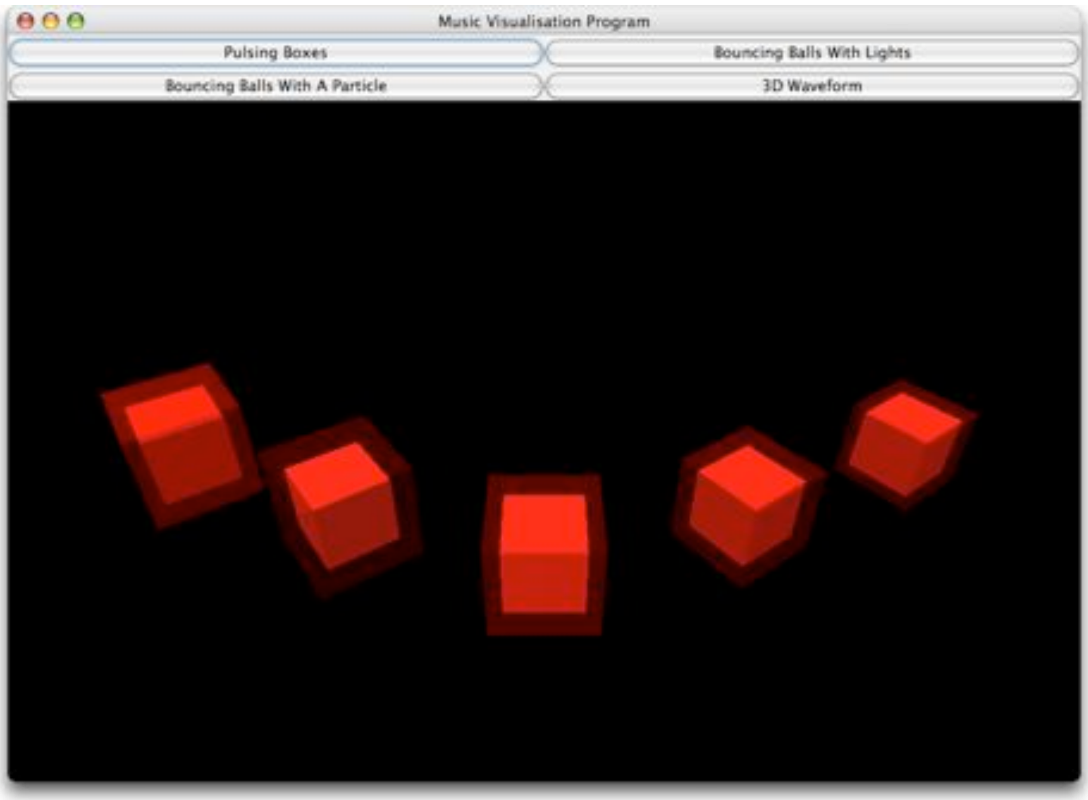

Music Visualisation Program
Pulsing Boxes
Bouncing Balls With Lights
Bouncing Balls With A Particle
3D Waveform

Figure 8.14 The pulsing boxes visualisation theme

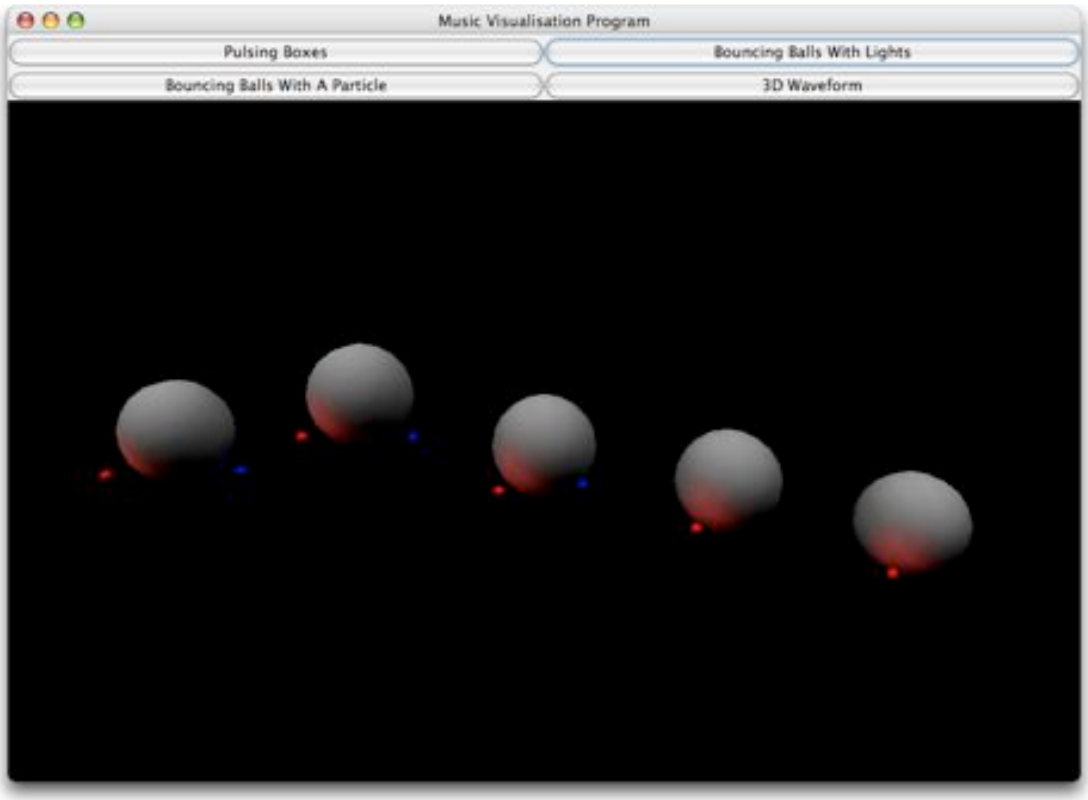

Music Visualisation Program
Pulsing Boxes
Bouncing Balls With Lights
Bouncing Balls With A Particle
3D Waveform

Figure 8.15 The bouncing balls with lights visualisation theme

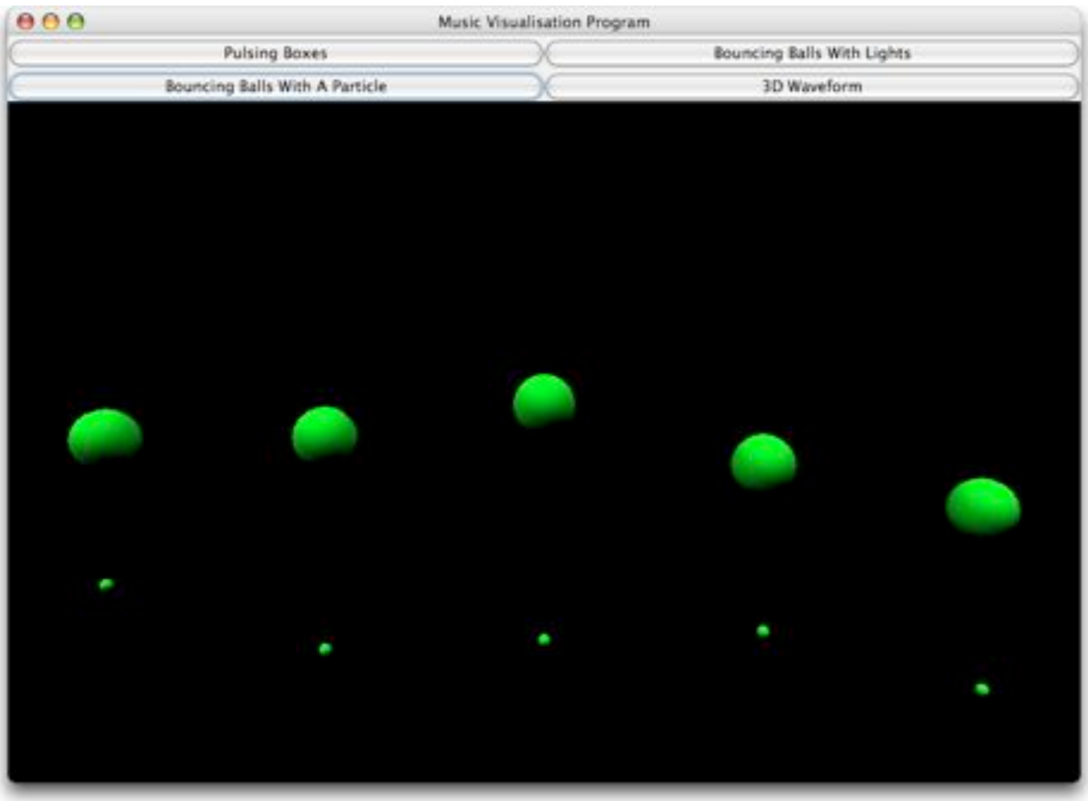

Music Visualisation Program
Pulsing Boxes
Bouncing Balls With Lights
Bouncing Balls With A Particle
3D Waveform

Figure 8.16 The bouncing balls with a particle visualisation theme

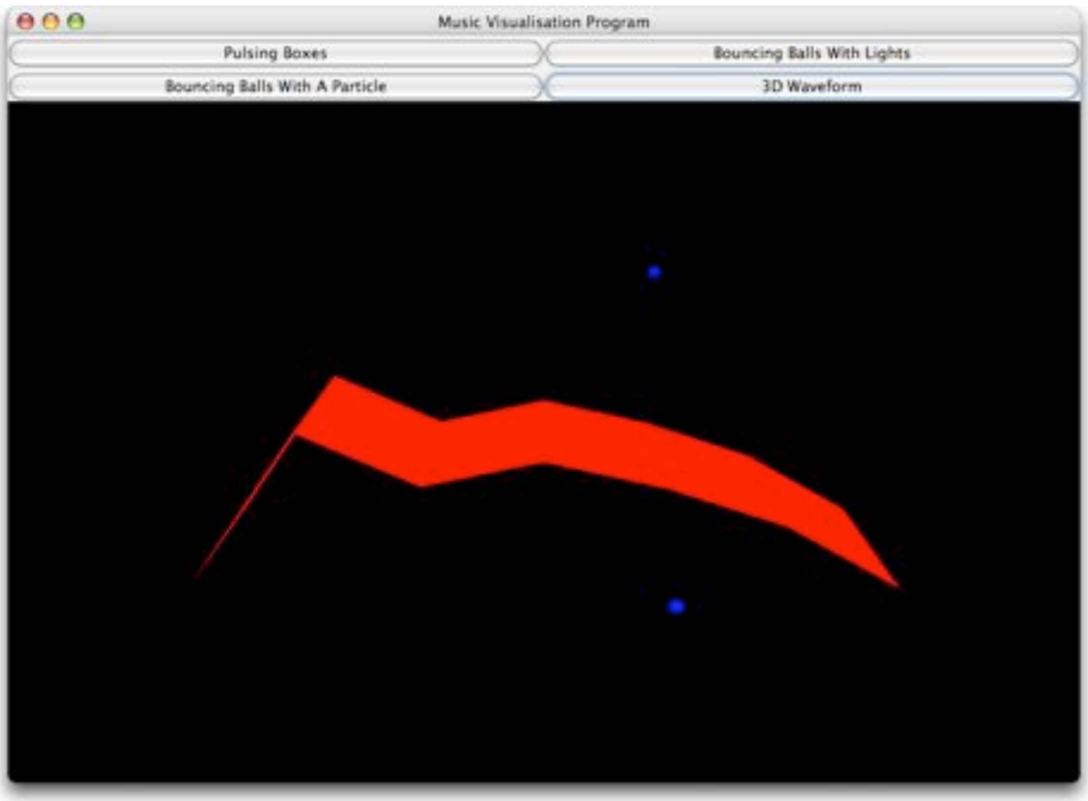

Music Visualisation Program
Pulsing Boxes
Bouncing Balls With Lights
Bouncing Balls With A Particle
3D Waveform

Figure 8.17 The 3D waveform visualisation theme

9 Testing

9.1 Iteration Testing

The lifecycle diagram in Figure 6.2 specifies that each iteration ends by testing the developed code. This allows for errors to be caught before they become too ingrained in the system. As discussed Section 6.1.2 in the application may be divided into two main sections – the audio processing and graphic generation.

9.1.1 Audio System Testing

The purpose of the audio system is to accept and process an input signal and produce a set of coefficients. The execution occurs invisibly within the operating system and so must be tapped into for its activity to be seen. As the nature of the audio system is one of calculation, a defined set of results will be expected from a specific input signal.

The amount of data being processed for audio is vast – 44100 bytes per second is far too much information to be analysed by printing to screen. Instead, a "snapshot" of the data may be written to file using the `FileWriter` class. This can then be copied and pasted into *Microsoft Excel*, allowing for the data to be plotted in a graph. Comparisons can be made between result sets, such as the effects of different FFT buffer sizes (see Appendix C.3.5).

It is difficult to prove the working of the audio system when using music as an input signal; not only does music contain wide and irregular signal content, comparing results against a source is difficult as coordinating both snapshots would be impossible. By using *Felt Tip Sound Studio* a sine wave of a specific frequency was generated and used as the input signal, making it easier to analyse and confirm results. This could be used to test both the successful input and frequency analysis of the audio.

9.1.2 Graphical Content Testing

The graphics generation classes create a visual output based on the coefficient values input from the `AudioController` class. Therefore, providing that the input values are known, the graphical output is sufficient for testing. By making the array of coefficients fixed the input values may be correlated with the rendered output. After the graphics have been confirmed to be correct using fixed coefficient levels the visualisation may be run using music.

9.2 Final Testing

With the application in its finishing stages, final testing allows for the assessment and investigation of its performance as a whole. More advanced testing techniques analyse the workings of the system, quantifying its performance rather than simply confirming it. The opinions of target users ensure that a rounded evaluation may be made.

9.2.1 Visualisation Frame Rate

The benchmark test for a graphical application is its frame rate. The greater the number of frames per second (fps), the better the quality of animation. Should the frame rate fall lower than 30 fps, as laid out in the Specifications, then the visualisations' impact will be affected.

To test the application's frame rate a custom `Behavior` class is added. `FpsBehavior` (see Appendix D.8) prints to terminal the fps of the scenegraph it is added to. By using the `WakeupOnElapsedFrames` subclass of `WakeupCondition` the fps may be calculated.

An `FpsBehavior` object was added to the `PulsingBoxes` class and the application run, initially with no audio input, to test the output of the fps. The terminal showed an output of frame rate values around 150 for the pulsing boxes theme. However, when another button was pressed the output continued, the frame rate dropping considerably. This is in spite of the `FpsBehavior` only being added to the `PulsingBoxes` class. The application was run again and the pulsing boxes button pressed at intervals so that the effect on frame rate could be seen (Figure 9.1).

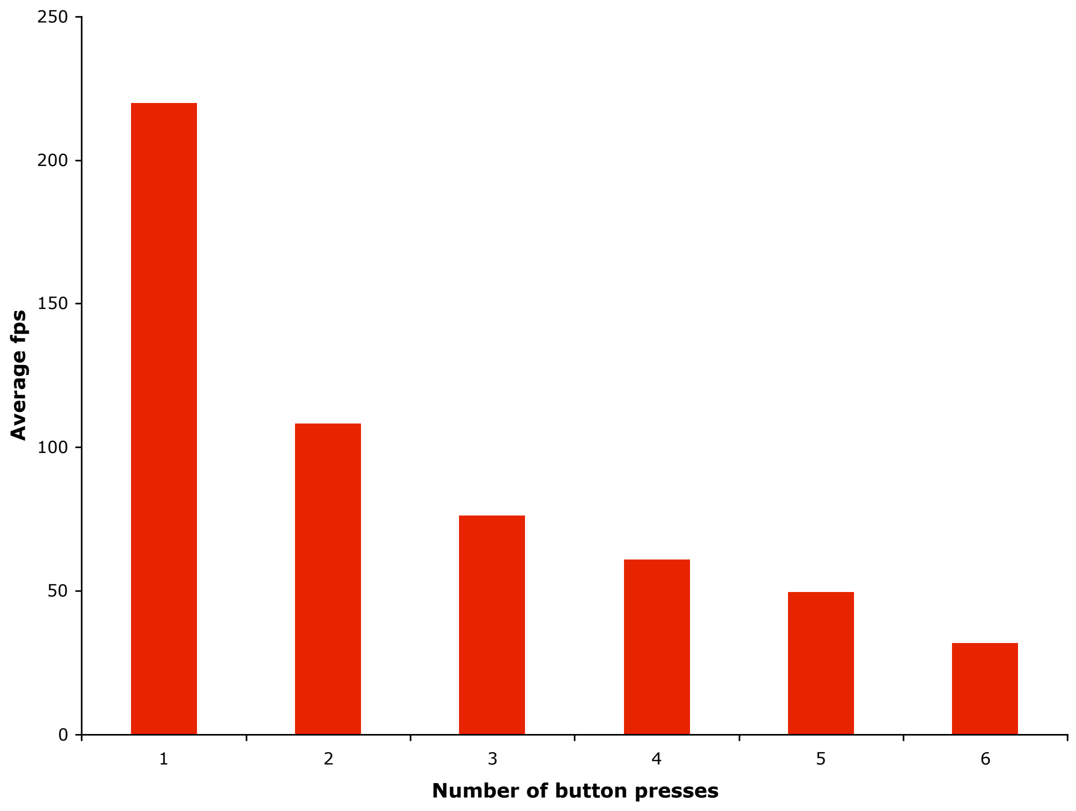


Figure 9.1 The effect of button presses on the frame rate

This depletion of frame rate is caused by scenes being unsuccessfully discarded, despite references to them being severed. A discussion of this, and the steps taken to solve it, are shown in Section 8.4.

With the resource leak fixed the application's frame rates could be tested fully. Figure 9.2 shows the frame rates for each of the themes, with and without an audio input.

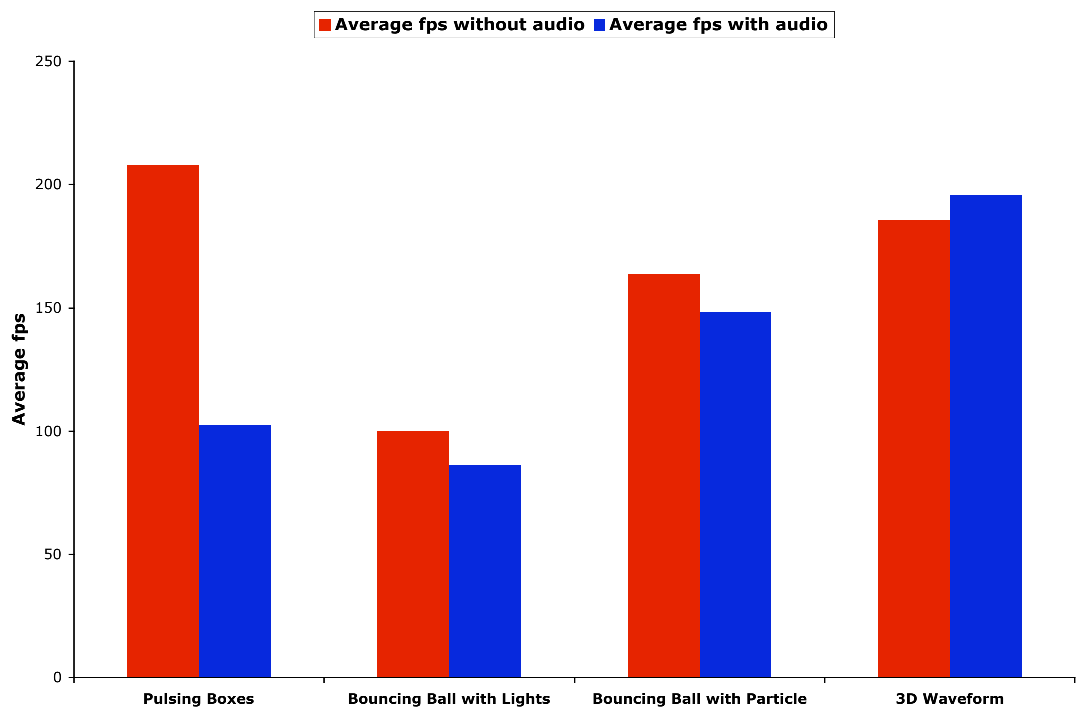


Figure 9.2 Comparison of frame rates with and without audio

The frame rate of all themes remains above 90 fps, even with the audio input running. The greatest drop in frame rate occurred with the pulsing boxes theme, in keeping with discussions made about the computational power required for transparency in Appendix C.8.5. The lowest frame rate appeared with the bouncing spheres with lights theme, because of the large amount of lighting implemented. Contrary to other themes the 3D waveform increased its frame rate when an audio input was used, probably due to the increased activity it demanded. Overall, the performance of the graphics is in keeping with the Specifications laid out in Section 4.1.

So that a bearing on the application's performance may be made the same test is carried out on *Apple's iTunes*, a similar piece of software discussed in Section 2.1. Within the application's options it is possible to display the frame rate of the visualisation. These values fluctuate greatly between 25 and 60 fps, although the complexity of *iTunes'* graphical content must be borne in mind.

9.2.2 Audio / Visual Latency

In order for the visualisation to be effective there must be minimal latency between the audio input and resulting graphical output. By taking two separate audio channels containing the same audio pulse this latency value can be compensated for when they are played simultaneously (Figure 9.3). The delay value is increased until the pulsing sound matches the visualisation. This makes the latency difficult to measure accurately as it is based on human perception. When carried out on the application the latency value was found to be ≈ 80ms.

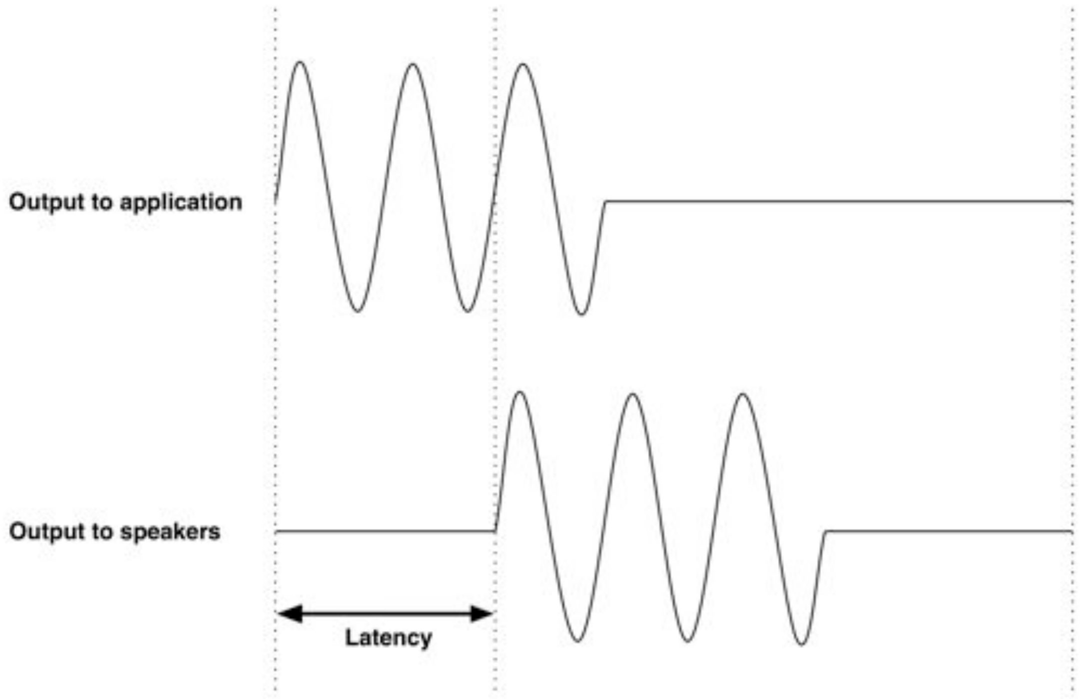


Figure 9.3 Method for measuring latency figure

9.2.3 System Resource Demands

Although when in deployment the application will almost certainly have all of a computer's resources available to it, an analysis of CPU and memory demands is still important. Again, *iTunes* is used as a comparative application so that the results may be gauged. In order to monitor CPU usage *App Monitor* is observed for each of the visualisations, as may be seen from Figures 9.4 to 9.8. The red line plots the average CPU usage, not the blue numerical value.

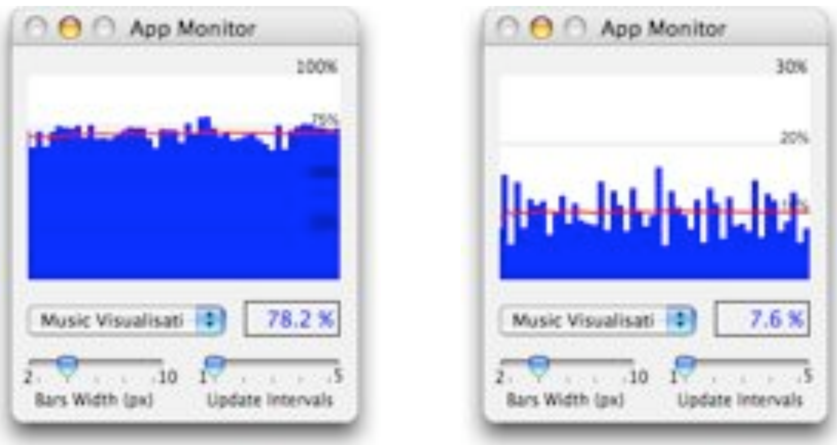


Figure 9.4 Comparison of CPU usage with and without audio for the pulsing boxes theme

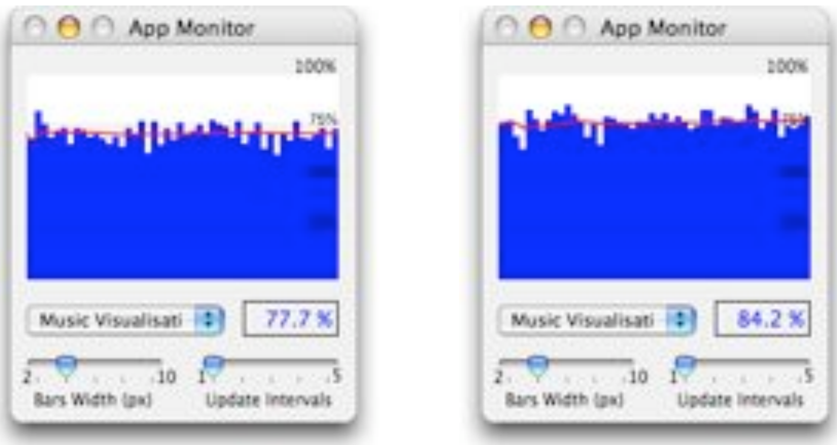


Figure 9.5 Comparison of CPU usage with and without audio for the bouncing spheres with lights theme

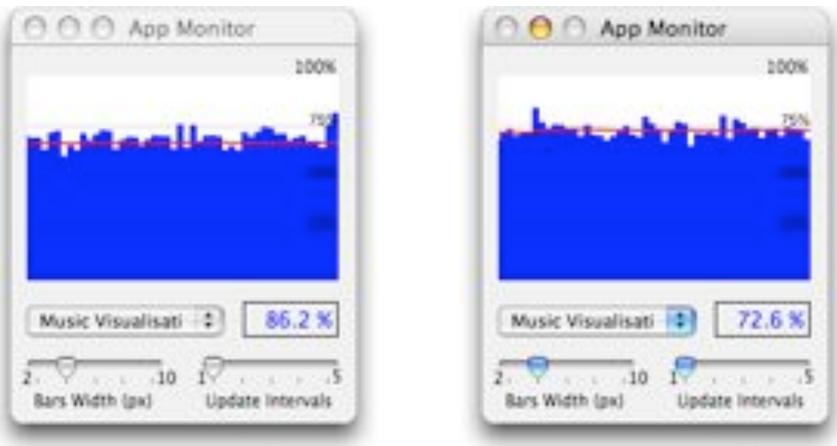


Figure 9.6 Comparison of CPU usage with and without audio for the bouncing spheres with particles theme

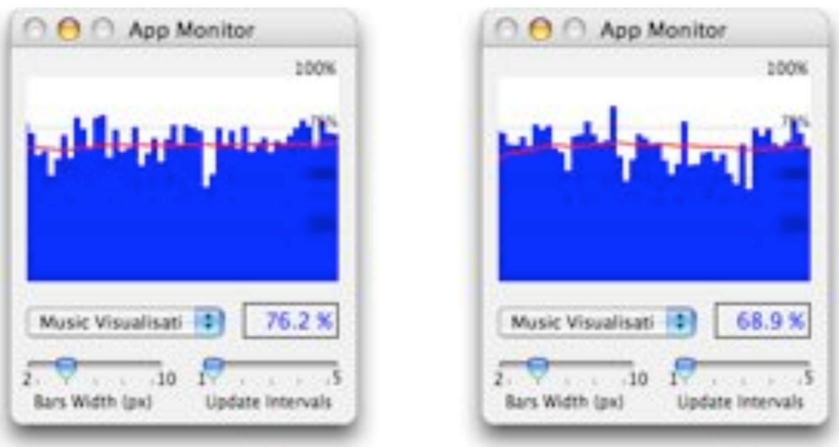


Figure 9.7 Comparison of CPU usage with and without audio for the 3D waveform theme

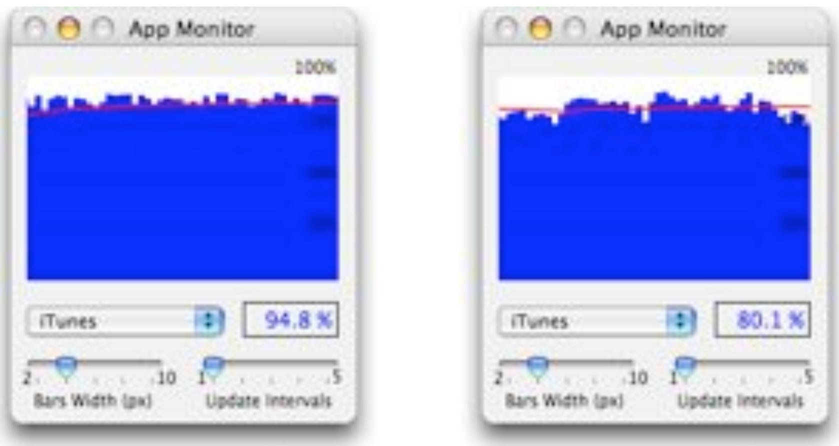


Figure 9.8 Comparison of CPU usage with and without audio for iTunes

What is important to ascertain from this test is that the CPU is not under full capacity, allowing the application to execute to its full potential. For all themes the CPU usage is around 75% when an audio input is used, as opposed to the 80% used by *iTunes*. Therefore, the application runs within the scope of the system.

9.2.4 The Influence of Different Musical Styles

There is a set of criteria that lends itself to better visualisation results; the presence of a distinct beat and a simplistic structure create the separate peaks which the application is suited to. The visualisation works best when emulating pulsing in an audio signal.

Music that uses a large amount of distortion, such as rock, floods the visualisation as a mass of frequencies are contained across the entire spectrum. However, electronic music contains a more minimal structure where the individual elements are easier to separate as the sampled sounds used do not have to same frequency spread as acoustic instruments.

9.2.5 User Feedback

The opinions of users are important when assessing a developed application, their views giving an insight into the success of the project. Users were shown the application with a suitable musical input playing and asked to give their opinions. No prior information was given about the software, other than its intended use.

The feedback received was very positive. Most users commented that they had never seen such a program used in live venues and thought it would match the "kitsch" style of some electronic music. It was noted how the application used the same graphical style that may be seen being used by many "electronica" record labels, such as Warp Records [21].

When asked about how the music correlated with the graphics a mixed response was given. The general consensus was that the pulsing boxes and 3D waveform both fitted the music well. The music's rhythmic characteristics were most apparent when viewing the pulsing boxes theme, but the differentiation between different frequencies not as strong as in the 3D waveform. Users found it more difficult to find a correlation in the two bouncing sphere themes, although all said that it would still look good at a live event.

The greatest threat to the quality of the visualisation is its latency value – if it becomes too large the effect of the graphical content will be lost. Users were asked to what degree the visualisation was in time with the music. Although a slight delay in the graphical output was perceived, they did not think that this was to the detriment of the application.

10 Conclusion

With development over, the deployed application may be assessed using the research material and test information produced. From this assessment suggestions for further development can be made.

10.1 Assessment of Success

Overall, the application carries out its intended purpose – it successfully takes an audio input and produces a correlating graphical output. Elements of mechanics have been included to add an element of realism. The specifications laid out in Section 4 have all been fulfilled and justified within this document.

Research and development have lead to the discoveries that:

- The implementation of a real time FFT with minimal latency is possible using Java
- Transparency issues within Java 3D still produce volatile results, despite the improvements made in version 1.3 (see Appendix C.8.5)
- A simple flocking algorithm may be achieved by knowing its position and velocity vector
- Mechanical functions and custom motion characteristics are most efficiently carried out by extending the `TransformInterpolator` class
- Audio input is most efficiently carried out using a `TargetDataLine` rather than a `Port` and `Mixer`.
- Resource collection of Java 3D scenegraph elements must be carried out manually to ensure no memory leaks occur.

10.2 Improvements and Future Work

So that the application could be viewed using a projector the implementation of a full screen mode would be essential. To complement this some form of keyboard control would be required to change the graphical content. With a more advanced set of graphical styles this keyboard control could be extended further to allow an operator to dynamically control the visual content. Instead of simple switching between scenes, graphical elements could be seamlessly integrated and manipulated. The utilisation of MIDI parameters would allow for the use of audio hardware such as the *Korg Kaoss Pad* [22] to control the graphical content of the visualisation.

These more advanced implementations would require a change of technology. C++ and OpenGL are capable of producing such dynamic content in real time. With this increase in speed the ability for more advanced particle systems could be achieved, as well as an extension of the audio processing system. By calculating the beats per minute of the audio signal an improvement on the “bouncing ball” style visualisation could be made. An advantage of C++ over Java is that multi-track audio input can be achieved using the correct hardware and drivers, allowing for an even greater abstraction of music.

11 References

1. http://www.apple.com/uk/ilife/itunes
2. http://www.microsoft.com/windows/windowsmedia/default.aspx
3. http://www.adobe.com/products/audition/main.html
4. http://archives.java.sun.com/archives/javasound-interest.html
5. http://java.sun.com/products/java-media/sound/samples/JavaSoundDemo/
6. Selman, D (2002) *Java 3D Programming*, Manning Publications Co.
7. Global Knowledge Inc (2000) *Software Engineering with Ada*, Global Knowledge Inc
8. Ambler, Scott W. (2001) *The Object Primer*, Cambridge University Press p.435
9. see 8, p.103
10. http://www.psbspeakers.com/FrequenciesOfMusic.html
11. see 10
12. Alhir, Sinan Si (1998) *UML in a Nutshell*, O'Reilly p71
13. Lindley, Criag A. (1999) *Digital Audio With Java*, p114
14. http://java.sun.com/j2se/1.4.2/docs/guide/sound/programmer_guide/contents.html
15. http://www.jsresources.org/faq_audio.html#no_ports
16. http://sound.westhost.com/project64.htm
17. see 6
18. John Hebborn & Jean Littlewood (1995) *Heinemann Modular Mathematics for London AS and A-Level: Mechanics 2*, Heinemann Publishers Ltd.
19. http://java.sun.com/docs/books/tutorial/essential/system/garbage.html
20. http://java.sun.com/docs/books/tutorial/java/data/garbagecollection.html
21. http://www.warprecords.co.uk
22. http://www.korg.com
23. http://www.pressurewave.com/~stoltz/Fft.html
24. http://www.nauticom.net/www/jdtaft/JavaFFT.htm
25. see 13
26. http://java.sun.com/products/java-media/3D/forDevelopers/J3D_1_3_API/j3dapi/index.html
27. java.sun.com/products/java-media/3D/java3d-features.html
28. see 6

12 Acknowledgments

Project supervisor:	Dr P. Newbury
Proofreader:	Dr P. Newbury

Appendix A: System Design

A.1 Execution Cycle Flowcharts

The following flowcharts illustrate the more detailed design concepts behind the workings of the application. Although the diagrams contain Java specific details, they could be used as the basis for an implementation in another language.

All the 3D graphics generator classes set up the scenegraph elements in much the same way (Figure A.1); the scenegraph elements are created and then linked together. The first two parts which are concerned with the view elements are the same for all of the graphics models, except for minor `viewPlatform` positioning. The content side of the scenegraph varies for each of the models, but the same process is followed.

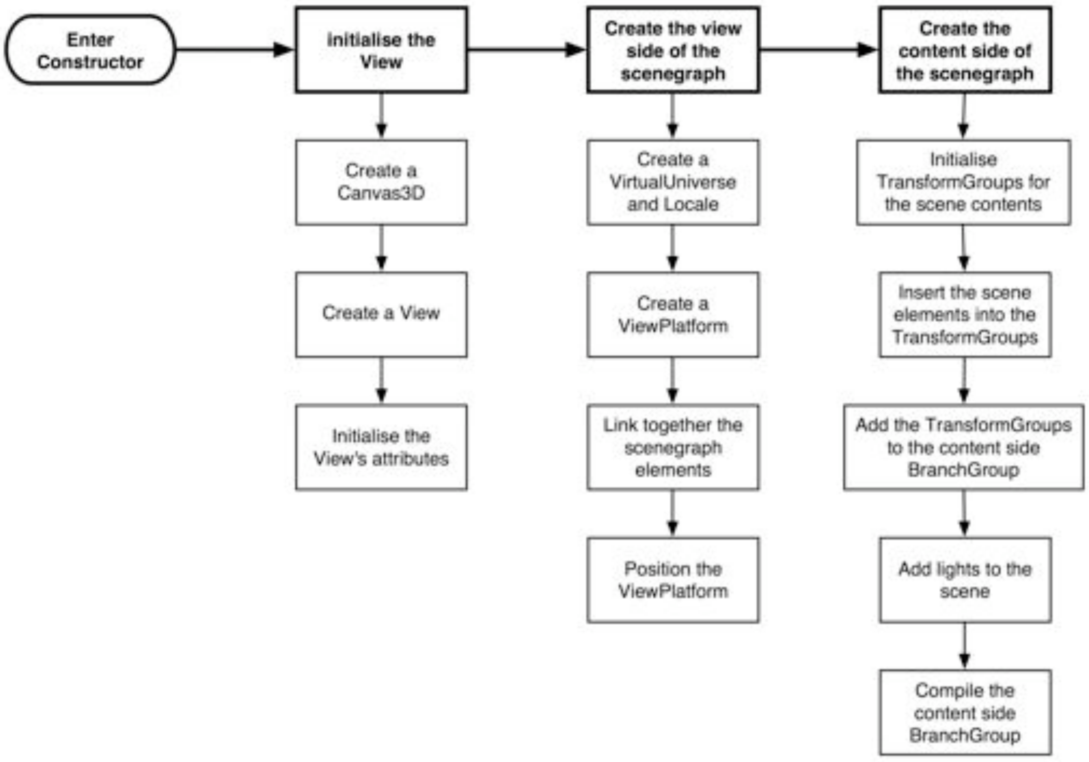

Enter Constructor
initialise the View
Create the view side of the scenegraph
Create the content side of the scenegraph
Create a Canvas3D
Create a View
Initialise the View's attributes
Create a VirtualUniverse and Locale
Create a ViewPlatform
Link together the scenegraph elements
Position the ViewPlatform
Initialise TransformGroups for the scene contents
Insert the scene elements into the TransformGroups
Add the TransformGroups to the content side BranchGroup
Add lights to the scene
Compile the content side BranchGroup

Figure A.1 Flowchart for initialising each of the 3D graphics generators

The `AudioController` can be split into two main sections – initialisation and execution loop. The while loop contains no exit route on the flowchart as the `MainFrame` deals with closing the application.

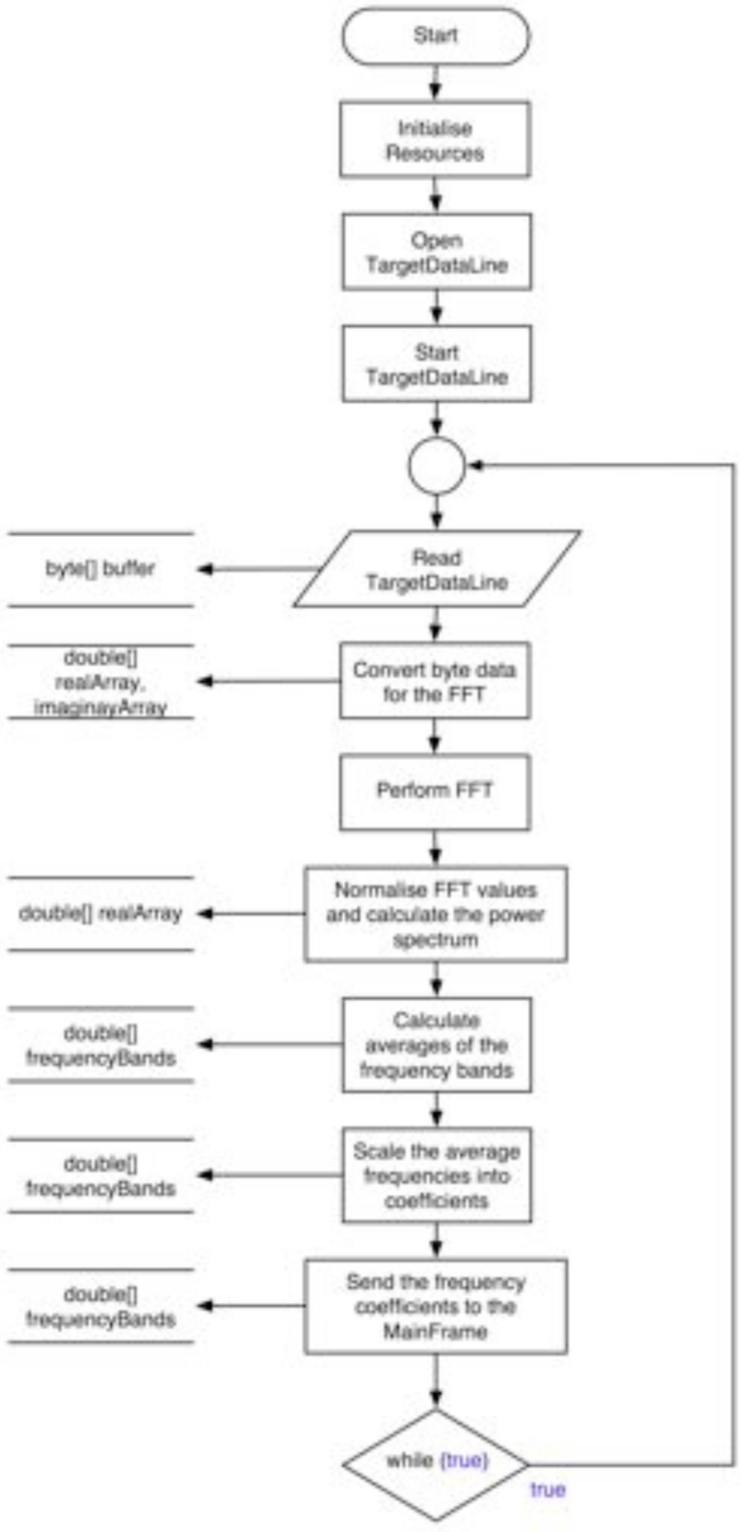

Start
Initialise Resources
Open TargetDataLine
Start TargetDataLine
Read TargetDataLine
byte[] buffer
Convert byte data for the FFT
double[] realArray, imaginayArray
Perform FFT
Normalise FFT values and calculate the power spectrum
double[] realArray
Calculate averages of the frequency bands
double[] frequencyBands
Scale the average frequencies into coefficients
double[] frequencyBands
Send the frequency coefficients to the MainFrame
double[] frequencyBands
while (true)
true

Figure A.2 Flowchart for the AudioController class

The `MainFrame` constructor (Figure A.3) carries out the initialisation of the GUI, meaning that it appears as soon as the application is run. The update method (Figure A.3) passes on the frequency data according to the option set when the button was pressed. Should no graphical content be currently displayed, i.e. when the application is first run, then the method returns without doing anything.

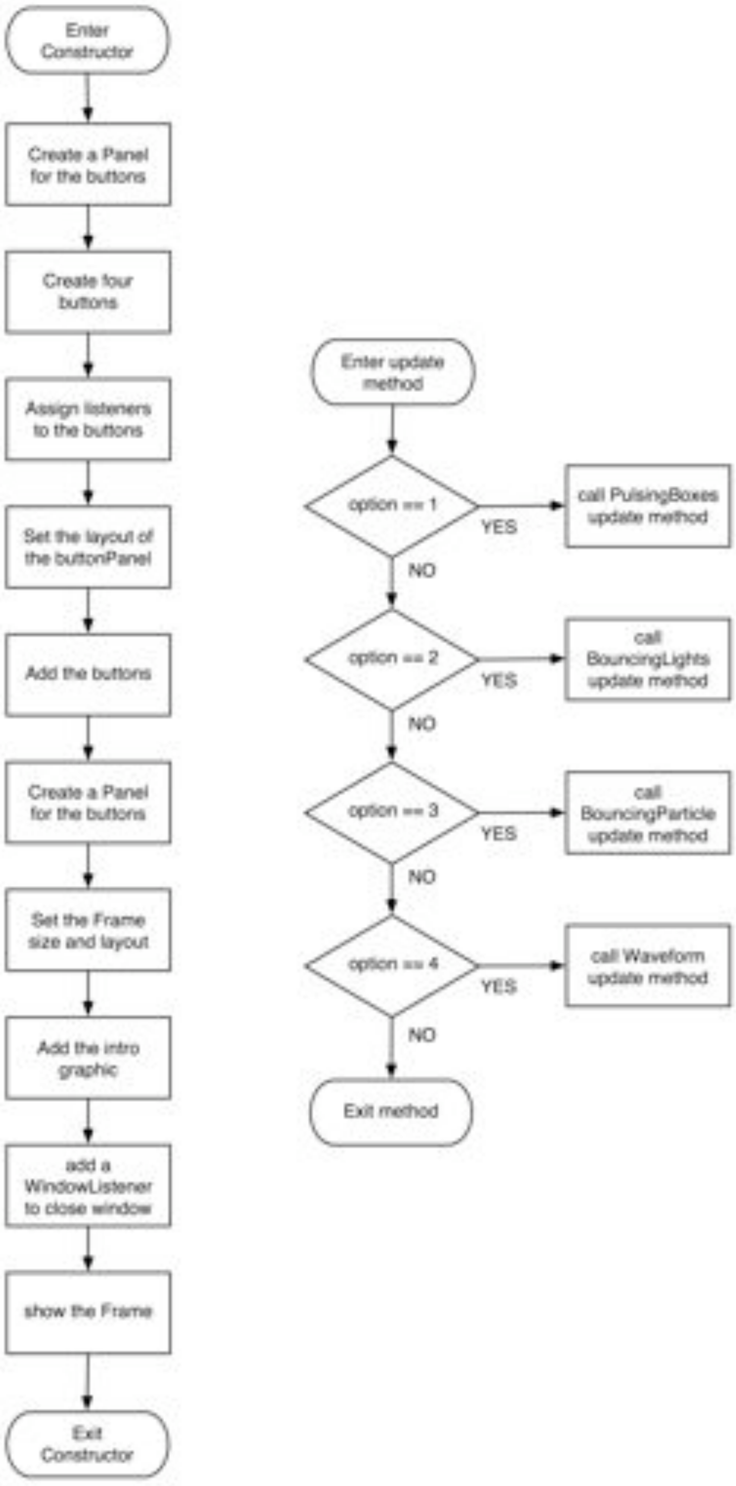

Enter Constructor
Create a Panel for the buttons
Create four buttons
Assign listeners to the buttons
Set the layout of the buttonPanel
Add the buttons
Create a Panel for the buttons
Set the Frame size and layout
Add the intro graphic
add a WindowListener to close window
show the Frame
Exit Constructor
Enter update method
option == 1
YES
call PulsingBoxes update method
NO
option == 2
YES
call BouncingLights update method
NO
option == 3
YES
call BouncingParticle update method
NO
option == 4
YES
call Waveform update method
NO
Exit method

Figure A.3 Flowchart for the MainFrame constructor and update method

By placing the boxes in an array of `TransformGroups` the update method can be optimised to a single for loop (Figure A.4). Three separate checks are made with the validity of each depending on the previous one, thus avoiding the unnecessary execution of code. All checks are to ensure that visual consistency is maintained, preventing the flow of the object's motion from being disturbed unless a progressive action is to be taken.

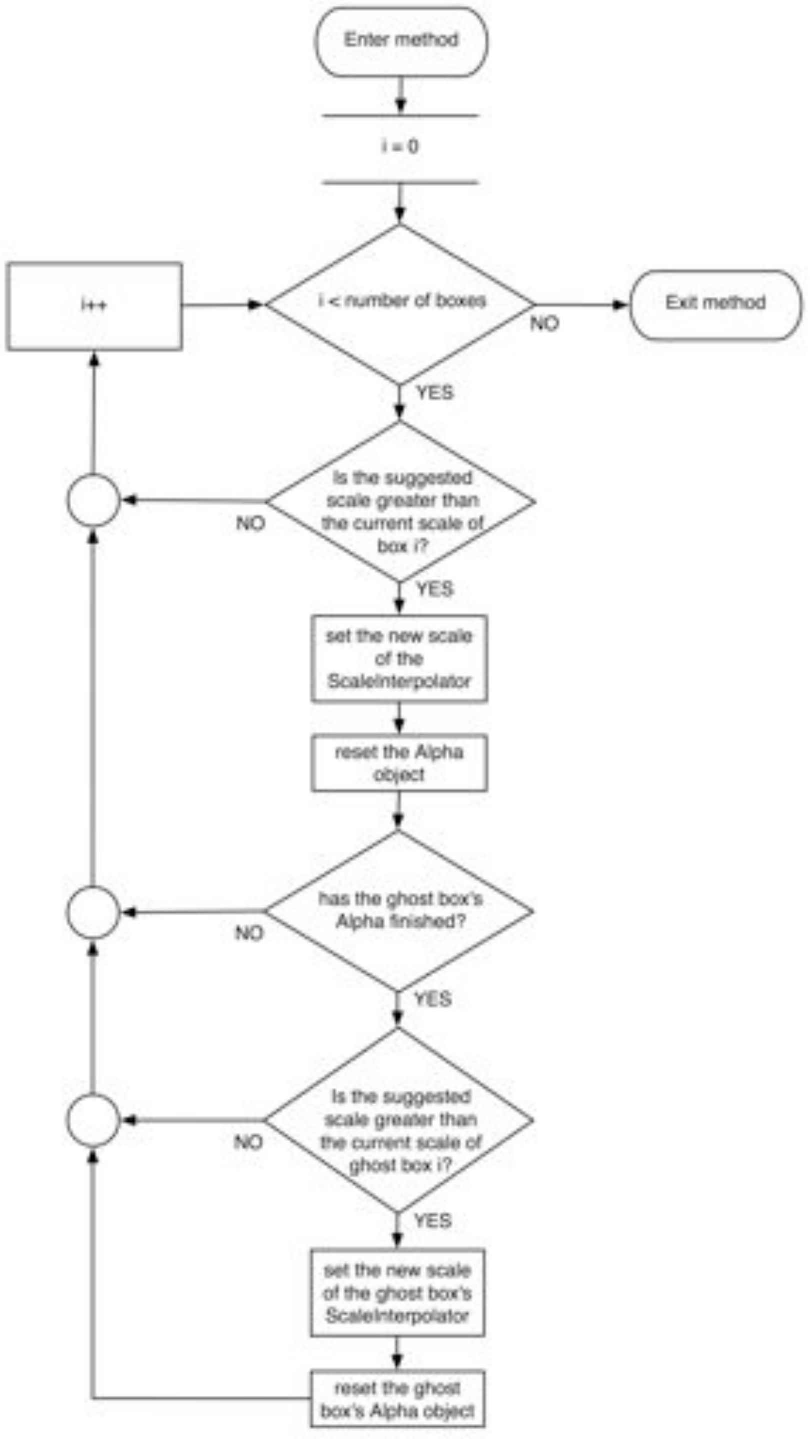

Enter method
i = 0
i++
i < number of boxes
NO
Exit method
YES
Is the suggested scale greater than the current scale of box i?
NO
YES
set the new scale of the ScaleInterpolator
reset the Alpha object
has the ghost box's Alpha finished?
NO
YES
Is the suggested scale greater than the current scale of ghost box i?
NO
YES
set the new scale of the ghost box's ScaleInterpolator
reset the ghost box's Alpha object

Figure A.4 Flowchart for the PulsingBoxes update method

All of the principles used in the `PulsingBoxes` update method are used here; the balls are placed in a `TransformGroup` array which is traversed using a for loop. In order to speed up execution a check is first made to see if the ball is falling as no updating is allowed unless this condition holds true.

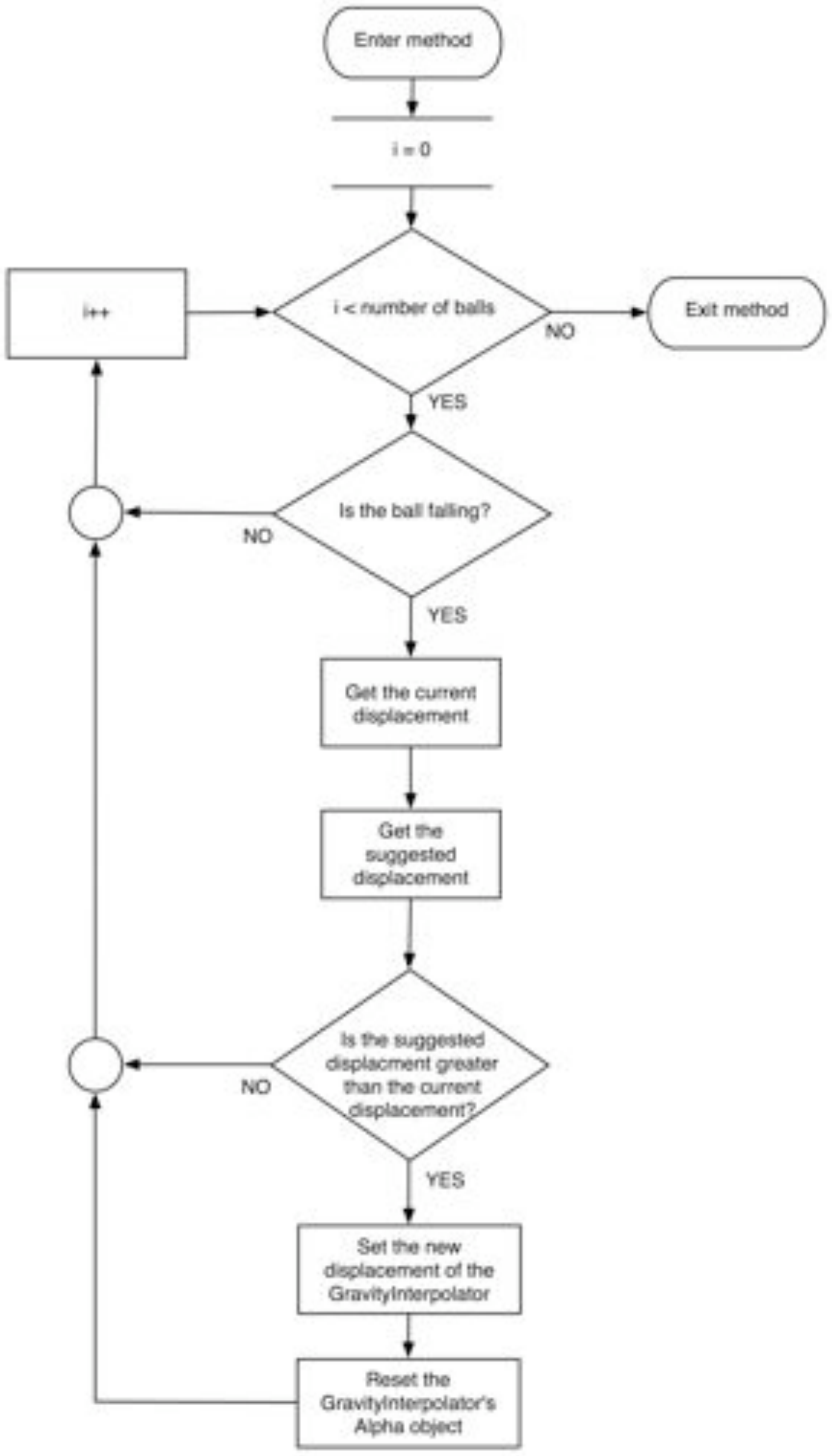

Enter method
i = 0
i++
i < number of balls
NO
Exit method
YES
Is the ball falling?
NO
YES
Get the current displacement
Get the suggested displacement
Is the suggested displacment greater than the current displacement?
NO
YES
Set the new displacement of the GravityInterpolator
Reset the GravityInterpolator's Alpha object

Figure A.5 Flowchart for a bouncing balls update method

As a set of six VU elements are applied to each ball the VU elements could be controlled by placing them in a 2-dimensional array; one dimension determines the ball and the other specifies the element. Figure A.6 illustrates how this may be achieved. Within a single for loop which specifies the assigned ball there are two for loops; one for the VU elements at or below the ball level and another for the VU elements above. By casting the current height of the ball it is possible to get the integer value of the VU element's index. The use of dynamic for loops enables more streamlined execution.

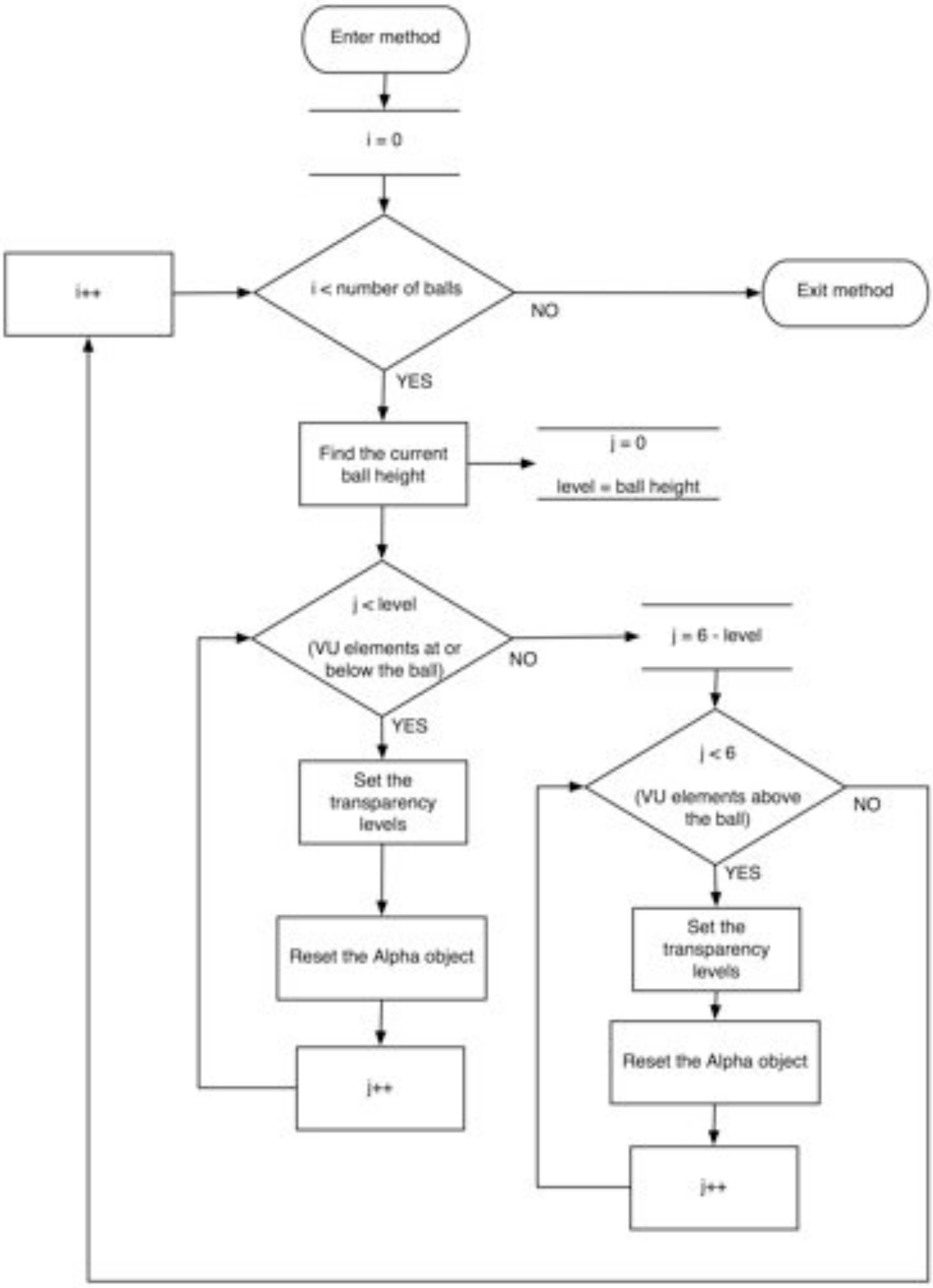


Figure A.6 Flowchart for the VU elements update method

The waveform is made up of six `Shape3D` objects, each governed by a `QuadArray` that specifies its geometry characteristics. By updating the vectors that specify the positional characteristics of the `QuadArray` the waveform's shape can be changed.

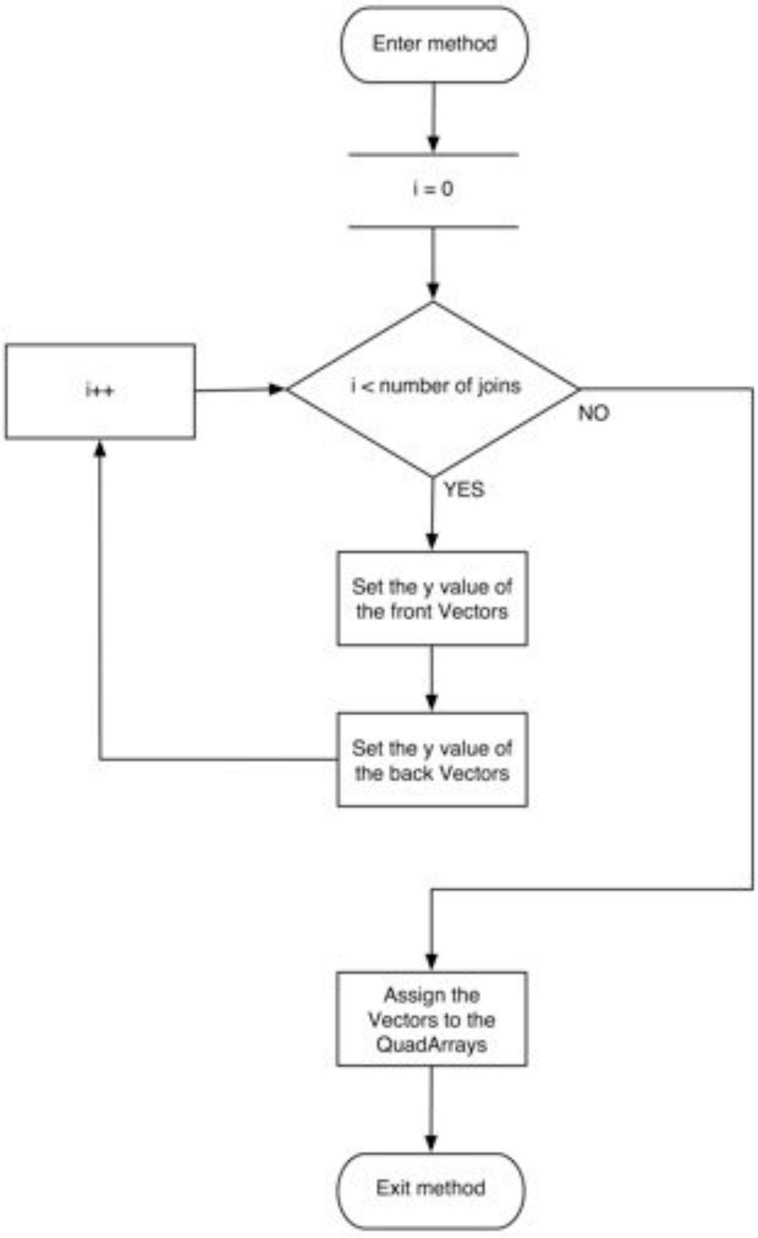

Enter method
i = 0
i++
i < number of joins
NO
YES
Set the y value of the front Vectors
Set the y value of the back Vectors
Assign the Vectors to the QuadArrays
Exit method

Figure A.7 Flowchart for the Waveform update method

A.2 Design Class Diagrams

As an augmentation to the design class diagram show in Figure 7.4, this section contains details concerning the individual classes' attributes and methods which could not be contained within a single diagram.

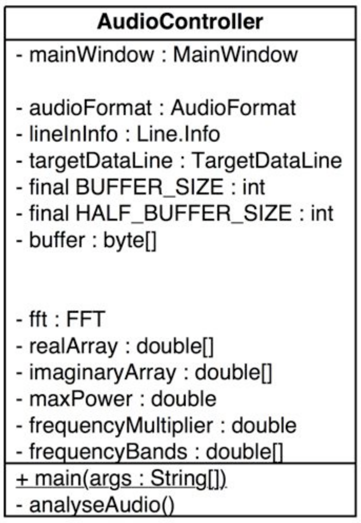

AudioController
- mainWindow : MainWindow
- audioFormat : AudioFormat
- lineInInfo : Line.Info
- targetDataLine : TargetDataLine
- final BUFFER_SIZE : int
- final HALF_BUFFER_SIZE : int
- buffer : byte[]
- fft : FFT
- realArray : double[]
- imaginaryArray : double[]
- maxPower : double
- frequencyMultiplier : double
- frequencyBands : double[]
+ main(args : String[])
- analyseAudio()

Figure A.8 AudioController class

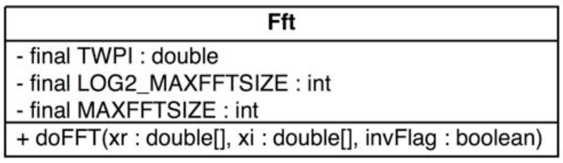

Fft
- final TWPI : double
- final LOG2_MAXFFTSIZE : int
- final MAXFFTSIZE : int
+ doFFT(xr : double[], xi : double[], invFlag : boolean)

Figure A.9 FFT class

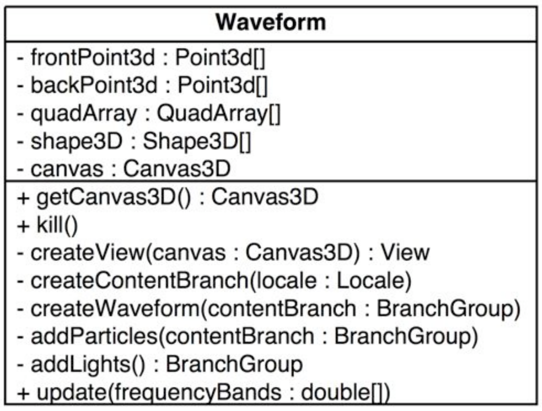
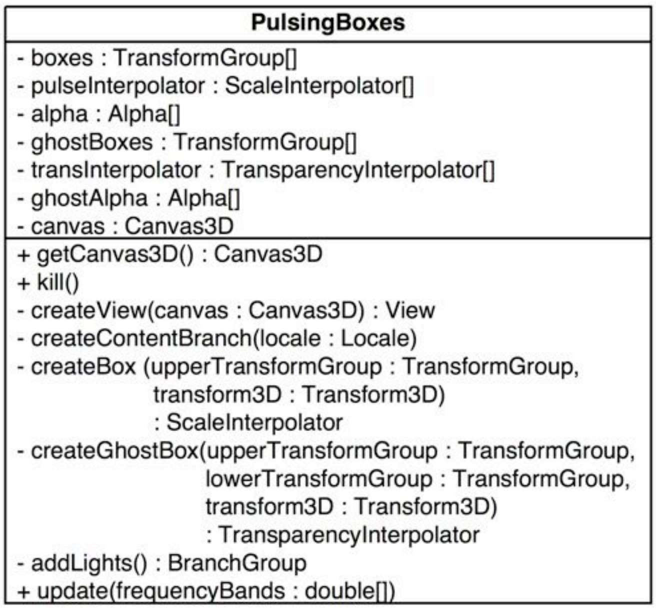
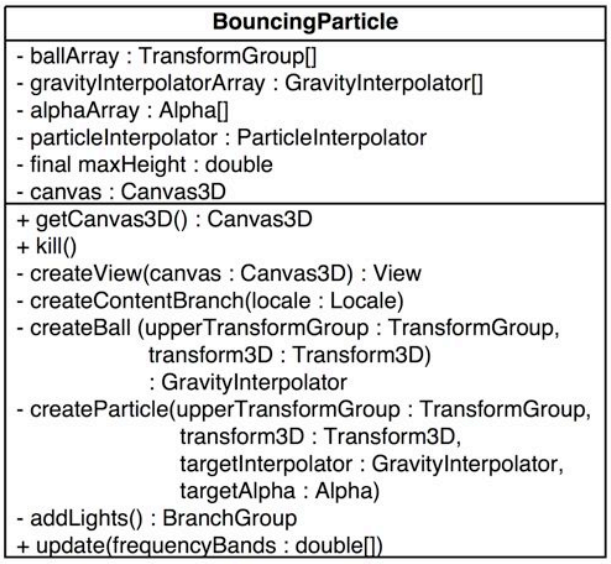

Waveform
- frontPoint3d : Point3d[]
- backPoint3d : Point3d[]
- quadArray : QuadArray[]
- shape3D : Shape3D[]
- canvas : Canvas3D
+ getCanvas3D() : Canvas3D
+ kill()
- createView(canvas : Canvas3D) : View
- createContentBranch(locale : Locale)
- createWaveform(contentBranch : BranchGroup)
- addParticles(contentBranch : BranchGroup)
- addLights() : BranchGroup
+ update(frequencyBands : double[])

Figure A.10 Waveform class



```
PulsingBoxes
- boxes : TransformGroup[]
- pulseInterpolator : ScaleInterpolator[]
- alpha : Alpha[]
- ghostBoxes : TransformGroup[]
- transInterpolator : TransparencyInterpolator[]
- ghostAlpha : Alpha[]
- canvas : Canvas3D
+ getCanvas3D() : Canvas3D
+ kill()
- createView(canvas : Canvas3D) : View
- createContentBranch(locale : Locale)
- createBox (upperTransformGroup : TransformGroup,
             transform3D : Transform3D)
             : ScaleInterpolator
- createGhostBox(upperTransformGroup : TransformGroup,
                 lowerTransformGroup : TransformGroup,
                 transform3D : Transform3D)
                 : TransparencyInterpolator
- addLights() : BranchGroup
+ update(frequencyBands : double[])
```

Figure A.11 PulsingBoxes class



```
BouncingParticle
- ballArray : TransformGroup[]
- gravityInterpolatorArray : GravityInterpolator[]
- alphaArray : Alpha[]
- particleInterpolator : ParticleInterpolator
- final maxHeight : double
- canvas : Canvas3D
+ getCanvas3D() : Canvas3D
+ kill()
- createView(canvas : Canvas3D) : View
- createContentBranch(locale : Locale)
- createBall (upperTransformGroup : TransformGroup,
              transform3D : Transform3D)
              : GravityInterpolator
- createParticle(upperTransformGroup : TransformGroup,
                 transform3D : Transform3D,
                 targetInterpolator : GravityInterpolator,
                 targetAlpha : Alpha)
- addLights() : BranchGroup
+ update(frequencyBands : double[])
```

Figure A.12 BouncingParticle class

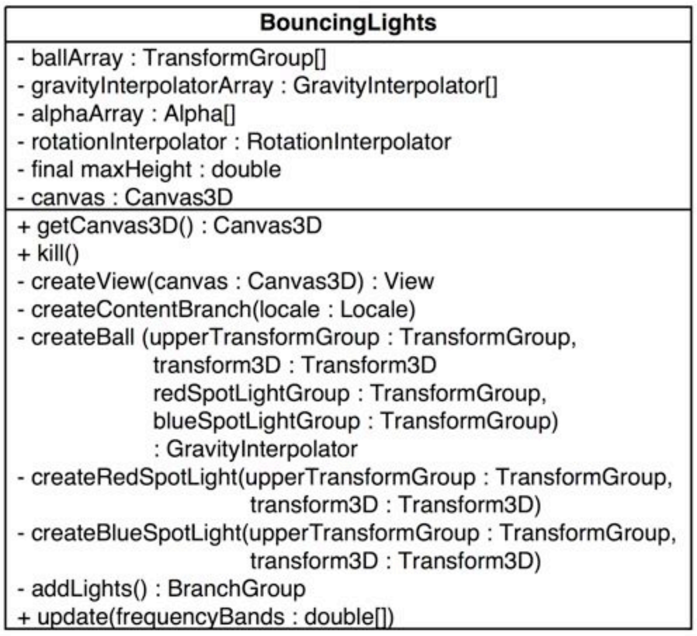

BouncingLights
- ballArray : TransformGroup[]
- gravityInterpolatorArray : GravityInterpolator[]
- alphaArray : Alpha[]
- rotationInterpolator : RotationInterpolator
- final maxHeight : double
- canvas : Canvas3D
+ getCanvas3D() : Canvas3D
+ kill()
- createView(canvas : Canvas3D) : View
- createContentBranch(locale : Locale)
- createBall (upperTransformGroup : TransformGroup, transform3D : Transform3D redSpotLightGroup : TransformGroup, blueSpotLightGroup : TransformGroup) : GravityInterpolator
- createRedSpotLight(upperTransformGroup : TransformGroup, transform3D : Transform3D)
- createBlueSpotLight(upperTransformGroup : TransformGroup, transform3D : Transform3D)
- addLights() : BranchGroup
+ update(frequencyBands : double[])

Figure A.13 BouncingLights class

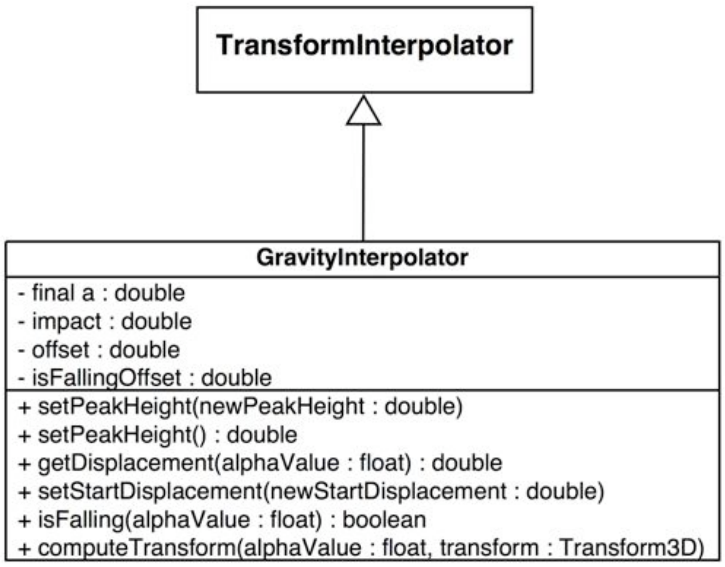

TransformInterpolator
GravityInterpolator
- final a : double
- impact : double
- offset : double
- isFallingOffset : double
+ setPeakHeight(newPeakHeight : double)
+ setPeakHeight() : double
+ getDisplacement(alphaValue : float) : double
+ setStartDisplacement(newStartDisplacement : double)
+ isFalling(alphaValue : float) : boolean
+ computeTransform(alphaValue : float, transform : Transform3D)

Figure A.14 GravityInterpolator class

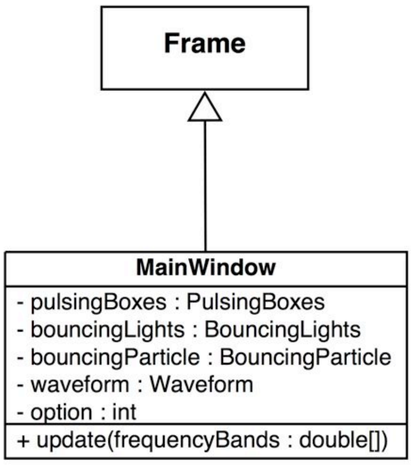

Frame
MainWindow
- pulsingBoxes : PulsingBoxes
- bouncingLights : BouncingLights
- bouncingParticle : BouncingParticle
- waveform : Waveform
- option : int
+ update(frequencyBands : double[])

Figure A.15 MainWindow class

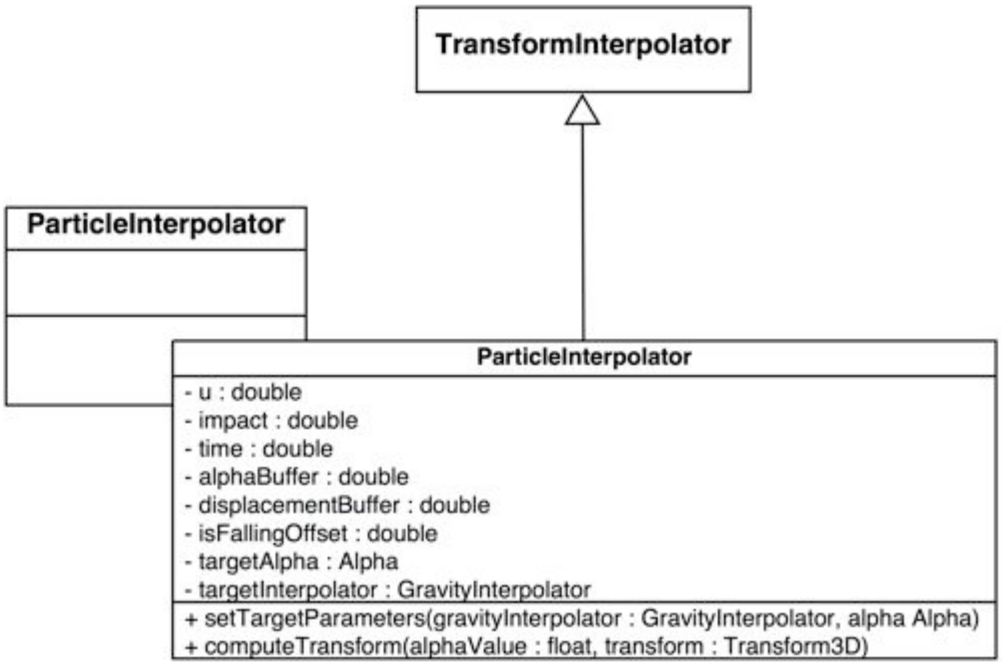

TransformInterpolator
ParticleInterpolator
ParticleInterpolator
- u : double
- impact : double
- time : double
- alphaBuffer : double
- displacementBuffer : double
- isFallingOffset : double
- targetAlpha : Alpha
- targetInterpolator : GravityInterpolator
+ setTargetParameters(gravityInterpolator : GravityInterpolator, alpha Alpha)
+ computeTransform(alphaValue : float, transform : Transform3D)

Figure A.16 ParticleInterpolator class

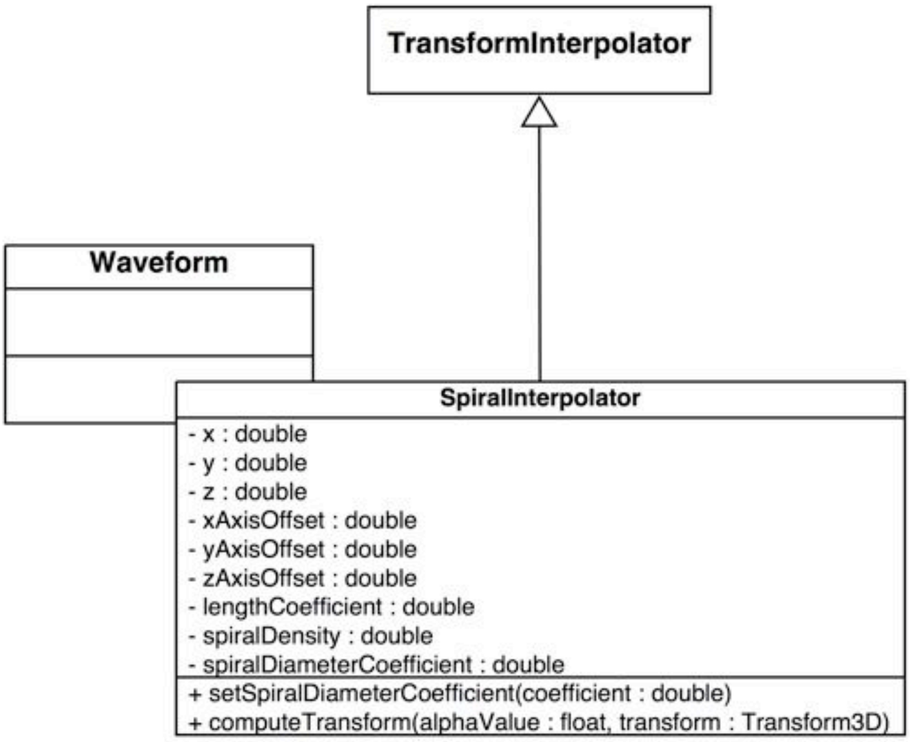

TransformInterpolator
Waveform
SpiralInterpolator
- x : double
- y : double
- z : double
- xAxisOffset : double
- yAxisOffset : double
- zAxisOffset : double
- lengthCoefficient : double
- spiralDensity : double
- spiralDiameterCoefficient : double
+ setSpiralDiameterCoefficient(coefficient : double)
+ computeTransform(alphaValue : float, transform : Transform3D)

Figure A.17 SpiralInterpolator class

A.3 Scenegraph Designs

The following diagrams show the structure of elements within the four visualisation themes deployed in the final application build. In Figure A.18 the red `TransformGroup` elements are all the same in structure.

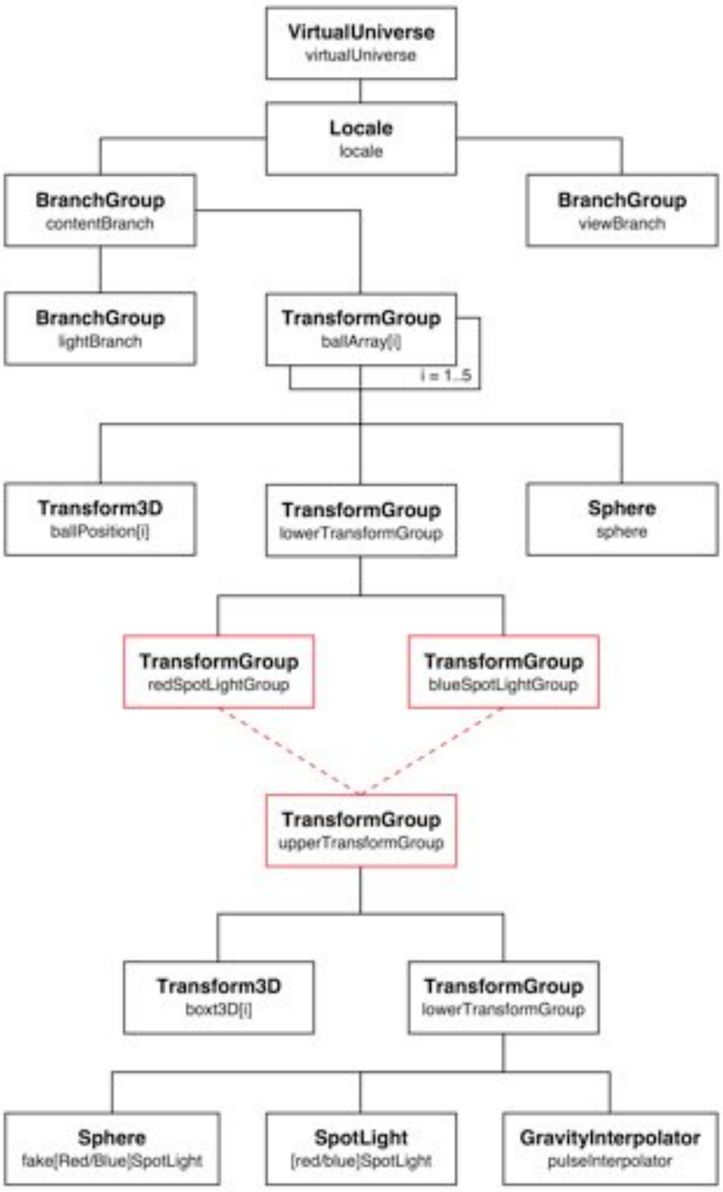


Figure A.18 Scenegraph for the BouncingLights class

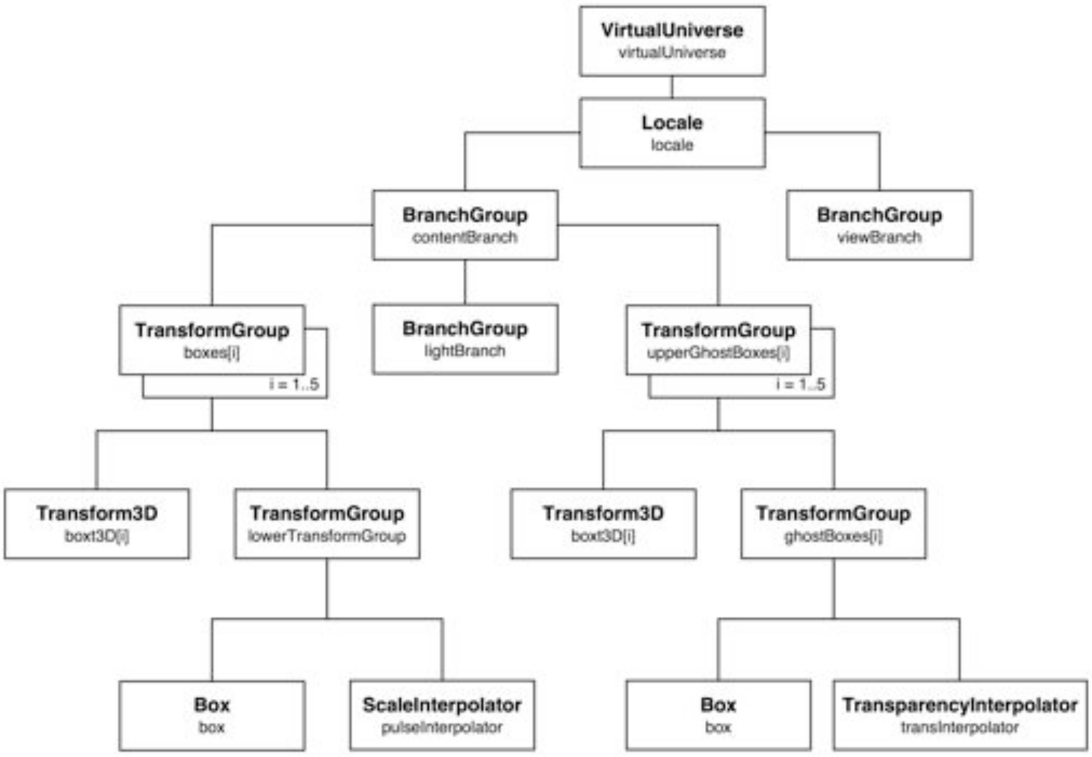


Figure A.19 Scenegraph for the PulsingBoxes class

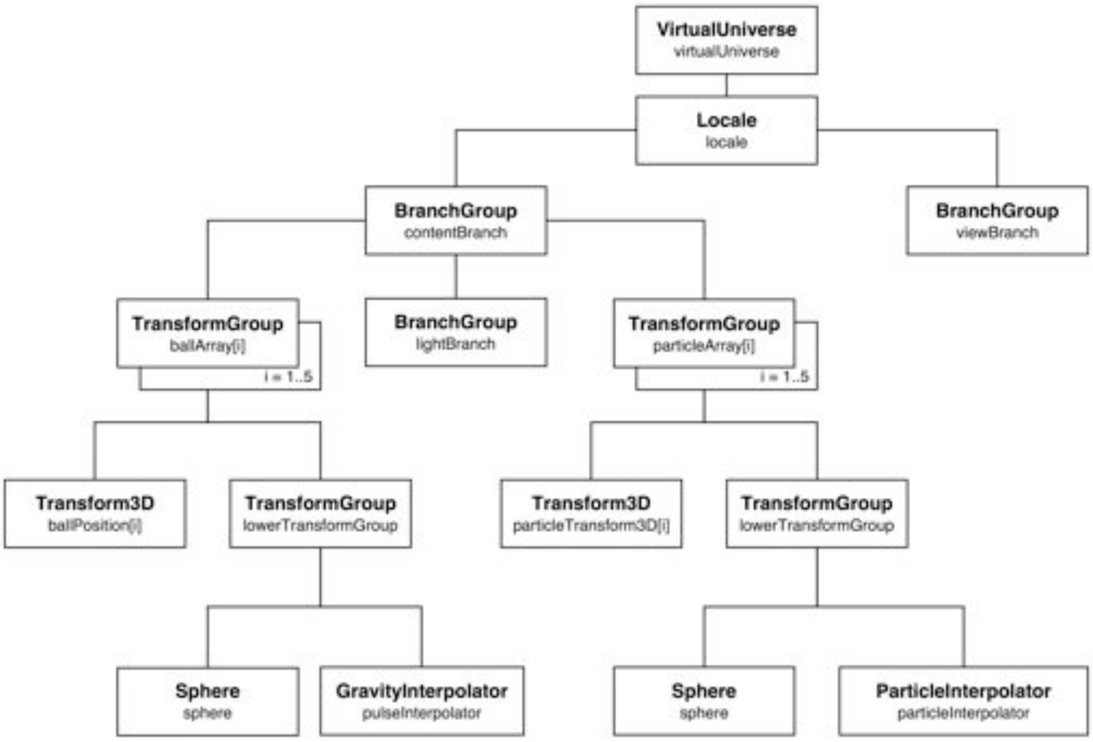


Figure A.20 Scenegraph for the BouncingParticle class

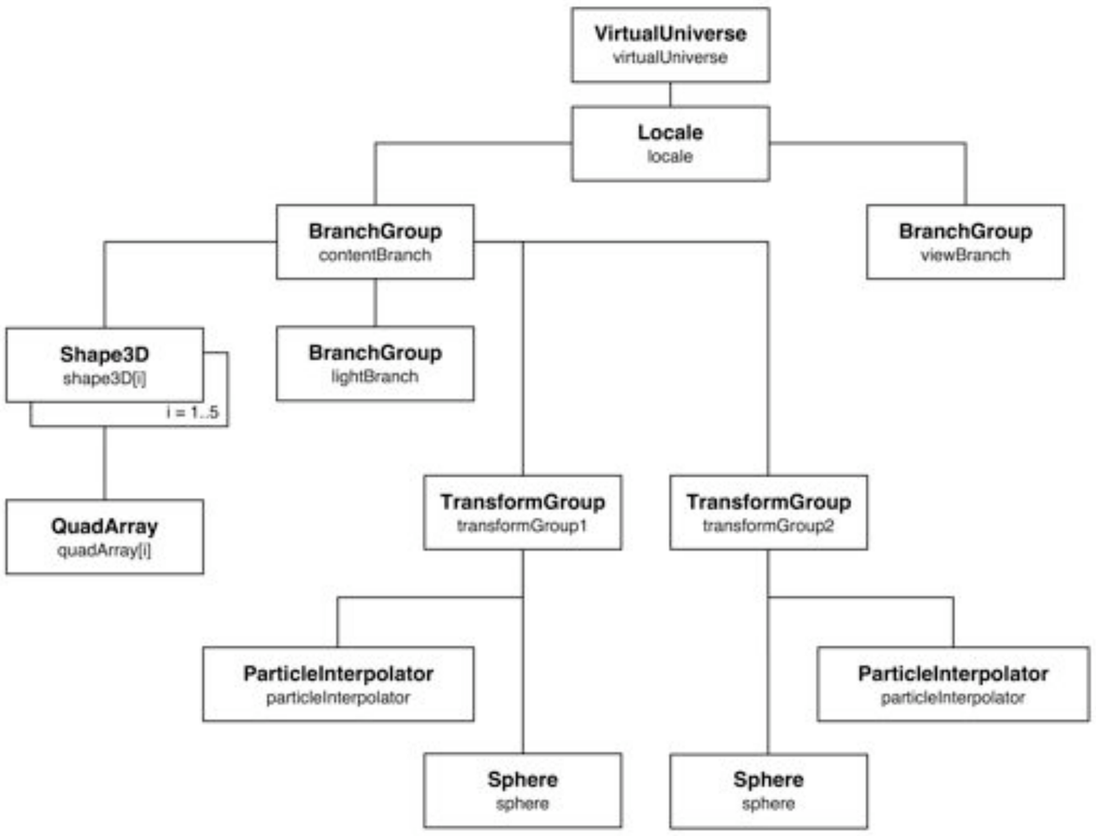


Figure A.21 Scenegraph for the Waveform class

Appendix B: Simulation of Mechanics

B.1 Flowchart for a Flocking Particle

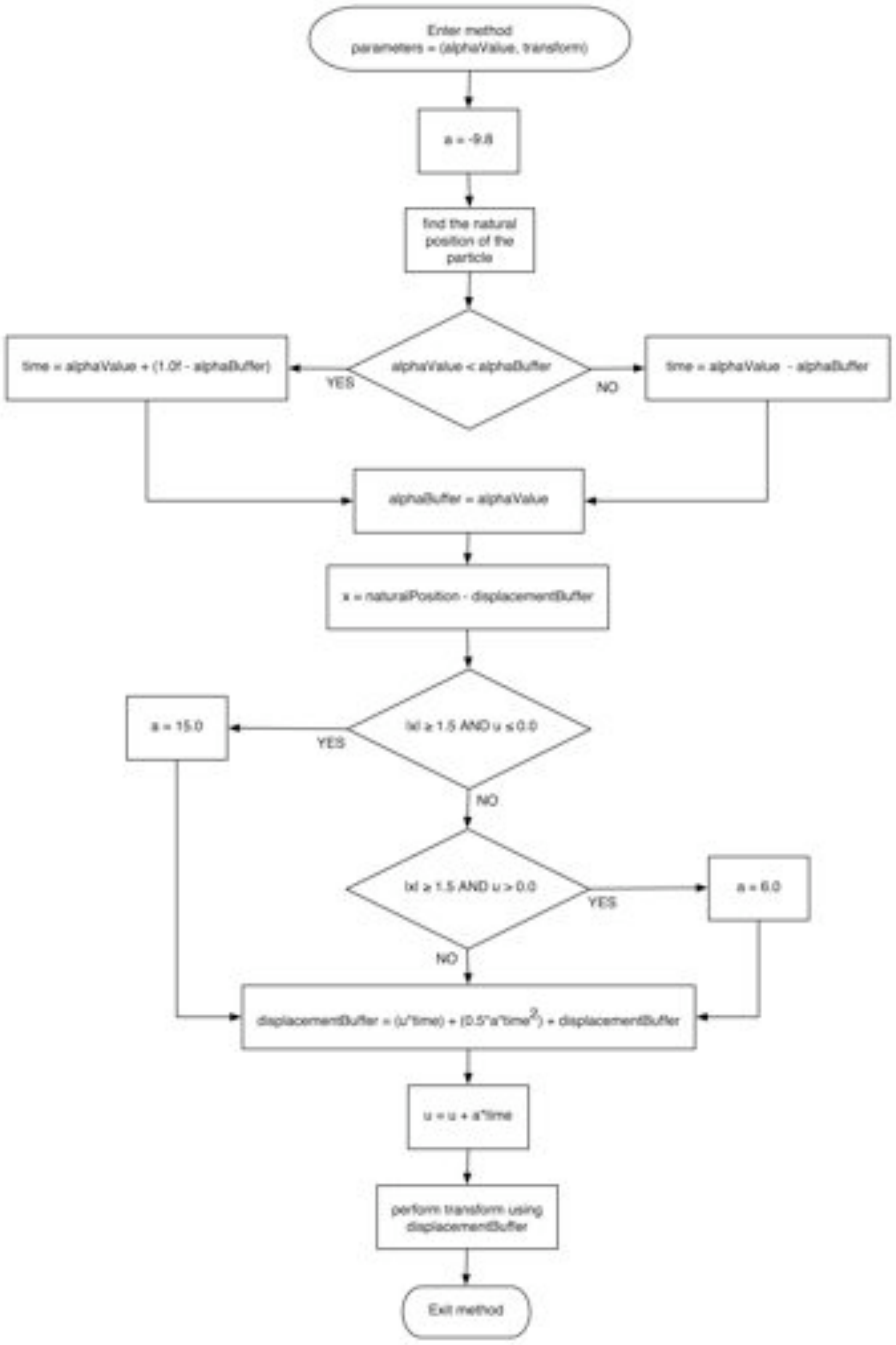

Enter method parameters = (alphaValue, transform)
a = -9.8
find the natural position of the particle
alphaValue < alphaBuffer
YES
time = alphaValue + (1.0f - alphaBuffer)
NO
time = alphaValue - alphaBuffer
alphaBuffer = alphaValue
x = naturalPosition - displacementBuffer
|x| ≥ 1.5 AND u ≤ 0.0
YES
a = 15.0
NO
|x| ≥ 1.5 AND u > 0.0
YES
a = 6.0
NO
displacementBuffer = (u*time) + (0.5*a*time^2) + displacementBuffer
u = u + a*time
perform transform using displacementBuffer
Exit method

Figure B.1 Flowchart for the ParticleInterpolator computeTransform method

B.2 Calculation for Alpha Offset Value

By taking Equation 8.4 the working out for Equation 8.5 is shown.

Equation 8.4,

$$\boldsymbol{s = ut + \frac{1}{2}at^2}$$

The offset values k are added,

$$s = u(t + k) + \frac{1}{2}a(t + k)^2$$

Making $a = -4.9$ and $u = -a$,

$$s = u(t + k)^2 - a(t + k)$$

$$s = u(t^2 + 2tk + k^2) - at - ak$$

$$s = at^2 + 2atk + ak^2 - at - ak$$

Taking the offset values to one side,

$$ak^2 + 2atk - ak = s - at$$

Arrange into a quadratic equation,

$$ak^2 + 2(at - a)k\ (at - s) = 0$$

Using the quadratic formula $x = \dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ where

$$\begin{aligned} &a = a \\ &b = (2at - a) c = (at - s) \end{aligned}$$

By order of substitution,

$$k = \frac{-(2at - a) \pm \sqrt{(2ax - a)^2 - 4a(at - s)}}{2a}$$

$$k = \frac{a - 2at \pm \sqrt{4a^2t^2 - 4a^2t + a^2 - 4a^2t + 4as}}{2a}$$

as $t = 0$ (see Figure 7.9) we eliminate it,

$$k = \frac{a \pm \sqrt{a^2 + 4as}}{2a} \qquad \text{(B.1)}$$

It must be found whether the ± should be a plus or minus. The peak height reached when there is no offset is at $t = 0.5$. Therefore, to find the peak height,

$$\boldsymbol{s = ut + \frac{1}{2}at^2}$$

$$s = 4.9*0.5 - \frac{1}{2}*9.8*0.5^2$$

$$s = 1.225$$

If we substitute the value $s = 1.225$ and $a = -4.9$ into Equation B.1 the offset value should be 0.5. Replacing ± with a plus,

$$k = \frac{-4.9 \pm \sqrt{-4.9^2 + 4*(-4.9)*1.225}}{-9.8}$$

$$k = \frac{-4.9 \pm \sqrt{24.01 - 24.01}}{-9.8}$$

$$k = 0.5$$

Therefore, the equation must contain a plus:

$$k = \frac{a + \sqrt{a^2 + 4as}}{2a} \qquad \text{(B.2)}$$

Appendix C: Iteration Breakdown

This section contains the breakdown of every iteration of code, showing the development of the application. Every iteration begins with a set of objectives to be achieved. The development of these goals is discussed and the results achieved are shown.

C.1 Iteration 1

C.1.1 Iteration 1a

- Attempt data input through a `Port` object
- Investigate available lines
- Attempt data handling using a `ByteArrayInputStream` object

This first iteration attempts the conventional method of audio input by requesting a `Port` object from the `Mixer` as specified in Figure 8.1. However, as discussed in Section 8.1.2, implementation of the Port class is not due until version 1.5.0. This was found to be true on execution of the following code,

```
lineInInfo = new Line.Info(Port.class);
TargetDataLine targetDataLine = (TargetDataLine)mixer.getLine(lineInInfo);
```

where an `IllegalArgumentException` is thrown,

```
Line unsupported: interface Port (in com.sun.media.sound.SimpleInputDevice)
```

C.1.2 Iteration 1b

- Implement the class as an extension of a Thread
- Initialisation of the audio system will be carried out in a constructor
- Audio input will be attempted without using a Mixer object

The iteration was unsuccessful in providing an audio input. The use of both the Thread and `ByteArrayInputStream` was cumbersome, so its development was put on hold.

C.1.3 Iteration 1c

- Use a while loop to input data
- Use the `TargetDataLine.read`() method to get the bytes
- Pass the audio data captured to a FFT routine which was adapted for Java by Ben Stoltz [23]

A simpler solution than Iteration 1b was sought after; a `TargetDataLine` was created and its `read` method placed in an indefinite while loop, providing constant handling of the audio input signal data. By using a `SourceDataLine` object the audio data could be written straight back out again and heard from the computer's audio output. This confirmed that the audio data was being successfully inputted into the system, the requirement of **BASELINE POINT 1**.

However, the FFT routine was not producing any intelligible results. The output was in the form of a primitive textual graph that was printed to the terminal. Although this could be disabled and the raw data handled manually, the poor documentation of code made modification difficult.

C.2 Iteration 2

C.2.1 Iteration 2a

- Implementation using an FFT class by Jeffrey D. Taft, PhD [24]

In order to test the results of the FFT a set of results was printed to screen to inspect the values being produced. The output is a stream of meaningless data,

```
Element 0 = [D@7b6643
Element 1 = [D@76e8a7
Element 2 = [D@a45536
Element 3 = [D@d66426
Element 4 = [D@490eb5
Element 5 = [D@64b09c
Element 6 = [D@86f247
Element 7 = [D@8c4a77
Element 8 = [D@6d0040
Element 9 = [D@2b9406
```

Such a serious problem with the data meant that no more time was spent on this FFT routine as other alternatives were available.

C.3 Iteration 3

C.3.1 Iteration 3a

- Implement an FFT routine created by Jef Pokanzer and adapted for Java by Craig A. Lindley [25]
- Utilise equations (see Section 8.1.3) to convert the output of the FFT into decibels

By implementing the equations to convert the FFT data into decibels, interpreting the results is made easier. A 1000 Hz sine wave is input into the system and the frequency content results are printed to screen,

```
Frequency = 947   dB = -72
Frequency = 958   dB = -60
Frequency = 968   dB = -59
Frequency = 979   dB = -98
Frequency = 990   dB = -84
Frequency = 1001   dB = -55
Frequency = 1012   dB = -92
Frequency = 1022   dB = -69
Frequency = 1033   dB = -58
Frequency = 1044   dB = -79
```

The decibel values should peak at 1000Hz, but instead the same random pattern as shown above occurs across the entire frequency range. This is not what is expected, but as the code is from a reliable source its usage is investigated further.

C.3.2 Iteration 3b

- Investigate the raw data being read by the `TargetDataLine`

In order to analyse the audio data being read from the audio input the byte values are printed to screen. A 1000 Hz sine wave is used again so that the sample pattern may be easily recognised. The program was allowed to run for 30 seconds to ensure that any starting glitches are avoided. This is a sample of the output,

```
Sample 0 = -91
Sample 1 = -28
Sample 2 = 44
Sample 3 = -35
Sample 4 = 105
Sample 5 = -42
Sample 6 = 123
Sample 7 = -48
Sample 8 = -125
Sample 9 = -53
Sample 10 = -101
Sample 11 = -57
Sample 12 = -36
Sample 13 = -60
Sample 14 = 74
Sample 15 = -61
Sample 16 = -10
Sample 17 = -62
Sample 18 = -33
Sample 19 = -61
Sample 20 = -8
```

There is no discernible pattern in this extract, or the rest of the output. The fact that printing to screen is so slow may affect the resulting values being output. A faster way of extracting the data must be found to ensure that the speed issue is ruled out.

C.3.3 Iteration 3c

- Take a "snapshot" of the audio data and store in a `Vector`
- After the audio inputting stops the `Vector` is written to a text file using the `FileWriter` class
- Channel the audio back out through a `SourceDataLine`

A high speed method of storing a snapshot of audio data was required. Writing a buffer-full of data to a `Vector` keeps the data handling internal, ensuring that no speed issues are incurred. The following code deals with capturing the audio data in the `Vector`,

```
int index = 0;
while(index < 50) {
        bytesRead = targetDataLine.read(buffer, 0, BUFFER_SIZE);

        if (index > 43) {
                store.add(buffer);
        }
        sourceDataLine.write(buffer, 0, bytesRead);
        index++;
}
```

The `Vector`'s add method is used to copy the entire buffer. Notice that the audio is also channelled back out through a `SourceDataLine`, just to make sure that the audio is being inputted properly. As in Iteration 1c the audio input can be heard being outputted from the system. The `Vector` is then written to file,

```
FileWriter fileWriter = new FileWriter("output.txt");
String output = new String();

buffer = (byte[])store.elementAt(0);

for (int j = 0; j < buffer.length; j++) {
        output = String.valueOf((byte)buffer[j] + "\n");
        fileWriter.write(output);
}
```

The data is written as a single columned series of decibel values, such that the data could be copied and pasted into Excel. The values were plotted onto a graph, shown in Figure C.1.

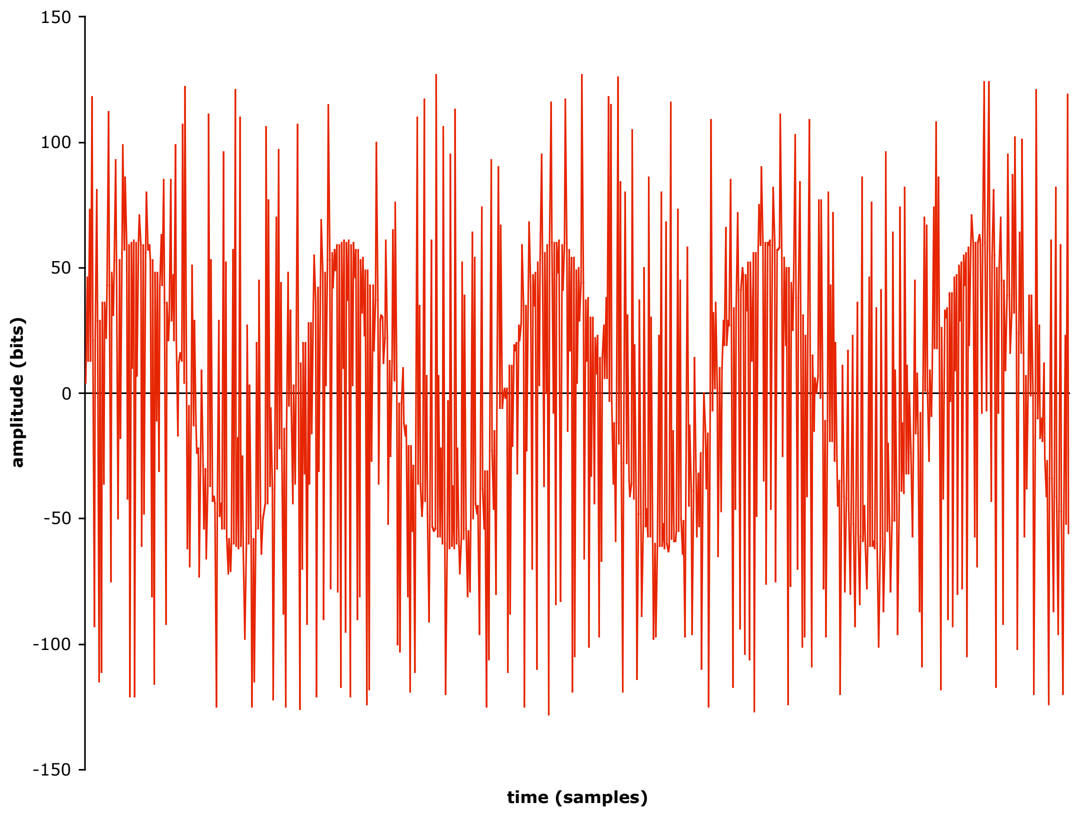


Figure C.1 Audio waveform extracted from 1000Hz input signal

What can be seen from the plotted waveform is a distorted pattern of a sine wave. This confirms that the audio is being inputted, but that the way it is being processed may be incorrect.

C.3.4 Iteration 3d

- Change the `TargetDataLine`'s `audioFormat` properties so that the sample size is 8 bits long

The application was tested in exactly the same way as in Iteration 3c with the extracted signal data plotted in Excel (Figure C.2).

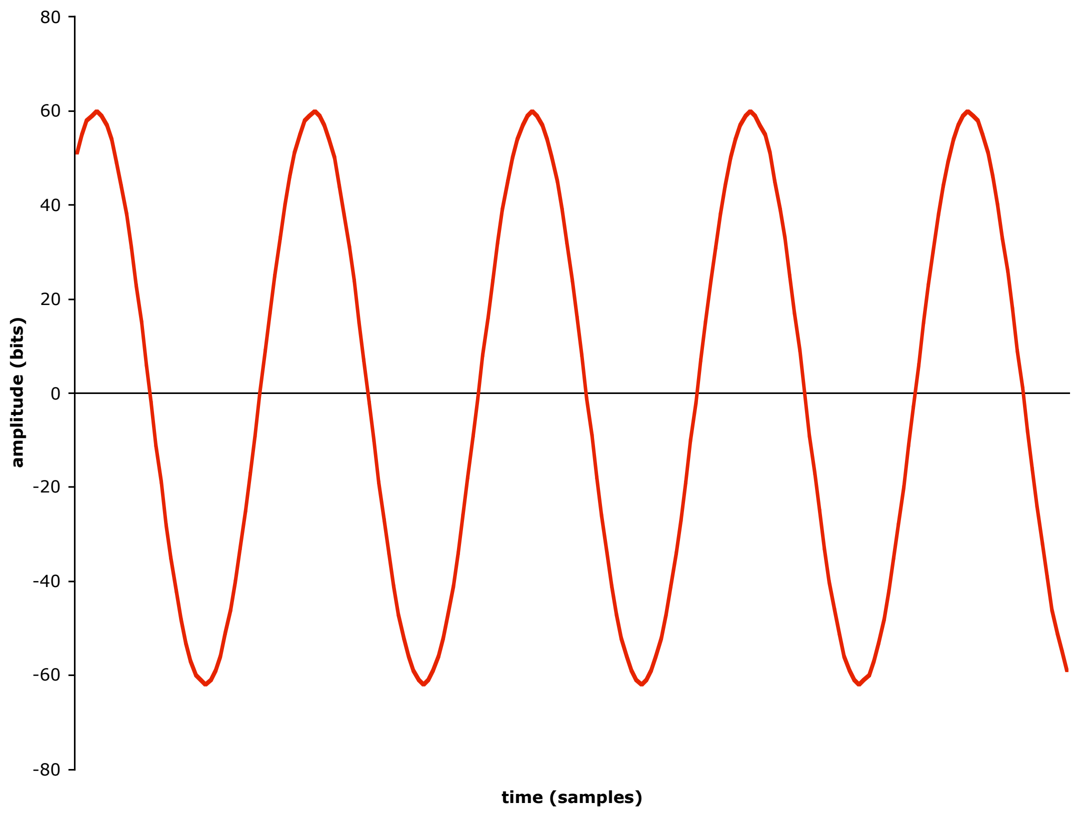


Figure C.2 Audio waveform extracted from a 1000Hz input signal

As was discussed in Section 8.1.2, if 16 bit audio data is handled then the 8 bit pairs it occupies must not be split. By inputting the audio signal in 8 bit format the byte buffer may be accessed directly, allowing for the successful handling of the audio byte data. This may appear to be where baseline point 1 is achieved, but this is an issue dealing with the processing, rather than input, of an audio signal

The problems of sample size handling becomes apparent when trying to implement a `SourceDataLine` to output the audio for monitoring purposes. As most modern soundcards do not support 8 bit audio output a 16 bit audio format must be used. However, Java does not support such audio format conversions, preventing the use of a `SourceDataLine`.

The same principal of writing to file was used to create a snapshot of the frequency spectrum data. By using Equation 8.3 the rows can be correlated to frequency values.

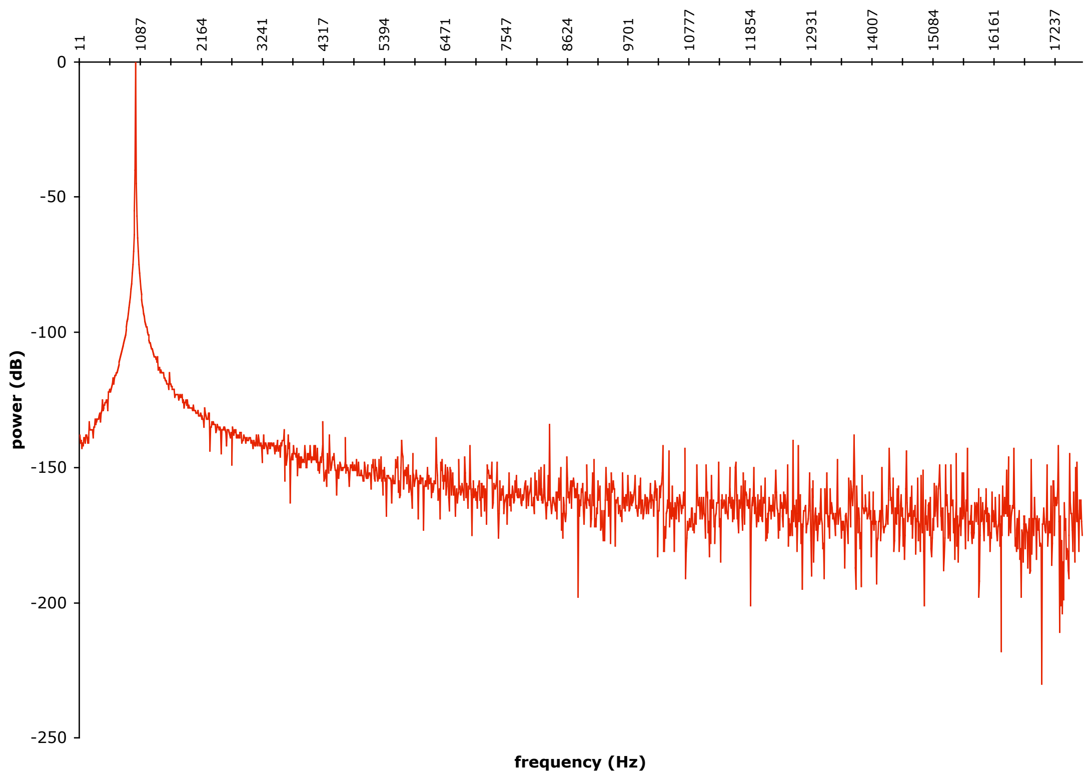


Figure C.3 Frequency content extracted from a 1000Hz input signal

Figure C.3 shows the correct frequency spectrum with a peak forming at 1000Hz. In theory only a spike should appear at 1000Hz; however, the noise generated through the digital to analogue and analogue to digital conversions creates the imperfections within the spectrum. Therefore, the successful frequency spectrum calculation allows for **BASELINE POINT 2** to be signed off.

C.3.5 Iteration 3e

- Investigate the effects of changing the size of the FFT sample set

The application was run using a 1000Hz sine wave as an audio input, like in Iteration 3d. The size of the sample set used to create Figure C.3 is 4096, so the two possible lower values of 2048 and 1024 were attempted, the results plotted in Figures C.4 and C.5.

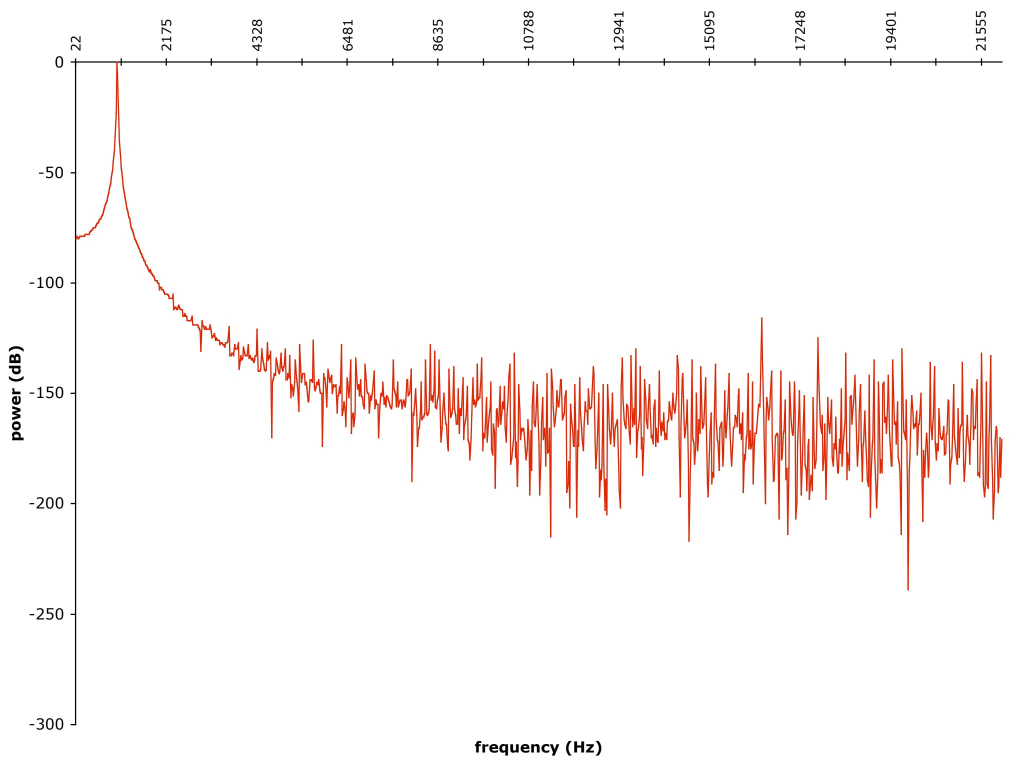


Figure C.4 Frequency content of a 1000Hz sine wave extracted from 2048 sample set

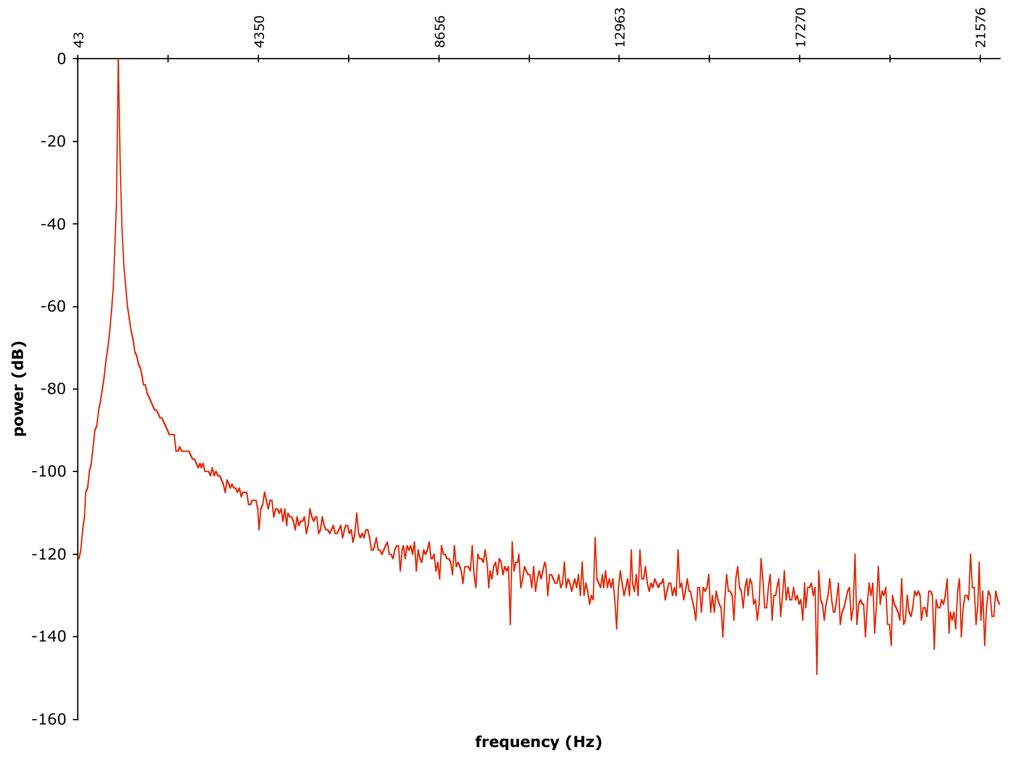


Figure C.5 Frequency content of a 1000Hz sine wave extracted from 1024 sample set

Using a 2048 sample set increased the amount of noise in the frequency spectrum, which may effect the accuracy of the resulting coefficients. However, using the lowest value of 1024 samples creates a simplified version of the spectrum without loosing important information. As discussed in Section 6.2.3 a detailed, high-fidelity frequency spectrum of the input signal is not required – the spectrum need only contain the necessary information for generating the frequency coefficients.

C.4 Iteration 4

C.4.1 Iteration 4a

- Create a simple 3D scenegraph hierarchy using a `SimpleUniverse`
- Place a box within the scene
- Make the box expand and contract (not reactively)

The box was successfully created and an interpolator used to make it expand and contract. A simple `Frame` was used to view the result, as may be seen in Figure C.6.

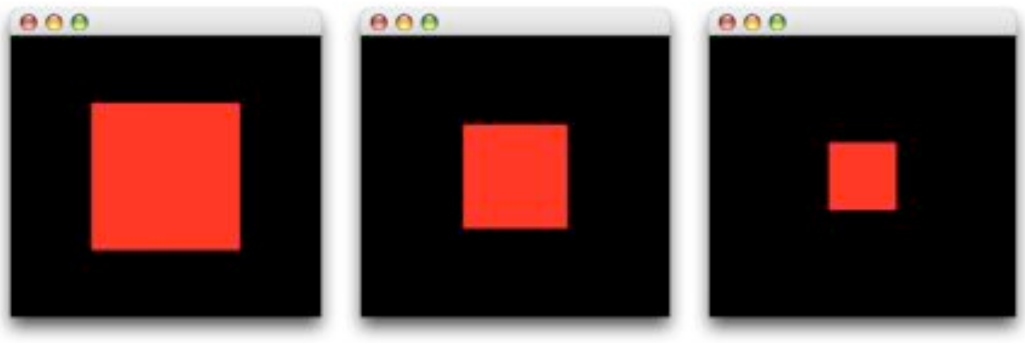


Figure C.6 Contracting box

C.4.2 Iteration 4b

- Implement a `ViewingPlatform` to access the current `ViewPlatform`
- Move the current `ViewPlatform`

In order to change the view, the `ViewPlatform` must have a transformation applied to it. Access through using a ViewingPlatform was not possible, the result remaining like Figure C.6.

C.4.3 Iteration 4c

- Request a `ViewingPlatform` from the `SimpleUniverse` object

- Request a `ViewPlatform`'s `TransformGroup` from the `ViewingPlatform`
- Attempt to move the requested `ViewPlatform`

The `SimpleUniverse` class contains a method `getViewingPlatform` which returns the current `ViewingPlatform` object. By accessing the `ViewPlatform`'s `TransformGroup` an attempt was made to apply a movement vector. However, the resulting output showed only a blank screen, even when a vector of (0, 0, 0) was applied.

C.5 Iteration 5

C.5.1 Iteration 5a

- Implement the view branch manually using a `VirtualUniverse`
- Make the class an extension of `Frame`
- Attempt to move the `ViewPlatform`

The scenegraph was changed to the design that can be seen in Figure 8.7. By extending the class from a `Frame` the `Canvas3D` object may be added to itself. The view point was successfully moved and a rotating box placed within the scene (Figure C.7).

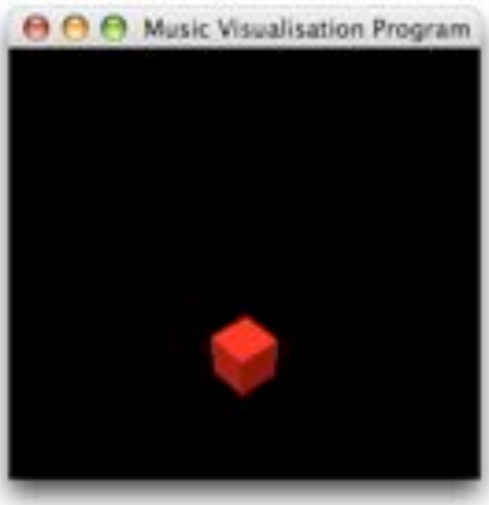


Figure C.7 The repositioned view point

C.5.2 Iteration 5b

- Assign a `ScaleInterpolator` to the box
- Trigger the `ScaleInterpolator` through the update method

The aim of this iteration is to get the box to pulse when the `update` method is called. By using `Alpha.setStartTime(System.currentTimeMillis())` the interpolator is in effect restarted. As the Alpha's loop count is set to 1 the box only performs a single pulse on every update.

C.6 Iteration 6

C.6.1 Iteration 6a

- Create a more extensive scenegraph containing 5 boxes

This forms the basis of the first visualisation theme. The boxes are arranged in a curved shape, ready for the frequency coefficients (Figure C.8). This fulfils the requirement of **BASELINE POINT 3**.

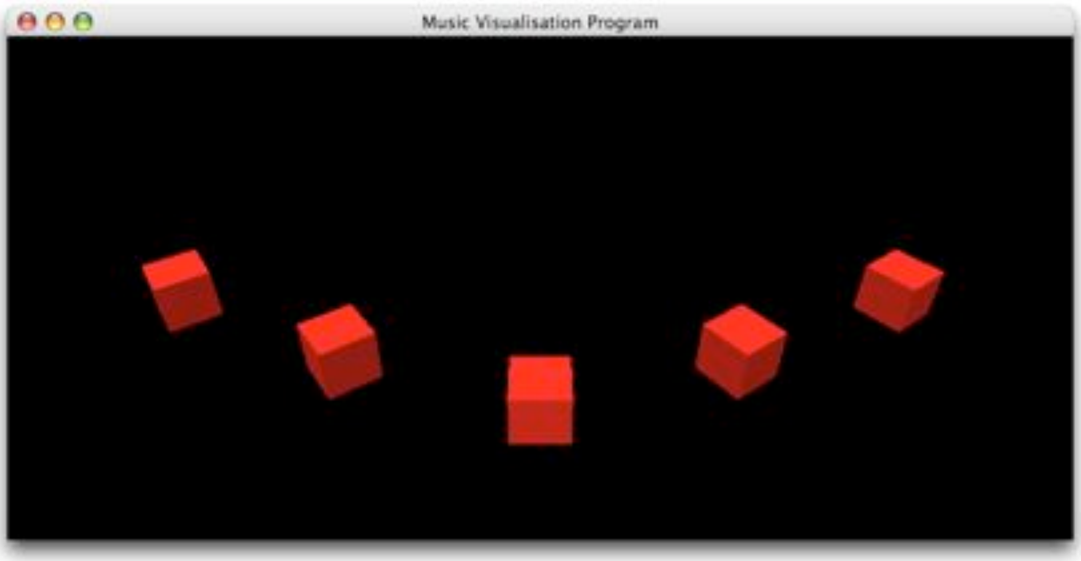


Figure C.8 The basic pulsing boxes theme

C.6.2 Iteration 6b

- Calculate the average amplitude value for each of the frequency bands
- Create a set of coefficients from the average amplitudes

The frequency bands specified in Figure 8.5 were used to divide the spectrum. In order to verify the calculated average values Excel was used to produce a correlating set of results.

As discussed in Section 8.1.4, the decibel values must be converted into a coefficient value between 0 and 1. The concept of suppressing the frequency averages to prevent loud frequencies from flooding the graphical output was employed. Equation C.1 was used to create such an effect.

$$k = e^{0.006p} \qquad \text{(C.1)}$$

Being an exponential equation the expression reaches $k = 0$ at $p = -\infty$, and as the signal power values may only reach a maximum of 0 dB there is no need for conditional statements. Again, Excel was used to produce a set of correlating results to verify the calculations, allowing for **BASELINE POINT 4** to be signed off.

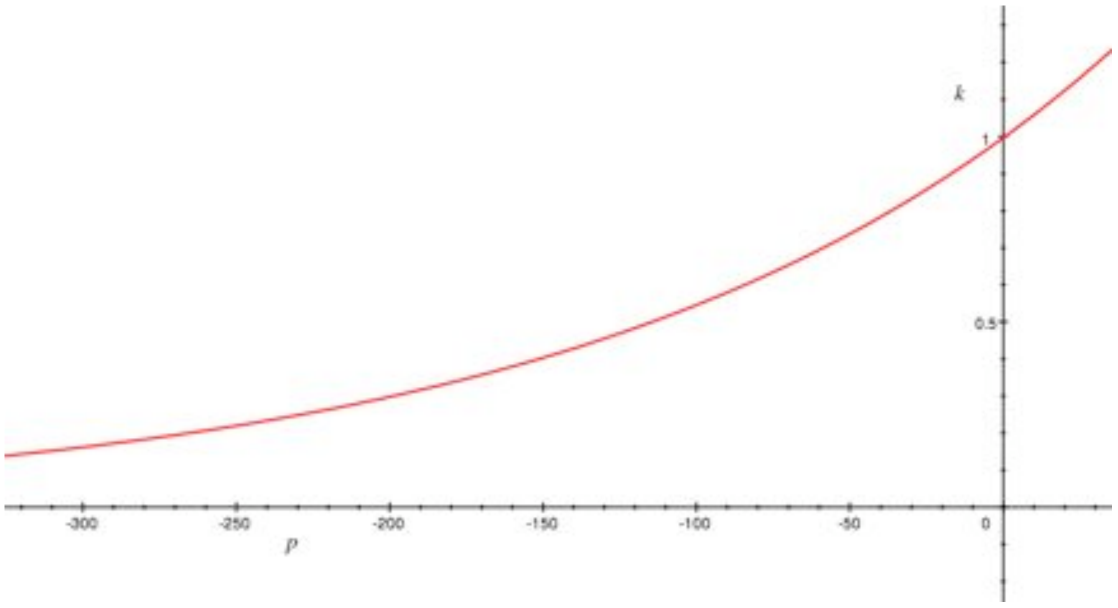


Figure C.9 The expression k = $e^{0.006p}$

C.6.3 Iteration 6c

- Update the boxes using the frequency coefficients

The following lines of code were placed in the update method for each of the boxes:

```
pulseInterpolator1.setMinimumScale((float)(frequencyBands[0]+1));
alpha1.setStartTime(System.currentTimeMillis());
```

The scale of the interpolator is set to the coefficient value + 1, the coefficient in effect setting the transformation percentage. The application was run using a 1000 Hz sine wave input signal and the successful reaction to the input signal observed (Figure C.10). This allows for **BASELINE POINT 5** to be written off.

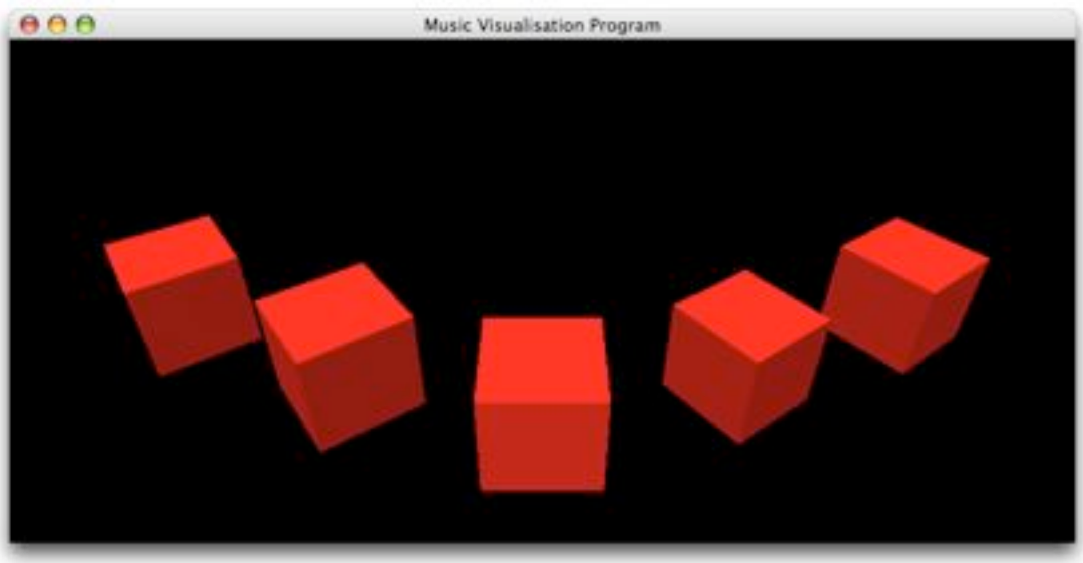


Figure C.10 The pulsing boxes theme reacting to an input signal

C.7 Iteration 7

C.7.1 Iteration 7a

- As an extension to the pulsing boxes theme create ghost boxes at the peaks of expansion

The implementation was carried out on a single box first to simplify development. For testing purposes instead of using an audio input coefficient values were hard coded,

```
private double[] frequencyBands = {0.0, 0.9, 0.0, 0.0, 0.0};
```

However the positioning of the ghost box is incorrect (Figure C.11), despite the clear positioning data given to it.

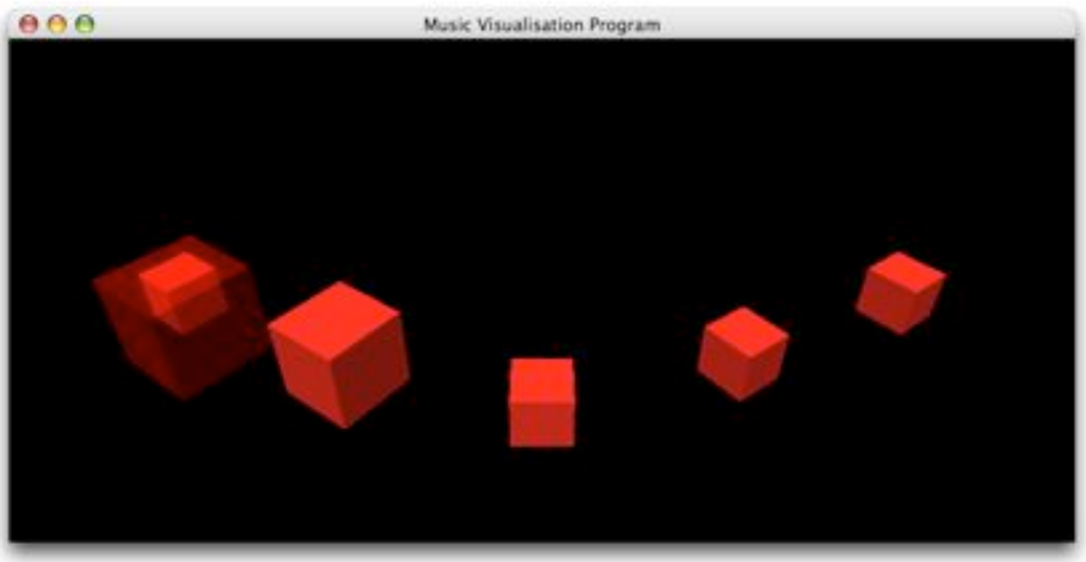


Figure C.11 The incorrect positioning of the ghost box

As the positioning data given to the ghost box was verified, the application was tested to see if the positioning was correct before any calls to the update method were made. The enlarged box's opacity was set to 50% so that it may be seen and the update method disabled (Figure C.12).

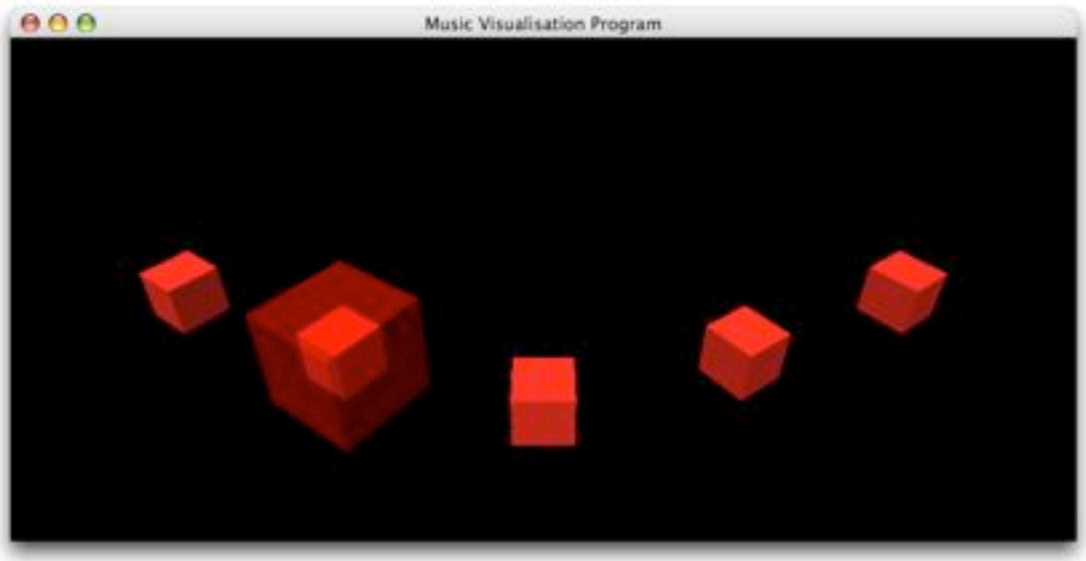


Figure C.12 Confirmation of the correct positioning data

The fact that the positioning is correct before any transformations occur suggests that there may be an issue with the scenegraph hierarchy.

C.7.2 Iteration 7b

- Place the scenegraph elements in arrays for easier management
- Create another level of `TransformGroup` objects, solely to be used for positioning the ghost boxes

The implementation of arrays is designed to streamline the code, placing repeated sections of code in for loops. As a result of Iteration 7a another layer of `TransformGroup` objects were used to ensure that the positional data was not being distorted by the scaling factor. Results of the successful 5 box implementation may be seen in Figure C.13.

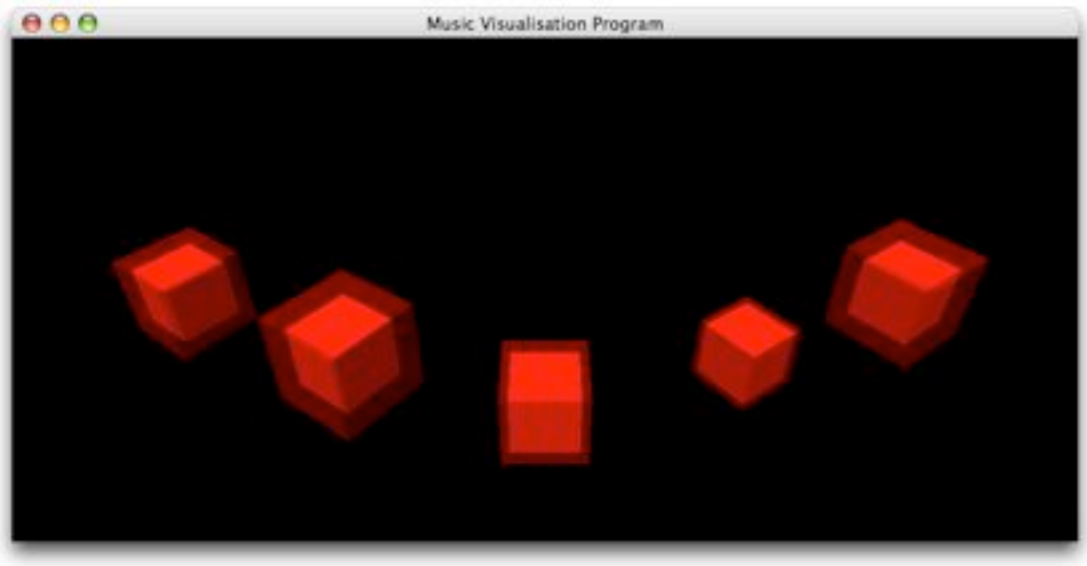


Figure C.13 Successful implementation of the ghost boxes

The reasoning behind the positioning problems is illustrated in Figure C.14; the `TransformGroup` in red separates the positional and scaling transformation factors, preventing the merging of the two. Without it the scaling factor effects both the positioning and dimensions of the ghost box, causing the problem seen in Figure C.11. This theme may be signed off as **BASELINE POINT 5**.

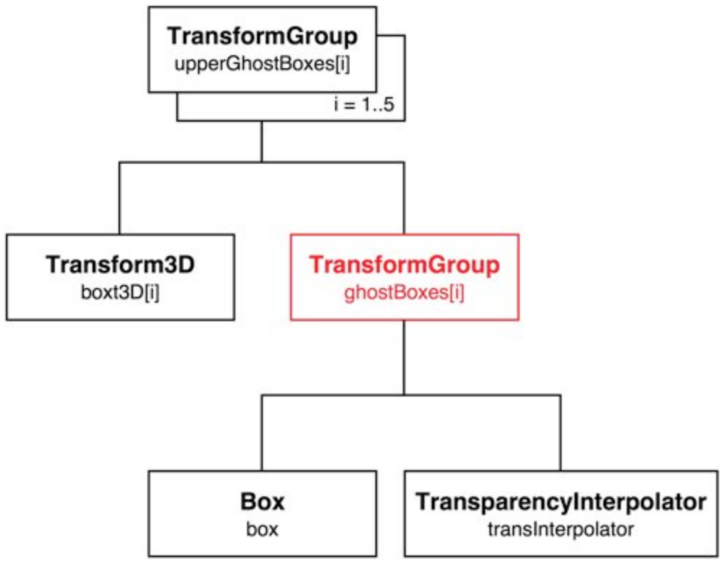


Figure C.14 The TransformGroup in red is necessary to preserve the positioning

C.8 Iteration 8

C.8.1 Iteration 8a

- Implement a scene containing 5 bouncing balls
- The motion under gravity should be achieved using the `Alpha` parameters

As discussed in Section 8.3.1, the parameters provided by the `Alpha` class are insufficient for simulating motion under gravity. This was evident in the observations made in this iteration.

C.8.2 Iteration 8b

- Create a `GravityInterpolator` class which extends `TransformInterpolator` to implement the expression $s = ut + \frac{1}{2}at^2$

The `GravityInterpolator` class may be used in exactly the same way as the `PositionInterpolator` (see Section 8.3.1 for more about custom interpolators). The motion of the ball within the scene became more realistic and could be updated with the frequency coefficients. The technique used is discussed further in Section 8.3.2.

C.8.3 Iteration 8c

- Create transparent VU meter elements which fade in when the ball is at or above them, much like on a conventional parametric VU meter.

Using the methodology shown in Appendix A.1 the VU meters' transparency levels were set. The key is the conversion of the spheres' height into an indexing value for the for loops. However, the results did not appear as expected.

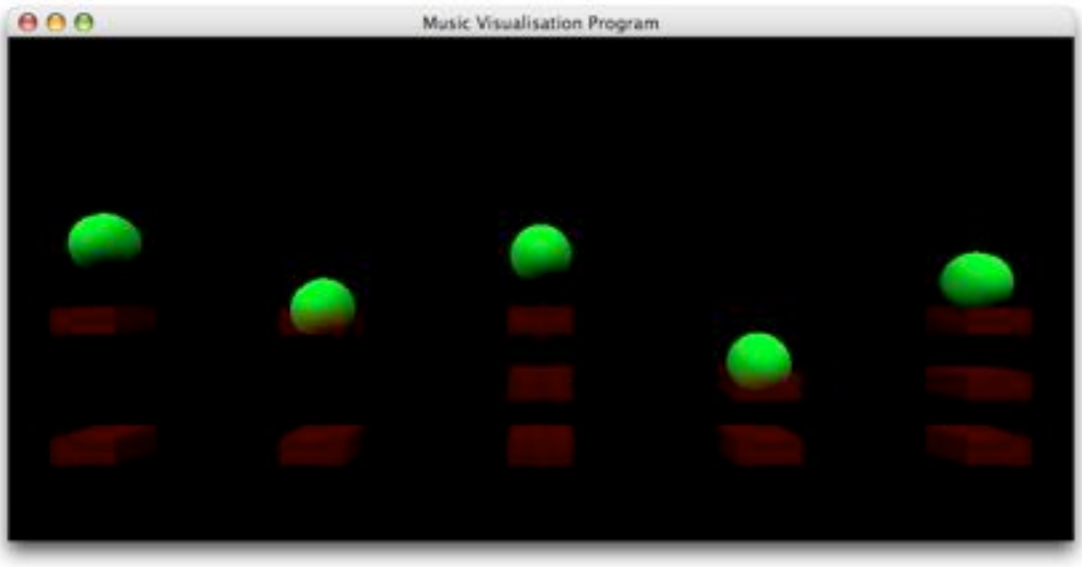


Figure C.15 VU meter implementation showing missing elements

As can be seen in Figure C.15 there are elements missing from the VU stacks. The same routine is used for every meter, clearing any errors in the `update` method.

C.8.4 Iteration 8d

- Remove fading of elements
- Experiment with alternative transparency modes

In order to reduce processing power the VU elements are switched rather than faded. However, the same effects as shown in Figure C.15 still occur so as a further investigation the transparency modes are altered. In Iteration 8c the mode `NICEST` was used, demanding the most processing power. Two other modes that require less processing power were tested; the mode `FASTEST` (Figure C.16) uses the fastest available method of rendering, but provides no improved results. Neither does the mode `SCREEN_DOOR` (Figure C.17) which utilises an on/off stipple pattern [26]

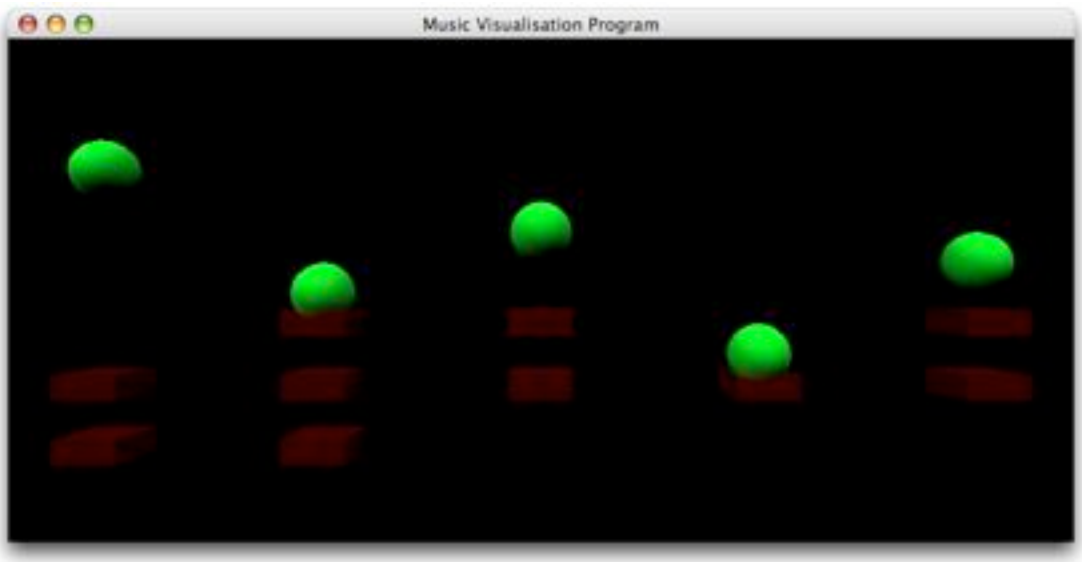


Figure C.16 VU meter implementation using the mode FASTEST

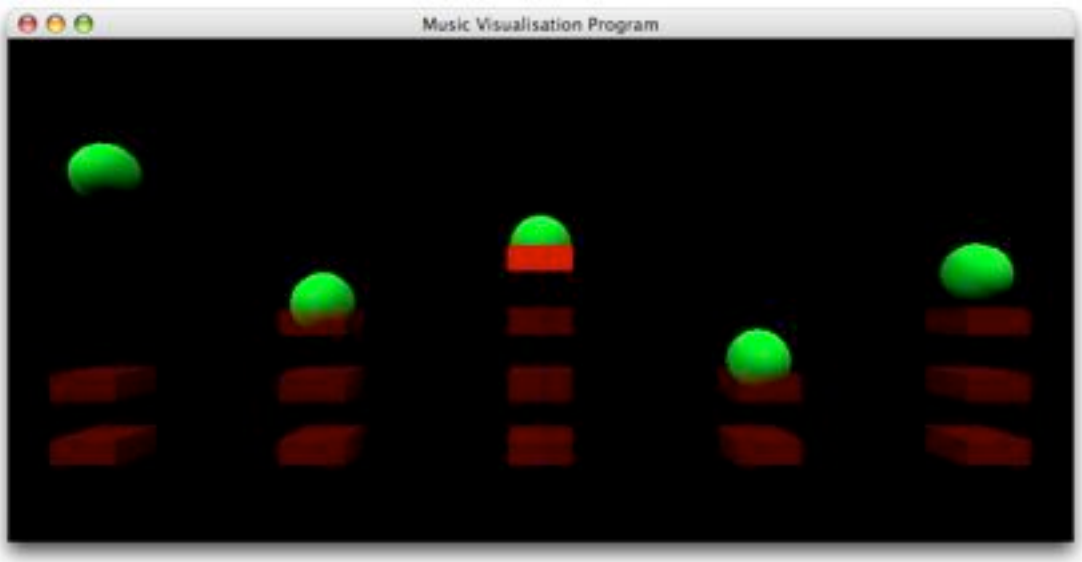


Figure C.17 VU meter implementation using the mode SCREEN_DOOR

C.8.5 Iteration 8e

- Create a separate method for updating the VU elements
- Implement the `GravityInterpolator` class as an inner class
- Call the new method from within the `GravityInterpolator`'s `computeTransform` method

To ensure that the elements are being updated as regularly as possible, the VU meter's update routine is called every time the balls' positional data is altered. This too leads to erratic results, as may be seen by the differing outputs in Figures C.18 and C.19.

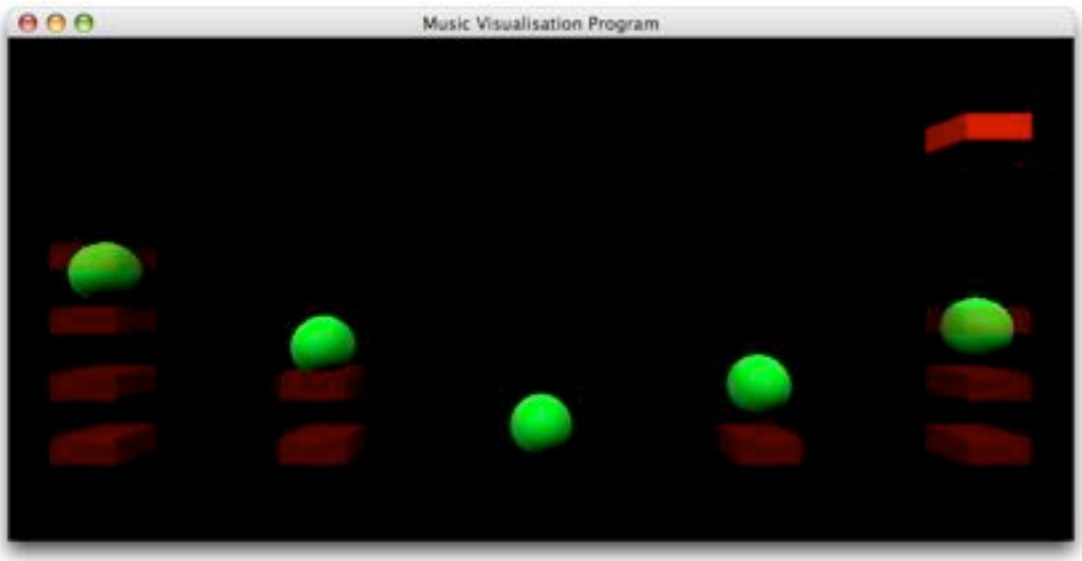


Figure C.18 Erratic results in the VU meters

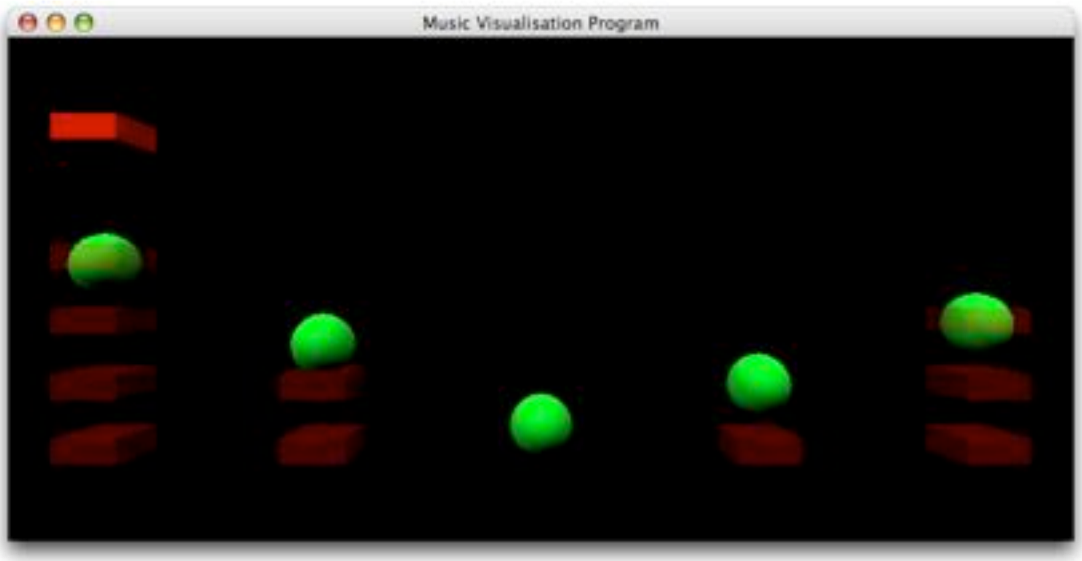


Figure C.19 Differing errors to the above

These transparency issues are not uncommon in implementing such applications. Using transparency makes rendering significantly slower due to the complex blending operations which must take place. Despite version 1.3 implementing depth-sorted transparency [27], is it still difficult to know what the rendered result will be. As the low-level functionality is performed using OpenGL, issues which occur with transparency are not the fault of Java 3D. However, Sun have not done enough to shield the developer such problems [28]. At this point the development of the theme was ended.

C.9 Iteration 9

C.9.1 Iteration 9a

- Continue from Iteration 8b
- Implement a particle which moves as though attached to an elastic string

The principals employed in Iterations 9a to 9d are explained in Section 8.3.3. Here, the acceleration is controlled using the expression $a = ks$, where k is a constant and s is the distance away from its target. The displacement is calculated using the expression $s = ut + \frac{1}{2}at^2$, where u is the initial velocity and t is time.

A particle is applied to a single sphere for testing. When run the particle follows the ball with the correct motion. However, its displacement resets at regular intervals due to the Alpha values returning to 0 as they loop.

C.9.2 Iteration 9b

- Employ constant acceleration which changes polarity when it passes the target.

This is more of a basic flocking system, directing the particle to accelerate in either direction. When the Alpha begins a new loop the initial velocity and position offset (which is added to the displacement) are updated using the previous final velocity and displacement values. However, the same resetting problems occur as in Iteration 9a.

C.9.3 Iteration 9c

- Calculate the displacement according to the time difference between `computeTransform` calls
- Calculate the acceleration relative to the distance from its target

To overcome the resetting of displacement an incrementing total of time was maintained. The resulting output did not reset as before and follows the target's movements. After approximately 20 seconds the particle entered into high frequency oscillations, the amplitude widening until eventually invisible within the scene. This behaviour was caused by the slowly increasing acceleration and time values, despite attempts to limit the acceleration when it rises too high.

C.9.4 Iteration 9d

- Recalculate all values, including time, on every method call

To overcome the high frequency oscillations that occur when using cumulative values, the approach of recalculation was employed. A detailed structure of program flow is shown in Appendix A.2.1 and is further discussed in Section 8.3.3. The resulting particle motion remains stable and is more visually realistic. The scene was signed off as **BASELINE POINT 5**.

C.10 Iteration 10

C.10.1 Iteration 10a

- Continue from Iteration 8b
- Increase dimensions of the spheres
- Implement orbiting lights which follow the spheres' movement

The working scene from Iteration 8b was taken and the dimensions of the spheres increased. The same technique as discussed in Section 8.2.2 was used to make the orbiting lights follow the motion of the bouncing balls.

Only the effects of a light are visible within a scene, so dummy glowing satellites were created to give the impression that they are creating the illumination. To increase this effect the orbiting objects' emissive and specular colour fields are increased, removing any dark shading and giving the impression that they are self illuminating.

On first testing, the scene ran very slowly with a low frame rate. It was considered that the large number of lights within the scene were adding an unreasonable load upon the system. This was overcome by reducing the lights' `BoundingSphere` objects so that their rays impact only the ball they are orbiting. At this point the scene could be signed off as **BASELINE POINT 5**.

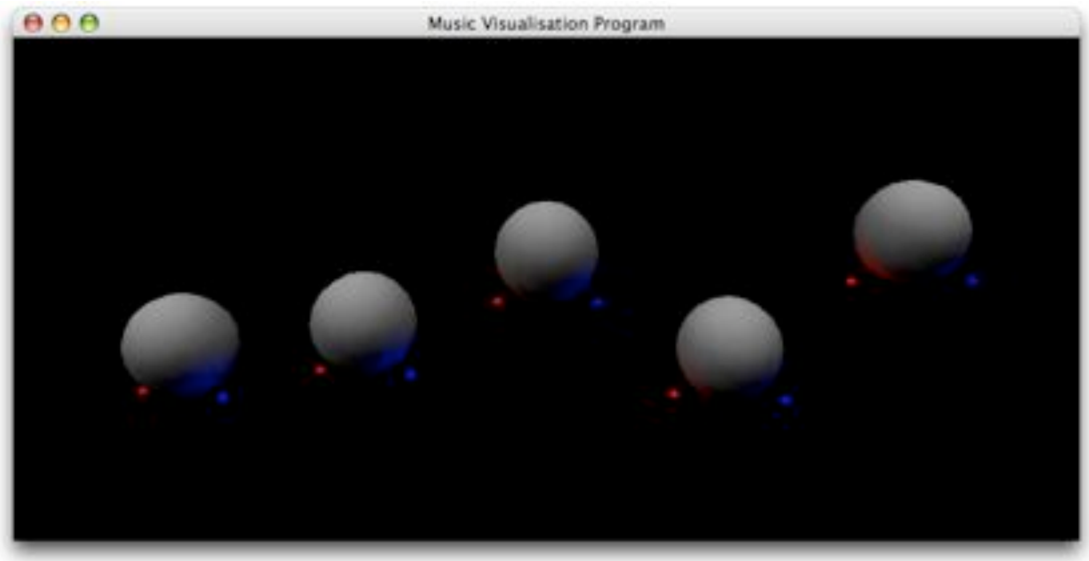


Figure C.20 Bouncing balls with lights

C.11 Iteration 11

C.11.1 Iteration 11a

- Implement a 3D waveform which plots the frequency coefficients

The waveform is constructed from a set of six `Shape3D` objects, the geometry of which is specified by a `QuadArray`. Each of the frequency bands correlates to a `Point3d` object that controls the `QuadArray` positioning. On every update call the new positioning data is applied to the `Point3d` objects and the coordinates of the `QuadArray` elements set. Figure A.7 shows a flowchart for the update method.

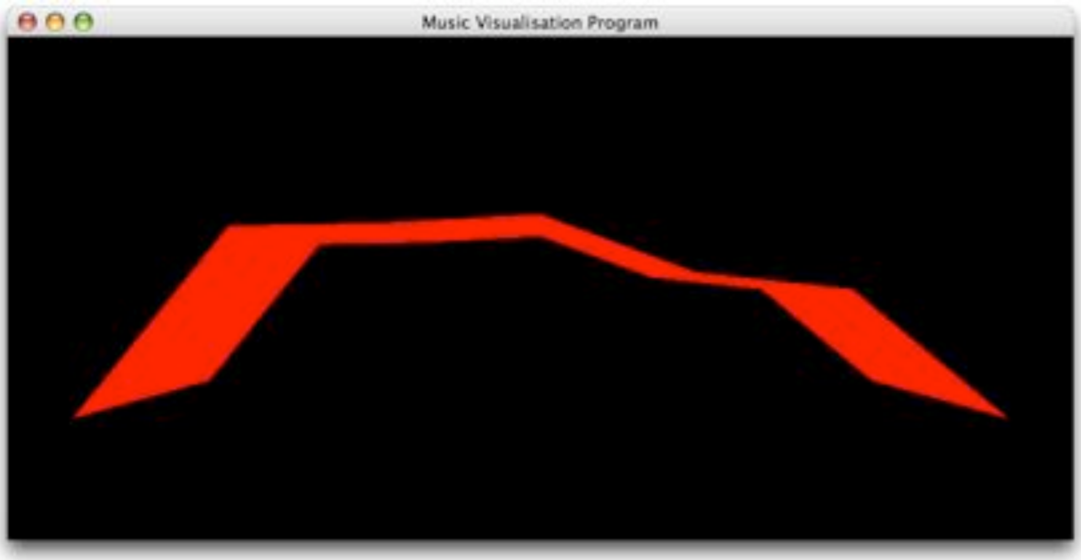


Figure C.21 The 3D waveform from a 1000Hz sine wave

A.11.2Iteration 11b

- Add two small `Sphere` objects which follow a spiral path

The `Sphere`'s motion is controlled by the custom interpolator discussed in detail in Section 8.3.4. The result may be seen in Figure C.22. The theme was signed off as **BASLINE POINT 5**.

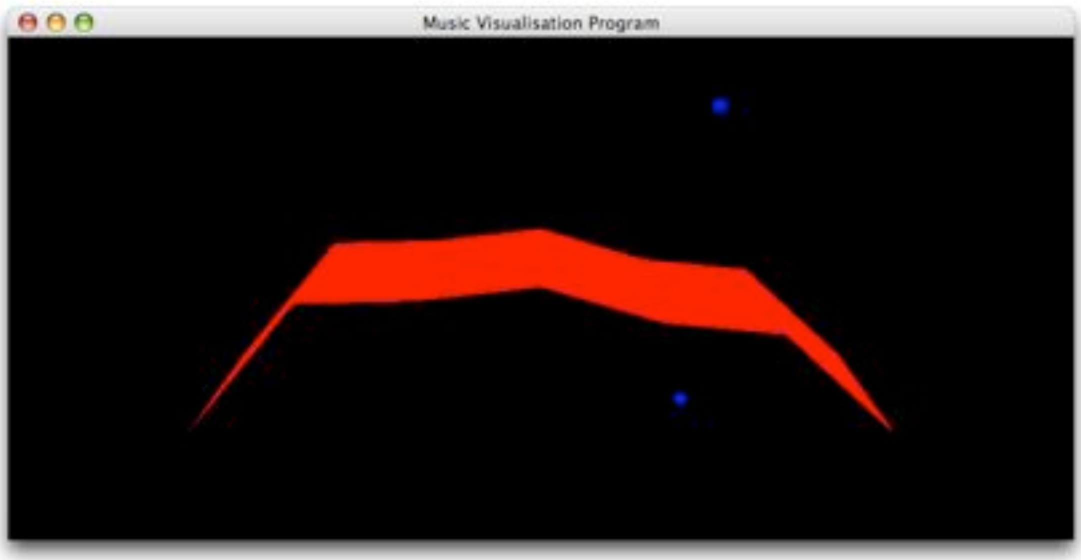


Figure C.22 The 3D waveform with additional Spheres

C.12 Window Iterations

C.12.1 Window Iteration 1

- Create the `MainFrame` class to house all four successful visualisations

The class `MainFrame` is a simple GUI containing a set of buttons which control the displayed visualisation. The buttons have `MouseListener` objects added to them which contain the necessary code to achieve this. The details of initialisation are shown in Figure A.3. With the graphics contained within a GUI the iteration could be signed off as **BASELINE POINT 6**.

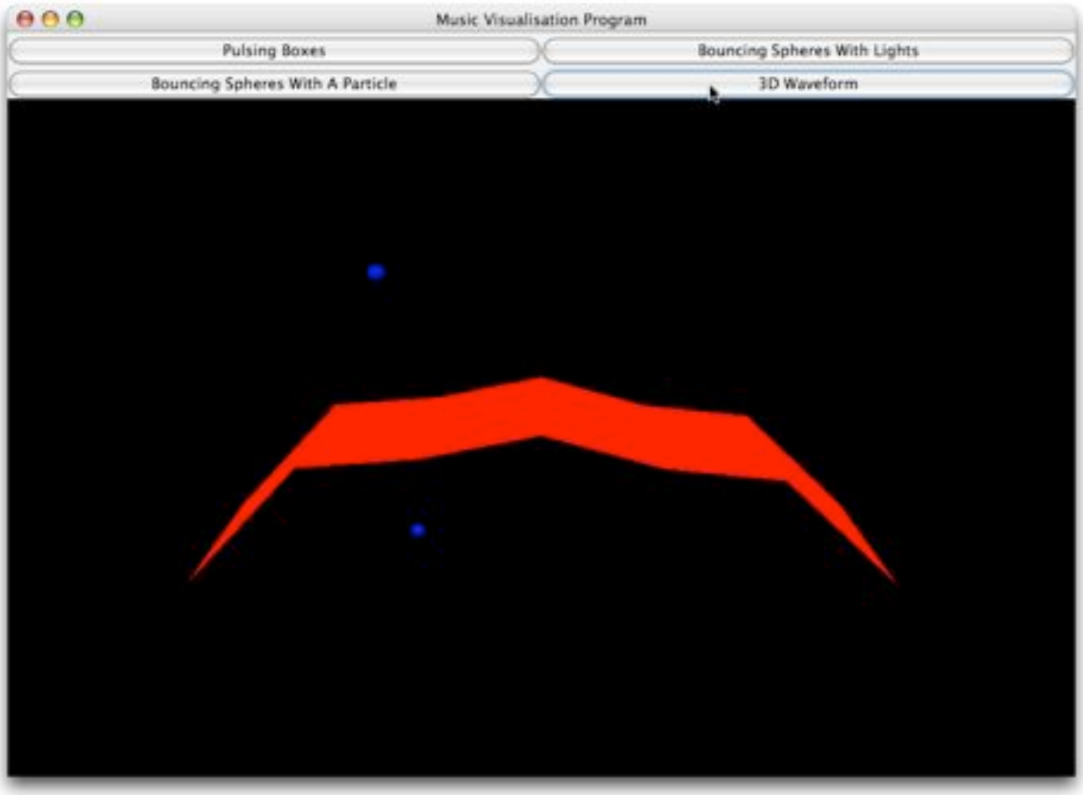

Music Visualisation Program
Pulsing Boxes
Bouncing Spheres With Lights
Bouncing Spheres With A Particle
3D Waveform

Figure C.23 Clicking on a button displays the visualisation

C.12.2 Window Iteration 2

- Investigate other frequency coefficient expressions and frequency band ranges

With the perspective of four visualisation themes, more work could be done on achieving the best frequency coefficient expression. The application was run and a variety of different music played into the audio input. Coefficient values were written to the terminal and observations made of the visual output. The range of different coefficient values produced spanned from around 0.45 to 0.65 and dropping sharply down to 0 when the volume is lowered.

This sudden drop in the coefficient suggests that the decibel range is smaller than the expression accommodates for. To investigate this the frequency band power averages were printed to screen for monitoring. The audio input was taken from a muted volume where the output reads –Infinity and increased to the maximum level possible. These values were copied into Excel where they cold be numerically sorted, thus giving the maximum and minimum achievable decibel values, -25dB and –200db respectively. Inspection of Figure C.24 provides an explanation for the sudden drop in the coefficient for this decibel range. In sight of this research the expression $k = e^{0.006p}$ was modified to Equation C.2, which covers a more appropriate decibel range, as can be seen in Figure C.24.

$$k = 1.5e^{0.02p} \tag{C.2}$$

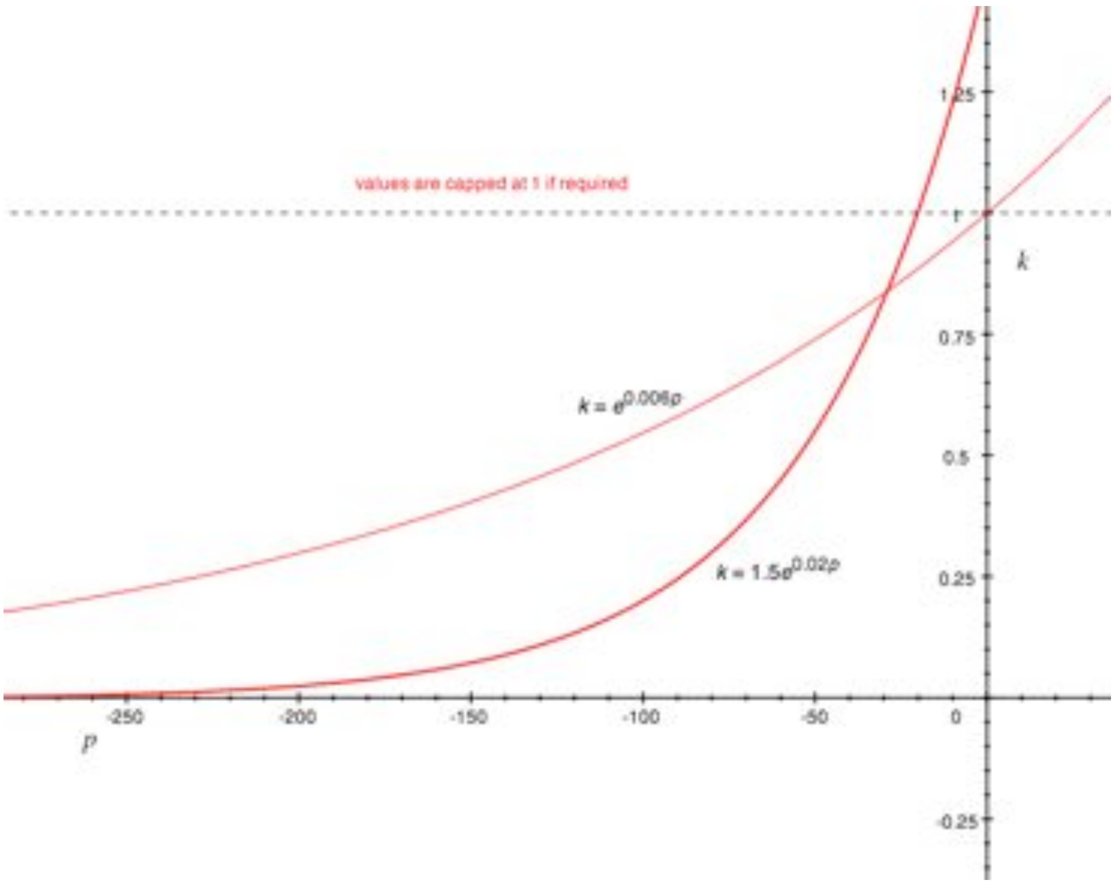


Figure C.24 The expression from Figure A.3.7 plus two alternatives, the one chosen being k = 1.5e$^{0.02p}$

C.12.3 Window Iteration 3

- Implement the `kill` method to fix the resource leak which occurs when button presses are made.
- Place a title graphic when the application is first opened

- Export the application as a single executable file.

The resource leak discussed in Section 8.4 was remedied. The introductory graphic was also inserted, as may be seen in Figure 8.13.

Appendix D: Source Code

D.1 AudioController.java

```java
/*
 * This is class handles the input of data into the program
 *
 * Nicholas Martin
 * April 2004
 */

import javax.sound.sampled.*;
import java.io.*;

class AudioController {

    private MainWindow mainWindow;

    private AudioFormat audioFormat;
    private Line.Info  lineInInfo;

    private TargetDataLine targetDataLine;

    // create a buffer to store the audio data
    private final int BUFFER_SIZE = 1024;
    private final int HALF_BUFFER_SIZE = BUFFER_SIZE/2;
    private byte[] buffer = new byte[BUFFER_SIZE];


    private Fft fft = new Fft(10);
    private double[] realArray = new double[BUFFER_SIZE];
    private double[] imaginaryArray = new double[BUFFER_SIZE];
    private double maxPower = 0.0;

    private double[] frequencyBands = new double[5];


    /*
     * constructor which contains the initialisation code to set up the system
     */
    public AudioController() throws LineUnavailableException, IOException{

        // create the GUI
        mainWindow = new MainWindow();

        // initialise variables to specify the TargetDataLine
        audioFormat = new AudioFormat(44100.0F, 8, 1, true, false);
        DataLine.Info targetDataLineInfo = new DataLine.Info(TargetDataLine.class, audioFormat);

        // create a TargetDataLine with the specifications above and open it
        targetDataLine = (TargetDataLine)AudioSystem.getLine(targetDataLineInfo);
        targetDataLine.open(audioFormat);

        // start the TargetDataLine
        targetDataLine.start();

        // MAIN PROGRAM LOOP
        int bytesRead = 0;
        while(true) {

            bytesRead = targetDataLine.read(buffer, 0, BUFFER_SIZE);
            analyseAudio();
            mainWindow.update(frequencyBands);
        }
    }
```

```
/*
 * this method calculates the frequency coefficients from the audio data
 */
private void analyseAudio() {

    // copy bytes aquired into realArray and place zero in imaginaryArray
    for (int i = 0; i < BUFFER_SIZE; i++) {
        realArray[i]      = (double)buffer[i];
        imaginaryArray[i] = 0.0;
    }

    // calculate the FFT of the signal
    fft.doFFT(realArray, imaginaryArray, false);

    // find the maximum power frequency
    for(int i = 0; i < HALF_BUFFER_SIZE; i++) {
        double power = Math.pow(realArray[i], 2) + Math.pow(imaginaryArray[i], 2);

        // Record the largest power value
        if (power > maxPower) maxPower = power;

        realArray[i] = power;
    }

    // normalise the values and then calculate the normalised power spectrum in dB
    for (int i = 0; i < HALF_BUFFER_SIZE; i++) {
        realArray[i] = 10 * Math.log(realArray[i] / maxPower);
    }

    // find the average amplitude of the different bands

    // realArray[1 to 3] gives the frequencies for 25 to 125 Hz
    double total = 0;
    int iterations = 0;

    for (int i = 1; i <= 3; i++) {
        total = total + realArray[i];
        iterations++;
    }
    frequencyBands[0] = total/iterations;

    // realArray[4 to 6] gives the frequencies for 125 to 250 Hz
    total = 0;
    iterations = 0;

    for (int i = 4; i <= 6; i++) {
        total = total + realArray[i];
        iterations++;
    }
    frequencyBands[1] = total/iterations;

    // realArray[7 to 11] gives the frequencies for 250 to 500 Hz
    total = 0;
    iterations = 0;

    for (int i = 7; i <= 11; i++) {
        total = total + realArray[i];
        iterations++;
    }
    frequencyBands[2] = total/iterations;

    // realArray[12 to 23] gives the frequencies for 500 to 1000 Hz
    total = 0;
    iterations = 0;

    for (int i = 12; i <= 23; i++) {
        total = total + realArray[i];
        iterations++;
    }
    frequencyBands[3] = total/iterations;

    // realArray[24 to 46] gives the frequencies for 1000 to 2000 Hz
    total = 0;
    iterations = 0;
```

```
        for (int i = 24; i <= 46; i++) {
            total = total + realArray[i];
            iterations++;
        }
        frequencyBands[4] = total/iterations;

        // from the average bands calculate a co-efficient
        for (int i = 0; i < frequencyBands.length; i++) {
            frequencyBands[i] = 1.5*Math.exp(0.02*frequencyBands[i]);

            // ensure that the co-efficient is never greater than 1
            if (frequencyBands[i] > 1.0) {
                frequencyBands[i] = 1.0;
            }
        }
    }



    public static void main(String[] args) {

        AudioController audioController;

        try {
            audioController = new AudioController();
        }
        catch (LineUnavailableException e){
            System.out.println(e);
            System.exit(1);
        }
        catch (IOException e) {
            System.out.println(e);
            System.exit(1);
        }


    }
}
```

D.2 Fft.java

```
// Fast Fourier Transform (FFT) Code
// Java implementation by: Craig A. Lindley
// Last Update: 02/27/99

//package craigl.spectrumanalyzer;


/* libfft.c - fast Fourier transform library
**
** Copyright (C) 1989 by Jef Poskanzer.
**
** Permission to use, copy, modify, and distribute this software and its
** documentation for any purpose and without fee is hereby granted, provided
** that the above copyright notice appear in all copies and that both that
** copyright notice and this permission notice appear in supporting
** documentation.  This software is provided "as is" without express or
** implied warranty.
*/

public class Fft {
     /**
     * This is a Java implementation of the fast Fourier transform
     * written by Jef Poskanzer. The copyright appears above.
     */

    private static final double TWOPI = 2.0 * Math.PI;

    // Limits on the number of bits this algorithm can utilize
    private static final int LOG2_MAXFFTSIZE = 15;
    private static final int MAXFFTSIZE = 1 << LOG2_MAXFFTSIZE;

    /**
     * FFT class constructor
     * Initializes code for doing a fast Fourier transform
     *
     * @param int bits is a power of two such that 2^b is the number
     * of samples.
     */
    public Fft(int bits) {

        this.bits = bits;

        if (bits > LOG2_MAXFFTSIZE) {
            System.out.println("" + bits + " is too big");
            System.exit(1);
        }
        for (int i = (1 << bits) - 1; i >= 0; --i) {
            int k = 0;
            for (int j = 0; j < bits; ++j) {
                k *= 2;
                if ((i & (1 << j)) != 0)
                    k++;
            }
            bitreverse[i] = k;
        }
    }

    /**
     * A fast Fourier transform routine
     *
     * @param double [] xr is the real part of the data to be transformed
     * @param double [] xi is the imaginary part of the data to be transformed
     * (normally zero unless inverse transofrm is effect).
     * @param boolean invFlag which is true if inverse transform is being
     * applied. false for a forward transform.
     */
    public void doFFT(double [] xr, double [] xi, boolean invFlag) {
        int n, n2, i, k, kn2, l, p;
        double ang, s, c, tr, ti;

        n2 = (n = (1 << bits)) / 2;

        for (l = 0; l < bits; ++l) {
            for (k = 0; k < n; k += n2) {
                for (i = 0; i < n2; ++i, ++k) {
                    p = bitreverse[k / n2];
                    ang = TWOPI * p / n;
                    c = Math.cos(ang);
                    s = Math.sin(ang);
                    kn2 = k + n2;

                    if (invFlag)
                        s = -s;

                    tr = xr[kn2] * c + xi[kn2] * s;
                    ti = xi[kn2] * c - xr[kn2] * s;

                    xr[kn2] = xr[k] - tr;
                    xi[kn2] = xi[k] - ti;
                    xr[k] += tr;
                    xi[k] += ti;
                }
            }
            n2 /= 2;
        }

        for (k = 0; k < n; k++) {
            if ((i = bitreverse[k]) <= k)
                continue;

            tr = xr[k];
            ti = xi[k];
            xr[k] = xr[i];
            xi[k] = xi[i];
            xr[i] = tr;
            xi[i] = ti;
        }

        // Finally, multiply each value by 1/n, if this is the forward
        // transform.
        if (!invFlag) {
            double f = 1.0 / n;

            for (i = 0; i < n ; i++) {
                xr[i] *= f;
                xi[i] *= f;
            }
        }
    }
    // Private class data
    private int bits;
    private int [] bitreverse = new int[MAXFFTSIZE];
}
```

D.2 MainWindow.java

```
/*
 * This is class creates a GUI for the graphics to be displayed
 *
 * Nicholas Martin
 * April 2004
 */


import java.awt.*;
import java.awt.event.*;

public class MainWindow extends Frame {

    private PulsingBoxes pulsingBoxes;
    private BouncingLights bouncingLights;
    private BouncingParticle bouncingParticle;
    private Waveform waveform;

    private int option = 0;

    /*
     * initialise the GUI
     */
    public MainWindow() {

        super("Music Visualisation Program",
              com.sun.j3d.utils.universe.SimpleUniverse.getPreferredConfiguration());

        // create two Panels: one for the buttons and one for the graphics
        Panel buttonPanel = new Panel();

        // add buttons to the buttonPanel
        Button pulsingBoxesButton = new Button("Pulsing Boxes");
        pulsingBoxesButton.addMouseListener(new Option1Listener());

        Button bouncingLightsButton = new Button("Bouncing Balls With Lights");
        bouncingLightsButton.addMouseListener(new Option2Listener());

        Button bouncingParticleButton = new Button("Bouncing Balls With A Particle");
        bouncingParticleButton.addMouseListener(new Option3Listener());

        Button waveformButton = new Button("3D Waveform");
        waveformButton.addMouseListener(new Option4Listener());

        // add the buttons to the button panel
        buttonPanel.setLayout(new GridLayout(2,2));
        buttonPanel.add(pulsingBoxesButton);
        buttonPanel.add(bouncingLightsButton);
        buttonPanel.add(bouncingParticleButton);
        buttonPanel.add(waveformButton);

        // create the layout for the Frame
        setLayout(new BorderLayout());
        setSize(900,650);

        // add the buttonPanel to the Frame
        add("North", buttonPanel);

        // add the intro graphic to the GUI
        //IntroGraphic introGraphic = new IntroGraphic();
        add("Center", new IntroGraphic());

        // allow the Window to be closed
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                dispose();
                System.exit(0);
            }
        });

        // show the Frame
        show();
    }


    /*
     * this passes on the frequency band coefficients
     */
    public void update(double[] frequencyBands) {

        if (option == 1) pulsingBoxes.update(frequencyBands);
        if (option == 2) bouncingLights.update(frequencyBands);
        if (option == 3) bouncingParticle.update(frequencyBands);
        if (option == 4) waveform.update(frequencyBands);
        else return;
    }


    /*
     * this is for the pulsing boxes option
     */
    private class Option1Listener implements MouseListener {

        public void mouseClicked(MouseEvent e) {
            // to prevent a NullPointerException in update() make the option = 0
            option = 0;
            cleanup();

            pulsingBoxes = new PulsingBoxes();
            add("Center", pulsingBoxes.getCanvas3D());
            option = 1;
        }

        public void mouseReleased(MouseEvent e) {}
        public void mouseEntered(MouseEvent e) {}
        public void mouseExited(MouseEvent e) {}
        public void mousePressed(MouseEvent e) {}
    }

    /*
     * this is for the bouncing balls and lights option
     */
    private class Option2Listener implements MouseListener {

        public void mouseClicked(MouseEvent e) {
            option = 0;
            cleanup();

            bouncingLights = new BouncingLights();
            add("Center", bouncingLights.getCanvas3D());
            option = 2;
        }

        public void mouseReleased(MouseEvent e) {}
        public void mouseEntered(MouseEvent e) {}
        public void mouseExited(MouseEvent e) {}
        public void mousePressed(MouseEvent e) {}
    }

    /*
     * this is for the bouncing balls and particle option
     */
    private class Option3Listener implements MouseListener {

        public void mouseClicked(MouseEvent e) {
            option = 0;
            cleanup();

            bouncingParticle = new BouncingParticle();
            add("Center", bouncingParticle.getCanvas3D());
            option = 3;
        }
        public void mouseReleased(MouseEvent e) {}
        public void mouseEntered(MouseEvent e) {}
        public void mouseExited(MouseEvent e) {}
        public void mousePressed(MouseEvent e) {}
    }

    /*
     * this is for the 3D waveform option
     */
    private class Option4Listener implements MouseListener {

        public void mouseClicked(MouseEvent e) {
            option = 0;
            cleanup();

            waveform = new Waveform();
            add("Center", waveform.getCanvas3D());
            option = 4;
        }

        public void mouseReleased(MouseEvent e) {}
        public void mouseEntered(MouseEvent e) {}
        public void mouseExited(MouseEvent e) {}
        public void mousePressed(MouseEvent e) {}
    }

    /*
     * clear up all graphics objects
     */
    private void cleanup() {
        if (pulsingBoxes != null) {
            pulsingBoxes.kill();
        }
        pulsingBoxes = null;

        if (bouncingLights != null) {
            bouncingLights.kill();
        }
        bouncingLights = null;

        if (bouncingParticle != null) {
            bouncingParticle.kill();
        }
        bouncingParticle = null;

        if (waveform != null) {
            waveform.kill();
        }
        waveform = null;

        System.gc();
    }

    // inner class for creating the intro screen
    private class IntroGraphic extends Canvas {

        private Image image;

        public IntroGraphic() {

            // set up the image handling elements
            Toolkit toolkit = Toolkit.getDefaultToolkit();
            image = toolkit.getImage("Intro graphic.jpg");
            MediaTracker mediaTracker = new MediaTracker(this);
            mediaTracker.addImage(image, 0);

            try {
                mediaTracker.waitForID(0);
            }
            catch (InterruptedException e) {
                System.out.println(e);
            }
        }
        public void paint(Graphics graphics) {
            graphics.drawImage(image, 0, 0, null);
        }
    }
}
```

D.3 PulsingBoxes.java

```
/*
 * This is class creates a 3D scene with 5 pulsing boxes
 *
 * Nicholas Martin
 * April 2004
 */

import java.awt.*;
import java.awt.event.*;
import java.util.Enumeration;

import javax.media.j3d.*;
import javax.vecmath.*;

import com.sun.j3d.utils.universe.*;
import com.sun.j3d.utils.geometry.*;

class PulsingBoxes {

    // initialise the scenegraph objects for the boxes
    private TransformGroup[] boxes = new TransformGroup[5];
    private ScaleInterpolator[] pulseInterpolator = new ScaleInterpolator[5];
    private Alpha[] alpha = new Alpha[5];

    // initialise the scenegraph objects for the ghost boxes
    private TransformGroup[] ghostBoxes = new TransformGroup[5];
    private TransparencyInterpolator[] transInterpolator = new TransparencyInterpolator[5];
    private Alpha[] ghostAlpha = new Alpha[5];

    private Canvas3D canvas;
    private VirtualUniverse universe;

    /*
     * GraphicsController constructor method which contains the initialisation
     * routine for creating the scene
     */
    PulsingBoxes() {

        // set up the GraphicsConfiguration and Canvas3D objects for the
        // rendering of the scene and make it visible
        canvas = new Canvas3D(SimpleUniverse.getPreferredConfiguration());

        // call the createView method which sets up the View objects contents
        View view = createView(canvas);

        // create the view side of the scenegraph
        Locale locale = createViewBranch(view);

        // create the content side of the scenegraph
        createContentBranch(locale);
    }

    /*
     * return the Canvas3D object for viewing in the GUI
     */
    public Canvas3D getCanvas3D() {
        return canvas;
    }

    /*
     * removes all the Locales from the scene
     */
    public void kill() {
        universe.removeAllLocales();
    }
```

```
/*
 * method which creates a View object and adds to it the Canvas3D, PhysicalBody
 * and PhysicalEnvironment objects
 */
private View createView(Canvas3D canvas) {

    View view = new View();

    view.addCanvas3D(canvas);
    view.setPhysicalBody(new PhysicalBody());
    view.setPhysicalEnvironment(new PhysicalEnvironment());
    view.setFieldOfView(1.3);

    return(view);
}


/*
 * method which sets up the view side of the scenegraph as well as the Locale
 * and VirtualUniverse
 */
private Locale createViewBranch(View view) {

    // create scenegraph objects from the top downwards
    universe     = new VirtualUniverse();
    Locale locale                = new Locale(universe);
    BranchGroup viewBranch       = new BranchGroup();
    TransformGroup viewTransform = new TransformGroup();
    ViewPlatform viewPlatform    = new ViewPlatform();

    // set capability bits to allow modification at runtime
    viewTransform.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    viewTransform.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
    viewPlatform.setCapability(ViewPlatform.ALLOW_POLICY_WRITE);
    viewPlatform.setViewAttachPolicy(View.RELATIVE_TO_FIELD_OF_VIEW);

    viewPlatform.setActivationRadius(100.0f);

    // link together the scenegraph from the bottom upwards
    viewTransform.addChild(viewPlatform);
    viewBranch.addChild(viewTransform);
    locale.addBranchGraph(viewBranch);

    // complete the chain by adding the ViewPlatform to the View
    view.attachViewPlatform(viewPlatform);

    // move the view
    Transform3D viewTransform3D = new Transform3D();
    viewTransform3D.rotX(Math.PI*0.2);
    viewTransform3D.setTranslation(new Vector3d(0.0, -1.8, -10.0));
    viewTransform3D.invert();
    viewTransform.setTransform(viewTransform3D);

    // return the Locale which contains a link to the whole chain
    return locale;
}


/*
 * create the content side of the scenegraph and attach it to the locale
 */
private void createContentBranch(Locale locale) {

    BranchGroup contentBranch = new BranchGroup();

    // create TransformGroups which allow the boxes to be generically
    // created and then moved
    for (int i = 0; i < boxes.length; i++) {
        boxes[i] = new TransformGroup();
        boxes[i].setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
        boxes[i].setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
    }

    // assign positioning values to the Transform3D objects
```

```
        Transform3D[] boxt3D = new Transform3D[boxes.length];
        for (int i = 0; i < boxt3D.length; i++) {
            boxt3D[i] = new Transform3D();
        }

        boxt3D[0].rotY(Math.PI*0.2);
        boxt3D[0].setTranslation(new Vector3d(-7.0, 0.0, -3.0));

        boxt3D[1].rotY(Math.PI*0.2);
        boxt3D[1].setTranslation(new Vector3d(-3.5, 0.0, -1.2));

        boxt3D[3].rotY(Math.PI*0.2);
        boxt3D[3].setTranslation(new Vector3d(3.5, 0.0, -1.2));

        boxt3D[4].rotY(Math.PI*0.2);
        boxt3D[4].setTranslation(new Vector3d(7.0, 0.0, -3.0));

        // set the ScaleInterpolator objects created at the top of the class
        // as well as their assigned Alpha objects this allows access during the
        // update method
        for (int i = 0; i < pulseInterpolator.length; i++) {
            pulseInterpolator[i] = createBox(boxes[i], boxt3D[i]);
            alpha[i] = pulseInterpolator[i].getAlpha();
        }


        // in order to allow the ghost boxes to be correctly scaled in the update method
        // another TransformGroup is created to house the current ghost box's TransformGroup
        // and its contents
        TransformGroup[] upperGhostBoxes = new TransformGroup[5];
        for (int i = 0; i < ghostBoxes.length; i++) {
            ghostBoxes[i] = new TransformGroup();
            ghostBoxes[i].setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
            ghostBoxes[i].setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
            upperGhostBoxes[i] = new TransformGroup();
        }

        // assign positioning values to the ghost boxes' Transform3D objects
        Transform3D[] ghostBoxt3D = new Transform3D[boxes.length];
        for (int i = 0; i < boxt3D.length; i++) {
            ghostBoxt3D[i] = new Transform3D();
        }

        ghostBoxt3D[0].rotY(Math.PI*0.2);
        ghostBoxt3D[0].setTranslation(new Vector3d(-7.0, 0.0, -3.0));

        ghostBoxt3D[1].rotY(Math.PI*0.2);
        ghostBoxt3D[1].setTranslation(new Vector3d(-3.5, 0.0, -1.2));

        ghostBoxt3D[3].rotY(Math.PI*0.2);
        ghostBoxt3D[3].setTranslation(new Vector3d(3.5, 0.0, -1.2));

        ghostBoxt3D[4].rotY(Math.PI*0.2);
        ghostBoxt3D[4].setTranslation(new Vector3d(7.0, 0.0, -3.0));

        // create the ghost boxes with the same positions as the normal boxes
        for (int i = 0; i < transInterpolator.length; i++) {
            transInterpolator[i] = createGhostBox(upperGhostBoxes[i], ghostBoxes[i],
ghostBoxt3D[i]);
            ghostAlpha[i] = transInterpolator[i].getAlpha();
        }


        // finally, add the TransformGroups to the contentBranch
        for (int i = 0; i < boxes.length; i++) {
            contentBranch.addChild(boxes[i]);
            contentBranch.addChild(upperGhostBoxes[i]);
        }

        // add the lights to the scene
        contentBranch.addChild(addLights());

        // perform optimisations on the contentBranch
        contentBranch.compile();
        // add the contentBranch to the locale object to complete the tree
        locale.addBranchGraph(contentBranch);
    }


    /*
     * method called to update the contents of the scene with new coefficients
     */
    public void update(double[] frequencyBands) {

        for (int i = 0; i < boxes.length; i++) {

            // this is where the current Scale Factor will be copied into
            Transform3D currentTransform = new Transform3D();

            boxes[i].getTransform(currentTransform);
            if (currentTransform.getScale() < frequencyBands[i]+1) {
                pulseInterpolator[i].setMinimumScale((float)(frequencyBands[i]+1));

                // allow this alpha to go first
                alpha[i].setStartTime(System.currentTimeMillis());

                Transform3D ghostBoxSize = new Transform3D();
                ghostBoxes[i].getTransform(ghostBoxSize);
                if (ghostAlpha[i].finished() == true ||
                        ghostBoxSize.getScale() < frequencyBands[i]+1) {

                    ghostBoxSize.setScale(frequencyBands[i]+1);
                    ghostBoxes[i].setTransform(ghostBoxSize);
                    ghostAlpha[i].setStartTime(System.currentTimeMillis());
                }
            }
        }
    }


    /*
     * return the ScaleInterpolator which is attached to the sent TransformGroup Box object
     */
    private ScaleInterpolator createBox(TransformGroup upperTransformGroup,
                                        Transform3D transform3D) {

        // set an extra TransformGroup to attach the Box and Interpolator to
        TransformGroup lowerTransformGroup = new TransformGroup();
        lowerTransformGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);

        // create an Appearance object to hold information on the
        // box's appearance in the scene
        Appearance appearance = new Appearance();

        // set two colours to create the Material object
        Color3f objectColor = new Color3f(0.9f, 0.1f, 0.1f);
        Color3f darkColor   = new Color3f(0.0f, 0.0f, 0.0f);
        Material material = new Material(objectColor, darkColor,
                                         objectColor, darkColor, 80.0f);

        // this may then be assigned to the Appearance
        appearance.setMaterial(material);

        // the box may now be created using the Primitive Box class
        Box box = new Box(0.5f, 0.5f, 0.5f, appearance);

        // add the lower TransformGroup to the upper TransformGroup
        // then the box can be added to the lower TransformGroup
        upperTransformGroup.addChild(lowerTransformGroup);
        lowerTransformGroup.addChild(box);

        // position the box in the scene
        upperTransformGroup.setTransform(transform3D);

        // apply a ScaleInterpolator to the box
        Transform3D interpolatorTransform3D = new Transform3D();
        Alpha alpha = new Alpha(1,0,0,200,150,0);

        ScaleInterpolator pulseInterpolator =
           new ScaleInterpolator(alpha, lowerTransformGroup, interpolatorTransform3D,
                                 2.0f, 1.0f);
        BoundingSphere bounds = new BoundingSphere(new Point3d(0.0,0.0,0.0), 100.0);
        pulseInterpolator.setSchedulingBounds(bounds);

        lowerTransformGroup.addChild(pulseInterpolator);

        return pulseInterpolator;
    }


    /*
     * create the ghost box
     */
    private TransparencyInterpolator createGhostBox(TransformGroup upperTransformGroup,
                                                    TransformGroup lowerTransformGroup,
                                                    Transform3D transform3D) {

        // create an Appearance object to hold information on the
        // box's appearance in the scene
        Appearance appearance = new Appearance();
        appearance.setCapability(Appearance.ALLOW_TRANSPARENCY_ATTRIBUTES_WRITE);

        // set two colours to create the Material object
        Color3f objectColor = new Color3f(1.0f, 0.0f, 0.0f);
        Color3f darkColor   = new Color3f(0.0f, 0.0f, 0.0f);
        Material material = new Material(objectColor, darkColor,
                                         objectColor, darkColor, 80.0f);

        // this may then be assigned to the Appearance
        appearance.setMaterial(material);

        // apply transparency to the appearance
        TransparencyAttributes transparency =
            new TransparencyAttributes(TransparencyAttributes.NICEST, 1.0f);
        transparency.setCapability(TransparencyAttributes.ALLOW_VALUE_WRITE);
        appearance.setTransparencyAttributes(transparency);

        // the box may now be created using the Primitive Box class
        Box box = new Box(0.5f, 0.5f, 0.5f, appearance);

        // add the lower TransformGroup to the upper TransformGroup
        // then the box can be added to the lower TransformGroup
        upperTransformGroup.addChild(lowerTransformGroup);
        lowerTransformGroup.addChild(box);

        // position the box within the scene
        upperTransformGroup.setTransform(transform3D);

        // add the interpolator
        Alpha alpha = new Alpha(1,0,0,1000,0,0);

        TransparencyInterpolator transInterpolator =
            new TransparencyInterpolator(alpha, transparency, 0.5f, 1.0f);

        BoundingSphere bounds = new BoundingSphere(new Point3d(0.0,0.0,0.0), 100.0);
        transInterpolator.setSchedulingBounds(bounds);

        lowerTransformGroup.addChild(transInterpolator);

        return transInterpolator;
     }




    /*
     * add many lights to the scene
     */
    private BranchGroup addLights() {
        BranchGroup lightBranch = new BranchGroup();

        // create a color and direction for the light
        Color3f color = new Color3f(0.8f, 0.8f, 0.8f);
        Vector3f direction1 = new Vector3f(-1.0f, -2.0f, -1.0f);
        Vector3f direction2 = new Vector3f(1.0f, -1.0f, -1.0f);

        // create a DirectionalLight using the color and direction specified
        DirectionalLight light1 = new DirectionalLight(color, direction1);
        DirectionalLight light2 = new DirectionalLight(color, direction2);

        // create the influencing bounds of the light
        BoundingSphere bounds =
              new BoundingSphere(new Point3d(0.0,0.0,0.0), 200.0);
        light1.setInfluencingBounds(bounds);
        light2.setInfluencingBounds(bounds);

        // add the light to the scene
        lightBranch.addChild(light1);
        lightBranch.addChild(light2);
        return lightBranch;
    }
}
```

D.4 Waveform.java

```
/*
 * This is class creates a 3D scene with a 3D waveform
 *
 * Nicholas Martin
 * April 2004
 */

import java.awt.*;
import java.awt.event.*;
import java.util.Enumeration;

import javax.media.j3d.*;
import javax.vecmath.*;

import com.sun.j3d.utils.universe.*;
import com.sun.j3d.utils.geometry.*;

class Waveform {

    // two arrays of Point3d information to speed up execution
    private Point3d[] frontPoint3d = new Point3d[5];
    private Point3d[] backPoint3d  = new Point3d[5];

    // the array of QuadArrays which contain the relevant Geometry for the waveform
    private QuadArray[] quadArray = new QuadArray[6];

    // the array of Shape3d objects which make up the waveform
    private Shape3D[] shape3D = new Shape3D[6];

    private Canvas3D canvas;
    private VirtualUniverse universe;

    /*
     * GraphicsController constructor method which contains the initialisation
     * routine for creating the scene
     */
    Waveform() {

        // set up the GraphicsConfiguration and Canvas3D objects for the
        // rendering of the scene
        canvas = new Canvas3D(SimpleUniverse.getPreferredConfiguration());

        // call the createView method which sets up the View objects contents
        View view = createView(canvas);

        // create the view side of the scenegraph
        Locale locale = createViewBranch(view);

        // create the content side of the scenegraph
        createContentBranch(locale);
    }


    /*
     * return the Canvas3D object for viewing in the GUI
     */
    public Canvas3D getCanvas3D() {
        return canvas;
    }


    /*
     * removes all the Locales from the scene
     */
    public void kill() {
        universe.removeAllLocales();
    }
```

```
/*
 * method which creates a View object and adds to it the Canvas3D, PhysicalBody
 * and PhysicalEnvironment objects
 */
private View createView(Canvas3D canvas) {

    View view = new View();

    view.addCanvas3D(canvas);
    view.setPhysicalBody(new PhysicalBody());
    view.setPhysicalEnvironment(new PhysicalEnvironment());

    view.setFieldOfView(1.0);

    return(view);
}


/*
 * method which sets up the view side of the scenegraph as well as the Locale
 * and VirtualUniverse
 */
private Locale createViewBranch(View view) {

    // create scenegraph objects from the top downwards
    universe     = new VirtualUniverse();
    Locale locale                = new Locale(universe);
    BranchGroup viewBranch       = new BranchGroup();
    TransformGroup viewTransform = new TransformGroup();
    ViewPlatform viewPlatform    = new ViewPlatform();

    // set capability bits to allow modification at runtime
    viewTransform.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    viewTransform.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
    viewPlatform.setCapability(ViewPlatform.ALLOW_POLICY_WRITE);
    viewPlatform.setViewAttachPolicy(View.RELATIVE_TO_FIELD_OF_VIEW);

    viewPlatform.setActivationRadius(100.0f);

    // link together the scenegraph from the bottom upwards
    viewTransform.addChild(viewPlatform);
    viewBranch.addChild(viewTransform);
    locale.addBranchGraph(viewBranch);

    // complete the chain by adding the ViewPlatform to the View
    view.attachViewPlatform(viewPlatform);

    // move the view
    Transform3D viewTransform3D = new Transform3D();
    viewTransform3D.rotX(Math.PI*0.11);
    viewTransform3D.setTranslation(new Vector3d(0.0, -3.7, -23.0));
    viewTransform3D.invert();
    viewTransform.setTransform(viewTransform3D);

    // return the Locale which contains a link to the whole chain
    return locale;
}


/*
 * create the content side of the scenegraph and attach it to the locale
 */
private void createContentBranch(Locale locale) {

    BranchGroup contentBranch = new BranchGroup();

    createWaveform(contentBranch);

    addParticles(contentBranch);

    // add the lights to the scene
    contentBranch.addChild(addLights());
```

```
        // perform optimisations on the contentBranch
        contentBranch.compile();

        // add the contentBranch to the locale object to complete the tree
        locale.addBranchGraph(contentBranch);
    }


    public void update(double[] frequencyBands) {

        // convert the frequencyBands data into Point3d values
        for (int i = 0; i < frontPoint3d.length; i++) {
            frontPoint3d[i].y = 6*frequencyBands[i];
            backPoint3d[i].y = frontPoint3d[i].y;
        }

        // apply the new values to the Shape3D objects
        quadArray[0].setCoordinate(2, frontPoint3d[0]);
        quadArray[0].setCoordinate(3, backPoint3d[0]);

        for (int i = 1; i < 5; i++) {
            quadArray[i].setCoordinate(0, backPoint3d[i-1]);
            quadArray[i].setCoordinate(1, frontPoint3d[i-1]);
            quadArray[i].setCoordinate(2, frontPoint3d[i]);
            quadArray[i].setCoordinate(3, backPoint3d[i]);
        }

        quadArray[5].setCoordinate(0, backPoint3d[4]);
        quadArray[5].setCoordinate(1, frontPoint3d[4]);
    }


    /*
     * method which creates a 3D waveform and places it in the contentBranch
     */
    public void createWaveform(BranchGroup contentBranch) {

        // create the appearance information for the Shape3D objects
        Appearance appearance = new Appearance();
        Color3f objectColor = new Color3f(1.0f, 0.0f, 0.0f);
        Color3f darkColor   = new Color3f(0.0f, 0.0f, 0.0f);
        Material material = new Material(objectColor, darkColor,
                                         objectColor, darkColor, 80.0f);
        appearance.setMaterial(material);

        // set the appearance's PolygonAttributes
        PolygonAttributes polygonAttributes = new PolygonAttributes();
        polygonAttributes.setCullFace(PolygonAttributes.CULL_NONE);
        appearance.setPolygonAttributes(polygonAttributes);

        // intialise the QuadArray objects
        for (int i = 0; i < quadArray.length; i++) {
            quadArray[i] = new QuadArray(4, QuadArray.COORDINATES | QuadArray.COLOR_3);
            quadArray[i].setCapability(GeometryArray.ALLOW_COORDINATE_WRITE);
        }

        // intialise the Point3d arrays
        for (int i = 0; i < frontPoint3d.length; i++) {
            frontPoint3d[i] = new Point3d();
            backPoint3d[i]  = new Point3d();
        }

        // initialise the positions of the Shape3D objects. the end point don't
        // move and are never changed
        // quadArray[0]
        Point3d endPoints = new Point3d();
        endPoints.set(-9.0, 0.0, -5.0);
        quadArray[0].setCoordinate(0, endPoints);
        endPoints.set(-9.0, 0.0, 0.0);
        quadArray[0].setCoordinate(1, endPoints);
        frontPoint3d[0].set(-6.0, 0.0, 0.0);
        quadArray[0].setCoordinate(2, frontPoint3d[0]);
        backPoint3d[0].set(-6.0, 0.0, -5.0);
        quadArray[0].setCoordinate(3, backPoint3d[0]);
        // quadArray[1]
        quadArray[1].setCoordinate(0, backPoint3d[0]);
        quadArray[1].setCoordinate(1, frontPoint3d[0]);
        frontPoint3d[1].set(-3.0, 0.0, 0.0);
        quadArray[1].setCoordinate(2, frontPoint3d[1]);
        backPoint3d[1].set(-3.0, 0.0, -5.0);
        quadArray[1].setCoordinate(3, backPoint3d[1]);
        // quadArray[2]
        quadArray[2].setCoordinate(0, backPoint3d[1]);
        quadArray[2].setCoordinate(1, frontPoint3d[1]);
        frontPoint3d[2].set(0.0, 0.0, 0.0);
        quadArray[2].setCoordinate(2, frontPoint3d[2]);
        backPoint3d[2].set(0.0, 0.0, -5.0);
        quadArray[2].setCoordinate(3, backPoint3d[2]);
        // quadArray[3]
        quadArray[3].setCoordinate(0, backPoint3d[2]);
        quadArray[3].setCoordinate(1, frontPoint3d[2]);
        frontPoint3d[3].set(3.0, 0.0, 0.0);
        quadArray[3].setCoordinate(2, frontPoint3d[3]);
        backPoint3d[3].set(3.0, 0.0, -5.0);
        quadArray[3].setCoordinate(3, backPoint3d[3]);
        // quadArray[4]
        quadArray[4].setCoordinate(0, backPoint3d[3]);
        quadArray[4].setCoordinate(1, frontPoint3d[3]);
        frontPoint3d[4].set(6.0, 0.0, 0.0);
        quadArray[4].setCoordinate(2, frontPoint3d[4]);
        backPoint3d[4].set(6.0, 0.0, -5.0);
        quadArray[4].setCoordinate(3, backPoint3d[4]);
        // quadArray[5]
        quadArray[5].setCoordinate(0, backPoint3d[3]);
        quadArray[5].setCoordinate(1, frontPoint3d[3]);
        endPoints.set(9.0, 0.0, 0.0);
        quadArray[5].setCoordinate(2, endPoints);
        endPoints.set(9.0, 0.0, -5.0);
        quadArray[5].setCoordinate(3, endPoints);


        for (int i = 0; i < quadArray.length; i++) {
            for (int j = 0; j < 4; j++) {
            quadArray[i].setColor(j, objectColor);
            }
        }

        // Apply the QuadArrays and appearance to the Shape3D objects and add
        // to the BranchGraph object
        for (int i = 0; i < shape3D.length; i++) {
            shape3D[i] = new Shape3D(quadArray[i], appearance);
            contentBranch.addChild(shape3D[i]);
        }
    }


    /*
     * create a particle object which spirals around the waveform
     */
    private void addParticles(BranchGroup contentBranch) {

        // create the appearance information for the sphere objects
        Appearance appearance = new Appearance();
        Color3f objectColor = new Color3f(0.0f, 0.0f, 1.0f);
        Color3f darkColor   = new Color3f(0.0f, 0.0f, 0.0f);
        Material material = new Material(objectColor, darkColor,
                                         objectColor, darkColor, 80.0f);
        appearance.setMaterial(material);

        // create the sphere objects
        Sphere sphere1 = new Sphere(0.2f, Primitive.GENERATE_NORMALS, 10, appearance);
        Sphere sphere2 = new Sphere(0.2f, Primitive.GENERATE_NORMALS, 10, appearance);

        // create the TransformGroups to house the sphere and interpolator objects
        TransformGroup transformGroup1 = new TransformGroup();
        transformGroup1.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
        transformGroup1.addChild(sphere1);
        TransformGroup transformGroup2 = new TransformGroup();
        transformGroup2.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
        transformGroup2.addChild(sphere2);

        // create the first interpolator
        Transform3D interpolatorTransform3D1 = new Transform3D();
        Alpha alpha = new Alpha(-1,Alpha.INCREASING_ENABLE | Alpha.DECREASING_ENABLE,
                                0, 0, 20000, 0, 0, 20000, 0, 0);

        SpiralInterpolator spiralInterpolator1 =
            new SpiralInterpolator(alpha, transformGroup1, interpolatorTransform3D1);
        BoundingSphere bounds1 = new BoundingSphere(new Point3d(0.0,0.0,0.0), 100.0);
        spiralInterpolator1.setSchedulingBounds(bounds1);
        transformGroup1.addChild(spiralInterpolator1);

        // create the second interpolator
        Transform3D interpolatorTransform3D2 = new Transform3D();
        //Alpha alpha = new Alpha(-1,Alpha.INCREASING_ENABLE | Alpha.DECREASING_ENABLE,
        //                        0, 0, 20000, 0, 0, 20000, 0, 0);

        SpiralInterpolator spiralInterpolator2 =
            new SpiralInterpolator(alpha, transformGroup2, interpolatorTransform3D2);
        BoundingSphere bounds = new BoundingSphere(new Point3d(0.0,0.0,0.0), 100.0);
        spiralInterpolator2.setSchedulingBounds(bounds);
        // make the second sphere move in antiphase
        spiralInterpolator2.setSpiralDiameterCoefficient(-5.0);
        transformGroup2.addChild(spiralInterpolator2);

        contentBranch.addChild(transformGroup1);
        contentBranch.addChild(transformGroup2);
    }






    /*
     * add many lights to the scene
     */
    private BranchGroup addLights() {

        BranchGroup lightBranch = new BranchGroup();

        // create a color and direction for the light
        Color3f color = new Color3f(0.8f, 0.8f, 0.8f);
        Vector3f direction1 = new Vector3f(-1.0f, -2.0f, -1.0f);
        Vector3f direction2 = new Vector3f(1.0f, -1.0f, -1.0f);

        // create a DirectionalLight using the color and direction specified
        DirectionalLight light1 = new DirectionalLight(color, direction1);
        DirectionalLight light2 = new DirectionalLight(color, direction2);

        // create the influencing bounds of the light
        BoundingSphere bounds =
              new BoundingSphere(new Point3d(0.0,0.0,0.0), 200.0);
        light1.setInfluencingBounds(bounds);
        light2.setInfluencingBounds(bounds);

        // add the light to the scene
        lightBranch.addChild(light1);
        lightBranch.addChild(light2);
        return lightBranch;
    }

    /*
     * inner class which specifies the a spiral path to an object
     */
    private class SpiralInterpolator extends TransformInterpolator {

        private double x, y, z;
        private double xAxisOffset = -9.0;
        private double yAxisOffset = 3.5;
        private double zAxisOffset = -2.5;
        private double lengthCoefficient = 2*Math.abs(xAxisOffset);
        private double spiralDensity = 36.0;
        private double spiralDiameterCoefficient = 5.0;


        SpiralInterpolator(Alpha alpha, TransformGroup target, Transform3D axisOfTransform) {

            super(alpha, target, axisOfTransform);
        }

        public void setSpiralDiameterCoefficient(double coefficient) {
            spiralDiameterCoefficient = coefficient;
        }


        // method which simulates the motion of the object about a spiral path
        // using the equations x = t, y = sin(6t) and z = cos(6t)
        public void computeTransform(float alphaValue, Transform3D transform) {

            // calculate the x, y and z values
            x = lengthCoefficient*alphaValue;
            y = spiralDiameterCoefficient*Math.sin(spiralDensity*alphaValue);
            z = spiralDiameterCoefficient*Math.cos(spiralDensity*alphaValue);

            // apply the calculations
            transform.setTranslation(new Vector3d(x + xAxisOffset, y + yAxisOffset, z +
zAxisOffset));
        }
    }
}
```

D.5 BouncingParticle.java

```java
/*
 * This is class creates a 3D scene with 5 bouncing spheres and particles
 *
 * Nicholas Martin
 * April 2004
 */

import java.awt.*;
import java.awt.event.*;
import java.util.Enumeration;

import javax.media.j3d.*;
import javax.vecmath.*;

import com.sun.j3d.utils.universe.*;
import com.sun.j3d.utils.geometry.*;

class BouncingParticle {

    // arrays containing the scenegraph objects for the bouncing balls
    private Alpha[] alphaArray = new Alpha[5];
    private GravityInterpolator[] gravityInterpolatorArray = new GravityInterpolator[5];
    private TransformGroup[] ballArray = new TransformGroup[5];

    private ParticleInterpolator[] particleInterpolatorArray = new ParticleInterpolator[5];
    private TransformGroup[] particleArray = new TransformGroup[5];

    private final double maxHeight = 4.9;

    private Canvas3D canvas;
    private VirtualUniverse universe;

    /*
     * GraphicsController constructor method which contains the
     * initialisation routine for creating the scene
     */
    BouncingParticle() {

        // set up the GraphicsConfiguration and Canvas3D objects for the
        // rendering of the scene and make it visible
        canvas = new Canvas3D(SimpleUniverse.getPreferredConfiguration());

        // call the createView method which sets up the View objects contents
        View view = createView(canvas);

        // create the view side of the scenegraph
        Locale locale = createViewBranch(view);

        // create the content side of the scenegraph
        createContentBranch(locale);
    }


    /*
     * return the Canvas3D object for viewing in the GUI
     */
    public Canvas3D getCanvas3D() {
        return canvas;
    }


    /*
     * removes all the Locales from the scene
     */
    public void kill() {
        universe.removeAllLocales();
    }
```

```
/*
 * method which creates a View object and adds to it the Canvas3D, PhysicalBody
 * and PhysicalEnvironment objects
 */
private View createView(Canvas3D canvas) {

    View view = new View();

    view.addCanvas3D(canvas);
    view.setPhysicalBody(new PhysicalBody());
    view.setPhysicalEnvironment(new PhysicalEnvironment());

    view.setFieldOfView(1.4);

    return(view);
}


/*
 * method which sets up the view side of the scenegraph as well as the Locale
 * and VirtualUniverse
 */
private Locale createViewBranch(View view) {

    // create scenegraph objects from the top downwards
    universe     = new VirtualUniverse();
    Locale locale                = new Locale(universe);
    BranchGroup viewBranch       = new BranchGroup();
    TransformGroup viewTransform = new TransformGroup();
    ViewPlatform viewPlatform    = new ViewPlatform();

    // set capability bits to allow modification at runtime
    viewTransform.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    viewTransform.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
    viewPlatform.setCapability(ViewPlatform.ALLOW_POLICY_WRITE);
    viewPlatform.setViewAttachPolicy(View.RELATIVE_TO_FIELD_OF_VIEW);

    viewPlatform.setActivationRadius(100.0f);

    // link together the scenegraph from the bottom upwards
    viewTransform.addChild(viewPlatform);
    viewBranch.addChild(viewTransform);
    locale.addBranchGraph(viewBranch);

    // complete the chain by adding the ViewPlatform to the View
    view.attachViewPlatform(viewPlatform);

    // move the view
    Transform3D viewTransform3D = new Transform3D();
    viewTransform3D.setTranslation(new Vector3d(0.0, 2.5, 9.0));
    viewTransform.setTransform(viewTransform3D);

    // return the Locale which contains a link to the whole chain
    return locale;
}


/*
 * create the content side of the scenegraph and attach it to the locale
 */
private void createContentBranch(Locale locale) {

    BranchGroup contentBranch = new BranchGroup();

    // create TransformGroups which allow the spheres to be generically
    // created and then moved
    for (int i = 0; i < ballArray.length; i++) {
        ballArray[i] = new TransformGroup();
    }

    for (int i = 0; i < ballArray.length; i++) {
        ballArray[i].setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
        ballArray[i].setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
        }


        // assign positioning values to the Transform3D objects
        Transform3D[] ballPosition = new Transform3D[5];
        ballPosition[0] = new Transform3D();
        ballPosition[0].setTranslation(new Vector3d(-7.0, 0.0, 0.0));

        ballPosition[1] = new Transform3D();
        ballPosition[1].setTranslation(new Vector3d(-3.5, 0.0, 0.0));

        ballPosition[2] = new Transform3D(); // Sphere number 3 does not need to be moved

        ballPosition[3] = new Transform3D();
        ballPosition[3].setTranslation(new Vector3d(3.5, 0.0, 0.0));

        ballPosition[4] = new Transform3D();
        ballPosition[4].setTranslation(new Vector3d(7.0, 0.0, 0.0));

        // set the GravityInterpolator objects created at the top of the class
        // as well as their assigned Alpha objects this allows access during the
        // update method
        for (int i = 0; i < gravityInterpolatorArray.length; i++) {
            gravityInterpolatorArray[i] = createBall(ballArray[i], ballPosition[i]);
            alphaArray[i] = gravityInterpolatorArray[i].getAlpha();
        }

        // fill the particles' TransformGroups
        for (int i = 0; i < particleArray.length; i++) {
            particleArray[i] = new TransformGroup();
        }

        // assign positioning values for the particles
        Transform3D[] particleTransform3D = new Transform3D[5];

        particleTransform3D[0] = new Transform3D();
        particleTransform3D[0].setTranslation(new Vector3d(-7.0, -0.8, 0.0));

        particleTransform3D[1] = new Transform3D();
        particleTransform3D[1].setTranslation(new Vector3d(-3.5, -0.8, 0.0));

        particleTransform3D[2] = new Transform3D();
        particleTransform3D[2].setTranslation(new Vector3d(0.0, -0.8, 0.0));

        particleTransform3D[3] = new Transform3D();
        particleTransform3D[3].setTranslation(new Vector3d(3.5, -0.8, 0.0));

        particleTransform3D[4] = new Transform3D();
        particleTransform3D[4].setTranslation(new Vector3d(7.0, -0.8, 0.0));


        // create the particles
        for (int i = 0; i < particleArray.length; i++) {
            //particleInterpolatorArray[i] =
                createParticle(particleArray[i], particleTransform3D[i],
gravityInterpolatorArray[i], alphaArray[i]);
        }

        // finally, add the TransformGroups to the contentBranch
        for (int i = 0; i < ballArray.length; i++) {
            contentBranch.addChild(ballArray[i]);
        }

        for (int i = 0; i < particleArray.length; i++) {
            contentBranch.addChild(particleArray[i]);
        }

        // add the lights to the scene
        contentBranch.addChild(addLights());

        // perform optimisations on the contentBranch
        contentBranch.compile();

        // add the contentBranch to the locale object to complete the tree
        locale.addBranchGraph(contentBranch);
    }


    /*
     * method called to update the contents of the scene with new coefficients
     */
    public void update(double[] frequencyBands) {

        for (int i = 0; i < ballArray.length; i++) {

            // check to see if the ball is falling
            if (gravityInterpolatorArray[i].isFalling(alphaArray[i].value()) == true) {

                // find the height suggested by the co-efficient.  If this is greater
                // than the current value the interpolator and alpha are reset
                double suggestedHeight = frequencyBands[i]*maxHeight;
                double displacement =
                    gravityInterpolatorArray[i].getDisplacement(alphaArray[i].value());

                if (suggestedHeight > displacement) {
                    gravityInterpolatorArray[i].setStartDisplacement(displacement);
                    gravityInterpolatorArray[i].setPeakHeight(suggestedHeight);
                    alphaArray[i].setStartTime(System.currentTimeMillis());
                }
            }
        }
    }


    /*
     * return the ScaleInterpolator which is attached to the sent
     * TransformGroup Sphere object
     */
    private GravityInterpolator createBall(TransformGroup upperTransformGroup,
                                           Transform3D transform3D) {

        // set an extra TransformGroup to attach the Sphere and Interpolator
        TransformGroup lowerTransformGroup = new TransformGroup();
        lowerTransformGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);

        // create an Appearance object to hold information on the
        // Sphere's appearance in the scene
        Appearance appearance = new Appearance();

        // set two colours to create the Material object
        Color3f objectColor = new Color3f(0.1f, 0.9f, 0.1f);
        Color3f darkColor   = new Color3f(0.0f, 0.0f, 0.0f);
        Material material = new Material(objectColor, darkColor,
                                         objectColor, darkColor, 80.0f);
        material.setCapability(Material.ALLOW_COMPONENT_WRITE);

        // this may then be assigned to the Appearance
        appearance.setMaterial(material);

        // the Sphere may now be created using the Primitive Sphere class
        Sphere sphere = new Sphere(0.5f, Primitive.GENERATE_NORMALS, 20, appearance);

        // add the lower TransformGroup to the upper TransformGroup
        // then the Sphere can be added to the lower TransformGroup
        upperTransformGroup.addChild(lowerTransformGroup);
        lowerTransformGroup.addChild(sphere);

        // position the sphere in the scene
        upperTransformGroup.setTransform(transform3D);

        // apply a GravityInterpolator to the sphere
        Transform3D interpolatorTransform3D = new Transform3D();
        interpolatorTransform3D.rotZ(0.5 * Math.PI);

        Alpha alpha = new Alpha(1,1000);

        GravityInterpolator pulseInterpolator =
            new GravityInterpolator(alpha, lowerTransformGroup, interpolatorTransform3D);
        BoundingSphere bounds = new BoundingSphere(new Point3d(0.0,0.0,0.0), 100.0);
        pulseInterpolator.setSchedulingBounds(bounds);

        lowerTransformGroup.addChild(pulseInterpolator);

        return pulseInterpolator;
    }


    /*
     * return the ScaleInterpolator which is attached to the sent
     * TransformGroup Sphere object
     */
    private void createParticle(TransformGroup upperTransformGroup, Transform3D transform3D,
                                GravityInterpolator targetInterpolator, Alpha targetAlpha) {

        // set an extra TransformGroup to attach the particle and Interpolator
        TransformGroup lowerTransformGroup = new TransformGroup();
        lowerTransformGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);

        // create an Appearance object to hold information on the
        // particle's appearance in the scene
        Appearance appearance = new Appearance();

        // set two colours to create the Material object
        Color3f objectColor = new Color3f(0.1f, 0.9f, 0.1f);
        Color3f darkColor   = new Color3f(0.0f, 0.0f, 0.0f);
        Material material = new Material(objectColor, darkColor,
                                         objectColor, darkColor, 80.0f);
        material.setCapability(Material.ALLOW_COMPONENT_WRITE);

        // this may then be assigned to the Appearance
        appearance.setMaterial(material);

        // the particle may now be created using the Primitive Box class
        Sphere sphere = new Sphere(0.1f, Primitive.GENERATE_NORMALS, 10, appearance);

        // add the lower TransformGroup to the upper TransformGroup
        // then the particle can be added to the lower TransformGroup
        upperTransformGroup.addChild(lowerTransformGroup);
        lowerTransformGroup.addChild(sphere);

        // position the particle in the scene
        upperTransformGroup.setTransform(transform3D);

        // apply a ScaleInterpolator to the particle
        Transform3D interpolatorTransform3D = new Transform3D();

        Alpha alpha = new Alpha(-1, 2000);

        ParticleInterpolator particleInterpolator =
            new ParticleInterpolator(alpha, lowerTransformGroup, interpolatorTransform3D);

        particleInterpolator.setTargetParameters(targetInterpolator, targetAlpha);
        BoundingSphere bounds = new BoundingSphere(new Point3d(0.0,0.0,0.0), 100.0);
        particleInterpolator.setSchedulingBounds(bounds);

        lowerTransformGroup.addChild(particleInterpolator);
    }


    /*
     * add many lights to the scene
     */
    private BranchGroup addLights() {

        BranchGroup lightBranch = new BranchGroup();

        // create a color and direction for the light
        Color3f color = new Color3f(0.8f, 0.8f, 0.8f);
        Vector3f direction1 = new Vector3f(-1.0f, -2.0f, -1.0f);
        Vector3f direction2 = new Vector3f(1.0f, -1.0f, -1.0f);

        // create a DirectionalLight using the color and direction specified
        DirectionalLight light1 = new DirectionalLight(color, direction1);
        DirectionalLight light2 = new DirectionalLight(color, direction2);

        // create the influencing bounds of the light
        BoundingSphere bounds =
              new BoundingSphere(new Point3d(0.0,0.0,0.0), 200.0);
        light1.setInfluencingBounds(bounds);
        light2.setInfluencingBounds(bounds);

        // add the light to the scene
        lightBranch.addChild(light1);
        lightBranch.addChild(light2);
        return lightBranch;
    }


    /*
     * inner class to specify the elastic motion of the particle
     */
    private class ParticleInterpolator extends TransformInterpolator {

        private double u = 0.0;  // default value as the particle always starts at rest

        private double impact = 1.0;

        private double time = 0.0;
        private double alphaBuffer = 0.0;
        private double displacementBuffer = 0.0;

        private Alpha targetAlpha;
        private GravityInterpolator targetInterpolator;

        ParticleInterpolator(Alpha alpha, TransformGroup target,
                             Transform3D axisOfTransform) {

            super(alpha, target, axisOfTransform);
        }

        public void setTargetParameters(GravityInterpolator gravityInterpolator, Alpha alpha) {
            targetInterpolator = gravityInterpolator;
            targetAlpha = alpha;
        }


        public void computeTransform(float alphaValue, Transform3D transform) {
            double a = -9.8;

            // find the value of the natural position and the vector containing
            // the current position
            double naturalPosition = targetInterpolator.getDisplacement(targetAlpha.value());


            // calculate the time since the last method call
            if (alphaValue < alphaBuffer) {
                time = alphaValue + (1.0f - alphaBuffer);
            }
            else {
                time = alphaValue - alphaBuffer;
            }

            alphaBuffer = alphaValue;

            double x = naturalPosition - displacementBuffer;

            // calculate the acceleration based on the position and direction
            // of the particle
            if (x >= 1.5 && u <= 0.0) {
                a = 15.0;
            }
            else if (x >= 1.5 && u > 0.0) {
                a = 6.0;
            }

            // calculate the new displacement
            displacementBuffer = (u*time) + (0.5f*a*(time*time)) + displacementBuffer;

            // calculate the velocity for the update above the next time this method is called
            u = (u + a*time);

            transform.setTranslation(new Vector3d(0.0, impact*displacementBuffer, 0.0));
        }
    }
}
```

D.6 BouncingLights.java

```
/*
 * This is class creates a 3D scene with 5 bouncing spheres with orbiting lights
 *
 * Nicholas Martin
 * April 2004
 */

import java.awt.*;
import java.awt.event.*;
import java.util.Enumeration;

import javax.media.j3d.*;
import javax.vecmath.*;

import com.sun.j3d.utils.universe.*;
import com.sun.j3d.utils.geometry.*;

class BouncingLights {

    // arrays containing the scenegraph objects for the bouncing spheres
    private Alpha[] alphaArray = new Alpha[5];
    private GravityInterpolator[] gravityInterpolatorArray = new GravityInterpolator[5];
    private TransformGroup[] ballArray = new TransformGroup[5];

    private RotationInterpolator rotationInterpolator;

    private final double maxHeight = 4.9;

    private Canvas3D canvas;
    private VirtualUniverse universe;

    /*
     * GraphicsController constructor method which contains the
     * initialisation routine for creating the scene
     */
    BouncingLights() {

        // set up the GraphicsConfiguration and Canvas3D objects for the
        // rendering of the scene and make it visible
        canvas = new Canvas3D(SimpleUniverse.getPreferredConfiguration());

        // call the createView method which sets up the View objects contents
        View view = createView(canvas);

        // create the view side of the scenegraph
        Locale locale = createViewBranch(view);

        // create the content side of the scenegraph
        createContentBranch(locale);
    }


    /*
     * return the Canvas3D object for viewing in the GUI
     */
    public Canvas3D getCanvas3D() {
        return canvas;
    }


    /*
     * removes all the Locales from the scene
     */
    public void kill() {
        universe.removeAllLocales();
    }
```

```
/*
 * method which creates a View object and adds to it the Canvas3D, PhysicalBody
 * and PhysicalEnvironment objects
 */
private View createView(Canvas3D canvas) {

    View view = new View();

    view.addCanvas3D(canvas);
    view.setPhysicalBody(new PhysicalBody());
    view.setPhysicalEnvironment(new PhysicalEnvironment());

    view.setFieldOfView(1.4);

    return(view);
}


/*
 * method which sets up the view side of the scenegraph as well as the Locale
 * and VirtualUniverse
 */
private Locale createViewBranch(View view) {

    // create scenegraph objects from the top downwards
    universe     = new VirtualUniverse();
    Locale locale                = new Locale(universe);
    BranchGroup viewBranch       = new BranchGroup();
    TransformGroup viewTransform = new TransformGroup();
    ViewPlatform viewPlatform    = new ViewPlatform();

    // set capability bits to allow modification at runtime
    viewTransform.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    viewTransform.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
    viewPlatform.setCapability(ViewPlatform.ALLOW_POLICY_WRITE);
    viewPlatform.setViewAttachPolicy(View.RELATIVE_TO_FIELD_OF_VIEW);

    viewPlatform.setActivationRadius(100.0f);

    // link together the scenegraph from the bottom upwards
    viewTransform.addChild(viewPlatform);
    viewBranch.addChild(viewTransform);
    locale.addBranchGraph(viewBranch);

    // complete the chain by adding the ViewPlatform to the View
    view.attachViewPlatform(viewPlatform);

    // move the view
    Transform3D viewTransform3D = new Transform3D();
    viewTransform3D.setTranslation(new Vector3d(0.0, 2.5, 11.0));
    viewTransform.setTransform(viewTransform3D);

    // return the Locale which contains a link to the whole chain
    return locale;
}


/*
 * create the content side of the scenegraph and attach it to the locale
 */
private void createContentBranch(Locale locale) {

    BranchGroup contentBranch = new BranchGroup();

    // create TransformGroups which allow the spheres to be generically
    // created and then moved
    for (int i = 0; i < ballArray.length; i++) {
        ballArray[i] = new TransformGroup();
    }

    for (int i = 0; i < ballArray.length; i++) {
        ballArray[i].setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
        ballArray[i].setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
    }
```

```
        // assign positioning values to the sphere objects
        Transform3D[] ballPosition = new Transform3D[5];
        ballPosition[0] = new Transform3D();
        ballPosition[0].setTranslation(new Vector3d(-7.0, 0.0, 0.0));

        ballPosition[1] = new Transform3D();
        ballPosition[1].setTranslation(new Vector3d(-3.5, 0.0, 0.0));

        ballPosition[2] = new Transform3D(); // sphere number 3 does not need to be moved

        ballPosition[3] = new Transform3D();
        ballPosition[3].setTranslation(new Vector3d(3.5, 0.0, 0.0));

        ballPosition[4] = new Transform3D();
        ballPosition[4].setTranslation(new Vector3d(7.0, 0.0, 0.0));


        // set up the scenegraph elements for the orbiting objects

        TransformGroup[] redSpotLightsGroup = new TransformGroup[5];
        TransformGroup[] blueSpotLightsGroup = new TransformGroup[5];
        for (int i = 0; i < redSpotLightsGroup.length; i++) {
            redSpotLightsGroup[i] = new TransformGroup();
            blueSpotLightsGroup[i] = new TransformGroup();
        }

        Transform3D redSpotLightTransform3D = new Transform3D();
        redSpotLightTransform3D.setTranslation(new Vector3d(-1.0, -1.0, 0.0));
        Transform3D blueSpotLightTransform3D = new Transform3D();
        blueSpotLightTransform3D.setTranslation(new Vector3d(1.0, -1.0, 0.0));

        for (int i = 0; i < redSpotLightsGroup.length; i++) {
            createRedSpotLight(redSpotLightsGroup[i], redSpotLightTransform3D);
            createBlueSpotLight(blueSpotLightsGroup[i], blueSpotLightTransform3D);
        }

        // set the GravityInterpolator objects created at the top of the class
        // as well as their assigned Alpha objects this allows access during the
        // update method
        for (int i = 0; i < gravityInterpolatorArray.length; i++) {
            gravityInterpolatorArray[i] = createBall(ballArray[i], ballPosition[i],
                                                      redSpotLightsGroup[i],
blueSpotLightsGroup[i]);
            alphaArray[i] = gravityInterpolatorArray[i].getAlpha();
        }


        // finally, add the TransformGroups to the contentBranch
        for (int i = 0; i < ballArray.length; i++) {
            contentBranch.addChild(ballArray[i]);
        }

        // add the lights to the scene
        contentBranch.addChild(addLights());

        // perform optimisations on the contentBranch
        contentBranch.compile();

        // add the contentBranch to the locale object to complete the tree
        locale.addBranchGraph(contentBranch);
    }


    /*
     * method called to update the contents of the scene with new coefficients
     */
    public void update(double[] frequencyBands) {

        for (int i = 0; i < ballArray.length; i++) {

            // check to see if the ball is falling
            if (gravityInterpolatorArray[i].isFalling(alphaArray[i].value()) == true) {
```

```
            // find the height suggested by the co-efficient.  If this is greater
            // than the current value the interpolator and alpha are reset
            double suggestedHeight = frequencyBands[i]*maxHeight;
            double displacement =
                gravityInterpolatorArray[i].getDisplacement(alphaArray[i].value());

            if (suggestedHeight > displacement) {
                gravityInterpolatorArray[i].setStartDisplacement(displacement);
                gravityInterpolatorArray[i].setPeakHeight(suggestedHeight);
                alphaArray[i].setStartTime(System.currentTimeMillis());
            }
        }
    }
}


/*
 * return the ScaleInterpolator which is attached to the sent TransformGroup sphere object
 */
private GravityInterpolator createBall(TransformGroup upperTransformGroup,
                                       Transform3D transform3D,
                                       TransformGroup redSpotLightGroup,
                                       TransformGroup blueSpotLightGroup) {

    // set an extra TransformGroup to attach the sphere and Interpolator
    TransformGroup lowerTransformGroup = new TransformGroup();
    lowerTransformGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);

    // create an Appearance object to hold information on the
    // sphere's appearance in the scene
    Appearance appearance = new Appearance();

    // set two colours to create the Material object
    Color3f objectColor = new Color3f(0.4f, 0.4f, 0.4f);
    Color3f darkColor   = new Color3f(0.0f, 0.0f, 0.0f);
    Material material = new Material(objectColor, darkColor,
                                     objectColor, darkColor, 80.0f);
    material.setCapability(Material.ALLOW_COMPONENT_WRITE);

    // this may then be assigned to the Appearance
    appearance.setMaterial(material);

    // the sphere may now be created using the Primitive Box class
    Sphere sphere = new Sphere(1.0f, Primitive.GENERATE_NORMALS, 20, appearance);

    // add the lower TransformGroup to the upper TransformGroup
    // then the sphere can be added to the lower TransformGroup
    upperTransformGroup.addChild(lowerTransformGroup);
    lowerTransformGroup.addChild(sphere);

    // position the sphere in the scene
    upperTransformGroup.setTransform(transform3D);

    // apply a ScaleInterpolator to the sphere
    Transform3D interpolatorTransform3D = new Transform3D();
    interpolatorTransform3D.rotZ(0.5 * Math.PI);

    Alpha alpha = new Alpha(1,1000);

    GravityInterpolator pulseInterpolator =
        new GravityInterpolator(alpha, lowerTransformGroup, interpolatorTransform3D);
    BoundingSphere bounds = new BoundingSphere(new Point3d(0.0,0.0,0.0), 100.0);
    pulseInterpolator.setSchedulingBounds(bounds);

    lowerTransformGroup.addChild(pulseInterpolator);
    lowerTransformGroup.addChild(redSpotLightGroup);
    lowerTransformGroup.addChild(blueSpotLightGroup);

    return pulseInterpolator;
}
```

```
    /*
     * return the ScaleInterpolator which is attached to the sent TransformGroup sphere object
     */
    private void createRedSpotLight(TransformGroup upperTransformGroup, Transform3D transform3D) {

        // set an extra TransformGroup to attach the Box and Interpolator to
        TransformGroup lowerTransformGroup = new TransformGroup();
        lowerTransformGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);

        // create an Appearance object to hold information on the
        // fake spotlight's appearance in the scene
        Appearance redAppearance = new Appearance();

        // set colour to assign to the red spot light object
        Color3f redColor = new Color3f(1.0f, 0.0f, 0.0f);
        Color3f redDarkColor   = new Color3f(1.0f, 0.0f, 0.0f);
        Material redMaterial = new Material(redColor, redDarkColor,
                                            redColor, redDarkColor, 100.0f);

        // create the red spotlight
        SpotLight redSpotLight = new SpotLight(redColor,
                                               new Point3f(0.0f, 0.0f, 0.0f),
                                               new Point3f(1.0f, 0.0f, 0.0f),
                                               new Vector3f(-1.0f, 0.0f, 0.0f),
                                               (float)Math.PI,
                                               10.0f);

        // keep the bounds low to prevent the slowing the system
        BoundingSphere lightBounds = new BoundingSphere(new Point3d(0.0,0.0,0.0), 1.0);
        redSpotLight.setInfluencingBounds(lightBounds);

        // create the fake red spotlight
        redAppearance.setMaterial(redMaterial);
        Sphere fakeRedSpotLight = new Sphere(0.1f, Primitive.GENERATE_NORMALS, 7, redAppearance);

        // add the lower TransformGroup to the upper TransformGroup
        // then the particle can be added to the lower TransformGroup
        upperTransformGroup.addChild(lowerTransformGroup);
        lowerTransformGroup.addChild(redSpotLight);
        lowerTransformGroup.addChild(fakeRedSpotLight);

        // position the particle in the scene
        upperTransformGroup.setTransform(transform3D);

        // apply a ScaleInterpolator to the box
        Transform3D interpolatorTransform3D = new Transform3D();
        interpolatorTransform3D.rotZ(Math.PI*0.25);
        interpolatorTransform3D.setTranslation(new Vector3d(2.0, 0.0, 0.0));

        Alpha alpha = new Alpha(-1, 800);

        rotationInterpolator =
            new RotationInterpolator(alpha, lowerTransformGroup, interpolatorTransform3D,
                                     0.0f, (float)Math.PI*2.0f);
        BoundingSphere bounds = new BoundingSphere(new Point3d(0.0,0.0,0.0), 100.0);
        rotationInterpolator.setSchedulingBounds(bounds);

        lowerTransformGroup.addChild(rotationInterpolator);
    }


    /*
     * return the ScaleInterpolator which is attached to the sent TransformGroup Box object
     */
    private void createBlueSpotLight(TransformGroup upperTransformGroup, Transform3D transform3D)
{

        // set an extra TransformGroup to attach the Box and Interpolator to
        TransformGroup lowerTransformGroup = new TransformGroup();
        lowerTransformGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
        // create an Appearance object to hold information on the
        // fake spotlight's appearance in the scene
        Appearance blueAppearance = new Appearance();

        // set colour to assign to the red spot light object
        Color3f blueColor = new Color3f(0.0f, 0.0f, 1.0f);
        Color3f blueDarkColor   = new Color3f(0.0f, 0.0f, 1.0f);
        Material blueMaterial = new Material(blueColor, blueDarkColor,
                                         blueColor, blueDarkColor, 100.0f);

        // create the red spotlight
        SpotLight blueSpotLight = new SpotLight(blueColor,
                                                new Point3f(0.0f, 0.0f, 0.0f),
                                                new Point3f(1.0f, 0.0f, 0.0f),
                                                new Vector3f(-1.0f, 0.0f, 0.0f),
                                                (float)Math.PI,
                                                10.0f);

        BoundingSphere lightBounds = new BoundingSphere(new Point3d(0.0,0.0,0.0), 1.0);
        blueSpotLight.setInfluencingBounds(lightBounds);

        // create the fake blue spotlight
        blueAppearance.setMaterial(blueMaterial);
        Sphere fakeBlueSpotLight = new Sphere(0.1f, Primitive.GENERATE_NORMALS, 7,
blueAppearance);

        // add the lower TransformGroup to the upper TransformGroup
        // then the particle can be added to the lower TransformGroup
        upperTransformGroup.addChild(lowerTransformGroup);
        lowerTransformGroup.addChild(blueSpotLight);
        lowerTransformGroup.addChild(fakeBlueSpotLight);

        // position the particle in the scene
        upperTransformGroup.setTransform(transform3D);

        // apply a ScaleInterpolator to the box
        Transform3D interpolatorTransform3D = new Transform3D();
        interpolatorTransform3D.rotZ(-Math.PI*0.25);
        interpolatorTransform3D.setTranslation(new Vector3d(-2.0, 0.0, 0.0));

        Alpha alpha = new Alpha(-1, 800);

        rotationInterpolator =
            new RotationInterpolator(alpha, lowerTransformGroup, interpolatorTransform3D,
                                    0.0f, (float)Math.PI*2.0f);
        BoundingSphere bounds = new BoundingSphere(new Point3d(0.0,0.0,0.0), 100.0);
        rotationInterpolator.setSchedulingBounds(bounds);

        lowerTransformGroup.addChild(rotationInterpolator);
    }


    /*
     * add many lights to the scene
     */
    private BranchGroup addLights() {

        BranchGroup lightBranch = new BranchGroup();

        // create a color and direction for the light
        Color3f color = new Color3f(0.8f, 0.8f, 0.8f);
        Vector3f direction1 = new Vector3f(-1.0f, -2.0f, -1.0f);
        Vector3f direction2 = new Vector3f(1.0f, -1.0f, -1.0f);

        // create a DirectionalLight using the color and direction specified
        DirectionalLight light1 = new DirectionalLight(color, direction1);
        DirectionalLight light2 = new DirectionalLight(color, direction2);

        // create the influencing bounds of the light
        BoundingSphere bounds =
              new BoundingSphere(new Point3d(0.0,0.0,0.0), 200.0);
        light1.setInfluencingBounds(bounds);
        light2.setInfluencingBounds(bounds);
        // add the light to the scene
        lightBranch.addChild(light1);
        lightBranch.addChild(light2);
        return lightBranch;
    }
}
```

D.7 GravityInterpolator.java

```java
/*
 * This class extends TransformInterpolator and provides the motion
 * characteristics of gravity
 *
 * Nicholas Martin
 * April 2004
 */

import javax.media.j3d.*;
import javax.vecmath.*;

public class GravityInterpolator extends TransformInterpolator {

    private final double a = -4.9f;
    private double impact = 4.0f;
    private double offset = 0.0f;
    private double isFallingOffset = -2.5;

    GravityInterpolator(Alpha alpha, TransformGroup target,
                        Transform3D axisOfTransform) {

        super(alpha, target, axisOfTransform);
    }


    // this method sets the impact variable which governs the height of the peak
    public void setPeakHeight(double newPeakHeight) {
        impact = newPeakHeight/1.225;
    }


    // this method returns the height at which the ball will peak
    public double getPeakHeight() {
        return impact*1.225;
    }


    // get the current height displacement of the ball
    public double getDisplacement(float alphaValue) {
        double time = (double)alphaValue + offset;
        return impact*(((-a)*time) + (a*time*time));

    }


    // this method is called to change the position at which the ball starts
    public void setStartDisplacement(double newStartDisplacement) {

        double startDisplacement = newStartDisplacement/impact;

        // calculate the new offset value
        offset = (a + StrictMath.sqrt((a*a) + (4*a*startDisplacement)))/(2*a);
    }


    // by calculating the velocity of the ball it may be found out whether it is
    // falling.  The equation used is v = u + at.
    public boolean isFalling(float alphaValue) {
        double velocity = 2*a*(alphaValue+offset) - a;

        // the velocity is compared to an offset value so the point at which
        // the descent of the ball may be interrputed can be gauged
        if (velocity < isFallingOffset) return true;

        else return false;
    }
    // overwrite this method to calculate the characteristics of the motion of a
    // ball travelling through the air in a vertical line.
    public void computeTransform(float alphaValue, Transform3D transform) {

        // calculate the positioning factor with relation to the alphaValue
        // using the equation s = ut + 1/2at^2 where s is the displacement,
        // u is the initial velocity and t is the time (alphaValue+offset).
        // displacementFactor ranges from 0 to 1.225
        double displacementFactor = getDisplacement(alphaValue);

        // to prevent the ball from falling lower than y = 0 a check must be made
        // this will only be necessary when the offset is not 0.
        if (displacementFactor < 0) {
            displacementFactor = 0.0f;
        }

        // multiply the impact to scale the height at which the ball peaks
        transform.setTranslation(new Vector3d(0.0, displacementFactor, 0.0));
    }
}
```

D.8 FpsBehavior.java

```
/**********************************************************
  Copyright (C) 2001    Daniel Selman

  First distributed with the book "Java 3D Programming"
  by Daniel Selman and published by Manning Publications.
  http://manning.com/selman

  This program is free software; you can redistribute it and/or
  modify it under the terms of the GNU General Public License
  as published by the Free Software Foundation, version 2.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  The license can be found on the WWW at:
  http://www.fsf.org/copyleft/gpl.html

  Or by writing to:
  Free Software Foundation, Inc.,
  59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.

  Authors can be contacted at:
  Daniel Selman: daniel@selman.org

  If you make changes you think others would like, please
  contact one of the authors or someone at the
  www.j3d.org web site.
**************************************************************/



import javax.media.j3d.*;
import javax.vecmath.*;

import com.sun.j3d.utils.geometry.*;

// this class implements a simple behavior that
// output the rendered Frames Per Second.

public class FpsBehavior extends Behavior
{
    // the wake up condition for the behavior
    protected WakeupCondition       m_WakeupCondition = null;
    protected long                  m_StartTime = 0;

    private final int               m_knReportInterval = 100;


    public FpsBehavior( )
    {
        // save the WakeupCriterion for the behavior
        m_WakeupCondition = new WakeupOnElapsedFrames( m_knReportInterval );
    }

    public void initialize( )
    {
        // apply the initial WakeupCriterion
        wakeupOn( m_WakeupCondition );
        System.out.println("initialize");
    }

    public void processStimulus( java.util.Enumeration criteria )
    {
        while( criteria.hasMoreElements( ) )
        {
            WakeupCriterion wakeUp = (WakeupCriterion) criteria.nextElement( );
            // every N frames, update position of the graphic
            if( wakeUp instanceof WakeupOnElapsedFrames )
            {
                if( m_StartTime > 0 )
                {
                    final long interval = System.currentTimeMillis( ) - m_StartTime;
                    System.out.println((m_knReportInterval * 1000) / interval );
                }

                m_StartTime = System.currentTimeMillis( );
            }
        }

        // assign the next WakeUpCondition, so we are notified again
        wakeupOn( m_WakeupCondition );
    }
}
```