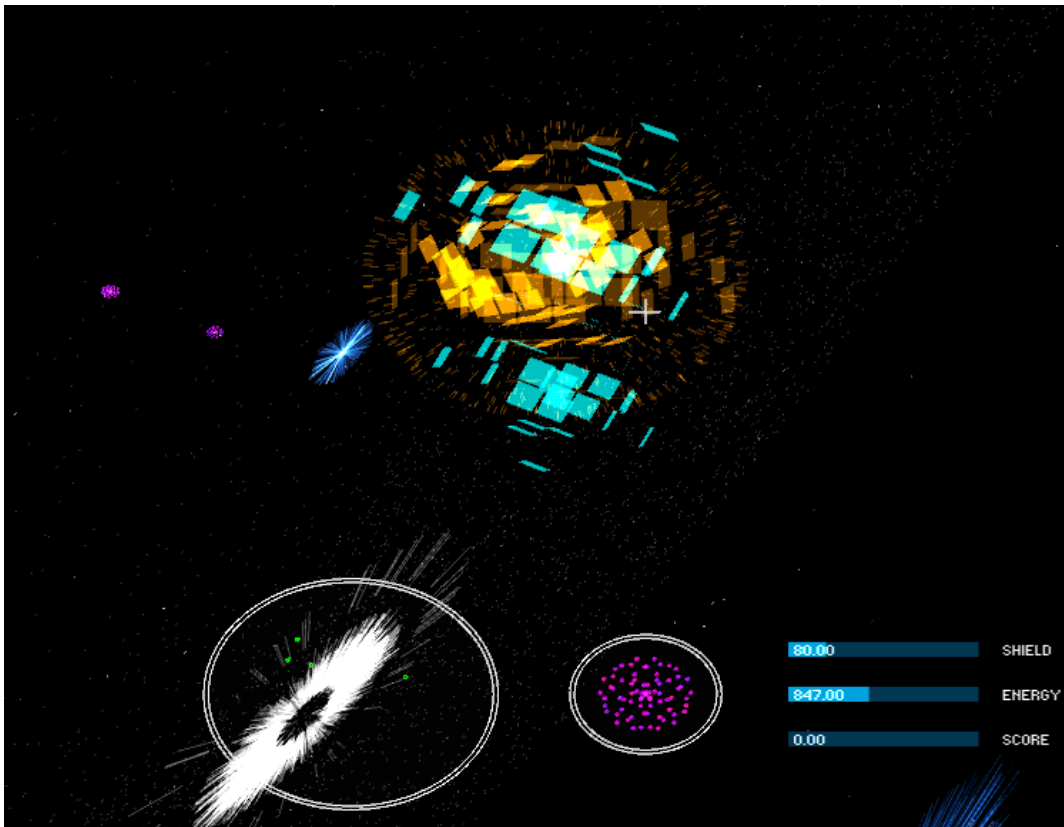


Audio Driven Games

Masters Thesis



Ashley Morrison (i7861963@bournemouth.ac.uk)
Msc Computer Animation & Visual Effects
N.C.C.A Bournemouth University
18th August 2009

Abstract

Two of the most key human senses are vision and hearing. Based on many shared features of these two phenomena as well as the powerful effects they can often elicit, this paper will demonstrate the author's attempts to link the two together in a synchronised fashion within a games context. This will also bring into play a third element, that of interaction. An overview of areas such as digital signal processing, mathematical art and synaesthesia will form the basis of the thesis and the programming language, Processing, used for the implementation, will be considered against the original design goals and final outcome.

Contents

1. Introduction	5
2. Background	5
1. Audio	6
1. Digital Signal Processing	6
2. Beat Detection Algorithms	16
2. Visual	20
1. Mathematical Art	20
2. Music Games Survey	23
3. Design	27
1. Game-Music Links, 'Flow' & Synaesthesia	27
1. Music Game Principles	27
2. Flow	28
2. Game Design Notes	28
1. Specific Game Elements	28
2. Key Focus	29
3. Early Game Ideas & Tests	29
4. Final Game Design Notes	32
4. Tools Overview	33
5. Implementation	33
1. Classes Overview & Diagram	33
2. Sequence Diagrams & Explanation	35
1. Menu Movement	35
2. Main Loop	36
3. Asteroids	37
4. Item Drops	38
3. Key Components & Pseudo-code	39
1. Finding Maximums & Averages in a Frequency Spectrum	40
2. Updating the Contractors	40
3. Asteroid Shape Distortion	41
4. Contractor Creation	41
5. Rose Creation	42
6. Wormhole Draw	42
7. Particle Gravitare	43
8. Menu Visualisation	43
9. Missile-Asteroid Collision & Asteroid Split	44
10. Radar Draw	44
11. Updating Asteroids	45
12. Updating Item Drops	46
13. Explosions	47
14. Asteroid & Item Drop Steering	48
4. Efficiency & Bugs	48
1. Efficiency	48

2. Bugs & Usability	49
5. General Issues of Implementation	51
6. Conclusions	52
1. Objectives Comparison	52
1. Music Game Influences	52
2. Standard Game Influences	53
3. Audio, Visual & Game Relationships	54
4. Approach to Audio Analysis	54
5. Specific Design/Implementation Goals	55
2. Improvements & Future Work	56
7. Bibliography	57
8. Appendix	62
1. Tools Overview	62
1. Processing	62
2. Minim	62
3. ControlP5	63
2. Beat Induction/Tracking Extra	63
1. Beat Induction	63
2. Beat Tracking	64
3. Code Usage	67
4. Development Notes	68
1. Overview of Progress	68
5. Code Listing	71

1. Introduction

This paper revolves around three general areas, namely: audio analysis, audio visualisation and finally user interaction between the two. The production of a game to explore the relationships between these areas will also cover phenomena such as synaesthesia or synchronisations between the senses and interaction produced through immersion. The feeling of losing oneself is common in both music and games as a harmony between medium and participant occurs, sometimes referred to as, “flow” [Csikszentmihalyi90].

This paper will attempt to outline some of these areas and discussions to encourage interest and also to use them as a basis upon which to develop a game which itself exemplifies said interactions. More specifically this will be accomplished by studying the balance needed between audio, visuals and interaction to immerse a player in a game.

Music will be considered both from a technical as well as artistic point of view, i.e., what *can* be extracted and what *needs* to be to best support the sound visually. The visuals themselves will initially be considered from a non interactive, procedural based approach through areas such as mathematical art and later more specifically through a games context. Lastly the links between these two areas will also be examined in different types of games, pro-active/re-active and what synaesthesia might tell us regarding game immersion for example.

Chapter two is the background and history of the areas noted above that will form the basis of this project. Audio centres around digital signal processing and beat detection. The visual element looks at mathematical art and procedural animation. Last of all is a music games overview.

Chapter three covers the design. This will include a consideration of synaesthesia and game immersion. Ideas and tests initially considered for the animation/game, then, final game design choices and inspirations that will serve as the backdrop for review in the conclusion as to success or failure. Chapter four covers the tools to be used and how they link in with the background areas, most of which is moved to the appendix. Next is the implementation itself which will cover more in depth class/method reviews, sequence diagrams and pseudo code examples of how it all fits together as well as problems encountered and any bugs etc.

After this is an overview of the final piece, considered from all three areas outlined in the design. The user guide will most fully cover how to use the system and what it is capable of with screenshots. Finally, any technical limitations of the outcome, problems encountered, general conclusions and lessons learned will be covered and ideas/expectations for future work/development. An appendix with extra audio analysis research and development details as well as a bibliography is attached also.

2. Background

2.1 Audio

2.1.1 Digital Signal Processing (DSP)

This section is devoted to giving an overview of the key concepts of DSP and beat tracking as they apply to the project as a whole and also as they are specifically applicable to the audio library used for implementation, “minim” [Minim].

a.Signal Domains

Continuous signals are a varying quantity expressed as a function of a real-valued domain, typically time or space. This means they are a function of a continuous argument. Analogue signals are continuous by nature however, in DSP, digital or discrete signals are used to allow computers to work with an approximation of these analogue values. Discrete signals are usually obtained via two methods, sampling and quantization of a continuous signal. As such, it is a function over a domain of discrete integers with each value in a sequence being a “sample”. Being an approximation of a continuous signal, discrete signals in this sense can be thought of as a compressed continuous signal, restricting how much information a digital signal can contain. [Rocchesso03]

b.Conversion

The process of conversion typically involves two distinct elements, firstly, “sampling” and secondly, “quantization”. These two processes degrade the continuous signal to an approximate form but in different ways. First we have an analogue signal to be digitized, like a voltage over time. Then following this are two sections of the process that correspond to sampling and quantization, sample-and-hold (S/H) and analog-to-digital converter (ADC) respectively. S/H is as it sounds, a sampling of a continuous value at periodic intervals, the value is held until the next period. Quantization through the ADC process is a conversion of these sampled voltages to a nearest integer number.

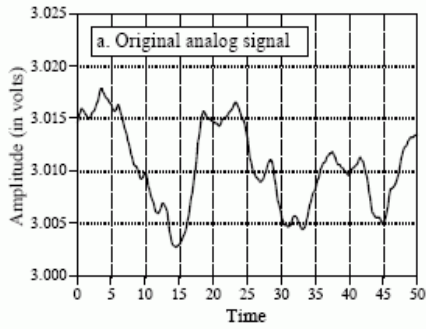


FIGURE 3-1 Waveforms illustrating the digitization process. The conversion is broken into two stages to allow the effects of *sampling* to be separated from the effects of *quantization*. The first stage is the sample-and-hold (S/H), where the only information retained is the instantaneous value of the signal when the periodic sampling takes place. In the second stage, the ADC converts the voltage to the nearest integer number. This results in each sample in the digitized signal having an error of up to $\pm\frac{1}{2}$ LSB, as shown in (d). As a result, quantization can usually be modeled as simply adding noise to the signal.

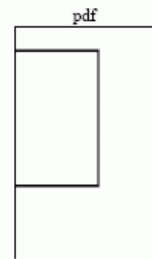
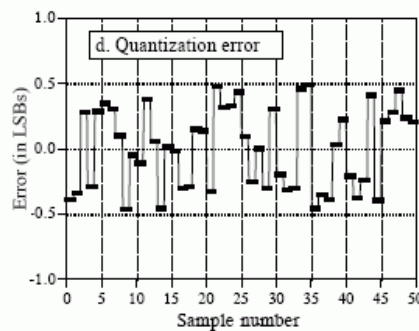
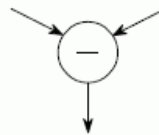
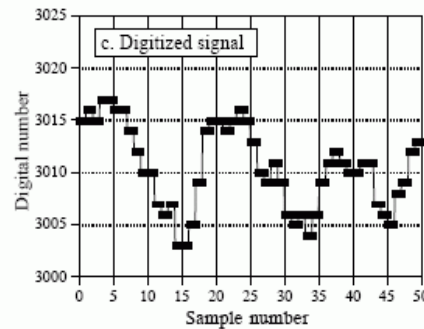
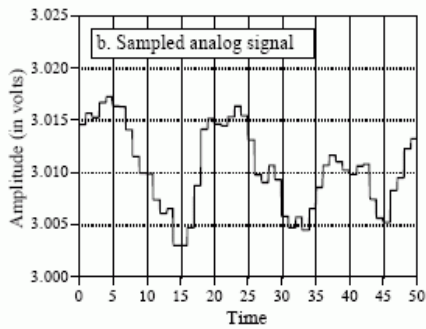
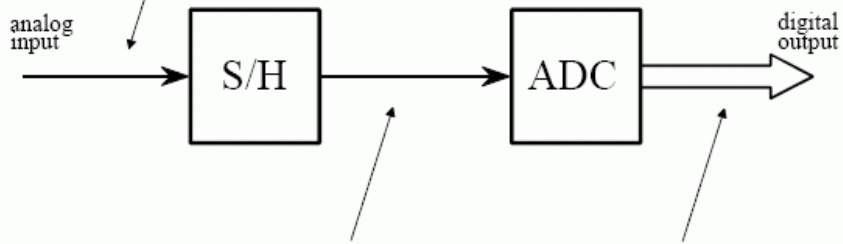


Figure 1.1 – Conversion Diagram. [Smith07]

Therefore sampling can be thought of as converting the “*independent variable* (time in this example) from continuous to discrete”. Quantization on the other hand, “converts the *dependent variable* (voltage in this example) from continuous to

discrete”. The nearest integer value that is chosen to most closely correspond will effectively give us the maximum error swing through, \pm ? Least Significant Bit (LSB). [Smith07]

c. Proper Sampling & The Sampling Theorem

The idea of proper sampling can be easily summed up through the question, “can I exactly reconstruct the analog signal from these digitized samples?” If this can be done, it will mean at least the key information has successfully been captured through the processes outlined above. This is apart from whether the digital markers “look” as if they correspond to the continuous wave.

The examples below show some different examples of analog signals that have been sampled at different rates. It may seem as if only the first two most accurately capture the analog signal but in fact (a), (b) and (c) all do, making them examples of proper sampling. This is because they are all a unique representation of the analog signal. However, (d) quite clearly represents a different wave than the one contained in the original signal. More specifically, “the original sine wave of 0.95 frequency misrepresents itself as a sine wave of 0.05 frequency in the digital signal.” The phenomenon of a wave changing frequency during sampling is known as, “aliasing”, from the word, “alias”, meaning an identity not your own. When there is nothing in the digitized samples to unambiguously reconstruct the analog signal, we have “improper” sampling.

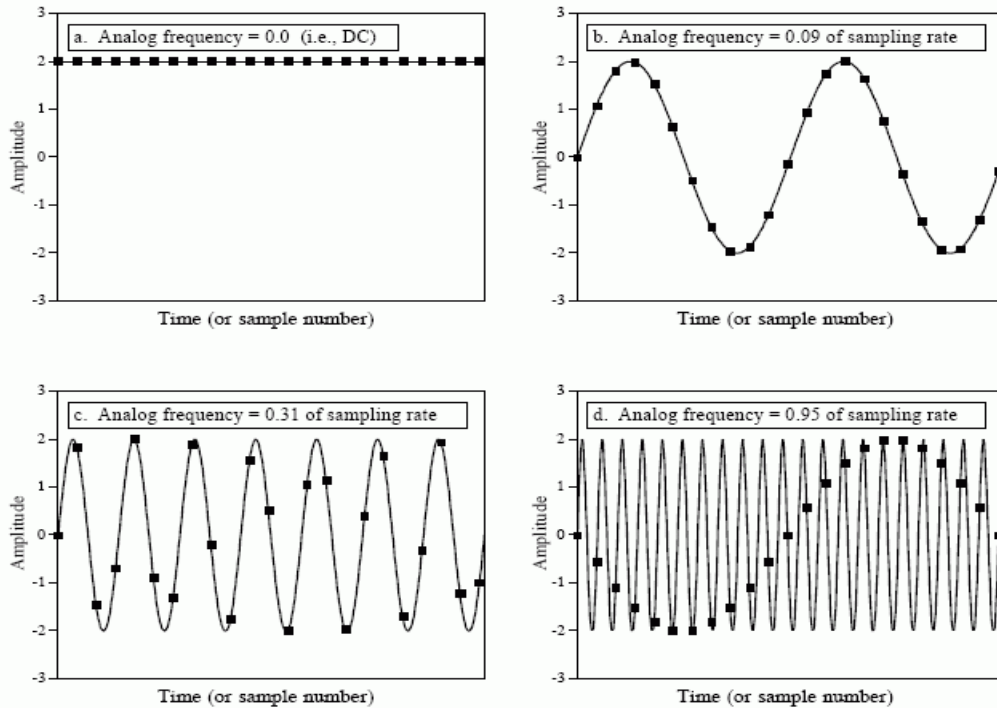


FIGURE 3-3
 Illustration of proper and improper sampling. A continuous signal is sampled *properly* if the samples contain all the information needed to recreate the original waveform. Figures (a), (b), and (c) illustrate *proper sampling* of three sinusoidal waves. This is certainly not obvious, since the samples in (c) do not even appear to capture the shape of the waveform. Nevertheless, each of these continuous signals forms a unique one-to-one pair with its pattern of samples. This guarantees that reconstruction can take place. In (d), the frequency of the analog sine wave is greater than the Nyquist frequency (one-half of the sampling rate). This results in *aliasing*, where the frequency of the sampled data is different from the frequency of the continuous signal. Since aliasing has corrupted the information, the original signal cannot be reconstructed from the samples.

Figure 1.2 – Sampling. [Smith07]

This problem is what lead to the “Sampling Theorem” or the “Nyquist/Shannon Sampling Theorem”, after the original authors themselves. “The sampling theorem indicates that a continuous signal can be properly sampled, only if it does not contain frequency components above one-half of the sampling rate.” An example given in [Smith07] is with a sampling rate of 2000 samples per second, this will require the analog signal to be composed of frequencies below 1000 cycles per second. The terms “Nyquist frequency” or “rate” are often used interchangeably and are not standardised, however the most common usage seems to be that they both refer to one-half the sampling rate. [Smith07]

d.Convolution

Convolution is the process of combining two signals to produce a third signal. Convolution relates the input signal, output signal and something called the “impulse response” and makes up a key element of DSP so a brief outline is given below. Essentially, signals can be decomposed into groups of components called, “impulses”, were impulses are a signal composed of all zeros except for one non-zero point. This means impulse decomposition allows for signal analysis one sample at a time. The

two main types of decomposition are “Fourier” and “Impulse”. Convolution is a mathematical operation that describes the process of impulse decomposition.

Two key terms are the “delta function” which is a normalized impulse, meaning sample zero is 1 and all other samples are 0. Second is, “Impulse response” which is the signal you get exiting a system when the input is a delta function. In other words the impulse response is what is applied to an input signal to produce an output, given the specific input itself.

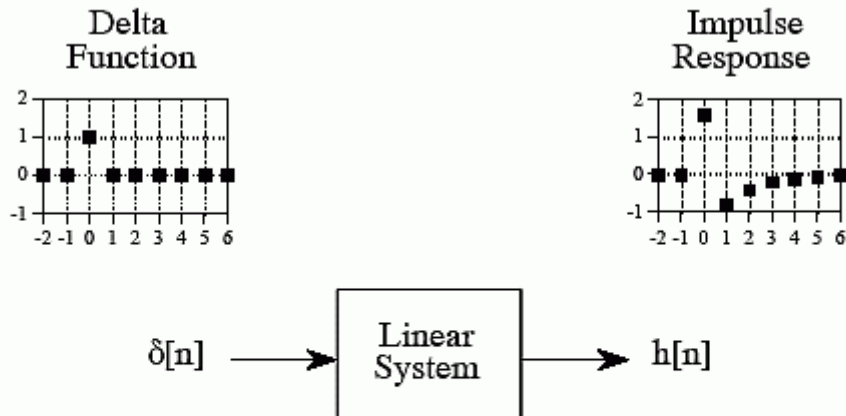
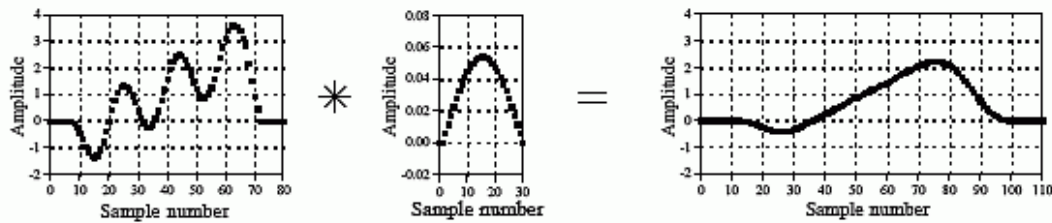


FIGURE 6-1
 Definition of *delta function* and *impulse response*. The delta function is a normalized impulse. All of its samples have a value of zero, except for sample number zero, which has a value of one. The Greek letter delta, $\delta[n]$, is used to identify the delta function. The *impulse response* of a linear system, usually denoted by $h[n]$, is the output of the system when the input is a delta function.

Figure 1.3 – Convolution Diagram. [Smith07]

The two signals combined in convolution in this context are the input delta function and the impulse response. In the audio library used for this implementation [Minim], the impulse response is referred to as the, “kernel”. The process of convolution comes under the class, “Convolver” which combines the input signal with the impulse response. Some examples of convolution in DSP are shown below which demonstrate how particular elements of the input signal are separated out using specifically chosen impulse responses or “kernels”. In the example convolution is used with respect to low and high pass filters, a common technique used in audio processing. Both effectively have a cut-off frequencies that allow frequencies above/below these values to pass but reduce the amplitude of those not meeting the requirement. [Smith07]

a. Low-pass Filter



b. High-pass Filter

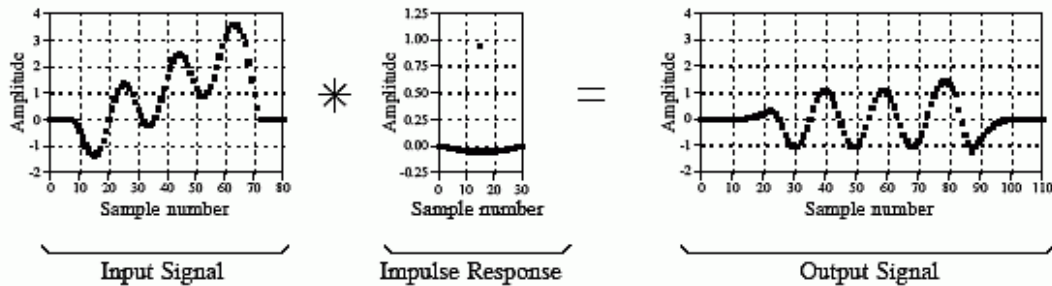


FIGURE 6-3 Examples of low-pass and high-pass filtering using convolution. In this example, the input signal is a few cycles of a sine wave plus a slowly rising ramp. These two components are separated by using properly selected impulse responses.

Figure 1.4 – Filter Passes Diagram. [Smith07]

e. Digital Filters

A digital filter can be thought of as, “any linear, time-invariant system operating on (a) discrete-time signal.” These systems are completely described in the impulse responses and how they’re applied to the input signals. In DSP, it is a question of what impulse response can be applied to return a satisfactory signal. There are two broad types, those that are essentially a “linear combination of a finite number of samples of the input signal”, also known as Finite Impulse Response (FIR) filters. Then there are those that “admit only recursive realizations, thus meaning that the output signal is always computed by using previous samples of itself.”, which are called, “Infinite Impulse Response (IIR) filters.

The audio library **[Minim]** uses IIR filtering and has options for setting cut off frequencies and specific signals to process the filter with. IIR filters stand against FIR filters given the previous points in so far as FIR are done via convolution noted above while IIR filters are “composed of sinusoids that exponentially decay in amplitude” meaning that without round off noise values, they could go on infinitely. [Rocchesso03]

f. Discrete Fourier Transform (DFT)

“Fourier analysis is a family of mathematical techniques, all based on decomposing signals into sinusoids.” Therefore the discrete variety is concerned with digitized

signals. This section will cover three key areas within this family of mathematical techniques, the real DFT which uses real numbers, the complex DFT which uses complex numbers and lastly the Fast Fourier Transform (FFT) which is a highly efficient way of calculating the complex DFT that minim [Minim] makes use of in the FFT class. Essentially, a Fourier transform takes a signal in the time domain like a sample buffer and transforms it into a signal in the frequency domain, typically referred to as the spectrum.

In this section, the Fourier transform family is laid out, next real and complex varieties are noted, then the three main methods of DFT calculation, one of which is the FFT which covers the last part. [Smith07]

1)Family of Fourier Transforms

Jean Baptiste Joseph Fourier who was a French mathematician and physicist presented a paper in 1807 with the claim “that any continuous periodic signal could be represented as the sum of properly chosen sinusoidal waves.” An objection was made that this was not possible as far as mathematical exactitude goes, for continuous signals. This has proven to be true, however, you can get so close as for the difference between the two to have “zero energy”, which is close enough.

An example of this break down would be a 16 point signal being decomposed into 9 cosine waves and 9 sine waves with the frequency of the sinusoid fixed, the amplitude is changed depending on the waveform being dealt with. This leads to the four categories of Fourier Transform, composed between the two features of continuous/discrete and periodic/aperiodic giving the following:

- Aperiodic – Continuous
 - Fourier Transform
- Periodic – Continuous
 - Fourier Series
- Aperiodic – Discrete
 - Discrete Time Fourier Transform
- Periodic – Discrete
 - Discrete Fourier Transform

How these categories of Fourier Transform apply to DSP and computers is connected with another problem to do with sin/cosine waves being defined as extending to infinity in both positive and negative directions while only having a discrete set of samples. This is solved by making the finite data “*look like* an infinite length signal.” This is accomplished with the idea of “imaginary samples” that *do* extend infinitely positive/negative but these values are all zero making it *effectively* discrete.

Finally, for the synthesis of an aperiodic signal, an infinite number of sinusoids are required which “makes it impossible to calculate the Discrete Time Fourier Transform in a computer algorithm.” This leaves the periodic, discrete category of the Discrete Fourier Transform (DFT). [Smith07]





Type of Transform	Example Signal
Fourier Transform <i>signals that are continuous and aperiodic</i>	
Fourier Series <i>signals that are continuous and periodic</i>	
Discrete Time Fourier Transform <i>signals that are discrete and aperiodic</i>	
Discrete Fourier Transform <i>signals that are discrete and periodic</i>	

FIGURE 8-2
Illustration of the four Fourier transforms. A signal may be continuous or discrete, and it may be periodic or aperiodic. Together these define four possible combinations, each having its own version of the Fourier transform. The names are not well organized; simply memorize them.

Figure 1.5 – Fourier Transform Types. [Smith07]

2) Real and Complex

As noted in (1) of this section, there are real and complex number varieties which means the four categories above can be further divided which gives us the two type of DFT. The Real version, “is the simplest, using ordinary numbers and algebra for the synthesis and decomposition.” The complex variety using complex numbers “comprising a “real number” part and an “imaginary number” part; is normally written in the form $a + bi$, where a and b are real numbers, and i is the square root of minus one.”

To sum up the basic elements of the DFT and why/how it does what it does we can say, “the discrete Fourier transform changes an N point input signal into two point output signals.” The “input signal contains the signal being decomposed, while the two output signals contain the *amplitudes* of the component sine and cosine waves.” The input signal is typically in the time domain.

The output is in the “frequency domain (which) is used to describe the amplitudes of the sine and cosine waves. The frequency domain contains exactly the same information as the time domain, just in a different form. If you know one domain, you can calculate the other. Given the time domain signal, the process of calculating the frequency domain is called decomposition, analysis, the forward DFT, or simply, the

DFT. If you know the frequency domain, calculation of the time domain is called synthesis, or the inverse DFT. Both synthesis and analysis can be represented in equation form and computer algorithms.” [Smith07]

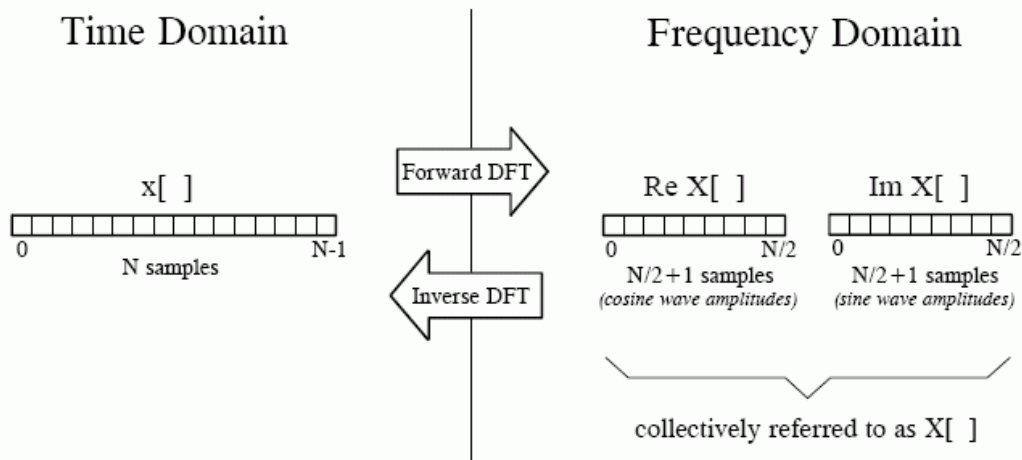


FIGURE 8-3 DFT terminology. In the time domain, $x[n]$ consists of N points running from 0 to $N-1$. In the frequency domain, the DFT produces two signals, the real part, written: $\text{Re } X[k]$, and the imaginary part, written: $\text{Im } X[k]$. Each of these frequency domain signals are $N/2 + 1$ points long, and run from 0 to $N/2$. The Forward DFT transforms from the time domain to the frequency domain, while the Inverse DFT transforms from the frequency domain to the time domain. (Take note: this figure describes the real DFT. The complex DFT, discussed in Chapter 31, changes N complex points into another set of N complex points).

Figure 1.6 – Domain Comparison. [Smith07]

It is felt the finer details of the notation for the real DFT and the mathematical details of the complex DFT are beyond the scope of this overview so instead the next section will cover the three ways DFT's are calculated and how the FFT applies.

3) Methods of DFT calculation

a. By Simultaneous Equation

This method is described as getting N values from the time domain and calculating N values in the frequency domain using basic algebra through solving for N values via N linearly independent equations. The equations are derived by summing relative samples from each sinusoid to get the N equations and using simultaneous equation solving methods to retrieve the answer. This method requires a lot of calculations and as such is almost never used practically. [Smith07]

b. By Correlation

In the correlation method, “to detect a known waveform contained in another signal, multiply the two and add the points in the resulting product. The single number that results from this procedure is a measure of how similar the two signals are.” Extending this, “each sample in the frequency domain is found by multiplying the time domain signal by the sine or cosine wave being looked for, and adding the resulting points.” If two signals are checked against a sin wave that makes three

cycles in a given range and one signal matches while another doesn't, the algorithm should return the "amplitude of the sine wave present in the signal". [Smith07]

c.FFT

Put simply this method "decomposes a DFT with N points, into N DFTs each with a single point." The major benefit the FFT method has that makes it so practically popular is a reduction of several orders of magnitude in computation time by avoiding the direct evaluation of the DFT formula as below:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi k \frac{n}{N}} \quad k = 0, \dots, N - 1.$$

"Evaluating this definition directly requires $O(N^2)$ operations: there are N outputs X_k , and each output requires a sum of N terms."

"The FFT operates by decomposing an N point time domain signal into N time domain signals each composed of a single point. The second step is to calculate the N frequency spectra corresponding to these N time domain signals. Lastly, the N spectra are synthesized into a single frequency spectrum."

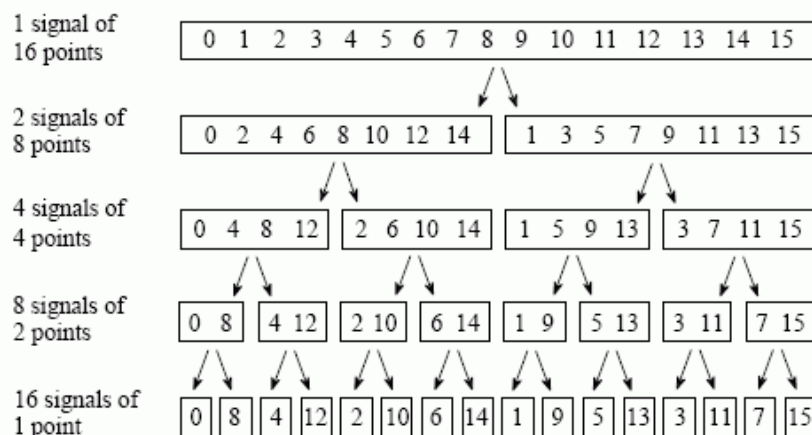


FIGURE 12-2
The FFT decomposition. An N point signal is decomposed into N signals each containing a single point. Each stage uses an *interlace decomposition*, separating the even and odd numbered samples.

Diagram 1.7 – FFT Decomposition. [Smith07]

The first step essentially amounts to a re-ordering of the samples in a signal. Finding the frequency spectra of the 1 point time domain signals is elementary because "the frequency spectrum of a 1 point signal is equal to *itself*". However now it has to be noted that the data is understood differently in the frequency domain. Lastly we have to "combine the N frequency spectra in the exact reverse order that the time domain

decomposition took place.” The “frequency spectra are combined in the FFT by duplicating them, and then adding the duplicated spectra together.” [Smith07]

2.1.2 Beat Detection Algorithms

See the appendix for a much larger summary of the beat induction/detection approaches.

This following section describes the overview and analysis of beat detection that the audio library *minim* [Minim] makes use of. It is a summary of Frederic Patin's paper [Patin]. Two main methods are outlined, connected with simple “sound energy” and a more specific frequency based analysis of sound energy. The first distinction that is made is between the idea of “loudness” and a beat itself. So signals intercepted by the brain will be associated with an energy and the higher that energy the louder the sound. However, a *beat* will only be detected “if his energy is largely superior to the sound's energy history, that is to say if the brain detects a brutal variation in sound energy.”

Meaning a constant high energy signal will not allow for beats while lower energy signals with sudden peaks will most clearly be associated with beats. This gives us an instant sound energy reading to compare against an average over a set period to correspond to a history. For the instant it is suggested to take an energy reading from approx 1024 samples which represents about 500ths of a second or an “instant”. With the average restricted to a set of samples that he chooses arbitrarily as representing about 1 second, or 44032 samples. This value is a compromise and *minim* allows for a “dampening” of the history selection to better suit specific needs.

i.Simple Sound Energy

The algorithm can be summarised as this:

- 1) Compute the instant sound energy 'e' on the 1024 new samples taken in (an) and (bn) using:

$$e = e_{\text{stereo}} = e_{\text{right}} + e_{\text{left}} = \sum_{k=i_0}^{i_0+1024} a[k]^2 + b[k]^2$$

- 2) Compute the average local energy $\langle E \rangle$ with (E) sound energy history buffer using:

$$\langle E \rangle = \frac{1}{43} \times \sum_{i=0}^{43} (E[i])^2$$

3) Compute the variance 'V' of the energies in (E) using the following formula:

$$V = \frac{1}{43} \times \sum_{i=0}^{43} (E[i] - \langle E \rangle)^2$$

4) Compute the 'C' constant using a linear regression of 'C' with 'V', using a linear regression with values (We can choose a linear decrease of 'C' with 'V' (the variance) and for example when $V \rightarrow 200$, $C \rightarrow 1.0$):

$$C = (-0.0025714 \times V) + 1.5142857$$

5) Shift the sound energy history buffer (E) of 1 index to the right. We make room for the new energy value and flush the oldest.

6) Pile in the new energy value 'e' at E[0].

7) Compare 'e' to 'C*⟨E⟩', if superior we have a beat!

Several points to note about this algorithm are:

1) The energy values taken from the 1024 samples are held in history rather than with the samples themselves. This buffer will correspond to the 1 second time frame noted earlier and means not having to compare a full 44100 sample buffer. The buffer is ⟨E⟩ in this equation, with ⟨E⟩[0] representing the latest 1024 instant history and ⟨E⟩[42] the oldest giving the overall 44032 samples that corresponds to the 1 second.

2) The constant C determines the algorithms sensitivity to beats. At point (4) above, C is calculated by computing the variance within the energy history buffer. For precise beats in say, techno music, C should be quite high while for more mixed/noisy beats in rock music C should be lower. So for a high variation in the history buffer, C is made higher while for a low variation, the opposite.

3) V corresponds to this variance calculation which is made at point (3) and this value is passed in to calculate C at (4).

The essential problem of this approach which is highlighted is that it is effectively colour blind. That is to say, the algorithm deals purely in terms of energy peaks and will detect only the most powerful beat amongst possibly a selection at any given moment. Therefore it's like a colour blind person seeing colours against a space but no variation within the colours themselves. This means we end up with drum beats that are sank among other noises that the listener picks up on (due to variance in *pitch* amongst different instruments for instance) but which is not recognised by the

algorithm. This being common amongst rock/punk type music it is important to deal with it to allow for the broadest ranges of application.

i.Frequency Selected Sound Energy

This extension of the above approach works essentially by identifying on which frequency subband the beat was detected thereby checking different subbands in isolation meaning different pitches, for instance, that can be accounted for. The more the frequency spectrum is divided up into subbands the more sensitive the detection can become but also the more specific its adaptation will be to songs/styles. **[Patin]** lists the basic algorithm as this:

For every 1024 samples:

- Compute the FFT on the 1024 new samples taken in (a_n) and (b_n) . The FFT inputs a complex numeric signal. We will say (a_n) is the real part of the signal and (b_n) the imaginary part. Thus the FFT will be made on the 1024 complex values of:

$$(a_n) + i \times (b_n)$$

- From the FFT we obtain 1024 complex numbers. We compute the square of their module and store it into a new 1024 buffer. This buffer (B) contains the 1024 frequency amplitudes of our signal.
- Divide the buffer into 32 subbands, compute the energy on each of these subbands and store it at (Es) . Thus (Es) will be 32 sized and $Es[i]$ will contain the energy of subband 'i':

$$Es[i] = \frac{32}{1024} \times \sum_{k=i \times 32}^{(i+1) \times 32} B[k]$$

- Now, to each subband 'i' corresponds an energy history buffer called (Ei) . This buffer contains the last 43 energy computations for the 'i' subband. We compute the average energy $\langle Ei \rangle$ for the 'i' subband simply by using:

$$\langle Ei \rangle = \frac{1}{43} \times \sum_{k=0}^{42} Ei[k]$$

- Shift the sound energy history buffers (Ei) of 1 index to the right. We make room for the new energy value of the subband 'i' and flush the oldest.
- Pile in the new energy value of subband 'i' : $Es[i]$ at $Ei[0]$.

$$Ei[0] = Es[i]$$

- For each subband 'i' if $Es[i] > (C * \langle Ei \rangle)$ we have a beat!

The major extension of this algorithm to really take into account the availability of the frequency spectrum analysis is the use of the human ear itself. Patin states:

“The second way to develop the accuracy of the algorithm uses the defaults of human ears. (The) human hearing system is not perfect; in fact its transfer function is more like a low pass filter. We hear more easily and more clearly low pitched noises than high pitch noises. This is why it is preferable to make a logarithmic repartition of the subbands. That is to say that subband 0 will contain only 2 frequencies whereas the last subband, will contain say 20.”

The algorithm is noted as:

- Linear increase of the width of the subband with its index:

$$wi = a \times i + b$$

- We can choose for example the width of the first subband:

$$w1 = a + b$$

- The sum of all the widths must not exceed 1024 ((B)'s size):

$$\sum_{i=1}^n wi = 1024 = n \cdot b + a \cdot \sum_{i=1}^n i = n \cdot b + a \cdot \frac{n \cdot (n+1)}{2} = 1024$$

This feature is a part of the *minim* [**Minim**] library under the *linAverages* and *logAverages* functions within the *FFT* class. *logAverages* takes a minimum bandwidth for an octave and the number of bands per octave. This splitting of the bands between octaves will increase like the example above as the octaves increase meaning the later bands will be larger in width.

In *minim* both of these beat detection algorithms are available depending on what level of granularity is needed. A more specific outline of the pseudo code and method calls for using *minim* and these techniques is shown in the *Implementation* section (5). For the original paper, see [**Patin**].

2.2 Visual

2.2.1 Mathematical art

Mathematical art in the broadest sense could be considered as “physical models that vividly demonstrate mathematical concepts, formulas and theorems”. With one of the great attractions being that they are either interaction oriented and encourage viewers to experiment with them to gain an intuitive understanding or they’re based on a pre-existing subject that is already understood and thus helps to build links to new and existing ideas. [Ruiz]

Lastly an example exhibition notes that mathematics, “as it is traditionally taught”, “fails to elicit wonder” or “foster creativity”. A key aim is “to restore some of this wonderment”, with models attempting to “demonstrate the power and beauty of mathematics”. [Ruiz]

Lorenz Attractor

An example of mathematical art could be the Lorenz attractor as the following image illustrates:

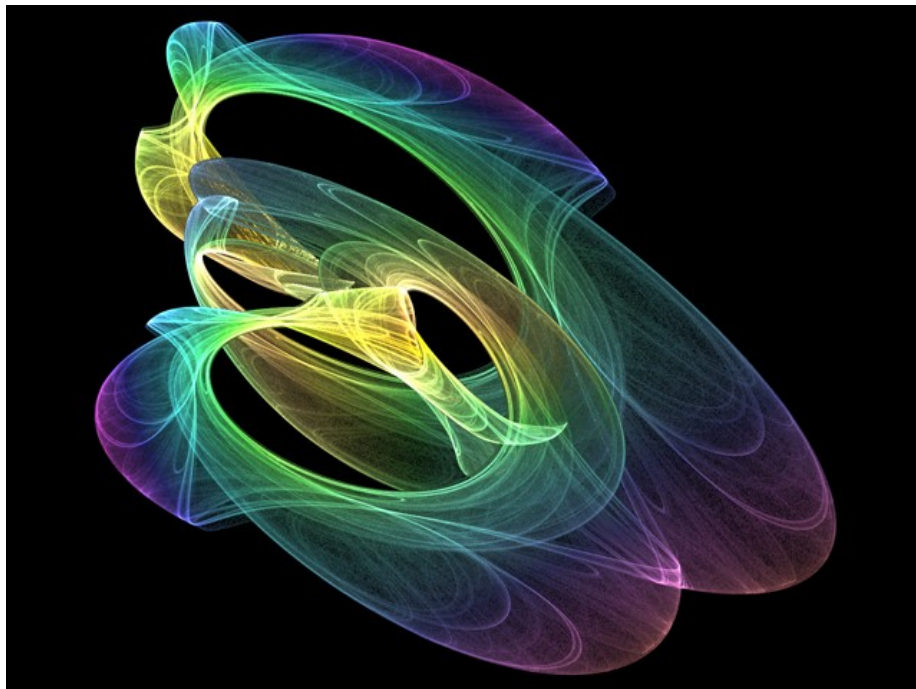


Figure 1.7 - 3d Lorenz Attractor example. [MathArt]

This developing pattern is governed by [MathArt]:

$$\frac{dy}{dt} = x(\rho - z) - y$$

$$\frac{dy}{dt} = x(\rho - z) - y$$

$$\frac{dz}{dt} = xy - \beta z$$

σ (Prandtl Number), ρ (Rayleigh Number)

Another example:

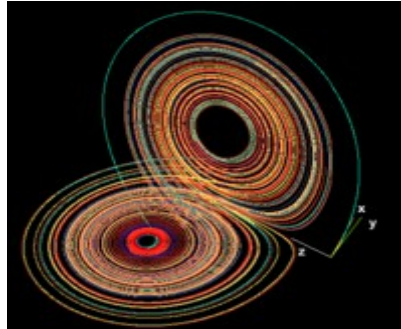


Figure 1.8 - 3d Lorenz Attractor Example 2. [Wikipedia1]

It's also well known that “Music theorists often use mathematics to understand musical structure and communicate new ways of hearing music. This has led to musical applications of set theory, abstract algebra, and number theory. Music scholars have also used mathematics to understand musical scales, and some composers have incorporated the Golden ratio and Fibonacci numbers into their work.” [Wikipedia2]

With procedural animation being automatically generated in real time and having a basis in the development of algorithms over time, these areas and links will form the crux for the visualisation investigation. Particular emphasis will be placed on visuals that are both striking and that re-enforce and emphasise musical elements. In a broader games context the link between audio and a visual will need to be considered against player input as well.

The Rose Equation

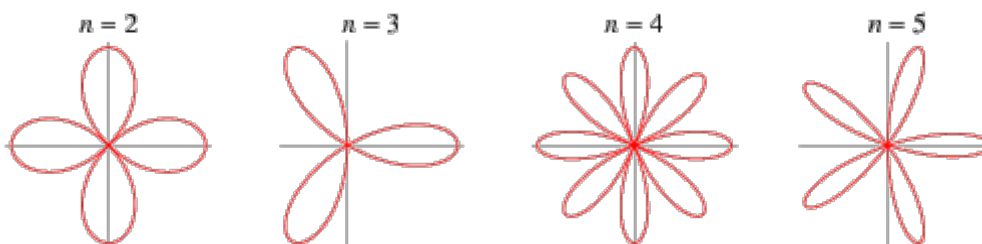


Figure 1.9 - Rose Example patterns 1 [Weisstein]

An example of math art is the rose equation. This is a curve which “has the shape of a petalled flower”, hence the name. The equation is:

$$r = a \sin(n\theta), \text{ [Weisstein]}$$

It has the property that if n is odd, the number of petals is odd, while if even, the number of petals becomes $2n$.

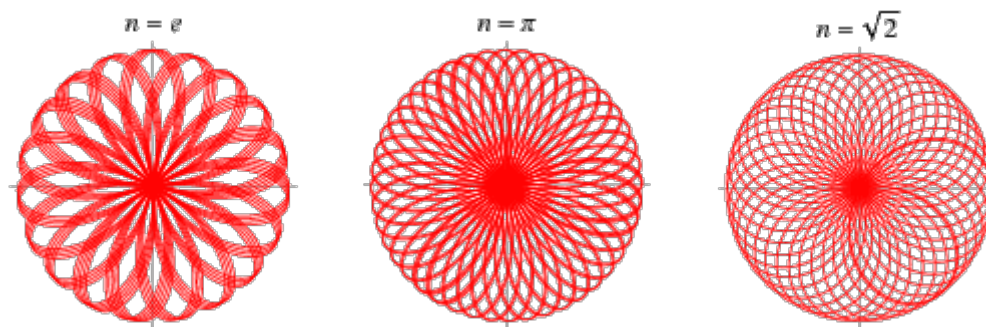


Figure 2.1 - Rose Example Patterns 2 [Weisstein]

There are many different variations of this theme producing different styles of patterns. Using a combination of cos and sin functions in a complementary manner produces the kinds of effects that are both striking and easily manipulated in real time based on another parameter like a frequency analysis of music. In this manner it's easy to see how a developing an ever changing visual pattern can re-enforce musical beats and changes in tempo etc. This is especially true if specific patterns become repeated following corresponding musical sequences.

The Mandala Pattern

“In common use, mandala has become a generic term for any plan, chart or geometric pattern that represents the cosmos metaphysically or symbolically, a microcosm of the Universe from the human perspective.” *[Wikipedia3]* They sometimes also exhibit a tendency to be both divisible in terms of discernible patterns within the whole but that these patterns themselves also form part of a connected structure. This clash between discernible components that also link together to form a coherent whole could easily be seen to correspond with music in so far as where the attention at any given moment is drawn or between the listener becoming alert to something or losing themselves in the piece.



Figure 2.2- Mandala Design Example. [Wikipedia3]

2.2.2 Music Games Survey

The following summary of music games is based upon a Wikipedia article [Wikipedia4].

Game Types

a. Music Memory

Music memory based games come in two main forms, sight-reading and eidetic, testing simple short term memory and more complex and demanding memory recall respectively.

i. Sight-reading music games

These games primarily focus upon one or more of rhythm, pitch and volume were “the goal of the player is to provide a direct injective response to each prompt (linked to an element of the music) from the game”. The prompt and reaction periods for these games mean they focus around short term memory and a “Simon says” type format.

1. Rhythm – Guitar Hero



Figure 2.3 - Guitar Hero Example. [IGN09]

An example of a rhythm based, sight-reading game would be “Guitar Hero”. Here “the player must press specific buttons, or activate controls on a specialized game controller, in sync with the game's music. The control scheme is usually fairly simplistic, and the moves required are usually predetermined rather than randomized.”

2.Pitch – Karaoke Revolution

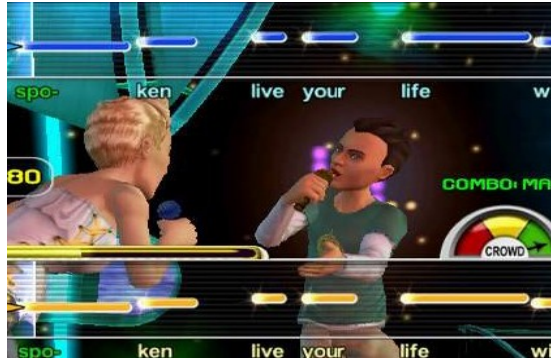


Figure 2.4 - Karaoke Revolution Example. [Geek09]

Pitch based games, like “Karaoke Revolution”, “test the player's ability to match the pitch of a piece of music provided by the game.” Linked to rhythm, instead the focus is for players to use their voice and they’re then ranked based on tonal accuracy.

3.Volume – Wii Music

Volume games seem to be much rarer and the link is simply the ability of the player to alter the volume through a gameplay action or button.

ii.Eidetic music games – Space Channel 5, Myst puzzle

Eidetic memory is often used as another term for photographic memory and relates to the recall of sounds/images etc through a brief time span for memorisation but with very high accuracy. These games are differentiated from the simpler short term memory requirement by the fact the goal of recall is continually extended to require more and more for recall. “Each successive prompt and response contains the entirety of the prior prompt or response as well as additional material determined by the

round.” This style of music game has also become a recurring puzzle as for instance found early on in the original *Myst* game within the spaceship.

b.Freeform – SimTunes

Freeform music based games are typically the equivalent of sand box games in a music environment. Like sandbox games, there often aren't any discernible goals except experimentation. Often the music creation is the focus and takes precedence over anything else. This is more of a sliding scale as well with games that focus more and less on the music, some are really just music synthesizers pure and simple. Some examples include *Fluid* and *Elektroplankton*.

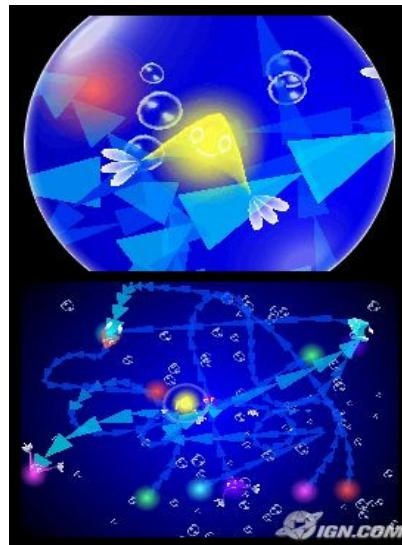


Figure 2.5- Elektroplankton Example. [Synth]

i.Music art games – Moondust, tranquillity

An extension or cousin of the freeform music game is the music art game whereby the “emphasis lies on the artistic aspects of musical gameplay rather than the ludological aspects”. They're typically quite abstract and lacking in any plot, focusing instead on visual/audible creativity and expression without lives or scores etc.

c.Hybrid

Lastly the Hybrid music video game is so called due to it containing meaningful and often complex interactions between the music and player while being part of a non music genre. For instance, shooters that are linked in or driven by the music in some way like the generation of enemies being based on the beat etc.

i.Generative – Rez



Figure 2.6- Rez Example. [Armchair]

“Generative”, hybrid music games tend to be a source of music/sound based on player input. To integrate the non music genre with sound in this way, the dynamics of the game in question will typically be quite simplistic. “Rez” is a famous example of such a game which is a shooter that combines “sound effects created by the actions of the player (as he completes the normal tasks of rail-shooting) with the soundtrack as a whole, the game is intended to permit the player's direct interaction with the soundtrack and to encourage the creation of a synaesthetic experience”.

ii.Reactive – Audiosurf

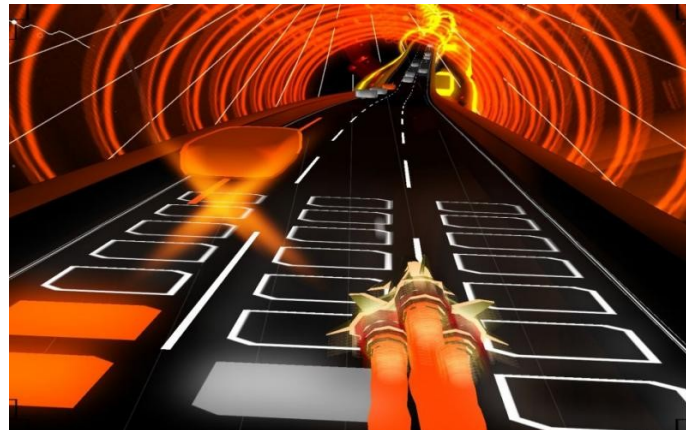


Figure 2.7 - Audiosurf Example. [Next09]

On the other hand “reactive” hybrid music games are based around the concept of players using the music and anticipation of its evolving nature through time to build and execute gameplay strategies. In other words “they employ music to determine gameplay”. A good example of such a reactive music game is “Audiosurf” which requires the player to catch certain coloured blocks amongst damaging ones with the placement and movements needed based on an analysis of the music played. Another example is the old playstation game “Vib-Ribbon”.

d.Mixed Genre – party games

Lastly mixed genre games tend to be party based games that mix a number of mini games into one “frame story”, i.e. an overall story leading to lots of mini self contained ones.

3. Design

3.1 Game-Music Links, ‘Flow’ & Synaesthesia

3.1.1 Music Game Principles

Through a qualitative analysis of music video games [Pichlmair07] found a number of typical features listed as “active scores, rhythm action, quantisation, synaesthesia, play as performance, free-form play, and sound agents.”

Differentiating between “action” and “rhythm” games, “Rez” of the former type, is described as “an easy and straightforward shooter game. The challenge is not to hit the beat but to create a steady flow of sound. The game cannot be played as an instrument in the default arcade mode, but an experienced player navigates through the musical capabilities of the game while still successfully playing the shooter presented as the surface of the game. The player acts on two levels: she plays a shooter and she reacts to the beat of the music game behind the shooter. Rez provides an immersive environment by tightly coupling visual and acoustic sensations.”

In [Mizuguchi07] inspirations for Rez are mentioned as also involving “rave culture. When I first saw a rave party in about 1993 there were many people dancing, and it was like they were jumping in time to the music. I had a big, big flash, and suddenly I just remembered about the concept of synaesthesia.”

While this synchronisation between onscreen events and audio samples is said to be a key part which “contributes to the pleasure of immersion by reliably providing feedback to the player, Rez offers an engaging but looser association between the on-screen action and the players interactivity. [Douglas01] describe the principles of immersion and engagement as the key factors of player involvement, ultimately leading to a flow experience introduced by Csikszentmihalyi. [Csikszentmihalyi90]”

[Pichlmair07] also gives another key example for the interest of this investigation which is that of Vib Ribbon, mentioned previously under the Reactive, Hybrid music games. The object of the game is noted as “to guide a female rabbit called Vibri along a line, a ribbon populated with obstacles the rabbit can pass by by pressing the correct button at the right time.”

Furthermore “the player can eject the game CD-ROM and insert an audio CD. The audio CD is analysed and the level is built on base of a track. While the player plays, she hears her own music. This way, Vib Ribbon fosters the attachment between a player and her music collection. The experience of playing Vib Ribbon is very personal. Interestingly, the mickey mousing sound effects of Vibri jumping and

walking over the obstacles can be turned off, turning the game into a visual performance tool.”

Audiosurf is an example of a music game which will serve as a key inspiration in this regard as well. Specifically, the obstacles presented are done so in such a way as to demand greater levels of concentration and reaction times depending on the tempo of the sound being played. This interplay between the nature of the interaction and a player’s relationship with the music will be a focus of the design and implementation.

Lastly [Pichlmair07] notes some interesting elements of free-form play in both Rez and Electropunk. Both games support a mode whereby “damage” is turned off, leaving what they call “a journey through sound”.

3.1.2 Flow

The experience of flow noted above is expanded upon in [Pace08] as involving “clear goals and timely feedback; a balance between the challenges of an activity and the skills required to meet those challenges; concentration on the task at hand; a sense of control; a merging of action and awareness; a loss of self consciousness; a distorted sense of time; and the autotelic experience. The term autotelic refers to an activity that is done without the expectation of some future benefit; the activity itself is the reward.”

It is also important to note that flow is an experience investigated in many different areas, cultures and over a considerable period of time and also that a substantial number of the elements above form a key part of many musical experiences. Moving beyond the passive activity of listening to music, given that games are interactive, a further suggestion might be that the process of learning to play a song is quite analogous to learning to play a level or game. The fine tuning of both these processes involves an intimacy between expectation and reaction that seems to be key to flow and is an area that this project will attempt in some manner to replicate or encourage.

Specific approaches regarding how this will be done and a summary of what existing games and features will be used follows in the design section.

3.2 Game Design Notes

3.2.1 Specific Game Elements

Of the general types listed it is the hybrid generative/reactive and the music art games that will be the main inspiration/focus of the game design.

Specifically, the abstract style of Rez, the musically driven pace and futuristic theme of Audiosurf and lastly the personalising aspect of Vib Ribbon will be the main aspects to emulate. Expanding on this list, the following non musical games will also provide inspiration:

a) Asteroids

a. Basic concept – destroying on-collision objects to remain alive etc.

b) Mono

a. Notion of something visually developing over time through player interaction – painting the background.

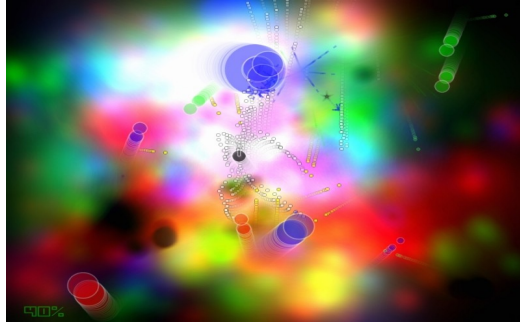


Figure 2.8 - Mono Screenshot example. [Binary05]

c) Arx Fatalis

a. Spell casting is done through specific movements of the mouse on screen, different spells, different patterns.



Figure 2.9 - Arx Fatalis Screenshot example. [Gametap09]

d) Polynomial

a. Visual space style.

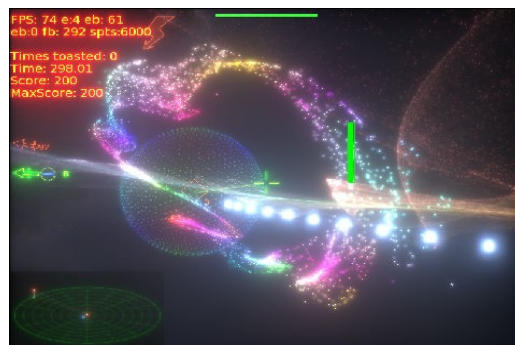


Figure 3.1 - Polynomial Screenshot example. [Indie]

3.2.2 Key Focus

The main issue will be the relationship between the player, the audio and the visualisation. If the visualisations are presented too frequently, it will be too hard and the player will lose connection with the visualisation. However, if too infrequent, the visualisation will lose connection with the audio and it will become too easy or less moving. The balance between these things will be the key to the gameplay.

The idea will not be to categorise the music as a whole as being one thing or another (based on emotion/feeling etc) and then transplant that onto a visual. Rather it will be to take the music, break it down into different elements that make up the whole and to apply these isolated elements to separate elements of the visual.

As a simple example, a song may have three types of beat, a kick, bass and snare drum. A sphere could have a size, position and colour. When a kick drum is onset, the size changes and likewise for the other elements. In this way, in a games context, the goal could be encouraging the user to concentrate and move between different elements of the piece and to synchronise this process with their manipulation of the visual element.

3.2.3 Early Game Ideas & Tests

The two main programs initially made in 2D were a pattern matching type game and something loosely based on a 'Geometry wars' style gameplay.

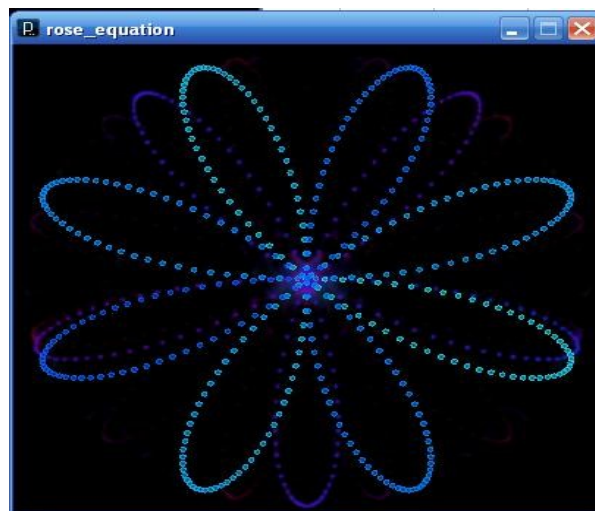


Figure 3.2 - Extended from: [Bumgardner]

The pattern matching game used the rose equation to draw a series of rose like patterns to the screen using points that the player would collect in a limited amount of time before the next pattern was generated. The patterns were influenced by the music being played through calculating the maximum peak value in the frequency spectrum and using this as the value for n , remembering that n primarily determines the number of curves and thus the complexity of the patterns.

The rate at which the patterns were generated was also determined by a rough estimate for fluctuations in beats per minute. This resulted in i) repeated patterns following repeated sequences of sound and ii) a tempo or pace of gameplay that was synchronised with the music as well.

Secondly, the geometry wars style game was very basic in design and implementation but extended on some of the points above by using the same rose patterns as the image drawn to screen for enemies chasing the player. This along with a shield or health and again, a moveable avatar in 2D for the player resulted in a link between the complexity of the patterns and the possible damage to be dealt. Greater complexity in pattern resulted in more damage.

The spawning of these enemies was done using a basic sound energy onset detection which worked quite well in also linking the tempo of sound with the flow of the gameplay. More beats, more enemies to evade or destroy. This combined with a greater chance of the game ending in defeat but also gives the player a greater potential for reward in the form of a higher points total.



Figure 3.3 - Screenshot of simple top down game test.

Lastly the other small test done concerned patterns, shapes or images developing on screen in a manner also connected with the sound. This was done using the boid class taken from [Shiffman] which was used in the geometry wars style example above. Extending this, extra influences are applied to the movement connected with beat onsets etc and not re-drawing the background. This results in interesting pseudo 3d line patterns that are linked with the beats as a viewer observes in real time and also result in something unique by the end after the song has finished.

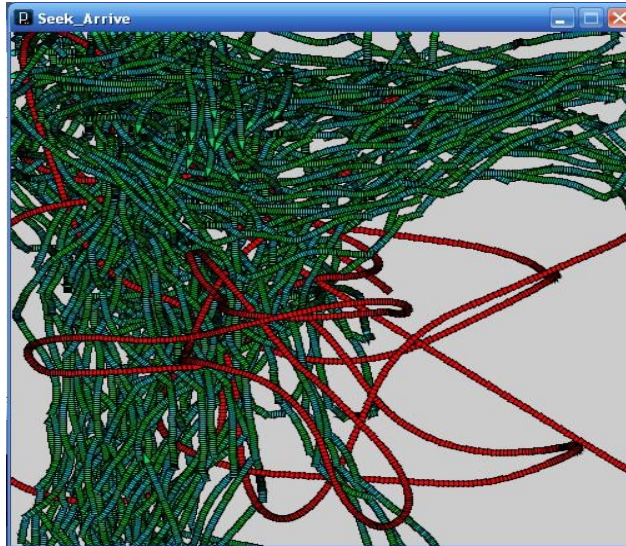


Figure 3.4 - Screenshot of the boid pattern making test.

3.2.4 Final Game Design Notes

By this point the decision was made to start working fully on 3D examples, because the end result offered greater possibilities both in terms of game mechanics and visual impact.

The development of the 3D work and the final game will be discussed in greater detail in the implementation section. Based on all the preceding notes in the background and design sections as well as the findings of [Pace08], the following were decided upon as being key game mechanic aims:

- 1) Clear goals and timely feedback
 - a. Clear objects (asteroids etc) moving towards the player with an aimer and scores clearly indicating a course of action
 - b. Clear gui updates
- 2) Balance between challenge and skill
 - a. Real time difficulty settings like beat sensitivity and object max velocity to allow the user to tailor the experience to the music and their skill.
- 3) Focused attention
 - a. Mix between radar/3d view - “a centering of attention on a limited stimulus field” [Pace08]
 - b. Shoot objects to remain alive, capture item drops to boost scores.
- 4) Reduced awareness
 - a. Simplistic game goals/scenariio to more easily transport the player into the game world, however abstract.
- 5) Presence
 - a. “All of the study participants’ descriptions of presence involved games that featured three-dimensional (3D) virtual environments. Although 3D graphics are not necessary for experiencing flow during game play, they appear to be very influential—and possibly a prerequisite—for

experiencing presence (i.e. situated immersion as opposed to diegetic immersion).”[Pace08]

- b. Player to be attacked from all sides, therefore demanding the need to rotate around “in-game” demands greater attention of action. Feel more in the game when action is “all around” and not just in a linear direction in front of you.

4. Tools Overview

The main tools used were 'Processing'[Fry], 'minim' [Minim], 'controlP5' [Schlegel] and 'Traer' [Bernstein], for the main language, audio, gui and physics needs respectively. See the Appendix for an expansion on the tools and why they were chosen.

5. Implementation

5.1 Classes Overview & Diagram

The key classes are:

- 1) **Main** – This is the main class of the program containing the setup, draw and key/mouse methods. It contains instances of GUI, MinimAudio for a menu song and MainGame.
- 2) **Background** - This class represents the state of the main menu visualisation. It contains lists of SpiralGalaxy objects and points representing stars.
- 3) **GUI** – This class contains the collection of widgets and control panels for the three main GUI's of the program, the main menu, the in-game player data and the in-game settings data. All three are shown/hidden at different times depending on the mode and user input.
- 4) **MinimAudio** – Contains all the audio related classes included a minim[Minim] object used to create a song object, play it and a FFT object used to analyse it amongst others.
 1. **FFTSample** – This is a custom class created in MinimAudio that contains the instance to the minim FFT class[Minim]. It is used to provide further custom analysis if needed.
- 5) **MainGame** – This is the main game class which is created and destroyed depending on the mode the program is in. It contains an instance of MinimAudio for in-game audio analysis as well as instances of Missile, the Traer Physics Library[Bernstein] classes ParticleSystem and Particle, Asteroid, ItemDrop, Contractor, Wormhole, Rose[Bumgardner] as well as references to the Main and GUI classes for updating some globals.

6)**Asteroid** – This is based on Daniel Shiffman's Boid [**Shiffman**] class mainly extended into 3D using the 'steer' method to direct itself at the player. A collection of Asteroid objects is in MainGame.

7)**Pulser** – This class has been modified from Claudio Gonzales' [**Gonzales**] Gravity Swarm example on OpenProcessing. It is a visual effect of a pulsing-like phenomenon that is linked to the beats of the song.

1.**PulserParticle** – This is the particle class Pulser uses, it has also been modified to distribute the particles around a 2D circle initially amongst other changes.

8)**Explosion** – Each Asteroid object has an Explosion object which uses 'Particles on a Sphere' by 'Starkes' [**Starkes**] on OpenProcessing as it's basis. The radius of the sphere is adjusted over time from a point to give the effect of an explosion which is merged with the disappearance of the Asteroid object itself before both are removed.

1.**ExplosionParticle** – The particle class used with the above example.

9)**ItemDrop** – This class is also based upon Daniel Shiffman's boid class [**Shiffman**] again using a 2D steering method given a target point. It is extended into 3D to allow the object to move from a spawn point to either the player or a pre-determined point on the wormhole object.

10)**Missile** – This class is instantiated each time the user fire the weapon. The pattern of ellipses that make up the design is based upon the Rose class below. It also uses the Traer Physics [**Bernstein**] classes to direct the object towards a target the player is facing. The number of points generated is reflected in the amount of damage caused to an asteroid upon impact.

11)**Rose** – This class is based upon Jim Bumgardner's 'Rose Display' [**Bumgardner**] sketch on OpenProcessing which uses the Rose equation outlined in the background/design. The audio analysis of the MinimAudio object is passed in to the equation to link the pattern generation with the sound.

12)**Wormhole** – This class uses the trigonometry of the Philippe Guglielmetti [**Guglielmetti**] sketch on OpenProcessing which in turn is based upon the work of Dr. Goulu at www.goulu.net regarding the principles of a spiral galaxy sequence. The central point is the target for ItemDrop objects.

13) **SpiralGalaxy** – This class is a different modification of Philippe Guglielmetti's [**Guglielmetti**] sketch like Wormhole that is used for the menu visualisations.

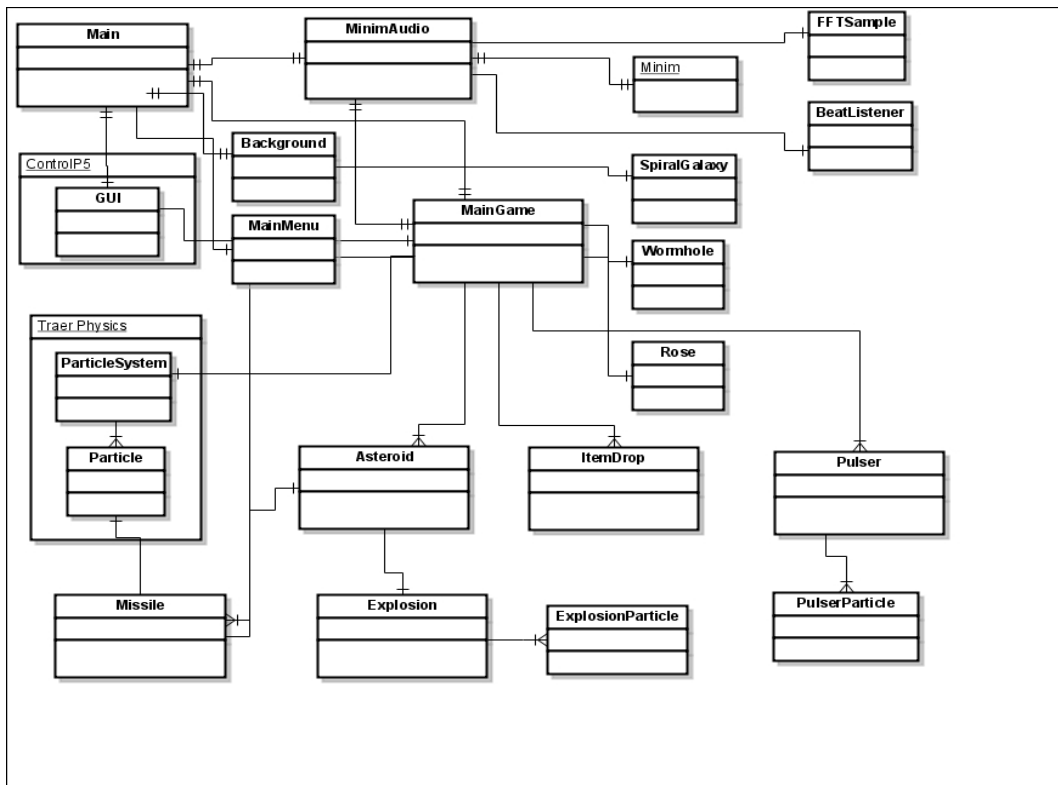


Figure 3.5 - Class Diagram of the System.

The examples used from elsewhere to base and extend these classes on are explicitly stated in the code and reference list.

5.2 Sequence Diagrams & Explanations

5.2.1 Menu Movement

The following diagram gives a basic impression of the movement that takes place in the program, from main menu to in-game and finally a game over screen. From the in-game point the user can also move both backward and forward to the menu and game over screens respectively. Boolean flags are used to determine exactly where the player is and what should be happening. In this way game objects are created and destroyed and gui's are hidden and displayed appropriately.

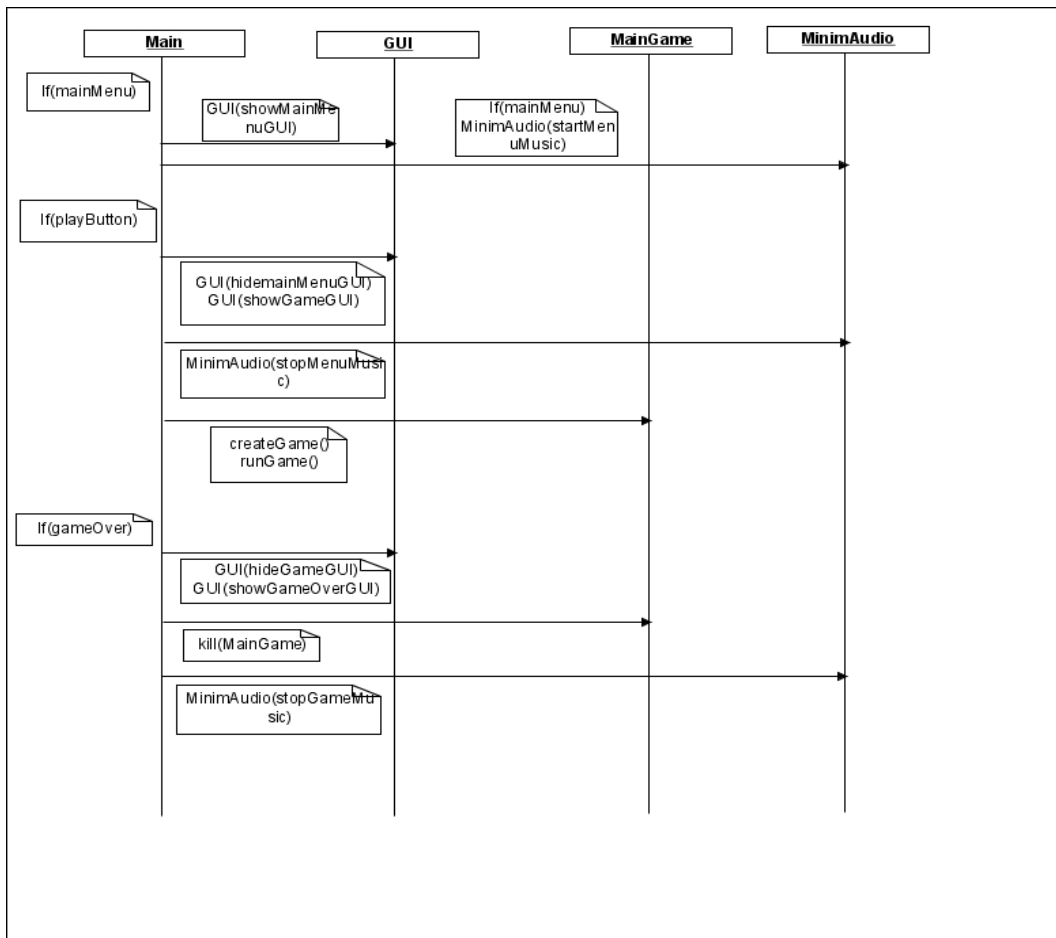


Figure 3.6 – Menu Movement Sequence Diagram.

5.2.2 Main Loop

Next is the main loop sequence diagram which is pretty simple and linear in execution. The relevant GUI is displayed depending on whether the player has pressed a button to change the settings in which case the main loop is paused or whether the game is running normally in which case all the current objects are updated and re-drawn. Lastly it shows that based on this information, the MainGame class updates the basic radar system to inform the player of relative object locations. The ItemDrop, Laser and Asteroid classes all check for collisions in varying ways too shown in more detail below.

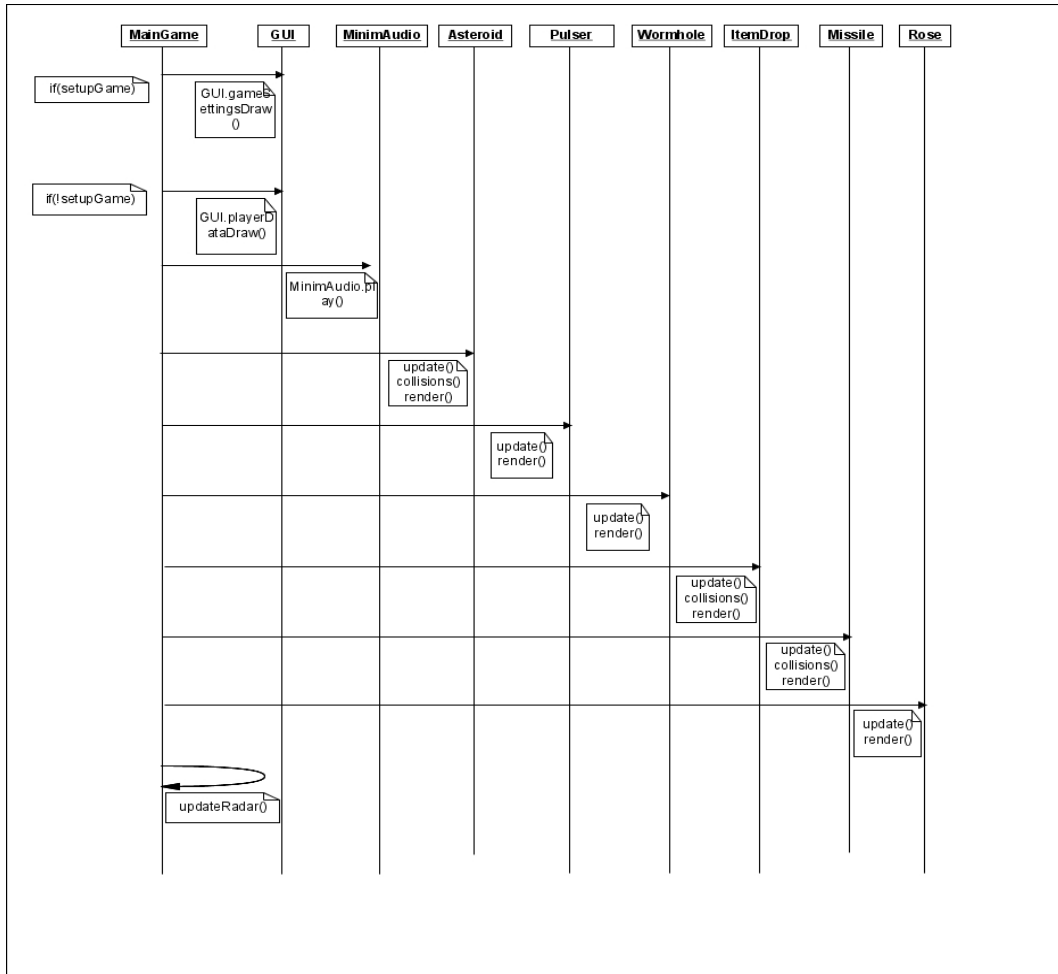


Figure 3.7 – Main Game Loop Sequence Diagram.

5.2.3 Asteroids

This is the basic setup for Asteroid object generation and updating. If a beat is detected, a point around a sphere is calculated given radius X, an asteroid object is placed at that position and told to steer towards the player position.

Different in this process is what happens when the asteroid is detected as having collided with a laser object. Not shown is the fact that if the asteroid is a 'parent', a splitAsteroid() method is called and two new asteroids are spawned, initially heading in opposing directions before steering at the player. This is akin to the Asteroid arcade game upon which this is based, breaking larger asteroids into smaller ones.

However this only happens if the user chooses 'hard' mode over 'normal' difficulty. Otherwise the asteroid object has a boolean flag 'removing' set to true. When this happens, the update and draw methods start to reduce an integer which represents the alpha value for the render which gives the impression of it slowly disappearing. On top of this an Explosion object is created from the centre of the asteroid. When the alpha value is zero, both objects are removed.

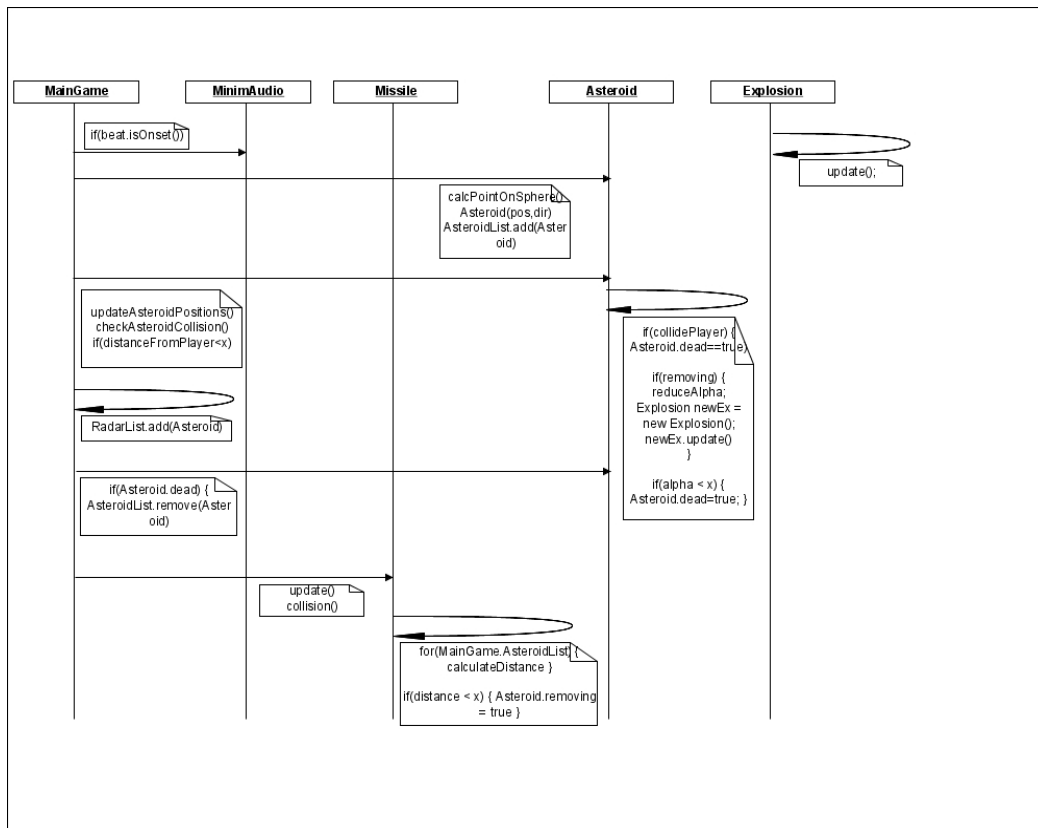


Figure 3.8 – Asteroid Generation and Updating Sequence Diagram.

5.2.4 Item Drops

Lastly is the similar process for item drops. ItemDrop objects use the Wormhole object as a default target position to move towards. They are generated if there is one of three types of frequency onset detected corresponding with a particular Contractor object currently being updated. Each Contractor object has one of three frequency beat detections, each of which are checked for per update. If Contractor.type = beat.type, spawn an item drop object of this type at the Contractor.centre.

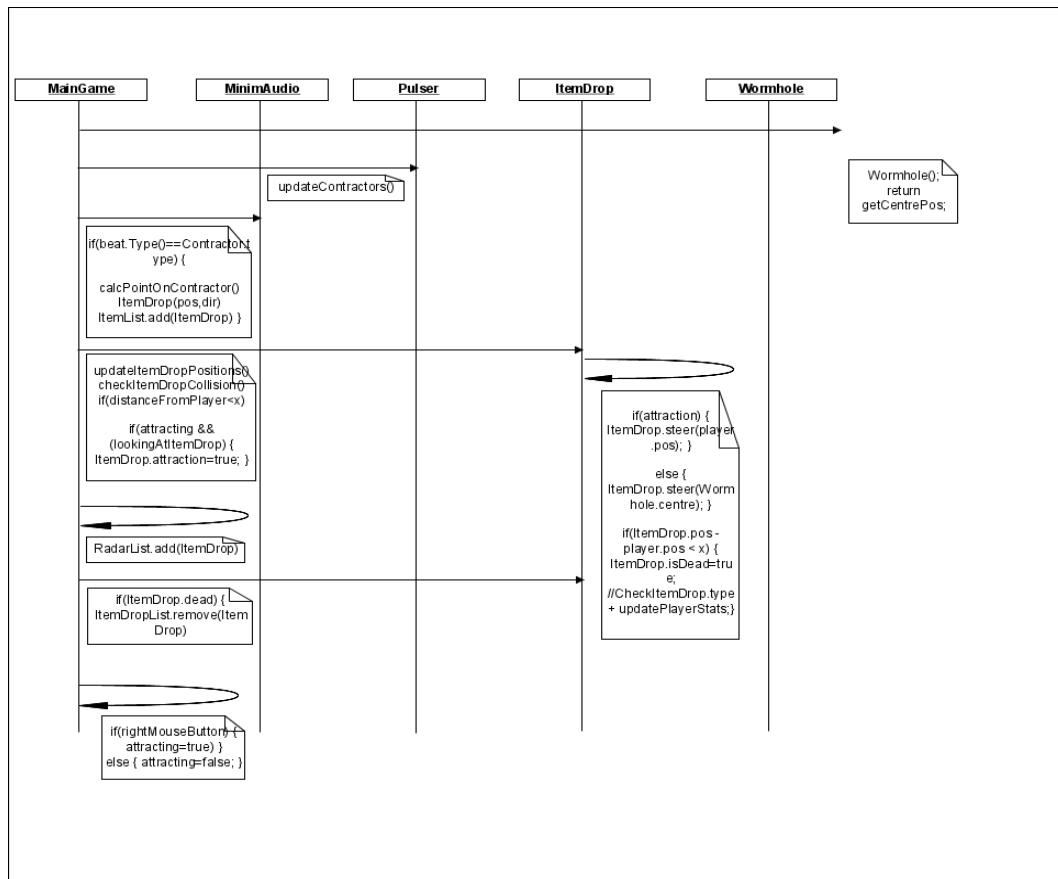


Figure 3.9 – Item Drop Generation and Updating Sequence Diagram.

The other difference is these item directions can be influenced by the player to head towards them. A boolean flag is used to determine this based on player direction, mouse input and the steer method from the [45] boid class is used to re-direct it. When items reach either the player or the centre of the wormhole, they're removed. If the former, it results in a bonus for the player.

5.3 Key Components & Pseudo-code

This section will be split between the audio analysis, visualisation and game mechanic related code structures, beginning with audio.

Audio Code

The audio analysis for this project has been done almost exclusively by the minim audio library that's been used in conjunction with Processing [40]. The two main classes as part of minim that have been used are the FFT and BeatDetect classes respectively, both of which have been covered in the background section as far as theory and implementation go.

For a broader description of the Discrete Fourier Transform see the section 2.1.2 (f) of this document. For the FFT part in particular, see section 3.c further on from this. For

the beat detection description that the author of minim references, see section 2.1.4 (v).

5.3.1 Finding Maximums & Averages in a Frequency Spectrum

```
currentMaximum=0;
average=0;

FFTSample.findMaxAvg() {
    // find the highest value and total up all values
    for(each fft band) {
        temp=fft(i);
        tempTotal+=temp;
        if(temp>currentMax) {
            currentMax=temp;
        }
    }
    // find local average of this loop
    localAvg=tempTotal/fft.size();
    average*=counter;
    counter++;

    // update global average over life time of song + local maximum
    average=(average+localAvg) / counter;
    currentMaximum=currentMax;
}
```

5.3.2 Updating the Contractors

```
MainGame.for(NoGalaxies) {
    // add new Pulser object to list, pass in random number to be the type
    pulsers.add(new Pulser(random(0,3)));
}

MainGame.updatePulsers() {
    Pulser = pulsers.get(index);
    for(Pulser.ParticleList) {
        if((beat.kick&&Pulser.type==0) || (beat.hat&&Pulser.type==1) ||
            (beat.snare&&Pulser.type==2) {
            Pulser.gravitate();
            Pulser.ParticleList.display();

            if(canAddItems) {
                items.add(ItemDrop());
            }
        }
    }
}
```


5.3.3 Asteroid Shape Distortion

```
Asteroid.render() {
    fftSize=fft.avgSize();
    // set end and increment values
    difference=start-end+increment;
    //set radius based on difference value + quad size variable
    incrementSize=fftSize/difference;

    for(dimensions of cube) {
        // depending on side, pass in x,y,z values with one of those values
        // being based on the fft.avgSize() call.

        // front passes in x, y, a+fft.getAvg(); because the front panels should
        // move in the z direction
        drawSide(front);
        drawSide(back);
        drawSide(bottom);
        drawSide(top);
        drawSide(right);
        drawSide(left);
    }
}

Asteroid.drawSide() {
    switch(side) {
        // 4 vertex calls, different values for different sides
    }
}
```

Visualisation Code

5.3.4 Contractor Creation

```
PulserParticle() {
    if(startup) {
        // distribute around circle
        x = radius*cos(theta)*sqrt(1-(u*u));
        y = radius*sin(theta)*sqrt(1-(u*u));
    }
}
```

5.3.5 Rose Creation

```
Rose.setPoints() {
    for(i=0; i<noDots; i++) {
        theta = i*PI/2 / noDots;
```

```

        offset = sin(fft.avg*theta);
        point.radius = rose.radius * (offset);

        offset = cos(theta)*point.radius;
        point.x = centre.x + (offset);
        point.y = centre.y + (offset);
        ellipse(point.x, point.y, 2,2);
    }
}

```

5.3.6 Wormhole Draw

```

Wormhole.colourInc() {
    if(colour increasing) {
        colour+=random();
    }
    else {
        colour-=random();
    }

    difference=colour-(max);
    if(difference >max && decreasing) { switch flags }
    else if(difference < min && increasing) { switch flags }
}

Wormhole.render() {
    colourInc();
    if(beat) {
        // increment twist and ratio values
    }

    for(i=0; i<stars; i++) {
        pos.x = offsetFromCentre*sin(angle);
        pos.y = offsetFromCentre*ellipseRatio*cos(angle);

        tmp=radius*TwistingFactor;
        tmp2=sin(tmp);
        tmp3=cos(tmp);

        xx=centreX+(tmp2*pos.x)+(tmp3*pos.y);
        yy=centreY+(tmp3*pos.x)-(tmp2*pos.y);

        point(xx,yy,-radius/2);
    }
}

```

5.3.7 Particle Gravitate

```
PulserParticle.gravitate() {
  if(Particle.pos != newParticle.pos) {
    force = Particle.mass * newParticle.mass;
    mX = ( mass * x + Z.mass * Z.x ) / ( mass + Z.mass );
    mY = ( mass * y + Z.mass * Z.y ) / ( mass + Z.mass );
    tmpAngle = findAngle( mX - x, mY - y )

    mX = force * path;
    mY = force * path;

    mX += magnitude * path;
    mY += magnitude * path;

    magnitude = sqrt( sq(mX) + sq(mY) );
    angle = findAngle( mX, mY );
  }
}

PulserParticle.display() {
  // decrement magnitude, particles slowly stop

  x += magnitude * path;
  y += magnitude * path;

  line(px, py, x, y);
  px=x;
  py=y;
}

```

5.3.8 Menu Visualisation

```
Background() {
  for(noStars) {
    stars.add(position(random(width), random(height)));
  }

  for(noGalaxies) {
    // param=random radius size
    spirals.add(SpiralGalaxy(random(300)));
  }
}

Background.update() {
  if(beat) {
    SpiralGalaxy = spirals.get(random);
    SpiralGalaxy.beat=true;
  }
}

```

```

SpiralGalaxy.colourChange=true;
SpiralGalaxy.twist+=random();

if(SpiralGalaxy.stars<max) {
    for(amountOfStarsToAdd) {
        // add extra points to angle/radius array lists to be used
        // in render() to calculate x/y positions
    }
    SpiralGalaxy.stars+= amountOfStarsToAdd;
}
}
}

SpiralGalaxy.render() {
    for( SpiralGalaxy.stars ) {
        // calculate new x/y positions
    }
}

```

Game Mechanics Code

5.3.9 Missile-Asteroid Collision & Asteroid Split

```

Missile.render() {
    if(hasCollided) {
        asteroid.health-=missile.damage;
        if(asteroid.health<=0) {
            if(asteroid.parent && (game.difficulty=hard) {
                asteroid.splitAsteroid();
                asteroid.removing;
            }
        }
    }
}

Asteroid.splitAsteroid() {
    lives--;
    if(lives>0) {
        asteroids.add(Asteroid(collisionOffset));
        asteroids.add(Asteroid(-collisionOffset*2));
    }
}

```

5.4 Radar Draw

```

//asteroid, itemDrop, wormhole
drawRadarPoint() {
    if(movingRight) {
        //state=how points currently being rotated to match player.rotation
    }
}

```

```

    if(state==increasing&&currentRotationValue>=180) {
        // set flags for what to do when mouse is central
        // increment rotational values specific to state
    }
    else if(otherStates) {
        // set flags accordingly
        // increment and rotate in specific direction/amount
    }
    else {
        // mouse central
        // don't increment rotate value, use last value according to
        // which flag set above.
    }
}
else {
    // moving left
    // opposite rotation changes to above
}

// centre on x, offset by an amount equivalent to distance calculated between
// real position in 3d
ellipse(0, distance, 2,2);
}

```

5.4.1 Updating Asteroids

```

MainGame.Asteroids() {
    if(beat detect) {
        // calculate point on sphere + offset
        p = random(-PI, PI);
        t = asin(random(-1, 1));

        // set x + z to offset asteroids from player in width/depth
        if(game.difficulty=hard) {
            // also offset the start positions by height, to make it harder
            y = sphere.point + offset
        }
        // no god mode for asteroids generated by the music, can be destroyed
        // instantly
        asteroids.add(Asteroid(x,y,z), god=false);
    }

    for(closeAsteroids) {
        // those close enough to be drawn on the radar
        if(distance > 2000) {
            remove.asteroid;
        }
    }
}

```

```

for(asteroids) {
    //full list
    distance=thisAsteroid.position.Sub(player.position);
    theta = distance.angleBetween(player.direction);

    if(theta < x) {
        // player directly lined up with asteroid
        onTarget=true;
    }

    if(distance < 2000) {
        closeAsteroids.add(asteroid);
    }

    if(asteroid.dead || asteroid.hitPlayer()) {
        if(not already removing) {
            if(sfx) { play.soundEffect; }
            if(gameMode) { playerHealth-=20; }
        }

        asteroids.remove(asteroid);
    }
}
}

```

5.4.2 Updating Item Drops

```

MainGame.Items() {
    for(items) {
        distance=thisItem.position.Sub(player.position);
        theta = distance.angleBetween(player.direction);

        if(theta < x) {
            // player directly lined up with item
            onTarget=true;
        }

        if(onAttract && playerWeaponEnergy >=10) {
            //if trying to attract an item and has enough energy
            item.attract=true;
        }

        if(item.dead) {
            items.remove(item);
        }

        item.update();
    }
}

```

```

        item.render();
        // to force continual use of weapon energy to attract item
        item.attraction=false;
    }
}

```

5.4.3 Explosions

```

Asteroid.render() {
    translate(asteroid.pos);
    if(asteroid.removing) {
        asteroid.explosion.update(asteroid.alpha);
    }
}

```

```

Explosion() {
    for(noParticles) {
        // used for points around sphere
        float theta = random(0,TWO_PI);
        float u = random(-1,1);

        myParticleList.add(ExplosionParticle(theta, u));
    }
}

```

```

Explosion.update(alpha) {
    for(myParticleList) {
        myParticleList[i].update(alpha);
        myParticleList[i].render();
    }
}

```

```

ExplosionParticle.update(alpha) {
    alphaValue=alpha;
    // slow down expansion
    expansionIncrement -=0.1;
    radius+=expansionIncrement;
    tmpx=x;
    tmpy=y;
    tmpz=z;

    x = radius*cos(theta)*sqrt(1-(u*u));
    y = radius*sin(theta)*sqrt(1-(u*u));
    z = u*radius;
}

```

```

ExplosionParticle.render() {
    stroke(colour, alpha);
}

```

```

    line(tmpx, tmpy, tmpz, x,y,z);
}

```

5.4.4 Asteroid & Item Drop Steering

```

Vector steer() {
    locationTargetDir = Target.Sub(location);
    distance= locationTargetDir.magnitude();

    if(distance > 0) {
        // still need to adjust further to move it to target
        // dampening process based on boolean flag for target to slow down
        // when approaching
        locationTargetDir.mult(maxspeed);

        steer=velocity.sub( locationTargetDir );
        return steer;
    }
}

```

5.5 Efficiency & Bugs

5.5.1 Efficiency

On the default settings the game is very playable which for real time interaction was a major priority beyond graphical quality. Adjustable settings and the option to just play the system as a visualisation with no input necessary was added as an alternative method for visualising music.

Due to the gameplay being relatively basic and actions being repeated to different levels of accuracy, the performance of the game is able to drop a bit without an equivalent drop in the game anyway. Increasing the number of particles per Pulser object, the number of Pulser objects themselves as well as the number of stars in the background is the best way to upgrade the visual quality but the first two do degrade performance more significantly as values are increased due to the number of comparisons.

For collisions between asteroids and missiles, a tree structure as part of cutting down on comparisons was considered. However, given the issue of a dynamic set of objects (the asteroids are moving about) and the need for an updated tree, as well as the fact that there are never too many asteroids for comparison anyway, the decision was made that implementing and updating a tree would not be worth the benefit and would in fact degrade performance overall.

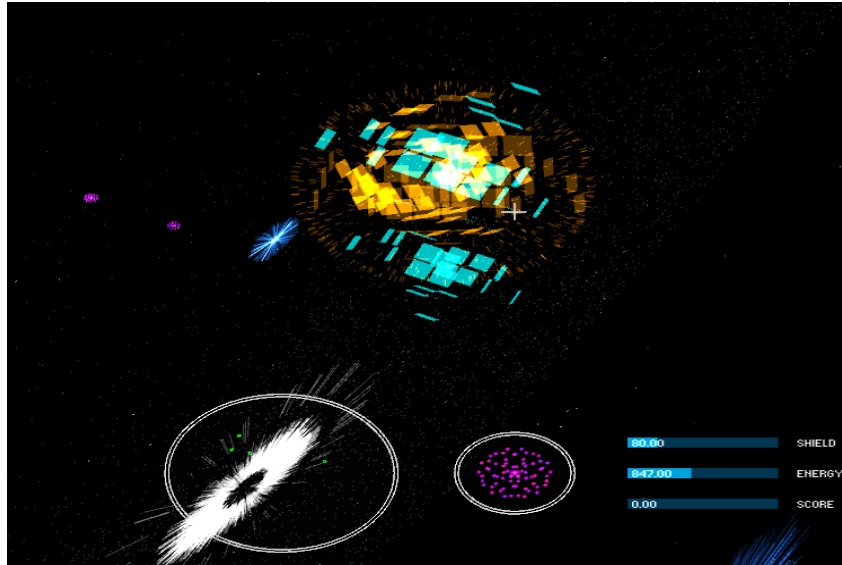


Figure 4.1 - An Example screenshot of an Asteroid splitting.

Both from a performance and a game point of view, various measures were taken to ensure that there was never too much happening at once anyway. For instance a limit of weapon energy means only so many missiles can be fired at a time, also missiles have a finite life span to ensure that they die quickly if they've missed the target. Item drops are based upon specific types of beat being recognised, this spawning was dampened both by adjusting a fixed beat sensitivity value and by ensuring that item drops are only spawned in accordance with a beat at a specific rate.

Asteroids move towards and are destroyed upon impact with the player meaning the continual creation of them is not slowly reducing performance either. Likewise item drops have a finite life span regardless of player interaction, this latter aspect with item drops forms a game purpose as well of forcing decision making up under pressure.

To take the system forward and really develop on it, multi-threading would need to be implemented because the game loop is simply a linear set of loops which results in periodic slow downs that while not hampering the gameplay significantly do impair somewhat on the level of fun and also arbitrarily limit the potential for expansion.

5.5.2 Bugs & Usability

Without a dedicated testing period it is difficult to say how bug free the system is but overall it seems to function as expected pretty consistently. The basic flow of the system, moving between main menu, game and game over screens to adjust settings, play and restart is fine. Using the default methods/shortcuts with the ControlP5 [Schlegel] GUI library sometimes left items on screen that made no sense in context but this was re-worked and custom shortcuts were added to allow the user more flexibility anyway.

The game itself seems to run exactly as expected. Sometimes expected beats are missed but this is a core issue of using audio in this way and to be expected. Settings

to adjust how the game flows in this regard give the user the ability to fine tune anyway. The game mechanics outlined in the user guide all function well and the system reacts as expected except for a few issues noted below:

1) Radar

The radar function on the GUI was implemented relatively early in development as an intuitive aspect of what would help the player and add an extra dimension to the gameplay. It is therefore something which, after a working version was done, has been left and in the future should be re-analysed for better solutions. On the whole it works fine and seems a real help to the player.

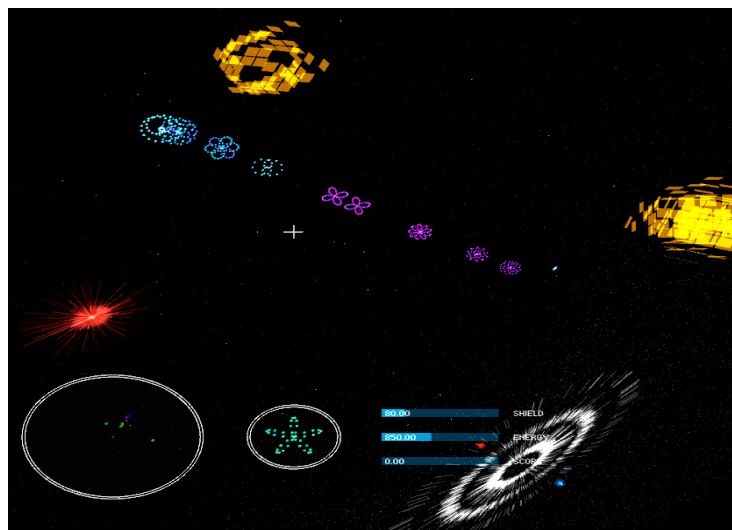


Figure 4.2 - Firing several missiles at an asteroid.

However the way in which angular differences are calculated and returned and how this should be updated on a 2d representation meant the system had to use this idea of states that the radar was in. This means that the display of the radar flicks between different states when the player is not rotating and this can be a little confusing. Also radar itself was not researched on exactly what it should offer and how it works but was implemented through a basic notion of what it should give you so a better understanding may lead to a better implementation.

Using it also takes some practice, as if an object is at an offset both in the X and Y in particular, to find the object requires a little trial and error. It becomes a mini game in itself of moving in some direction, seeing how this affects the position relative to the player, and adjusting the movement accordingly with the goal of trying to move the point to directly above the central red one representing the player. With a little practice this becomes quite easy but how it should work and whether it is a fun mechanic requires more research than is now possible.

2) Missile-Asteroid Collisions

This is only a slight issue but these collisions are not hugely accurate. Sometimes shots that seem to have missed will connect. This isn't really a bug as such because

this has been deliberately left in due to trying to encourage a certain style of play. The emphasis is less on the concentration of one shot and more on a broader picture of maintaining the player's overall safety which means a faster pace and less time for ensuring an asteroid has been removed before moving on to another.

Also, even if an asteroid is in the process of being removed with the alpha almost at zero, while it is still there, it will collide with a missile in exactly the same fashion as it always would. This can lead to some unintuitive moments when trying to blast through several asteroids, one behind the other. The one furthest back is completely protected until the nearest one is completely removed.

Despite some of these minor issues, the system itself reacts and runs exactly as it should do and based on the continued testing throughout later development, it works quite nicely and as expected.

See Appendix for more in-detail, documented code and development notes.

5.6 General Issues of Implementation

Some of the general issues throughout design and implementation were:

- 1) Switching in consideration from 2d to 3d in the game design. It needed some thought and a lot of ideas were thrown out/considered before starting to finally settle down on something.
- 2) Tending to favour audio-game related points over audio-visual ones that were more aesthetic based. The aspect of flow and immersion could be said to be largely connected with visuals and this lack of consideration possibly hampers the effectiveness of the final solution but it was a balancing act.
- 3) The environment that was not connected with game mechanics became very secondary and was not developed hardly at all which affected the final game in a substantial way.
- 4) Lack of more customisation and higher level analysis of the audio beyond the framework of minim rendered the solution somewhat restricted and narrower in scope but again this was a balancing act between using what was on offer to get something concrete done and spending more time on theory.
- 5) Didn't use as much of the initial research as planned much earlier on.
- 6) Performance tweaking became more of an issue throughout and needed to devote more time to it. Also therefore, more time on non performance related game enhancements rather than visual ones.

6. Conclusions

6.1 Objectives Comparison

On review of the design and a comparison with laid out goals, several areas will be approached:

6.1.1 Music Game Influences

Firstly the two main genres of music games to investigate, music art and hybrid types. The implemented system is essentially a hybrid music game so the latter has become the major focus of the approach and in this sense is a relative success depending on how fun the game is judged to be. It's a hybrid because it mixes a non musical game mechanic of this 3D asteroid modification with musical influences on the gameplay itself through driving enemy spawns with beat detection.

On the other hand, music art has not really been approached, mostly because the hybrid type is more concrete and well understood and in actually implementing a game, something simple and coherent took precedence over experimentation in this regard.

Next the specific games listed and why. Firstly, Rez based on it's abstract style. This was an interest due to wanting to remove much inherent meaning/complexity that might make up a more traditional setting because limiting the player focus on a narrower range is one way hoped to attune the player to a flow like experience in combination with an audibly up-lifting/heightening experience of the music. This aim has been kept to throughout, partly due to necessity of not using more complex, detailed, life like shapes in the system but also for the above reasons and the shape-like theme.

Audiosurf was noted for its use of audio to drive the pace of a game. This idea has been modified to spawn enemy objects that the player must deal with in a limited time range to successfully use audio to drive game pace.

Lastly, Vib-Ribbon was noted for allowing users to play it their own music collection and how in this way “Vib Ribbon fosters the attachment between a player and her music collection. The experience of playing Vib Ribbon is very personal”. This influence has also been heavy and corresponds strongly to the Audiosurf reference above because the system allows players to play their own songs and because the beat is such a fundamental and easily identifiable aspect of a song and because this varies the gameplay significantly, it is felt that a personalising aspect has also been achieved.

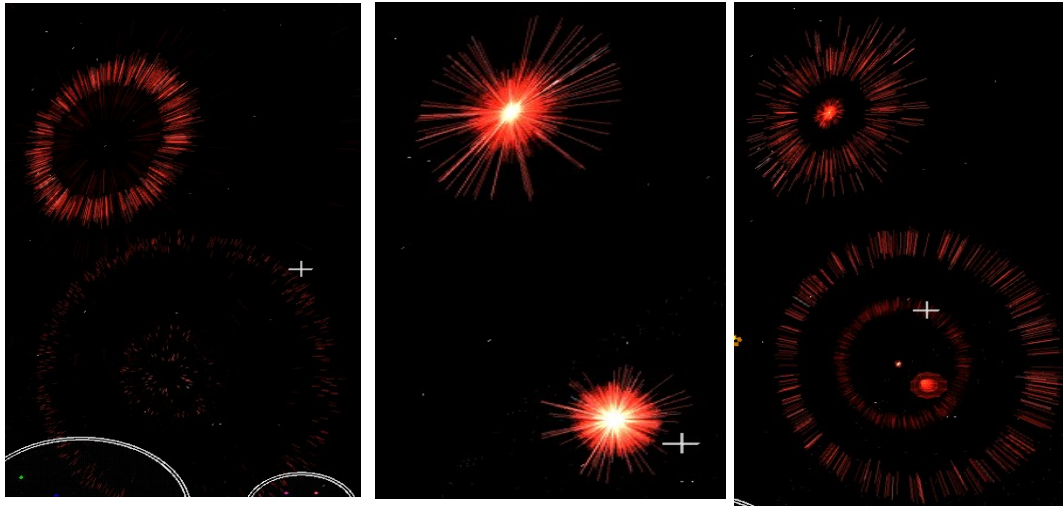


Figure 4.3 - Pulsar object gravitate method call example.

6.1.2 Standard Game Influences

In terms of non musical games, obviously Asteroids was a major design influence and what has emerged is really a 3d modification which extends the action in terms of goals. While it lacks the spaceship enemy that periodically appears in the original, it instead brings a cognizance to the asteroid like substitutes themselves by having them target the player instead of just randomly moving about.

This decision was made because in a larger 3d environment, the randomness would not have created a pace and tension that was desired. Also there is the inclusion of a broader range of goals from destroying asteroids, to collecting player boosts to the influence the music has over proceedings by focusing the player on survival up to a point in time.

Mono, the 2d game that developed a background environment over time through player interaction was more connected with the music art aspect and as such hasn't been implemented. Different approaches were considered such as slowly developing the brightness of stars and galaxies in the background based on where the player fired to reveal a complex and visually arresting environment by the end of the song but time has not allowed for this.

Arx Fatalis was noted for its use of mouse motion and understood patterns that lead to specific game functions like spellcasting. In this sense, the way missiles are fired and appear on screen was an influence that carries through, however there is no connection to a game mechanic besides aiming at asteroids to destroy them.

Lastly Polynomial was the major influence on the visual style and quality. The visual style is somewhat similar in so far as space environment always will be but also in the high contrast between bright and dark areas as well as strong colours. The way the music influences this is a positive. The visual quality of the final piece is not developed enough but is more held back by performance than anything so this is clearly something to be investigated further.

6.1.3 Audio, Visual & Game Relationship

The three main elements of this investigation were the audio analysis, the visual representation and the interaction aspect. Of this the balance between the audio-visual relationship and the visual-game relationship was noted in the design as being particularly important. This definitely turned out to be the case and was the source of most of the general issues that needed to be considered throughout.

Overall it was found that adding any game component to some visual effect is a non trivial issue and this aspect of the project ended up dominating most of the time due to the need to create good gameplay and constantly take the player into account. As such, and with the more concrete hybrid style being the focus, more experimental visuals and audio connections became somewhat neglected.

For instance, there is really only the manipulation of the wormhole through a phasing of its alpha value and a developing of its shape that does not serve some game purpose as far as visual aspect go. Everything else has some purpose of allowing the player to more easily comprehend their environment and actually play the game. For the game element, this is a positive and it seems to add to the consistency and interaction the player has.

Having said that, this has been at the expense of implementing features from a purely aesthetic point of view which could be said to hamper how enveloping the game is. So in conclusion of this aspect, to bring these two issues together probably requires a further iteration of audio analysis over the base game elements that are now in place or a re-think at the start of a project as to how the two are going to come together.

6.1.4 Approach to Audio Analysis

The other general approach worth noting is the philosophy of reducing the music to constituent parts and applying these aspects to isolated parts of the visuals. As opposed to a broader analysis of the music in terms of categories like emotion. This is another part of the design that has been kept to and really from a game aspect has paid off. For instance, interesting dynamics between the general noise of the song at any moment, how this visually transpires and what this means in a game for the player have developed out of the implementation.

One example would be the fact that during a quieter period, less or no asteroids are going to appear but the missile pattern and hence damage is also reduced meaning the player is safe during these times as long as they took advantage of the rush of beats earlier etc.

The sphere example in the design section of adjusting size, colour and position based on different kinds of beats etc is a good demo of the general approach but in reality it has been more limited and refined to maintain a good coherency and focus of attention. For example, the item drops are adjusted by size through the audio as they travel between two points. The position is based on fixed game elements or the

interaction of the player. The colour is one of three possibilities based on a beat detection but also that corresponds to a game related player boost.



Figure 4.4 - Background visualisation screenshot example.

6.1.5 Specific Design/Implementation Goals

Lastly, in [Pace08] a number of key game mechanics/goals were noted for encouraging immersion and flow. A re-cap of them alongside how they've been approached in the solution is below:

- 1) Clear goals and feedback
 1. The solution tries to offer a clear space environment/theme with simple controls and a clear link with the audio and asteroid generation. It combines this with the heads-up-display that shows stats such as a shield/weapon energy values and a basic radar that means even if the player isn't looking at a threat, the asteroid positions will be viewable at all times.
- 2) Game-skill balance
 1. This issue has been approached in two ways, offering a highly adjustable settings option particularly for difficulty and allowing the possibility of the player restoring themselves throughout or during ebbs in the flow through item drops.
- 3) Focused Attention
 1. The mix between the radar representation of the game world and the 3d view itself and how they relate and can be used interchangeably was intended to focus the attention on being surrounded and how the play should be approached. The radar gives a very clear picture of being surrounded and in danger in a way that a 3d view alone cannot. So they complement each other and create a more complete environment.
- 4) Reduced Awareness through simplistic goals

1. Asteroids was taken to be a basis for this reason and the item drops are not absolutely necessary for the completion of the game so the basic game mechanic is still the same throughout.
- 5) Feeling of presence in the game world.
 1. The pacing of the game along with the fact of quickly being surrounded on all sides and needing to rotate fast was hoped to engender a feeling of “being there” in the same way racing games tend to lead people to moving in the direction their car is adjusted etc.

6.2 Improvements and Future Work

- 1) Efficiency / Performance tweaking.
- 2) Visual quality/aesthetics like Asteroid trails and planets etc, more complete environment.
- 3) A further audio analysis iteration of development and use of music theory to add an extra layer of player understanding to the relationship with the audio
- 4) Incorporate a development of the background with the music/play a la 'Mono' **[Binary05]**
- 5) Player action-audio feedback loop. Player's can create a sound which is fed back into the audio analysis/becomes part of it.
- 6) Greater variety of dangerous objects in accordance with broader audio analysis.
- 7) Complete physics system of attraction/repulsion/bounce etc between all objects inheriting from a generic base.

7. Bibliography

- 1) [**Bernstein**] Bernstein, J., T., “Traer Physics”, Princeton. Accessed on July 5 2009. Available from: <<http://www.cs.princeton.edu/~traer/physics/>>
- 2) [**Binary05**] “Binary Zoo”, “Mono”. Las6 2005. Accessed on July 10 2009. Available from: <http://www.binaryzoo.com/games/mono/images/mono_4.jpg>
- 3) [**Brown93**] Brown, J. C., “Determination of the meter of musical scores by autocorrelation” 1993, J. Acoust. Soc. Am. 94, 1953–1957. Accessed on July 10 2009. Available from: <<http://www.wellesley.edu/Physics/brown/pubs/meterACv94P1953-P1957.pdf>>
- 4) [**Bumgardner**] Bumgardner, J., “Rose Equation”, OpenProcessing. Accessed on July 5 2009. Available from: <<http://openprocessing.org/visuals/?visualID=1555>>
- 5) [**Csikszentmihalyi90**] Csikszentmihalyi, M., 1990. “Flow: the psychology of optimal experience”. Harper, New York.
- 6) [**Desain99**] Desain, P., Honing, H. “Computational Models of Beat Induction: The Rule-Based Approach. Journal of New Music Research.” 1999. Accessed on July 10 2009. Available from: <<http://www.nici.kun.nl/mmm/papers/dh-100/dh-100.pdf>>
- 7) [**Dixon**] Dixon, S., “Automatic Extraction of Tempo and Beat from Expressive Performances”. Austrian Research Institute for Artificial Intelligence. Accessed on July 10 2009. Available from: <<http://www.ofai.at/cgi-bin/get-tr?paper=oefai-tr-2001-19.pdf>>
- 8) [**Douglas01**] Douglas, J. Y. & Hargadon (2001): A. “The pleasures of immersion and engagement: schemas, scripts and the fifth business”. Digital Creativity, 2001, Vol. 12, No. 3, pp. 153–166.
- 9) [**Eliassen07**] Eliassen, Jedrik., 2007, “Beat Tracking to Control Lighting”, Jedrik Eliassen. Accessed on July 6 2009. Available from: <<http://nccastaff.bournemouth.ac.uk/jmacey/MastersProjects/MSc07/Jed/theses.pdf>>
- 10) [**Ellis**] Ellis, Daniel P.W. “Beat Tracking with Dynamic Programming.” Accessed on July 10 2009. Available from: <<http://www.ee.columbia.edu/~dpwe/pubs/Ellis06-beattrack.pdf>>
- 11) [**Foote**] Foote, J., “Visualizing Music and Audio using Self-Similarity”. FX Palo Alto Laboratory, Inc. Accessed on July 10 2009. Available from: <<http://www.fxpal.com/publications/XPAL-PR-99-093.pdf>>

- 12) [**Flyud**] Flyud, Y., “Music Box”, OpenProcessing. Accessed on July 10 2009. Available from: <<http://www.openprocessing.org/visuals/?visualID=2545>>
- 13) [**Fry**] Fry, B., Reas, C., “Processing”, Accessed on July 1 2009. Available from: <<http://processing.org/>>
- 14) [**Gametap09**] “GameTap LLC”, “Arx Fatalis”, TM & © 2009 GameTap LLC. Accessed on July 10 2009. Available from: <<http://assets.gametap.com/repository/ARFA/us/images/screenshots/original/screenshot4.jpg>>
- 15) [**Geek09**] “Geek.com, LLC”, “Karaoke Revolution”, 1996-2009, Accessed on July 10 2009. Available from: <http://www.geek.com/wp-content/uploads/2008/03/karaoke_revolution_422x317.jpg>
- 16) [**Gendou**] 'gendou', “Nightsky3d”, OpenProcessing. Accessed on July 10 2009. Available from: <<http://www.openprocessing.org/visuals/?visualID=2277>>
- 17) [**Gonzales**] Gonzales, C., “Gravity Swarm”, OpenProcessing. Accessed on July 10 2009. Available from: <<http://www.openprocessing.org/visuals/?visualID=2363>>
- 18) [**Goto94**] Goto, Masataka and Yoichi Muraoska. “A Beat Tracking System for Acoustic Signals of Music.” Proceedings of the second ACM international conference on Multimedia. 1994. Accessed on July 10 2009. Available from: <<http://staff.aist.go.jp/m.goto/PAPER/ACM94goto.pdf>>
- 19) [**Gouyan**] Gouyan, F., Herrera, P., “A Beat Induction Method for Musical Audio Signals”, Accessed on July 10 2009. Available from: <<http://www.iaa.upf.es/mtg/publications/WIAMIS03-fgouyonbeat.pdf>>
- 20) [**Guglielmetti**] Guglielmetti, P., “Spiral Galaxy”, OpenProcessing. Accessed on July 10 2009. Available from: <<http://www.openprocessing.org/visuals/?visualID=699>>
- 21) [**IGN09**] “IGN Entertainment, Inc”, “Guitar Hero ii”, 1996-2009, Accessed on July 10 2009. Available from: <<http://ps2media.gamespy.com/ps2/image/article/709/709091/guitar-hero-ii-20060517053840543.jpg>>
- 22) [**Indie**] “Indie Games, Think Services.”, “Polynomial”. Think Services. Accessed on July 10 2009. Available from: <<http://www.indiegames.com/blog/images/timw/polynomi3a.jpg>>

- 23) **[MathArt]** “Math Art”, “Lorenz Attractor – A 3D Render”, Accessed on July 15 2009. Available from: <<http://math-art.net/2007/12/02/lorenz-attractor-a-3d-render/>>
- 24) **[Meudic]** Meudic, B., “A Causal algorithm for beat-tracking.” Accessed on July 10, 2009. Available from: <<http://mediatheque.ircam.fr/articles/textes/Meudic02b>>
- 25) **[Minim]** Unknown, “Minim”, Accessed on July 2 2009. Available from: <<http://code.compartmental.net/tools/minim/>>
- 26) **[Mizuguchi07]** Mizuguchi, T., “Q&A: Every Extend Extra's Tetsuya Mizuguchi”. Interview 2007. Accessed on July 10 2009. Available from: <<http://www.gamespot.com/news/6164638.html?sid=6164638>>
- 27) **[Next09]** “nextmedia Pty Ltd”, “Audiosurf”, 2009, Accessed on July 10 2009. Available from: <<http://www.pcpowerplay.com.au/content/images/stories/audiosurf.jpg>>
- 28) **[Pace08]** Pace, S., “Immersion, Flow and the Experiences of Game Players”. Central Queensland University, SimTecT 2008. Accessed on July 10 2009. Available from: <www.siaa.asn.au/get/2451314720.pdf>
- 29) **[Patin]** Patin, F., “Beat Detection Algorithms” Accessed on July 10 2009. Available from: <<http://www.flipcode.com/misc/BeatDetectionAlgorithms.pdf>>
- 30) **[Pichlmair07]** Pichlmair, M., Kayali, “Levels of Sound: On the Principles of Interactivity in Music Video Games”. Situated Play, Proceedings of DiGRA 2007 Conference. Accessed on July 10 2009. Available from: <<http://www.digra.org/dl/db/07311.14286.pdf>>
- 31) **[Ruiz]** Ruiz, M., P., “Mathematical Art in Japan”. Ateneo de Manila University. Accessed on July 15 2009. Available from: <<http://www.eurocg.org/www.us.es/ewcg04/mathlartinjapan.PDF>>
- 32) **[Rocchesso03]** Rocchesso, Davide, 2003, “Introduction to Sound Processing”, Davide Rocchesso. Accessed on July 8 2009. Available from: <<http://www.faqs.org/docs/sp/>>
- 33) **[Scheirer97]** Scheirer, E., D., “Tempo and beat analysis of acoustic musical signals.” 1997. Machine Listening Group, E15-401D MIT Media Laboratory. Accessed on July 10 2009. Available from: <http://www.iro.umontreal.ca/~pift6080/H09/documents/papers/scheirer_jasa.pdf>
- 34) **[Schlegel]** Schlegel, A., “ControlP5”, 2000-2009, Accessed on July 20 2009. Available from: <<http://www.sojamo.de/libraries/controlP5/index.html>>

- 35) [**Seppanen01**] Seppanen, J., “Computational models of musical meter recognition”. M.Sc. Thesis, Tampere University of Technology, 2001. Accessed on July 10 2009. Available from: <http://www.cs.tut.fi/sgn/arg/music/jams/mscthesis-seppanen2001.pdf>>
- 36) [**Shiffman**] Shiffman, D., “Seek-Arrive”, Daniel Shiffman. Accessed on July 5 2009. Available from: http://www.shiffman.net/itp/classes/nature/week06_s09/seekarrive/Boid.pdf>
- 37) [**Smith07**] Smith, W. Steven, 1997-2007, “The Scientist and Engineer's Guide to Digital Signal Processing”, California Technical Publishing. Accessed on July 8 2009. Available from: <http://www.dspguide.com/pdfbook.htm>>
- 38) [**Starkes**] 'Starkes', “Particles on a Sphere”, OpenProcessing. Accessed on July 10 2009. Available from: <http://www.openprocessing.org/visuals/?visualID=861>>
- 39) [**Synth**] “Synthesis Paradigms”, “Elektroplankton”, Accessed on July 10 2009. Available from: http://www.synthesisparadigms.com/articles_datas/memory/pictures/Elektroplankton_1.jpg>
- 40) [**Armchair**] “The Armchair Empire”, “Rez”, Accessed on July 10 2009. Available from: <http://www.armchairempire.com/images/Reviews/Playstation2/rez/rez-2.jpg>>
- 41) [**Tzanetakis1**] Tzanetakis, G., “Tempo Extraction using Beat Histograms.” University of Victoria. Accessed on July 10 2009. Available from: <http://www.music-ir.org/evaluation/mirex-results/articles/tempo/tzanetakis.pdf>>
- 42) [**Tzanetakis2**] Tzanetakis, G., Essl, Cook. “Audio Analysis using the Discrete Wavelet Transform” Accessed on July 10 2009. Available from: http://soundlab.cs.princeton.edu/publications/2001_amta_aadwt.pdf>
- 43) [**Weisstein**] Weisstein, E., “Rose”, Wolfram Research, Eric Weisstein. Accessed on July 15 2009. Available from: <http://mathworld.wolfram.com/Rose.html>>
- 44) [**Wikipedia1**] “Wikipedia”, “Lorenz Attractor”, Accessed on July 15 2009. Available from: http://en.wikipedia.org/wiki/Lorenz_attractor>
<http://en.wikipedia.org/wiki/File:Lorenz_Ro28-200px.png>
- 45) [**Wikipedia2**] “Wikipedia”, “Music and Mathematics”, Accessed on July 15 2009. Available from: http://en.wikipedia.org/wiki/Music_and_mathematics>
- 46) [**Wikipedia3**] “Wikipedia”, “Mandala”, Accessed on July 15 2009. Available from: <http://en.wikipedia.org/wiki/Mandala>>

<http://en.wikipedia.org/wiki/File:Mandala_gross.jpg>

47) [Wikipedia4] “Wikipedia”, “Music Video Game”, Accessed on July 15 2009.
Available from: <http://en.wikipedia.org/wiki/Music_video_game>

8. Appendix

8.1 Tools Overview

8.1.1 Processing [Fry]

Processing is “an open project initiated by Ben Fry and Casey Reas. It evolved from ideas explored in the Aesthetics and Computation Group at the MIT Media Lab.” It’s described as “an open source programming language and environment for people who want to program images, animation, and interactions.” Based on a survey of example programs, mostly found at (OpenProcessing), it hasn’t been used a great deal for games, especially 3D games.

However, it does contain a basic framework for a game to be built upon in the form of draw, setup and loop methods as well as input control. It’s also java based of which the author has some experience with and based upon some early experimentation was found to be very good at allowing quick and easy implementation of ideas which is considered to be a particularly important benefit for a more experimental project of this sort.

Over time through volunteer development it has also become capable of some very striking visual effects and contains OPENGL support in the form of a wrapper library called Java OpenGL(JOGL) to deal with the greater processing power this will offer over the Java2D engine.

Lastly, the core of the Processing library is quite small, efficient and elegant in design and it’s kept separate from the growing number of varied libraries that can be used to provide specific support such as GUI’s, physics support and importantly audio libraries. This will hopefully aid in easier, incremental and component based design of the final product.

<http://processing.org/>

8.1.2 Minim audio library [Minim]

‘Minim’ is one of several audio libraries now available for use with Processing. It’s noted as “an audio library that uses the JavaSound API, a bit of Tritonus, and Javazoom’s MP3SPI to provide an easy to use audio library for people developing in the Processing environment.”

<http://code.compartmental.net/tools/minim/>

The specific details regarding some of the techniques the library uses for audio analysis that are relevant to this work have already been outlined in the background but how the classes and methods are used will be expanded upon in the Implementation section (5).

It was primarily chosen due to the clear design and also the thorough documentation listed on the website as well as the example files. These things also helped to tie the more technical research element with specific implementation examples like the FFT class.

8.1.3 ControlP5 GUI Library [Schlegel]

The ControlP5 GUI library will be used to provide extra controls, flexibility and information to the player for both setting up and playing the game. It was made by Andreas Schlegel and describes on the main page how “controllers can be added to a processing sketch itself, to separate control windows, and can be organized in tabs.” The standard look and feel of the widgets also fits in nicely with the futuristic theme of the game too.

Lastly it provides shortcut support for the GUI to be hidden allowing for more player customisation and possibly as part of an alternative, free-form/visualisation mode which is a nice benefit as well. Again, the documentation is well laid out and provides enough information to get started including example files.

8.2 Beat Induction/Tracking Extra

8.2.1 Beat Induction

There are a variety of different uses of the term “beat tracking” and they often coincide with another term, “beat induction”. The following sections will give a brief outline of the literature found regarding these areas in DSP and the relation to this project. The most common understanding of these terms seems to label beat induction as identifying the beat rate while beat tracking as finding the beat locations.

Beat induction is typically subsumed or becomes a part of the larger process of beat tracking. So we can say beat induction is the processing of identifying a sufficient pulse to categorise it as a possible beat. Beat tracking takes a list of possible events and analyses them against several criteria for the “best beat”.

Gouyon [**Gouyon**] describes it this way, “A beat is characterized by a period and a phase, that is, the distance between two beats and the temporal location of the first beat. The tempo is inversely proportional to the beat period. If the tempo changes with time (as it occurs in real-life musical examples), beat period and phase have to be regularly updated. This is the process of beat tracking. A different process is that of beat-induction: the determination of one (or possibly several, ranked) candidate(s) as input for a beat-tracker.”

With this distinction made, the interest of this thesis is really with the (higher level), “beat tracking” so more attention will be given to that in the next section but below briefly outlines some of the key authors, papers and methods of beat induction found, sometimes as part of a larger beat tracking system.

Brown [Brown93] uses an auto-correlation method along with typical onset detection. A paper by [Desain99] describes a rule based approach. Foote [Foote] uses the idea of self similarity against lag time through a matrix representation of the similarity between frames with a beat represented by a maximal peak. Scheirer [Schreier97] uses a comb filter as opposed to auto correlation and argues that “a rhythmic processing algorithm should treat frequency bands separately, combining results at the end, rather than attempting to perform beat tracking on the sum of filterbank outputs”.

Tzanetakis et al [Tzanetakis1] propose the “beat histogram”, of which the aim is to collect statistics about the amplitude envelope periodicities of multiple frequency bands. Seppanen [Seppanen01] uses tick indexes to answer the question, “is this tick strong, i.e. a beat or weak, i.e. in between beats?” This summary of techniques was found in [Gouyan] who essentially uses a technique similar to [Brown93] but without the onset detection based on it being, “a difficult process without prior information regarding the sources making up the signal”. Instead they make use of [Foote] and the idea of low level descriptors instead.

8.2.2 Beat Tracking

i. Beat Tracking with Dynamic Programming [Ellis]

It's stated that “Beat tracking – i.e. deriving from a music audio signal a sequence of beat instants that might correspond to when a human listener would tap his foot – involves satisfying two constraints: On the one hand, the selected instants should generally correspond to moments in the audio where a beat is indicated, for instance by the onset of a note played by one of the instruments. On the other hand, the set of beats should reflect a locally-constant inter-beat interval, since it is this regular spacing between beat times that defines musical rhythm.” [Ellis]

The system is broken down into three main areas: [Ellis]

- 1) Onset strength signal
 - a. “The first stage of processing is to convert the audio into a one-dimensional function of time at a lower sampling rate that reflects the strength of onsets (beats) at each time.”
- 2) Tempo Estimation
 - a. Next the onset strength is auto-correlated with this raw data and scaled by a window that captures so called, “intrinsic bias of listeners towards a particular range of tempi” rendering multiple peaks to a single dominant one.
- 3) Beat Tracking
 - a. Finally the best BPM (taken from the tempo estimation) is passed to this module that “attempts to find a sequence of beat times that all correspond to large values in the onset waveform”. The usage of a beat history provides a balancing act between good local matches and ones that take prior history into more direct account.

- b. Further, “The advantage of dynamic programming is that it effectively searches all possible sets of beat instants, since it is guaranteed to find the best-scoring sequence up to any point. This allows the best global beat sequence to be found, even if it involves some locally-poor matching, for instances beats that occur during silence or uninflected sustained notes.” [Ellis]

ii. A Causal Algorithm for Beat Tracking [Meudic] vs Automatic Extraction of Tempo and Beat from Expressive Performances [Dixon]

In [Meudic] a comparison is made with [Dixon] to implement something similar but with real time functionality and making use of markings to “detect salient rhythmic events”. [Meudic] identifies the three stages that are common to both systems as:

- 1) Induction
 - a. Where possible beats are listed from the beginning music sequence
- 2) Propagation
 - a. Beats along the analysed sequence are propagated whereby the events in the sequence which *could* correspond to beat occurrences are chosen.
- 3) Extraction
 - a. Lastly this list is sorted, “according to several criteria (among which a kind of musical knowledge is used) in order to select the 'best beat'.”

[Dixon] describes the use of an agents for beat tracking whereby “The beat locations are determined by an agent-based architecture which simultaneously examines multiple hypotheses about the frequency and phase of the beat throughout the music. The agents are characterized by their state and history. The state is the agent’s current hypothesis of the beat frequency and phase, and the history is the sequence of beat locations selected so far by the agent. Each agent is evaluated on the basis of its history, with higher scores being awarded for greater regularity in the spacing between events, greater salience of chosen events, and fewer gaps in the sequence.”

[Meudic] in discussing propagation states “the algorithm selects the event the most weighted in the tolerance window whereas Dixon duplicates the agents when ambiguity arises. Doing this, we dramatically reduce the number of possible beats, which makes our algorithm faster”. It is also noted that “the markings are used not only in the final beat extraction step, but also in the two other steps : Concerning the beat induction, the markings filter the events so that the only most weighted ones are taken as possible positions for the phase value of new beats (whereas Dixon considers all the positions of all the events contained in a given window).

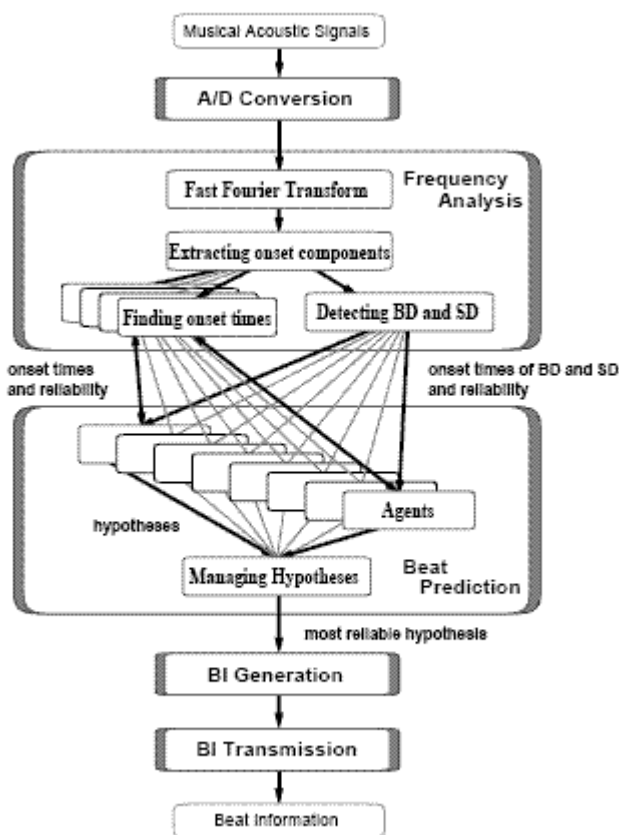
Lastly in [Meudic] if an event occurs as expected but another event occurs in the same window that is more weighted, this new event takes precedence which allows for a finer degree of tempo variance. On the other hand in [Dixon] such an initial event would be listed definitively.

iii. A Beat Tracking System for Acoustic Signals of Music [Goto/Muraoka]

The intent of their system against existing implementations as “Previous systems were not able to deal with acoustic signals that contained sounds of various instruments, especially drums. They dealt with either MIDI signals or acoustic signals played on a few instruments, and in the latter case, did not work in real time. Our system deals with popular music in which drums maintain the beat. Because our system examines multiple hypotheses in parallel, it can follow beats without losing track of them, even if some hypotheses become wrong.” [Goto94]

In discussing issues of signal processing they note that a “musical beat” may not even correspond to a signal at a given sample point. In describing the beat tracking system they state they do so by “managing multiple evidence such as onset times in several different frequency ranges, onset times of two different kinds of drum-sounds (a bass drum and a snare drum)”. [Goto94]

This is something that minim easily allows for and of which will be a key part of the implementation of this project. The frequency analysis and its relation to actual musical instruments will be key but in a games/music context, such analysis must be broadly successful over as wide a range of music as possible to encourage as much diversity in a player base as possible while maintaining the music/visual/interaction synchronisation at the heart of the experience. A diagram listing the basic setup which also summarises how several of the topics covered link in is given in their paper and represented again below:



Here, A/D is acoustic/digital conversion and BD/SD corresponds to base and snare drums respectively.

“First, Frequency Analysis finds notes’ onset times in an input acoustic signal digitized by A/D Conversion and also detects BD and SD. Second, multiple agents in Beat Prediction interpret the onset times found previously and make parallel hypotheses: each agent first calculates the IBI (inter-beat-interval); it then predicts the next beat time, and infers its beat type, and finally evaluates its own reliability. BI Generation assembles BI (beat information) on the basis of the most reliable hypothesis. Finally, BI Transmission transmits the BI to other application programs via a computer network.” [Goto94]

As you can see it is using a similar general idea of multiple agents/hypothesis as other implementations in combination with musical knowledge in terms of beat relations etc.

iv. Audio Analysis using Discrete Wavelet Transform (DWT) [Tzanetakis2]

The WT is described as “a technique for analyzing signals. It was developed as an alternative to the short time Fourier Transform (STFT) to overcome problems related to its frequency and time resolution properties.” With the DWT more specifically noted as “a special case of the WT that provides a compact representation of a signal in time and frequency that can be computed efficiently.” [Tzanetakis2]

In terms of the feature extraction and classification the following features are present:

- “The mean of the absolute value of the coefficients in each subband. These features provide information about the frequency distribution of the audio signal.
- The standard deviation of the coefficients in each subband. These features provide information about the amount of change of the frequency distribution
- Ratios of the mean values between adjacent subbands. These features also provide information about the frequency distribution.” [Tzanetakis2]

The beat detection itself is comprised of the following steps:

- 1)Low pass Filter
- 2)Full wave Rectification
- 3)Downsampling
- 4)Normalization
- 5)Auto-correlation

8.3 Code Usage

1)Wormhole + SpiralGalaxy

1.Philippe Guglielmetti – Spiral Galaxy - <http://www.openprocessing.org/visuals/?visualID=699>

2)Camera rotation

- 1.'gendou' – Nightsky3d - <http://www.openprocessing.org/visuals/?visualID=2277>

3)Pulser

- 1.Claudio Gonzales – Gravity Swarm - <http://www.openprocessing.org/visuals/?visualID=2363>

4)Asteroid

- 1.Yurko Flyud – Music Box - <http://www.openprocessing.org/visuals/?visualID=2545>

5)Steer and Collision Methods

- 1.Daniel Shiffman - Boid class - http://www.shiffman.net/itp/classes/nature/week06_s09/seekarrive/Boid.pde

6)Particle Spherical distribution

- 1.stars/galaxies/asteroids
- 1.'gendou' – Nightsky3d - <http://www.openprocessing.org/visuals/?visualID=2277>
- 2.Expllosion
- 1.'Starkes' – Particles on a Sphere - <http://www.openprocessing.org/visuals/?visualID=861>

7)Rose

- 8)Jim Bumgardner – Rose Equation - <http://openprocessing.org/visuals/?visualID=1555>**

8.4 Development Notes

8.4.1 Overview of Progress

- 1.Extended Particle Swarm example to be distributed around a circle and located at points around a sphere surrounding the player. Named 'Contractor'.
- 2.Added an audio link to the attract() method call of each contractor object to “pulse” when there was a beat onset.
- 3.Modified the 'cubus', beating cube example to take the place of asteroids in this space environment. Made up of outer sides only rather than full cubes. Distributed them all initially around randomly chosen points around the same sphere dimensions as above. Random direction.
- 4.Re-implemented the 'Rose' example, changing pattern based on the maximum peak value in a frequency spectrum reading. Displayed as part of the GUI.

5. Implemented and associated a 'Laser' class with the Rose class to fire objects in the direction the player is facing. Display of the laser object is the current rose pattern in larger form.
6. Added collision detection between laser objects and asteroids.
7. Added a radar like system to the main in-game interface to display asteroid positions relative to the player. Player position at the centre, all other asteroids plotted around centre depending on the dot product value between player direction and vector facing asteroid. Also takes into account distance.
8. Extended the asteroid class to use Boid steer() method to aim all asteroids at the player.
9. Added collision detection between asteroid and player.
10. Added player stats like a 'shield' value to form the basic gameplay for testing purposes. Experimented with different asteroid velocities, beat sensitivities and some different songs.
11. Implemented the galaxy method from the relevant sketch and distributed it randomly at some point around the player. Adjusted and modified it to look more like a wormhole type structure. Also made other changes visually to improve the appearance.
12. Created an item drop class that would serve as boosters for the player stats. Basic idea of items being pulled to the wormhole at which point they are destroyed. The player has the ability to interrupt this pull and attract items to him/herself and upon "collision" with the player, depending on an item type variable would receive one of several types of boost like shield increase.
13. Spawned item drops from centre of some contractor.
14. Added more player stats: shield, weapon energy, weapon modifier and score.
15. Correlated types of items with types of Contractor's and associated them with the player stats. Attracting items to player decreases weapon energy but adds extra level of consideration depending on gameplay situation.
16. Added item drop positions to radar as well as wormhole.
17. Added explosion class, one instance per asteroid.
18. Implemented all GUI's and menu systems and movement between different screens. Player stats now visually available for player during the game. Menu allows for initial tweaking of system, varying complexity and visual quality. In-game settings more connected with real time variables that affect gameplay and difficulty. Final scores upon song end or player death are displayed and player returns to main menu.

19. Made new main class and split all game relevant code into separate class with menu system and gui stuff as well as a game object all being handled in new main class.

20. New class for containing all of the audio analysis objects, one for main program and menu music, another for game music analysis.

8.5 Code Listing

```
/* ObjectSpace Game
13/08/09

author: Ashley Morrison
class: Main/ObjectSpace
brief: Main program file containing setup,loop and input functions.
       Contains game, menu and audio objects and flags to move user between screens.
*/

import processing.opengl.*;
import javax.media.opengl.*;
import traer.physics.*;
import ddf.minim.*;
import ddf.minim.analysis.*;
import controlP5.*;
import javax.swing.*;

/* OpenGL graphics objects used for extra effects, i.e. blending. */
PGraphicsOpenGL pgl;
GL gl;

/* main - class containing text outputs for the start and game over screens */
MainMenu main;

/* game - class of the game object containing all game related methods/items. */
MainGame game;

/* newGUI - class that uses the ControlP5 library for loading in and displaying the relevant user
interface.
       Includes main menu, in-game settings and in-game player stats panels. */
GUI newGUI;

/* titleSong - class containing all the related minim audio library objects and calls for playing and
       analysing the audio for the main menu and in-game song as well. This object is for the main
       menu. */
MinimAudio titleSong;

/* laser - minim library class item for loading in and accessing the laser sound effect played when the
       user fires the weapon */
AudioSnippet laser;

/* explosion - minim library class item for loading in and accessing the explosion sound effect played
       when the player collides with an asteroid */
AudioSnippet explosion;

/* newBackground - class containing the Spiral Galaxy objects used for the main menu visualisation */
Background newBackground;

/* R - default radius used for the sphere around which most items are distributed. */
final static int R = 1000;

/* NoStars, NoGalaxies - Default values displayed for the menu gui for how many stars and galaxies
       will be drawn in game. */
int NoStars = 1000, NoGalaxies = 5;//, NoAsteroids = 10;//, NoStars2 = 10000;
```

```

/* wormholeNoStars - Default value displayed in the menu gui for how many points to be used to draw
the wormhole object. */
int wormholeNoStars=10000;

/* wormholeRmax - Default value for the radius of the wormhole used in the render method for
distributing the points. */
int wormholeRmax=500;

/* galaxyParticleNo - Default value for the number of lines used to render the Pulsar objects to screen.
*/
int galaxyParticleNo=500;

/* galaxyRadius - Default value for the radius of the Pulsar object particles to be distributed around. */
int galaxyRadius=750;

/* beatSensitivity - Default value for the beat detection dampening amount. Larger value means more
dampening, less beats detected, less asteroids generated */
int beatSensitivity = 5000;

/* filename - Empty string value for the filename of the song to use for the game. If left empty, a
default song is set in the game.minimObj object, otherwise, gets set
in the fileButton() method in this file. */
String filename = "";

/* filename2 - Default string for the titleSong object, the song played on the menu screen */
String filename2 = "prodigy.mp3";

/* mainMenu, inGame, gameOver,win - Boolean flags used to determine what state the system is in and
what actions should be taken. 3 main states, menu screen, in-game and game
over screens. */
boolean mainMenu=true,inGame=false,gameOver=false;

/* setup - Setup method, each processing sketch contains one. Used to set the size, graphics API to use
and also creates the Background, MainMenu, newGUI and titleSong objects needed to start the game in
the menu screen state for options to pick and start a new game */
void setup() {

    try {
        UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
    }
    catch (Exception e) {
        e.printStackTrace();
    }

    size(1024, 768, OPENGL);
    //hint(ENABLE_OPENGL_4X_SMOOTH);
    perspective(PI/3.0, width/height, 0.1, 50000);

    newBackground = new Background();
    main = new MainMenu();
    newGUI = new GUI(this);
    titleSong = new MinimAudio(this, true);
}

```


/ draw - Draw method, also used once by a processing sketch, this is the main loop method. Sets up options for the graphics blending and determines the state of the system and what actions to take. If inGame, render the current state of the game object. If gameOver, render the MainMenu object in this state and if MainMenu, update the background visualisation, the gui controls and render relevant text to the screen (title).*

```

*/
void draw() {
  background(0);

  /* OPENGL FUNCTIONS */
  pgl = (PGraphicsOpenGL) g;
  gl = pgl.gl;

  gl.glDisable(GL.GL_DEPTH_TEST);
  gl.glEnable(GL.GL_BLEND);
  gl.glBlendFunc(GL.GL_SRC_ALPHA, GL.GL_ONE);
  /* OPENGL FUNCTIONS */

  /* If not in-game and game object has been set, also check if the game music is playing and stop it, if
  it is. */
  if(!inGame && game!=null) { if(game.minimObj.song.isPlaying()) { game.minimObj.song.pause(); }
  }

  /* If in game over state, render the MainMenu object which has it's own state check and renders
  differently depending on state. */
  if(gameOver) { main.render(); }

  /* if main menu state, update the background vis, the menu gui and ensure the in-game player data is
  hidden just in case and lastly render the MainMenu object. */
  if(mainMenu) {
    newBackground.update();
    newBackground.render();
    newGUI.update2();
    if(newGUI.gameData!=null) { newGUI.gameData.hide(); }
    main.render();
  }

  /* if in-game and game object is not null, ensure the meny gui is hidden and render the game object
  with it's current status. */
  if(inGame && game!=null && newGUI.gameData!=null) { newGUI.mainmenu.hide();
  game.render(); }

}

```

/ stop - Stop method, third main processing method, called when the system is exited. Uses relevant state to determine exactly which objects to close etc */*

```

void stop() {
  if(game!= null) {

    /* if leaving system and in menu, end the menu song */
    if(mainMenu) { titleSong.song.close(); titleSong.minim.stop(); }

    /* If leaving system and game object music is playing, stop and close it */
    if(game.minimObj.song.isPlaying()) {
      game.minimObj.song.close();
      game.minimObj.minim.stop();
    }
  }
}

```

```

    }

    /* If sfx on and laser is currently set, close it */
    if(laser!=null) { laser.close(); }
    }
    super.stop();
}

/* playButton - Play Button method, called when the play button on the menu gui is called. Launches
the game constructor after stopping the current song being played and updating the gui values also
editable in-game. Lastly, loads in sfx objects if that option has been chosen and changes the state flags.
*/
void playButton(float theValue) {

    /* Going to game, so close title song. */
    titleSong.song.close();
    titleSong.minim.stop();

    /* Update the gui variables for the game to use such as NoStars and beatSensitivity based on what the
menu gui has them at. */
    newGUI.update2();
    /* Create a new game item */
    game = new MainGame(this);
    /* Create the game item related gui objects which are the player data HUD and in-game settings
accessed through TAB */
    newGUI.createGameGUI(this);

    /* If sfx on, create the sfx objects. */
    if(newGUI.sfx.value()==0) {
        explosion = game.minimObj.minim.loadSnippet("explosion.wav");
        laser = game.minimObj.minim.loadSnippet("laser.wav");
    }

    /* Change the flags accordingly */
    mainMenu = false;
    inGame = true;
}

/* fileButton - File Button method, called when the user clicks the fileButton and chooses a song to be
played in-game. Sets the filename variable to the name of the file chosen. For safety, filters out all but
mp3 and wav files as usable. Also uses ths swing component JFileChooser and the Runnable class to
avoid conflicts between updating the file chooser and main draw method. */
void fileButton() {
    noLoop();

    // thanks to ---> http://processing.org/discourse/yabb2/YaBB.pl?board=Integrate;action=display;num=1147684168
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            try
            {
                try
                {
                    UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
                }
            }
            catch (Exception e) {

```

```

        e.printStackTrace();
    }

    File file = new File("");
    JFileChooser chooser = new JFileChooser(file);

    try {
        int returnVal = chooser.showOpenDialog(null);
        if (returnVal == JFileChooser.APPROVE_OPTION) {
            file = chooser.getSelectedFile();

            String fileName = file.getName().toLowerCase();
            /* Filter file extensions to just use mp3 and wavs */
            if (fileName.endsWith("mp3") || fileName.endsWith("wav"))
            {
                /* set global filename for game music here based on filename accessed */
                filename = fileName;
            }
            else {
                println("Unsupported file selected by user.");
            }
        }

    } catch (Exception e) {
        e.printStackTrace();
    }

    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
    });

    loop();
}

```

/* mousePressed - Mouse Pressed Event method, used for left and right mouse click in-game. Left fires the player weapon and plays relevant sound if sfx on,

also decreases the weapon energy value and sets a boolean value for firing to be true. Right sets a boolean value for whether the playing

is pulling an object towards them (item drops) or not. */

```

void mousePressed() {
    if(game!= null) {
        if (mouseButton == LEFT) {
            if(newGUI.sfx2.value()==1 && (!newGUI.gameSettings.isVisible())) { laser =
game.minimObj.minim.loadSnippet("laser.wav"); laser.play(); }
            game.playerWeaponEnergy--;
            game.isFire = true;
        } else if(mouseButton == RIGHT) {
            game.onAttract = true;
        }
    }
}
}
}

```

```

/* mouseReleased - Mouse Released Event method used in conjunction with right clicking above, if
released, attraction is set to false */
void mouseReleased() {
    if(game!= null) {
        if(mouseButton == RIGHT) {
            game.onAttract = false;
        }
    }
}

/* keyPressed - Key Press Event - Used for moving back to the main menu, and for showing/hiding
settings options and main HUD */
void keyPressed() {

    /*If in-game and enter pressed, return to main menu and set relevant flags. If in gameOver state, also
return to main menu
        and sets flags accordingly. Playing titleSong as well.*/
    if(game!= null) {
        if(keyCode == ENTER && gameOver==true) {
            gameOver=false;
            mainMenu = true;
            newGUI.gameSettings.hide();
            newGUI.mainmenu.show();
            titleSong = new MinimAudio(this, true);
        }
        else if(keyCode == ENTER && inGame==true) {
            inGame=false;
            mainMenu=true;
            newGUI.gameSettings.hide();
            newGUI.mainmenu.show();
            game.minimObj.song.close();
            game.minimObj.minim.stop();
            titleSong = new MinimAudio(this, true);
        }
    }
}

/* If the key is TAB, show/hide relevant gui depending on state and what is currently shown/hidden.
Used for gui display. */
if(keyCode == TAB) {
    if(mainMenu && newGUI.mainmenu.isVisible()) {newGUI.mainmenu.hide();}
    else if(mainMenu && !newGUI.mainmenu.isVisible()){newGUI.mainmenu.show();}

    if(inGame) {
        if(newGUI.gameSettings.isVisible()) { newGUI.gameSettings.hide(); game.setupGame = false;
game.minimObj.song.play();}
        else if(!newGUI.gameSettings.isVisible()) { game.setupGame = true;
game.minimObj.song.pause(); newGUI.gameSettings.show(); }
    }
}

/* If the key is SHIFT, show/hide HUD. */
else if(keyCode == SHIFT) {
    if(inGame) {
        if(newGUI.mode.value()==0) { newGUI.mode.setValue(1); }
        else if(newGUI.mode.value()==1) { newGUI.mode.setValue(0); }
    }
}
}

```

```
/* ObjectSpace Game
13/08/09
```

```
author: Ashley Morrison
class: Asteroid
brief: Class that represents the beating cubes that are the "asteroids" of the game. Contains a
constructor, update, steer, HitPlayer, splitAsteroid, TextureCube, drawSide, render and drawRadarPoint
methods.
```

This class uses the steer method taken from Daniel Shiffman's example boid class :
http://www.shiffman.net/itp/classes/nature/week06_s09/seekarrive/Boid.pde

Asteroid objects are created and added to a list on a beat OnSet. The steering method is used to direct them towards the player. The objects are made up of a number of mini cube outer edges which move back and forth depending on the frequency spectrum value. More noise, cube sides move more etc.

The TextureCube, drawSide and the calling of drawSide in render() are all extended from another example : <http://www.openprocessing.org/visuals/?visualID=2545>

```
*/
class Asteroid {

  /* loc - Asteroid location vector. */
  PVector loc;

  /* vel - Asteroid velocity vector */
  PVector vel;

  /* dir - Asteroid direction vector, adjusted by the steer method and applied to vel. */
  PVector dir;

  /* collOffset - Asteroid offset value determined for the splitting of an asteroid into two, smaller
versions. The offset is applied to the steer method direction to send the asteroids in opposite directions
before coming back to the player */
  PVector collOffset;

  /* r - Default value for the radius of the asteroid */
  float r = 20;

  /* maxspeed - Float value used for the limiting speed of an asteroid calculated in the steer method. */
  float maxspeed;

  /* maxforce - Float value used in a similar manner to maxspeed but applied at the end of the steer
method */
  float maxforce;

  /* sizeQuad - Used to determine how large the cubes that make up the asteroid should be spaced apart
*/
  float sizeQuad;

  /* rotx - Rotates the asteroid around the x axis. */
  float rotx = 0;
  /* roty - Rotates the asteroid around the y axis. */
  float roty = 0;
```

/ prevTheta, theta, rotate, distance - All used for the drawing of the point on the radar corresponding to asteroid-player location discrepancy. prevTheta and theta are the difference in dot products between the direction of the player and vector from player-asteroid from the last update/current update. rotateBy, the amount to rotate around the centre point. distance, the distance to offset the point from the centre corresponding to actual distance calculated.*/*

```
float prevTheta,theta,rotateBy,distance = 0;
```

/ radar1, radar2, radar3, radar4, goRight - Used to determine which of 4 states the radar is in, between increasing/decreasing past 180 and 0 respectively. The rotation needs to be different depending on which direction the player is heading in etc. goRight signifies whether the player is rotating roughly right or left and is also needed to update the radar correctly.*/*

```
boolean radar1,radar2,radar3,radar4, goRight=false;
```

/ lives - The number of lives the asteroid has left.*/*

```
int lives;
```

/ start - A start value for use in the render method, calls to drawSide()*/*

```
int start = -100;
```

/ timer - Used for child asteroids created in hard mode to apply the collOffset value over a period of x.*/*

```
float timer = 6;
```

/ timerDec - Used to decrement the timer value above.*/*

```
float timerDec = 0.6;
```

/ spawnTimer - Used to determine how long the asteroid has been in existence and how long to leave it invulnerable to damage,*

only used for child asteroids./*

```
float spawnTimer;
```

/ god - Boolean flag used with asteroids spawned due to a parent asteroid being destroyed, for a period, the child asteroid cannot be damaged.*/*

```
boolean god = true;
```

/ parent - Boolean flag for whether this asteroid is a parent, if true & if difficulty=hard, spawnAsteroid() will be called.*/*

```
boolean parent = true;
```

/ removing - Boolean flag to check whether item has been hit and is being removed through creating an explosion/fading out.*/*

```
boolean removing = false;
```

/ isHit - Boolean flag to determine whether this asteroid has simply been hit and not yet reduced to 0 health*/*

```
boolean isHit = false;
```

/ colliding - Is this asteroid colliding with a missile object.*/*

```
boolean colliding = false;
```

/ isDead - If colliding with player, set flag, it will be auto removed in the game loop*/*

```
boolean isDead = false;
```

/ alphaValue - Set to full for the life span of the asteroid except one removing=true, then this value is decremented to fade it out.*/*

```
float alphaValue = 0.85;
```

/ testExp - This asteroids explosion item to be called upon removing=true*/*

```

Explosion testExp;

/* asteroidHealth - The int value of the health, decremented each time it is hit by a missile based on
the number of points of the missile pattern. */
int asteroidHealth = 100;

/* Asteroid Ctr - Sets all major variables up, parent, sizeQuad, Start will all be based upon whether
asteroid is a child or not etc. */
Asteroid(PVector l, PVector v, float ms, float mf, float sizequad, int Lives, int Start, int SpawnTimer,
boolean Parent) {
    loc = l;
    vel = new PVector(0,0,0);
    dir = new PVector(0,0,0);
    collOffset = new PVector(0,0,0);
    maxspeed = ms;
    maxforce = mf;
    sizeQuad = sizequad;
    lives = Lives;
    start = Start;
    spawnTimer = SpawnTimer;
    testExp = new Explosion(Parent);
    parent=Parent;
}

/* update - Used to add the offset & invincibility if relavant, to determine the direction to head in
based on a target. */
void update(PVector target) {
    timer-=timerDec;
    timerDec+=0.1;
    spawnTimer--;

    /* if child, apply the offset as Vector tmp while the timer is greater/equal to 0. */
    if(!parent) {
        PVector tmp = new PVector(0,0,0);
        if(timer >=0) {
            tmp = collOffset;
            tmp.mult(timer);
        }
        vel.add(tmp);
    }

    /* Calculate direction vector by supplying target position, true indicates whether the calculation
should take into account slowing down as approaching. */
    dir = steer(target, true);
    vel.add(dir);
    loc.add(vel);

    /* if existed for x period, remove god mode. */
    if(spawnTimer<=0) {
        god = false;
    }

    render();
    isHit = false;
}

```

/* steer - Creates a direction vector based on current position and target position.
 Method directly taken from Daniel Shiffman's example boid class : http://www.shiffman.net/itp/classes/nature/week06_s09/seekarrive/Boid.pde
 with the addition of a boolean flag when the distance between asteroid-player is not greater than 0. */

```
PVector steer(PVector target, boolean slowdown) {
  PVector steer; // The steering vector
  PVector desired = PVector.sub(target,loc); // A vector pointing from the location to the target
  float d = desired.mag(); // Distance from the target is the magnitude of the vector
  // If the distance is greater than 0, calc steering (otherwise return zero vector)
  if (d > 0) {
    // Normalize desired
    desired.normalize();
    // Two options for desired vector magnitude (1 -- based on distance, 2 -- maxspeed)
    if ((slowdown) && (d < 100.0f)) desired.mult(maxspeed*(d/100.0f)); // This damping is somewhat
    arbitrary
    else desired.mult(maxspeed);
    // Steering = Desired minus Velocity
    steer = PVector.sub(desired,vel);
    steer.limit(maxforce); // Limit to maximum steering force
  } else {
    steer = new PVector(0,0,0);
    isDead = true;
  }
  return steer;
}
```

/* HitPlayer - Takes the position of the player p and determines whether the asteroid has collided with the player based on the sum of the radius.

This method has been simplified from Daniel Shiffman's collision example:
http://www.shiffman.net/itp/classes/nature/collisions_s09/ballvsball_equalmass/Thing.pde */

```
boolean HitPlayer(PVector p, boolean single) {
  /* distance between player and asteroid location. */
  float d = PVector.dist(loc,p);

  /* Sum of the radius. */
  float sumR = r + 1.0;

  /* If not already colliding and the distance is less than the sum, asteroid boolean flag set to be
  checked/removed next iteration.*/
  if (!colliding && d < sumR) {
    isDead = true;
    return true;
  }
  /* else if greater, return false. */
  else if (d > sumR) {
    return false;
  }
  return false;
}
```

/* splitAsteroid - Split Asteroid method called if the asteroid is a parent and if this difficulty is set to hard. This method removes this object after creating and adding two new asteroid objects to the list to be iterated through in MainGame. The two new asteroids are set with data based on the parent except

with some values like lives and size decremented. The offset is also calculated to ensure the two asteroids move off in different directions to differentiate them visually for the player. */

```

void splitAsteroid() {
  lives--;
  if(lives <= 0) {
    //spawnNew = false;
  }
  else {
    int cLives = lives;
    int cStart = start+100;

    /* 1st asteroid */
    Asteroid newAsteroid = new Asteroid(new PVector(loc.x, loc.y, loc.z), new PVector(random(-10,10),random(-10,10),random(-10,10)), maxspeed*3, 0.2, 45, cLives, cStart, 10, false);

    /* offset determined for asteroid 1 */
    newAsteroid.collOffset = new PVector(random(-0.005,0.005),random(-0.005,0.005),0);

    /* direction for asteroid 2 set based on (1) */
    PVector tmpDir = new PVector(-newAsteroid.collOffset.x*2, -newAsteroid.collOffset.y*2, -newAsteroid.collOffset.z*2);
    PVector tmpPos = PVector.add(loc,tmpDir);

    /* 2nd asteroid */
    Asteroid newAsteroid2 = new Asteroid(new PVector(tmpPos.x, tmpPos.y, tmpPos.z), new PVector(-newAsteroid.loc.x,-newAsteroid.loc.y,-newAsteroid.loc.z), maxspeed*3, 0.2, 45, cLives, cStart, 10, false);
    newAsteroid2.collOffset = new PVector(tmpDir.x, tmpDir.y, tmpDir.z);

    /* add to list */
    game.asteroidList.add(newAsteroid);
    game.asteroidList.add(newAsteroid2);

    //spawnNew = false;
  }
}

```

/* TextureCube - This method is used to draw the asteroids to screen based on a size value passed in depending on parent/child.

It is based upon the MusicBox sketch:

<http://www.openprocessing.org/visuals/?visualID=2545>

The difference being, only certain sides of the cubes are drawn depending on which side of the overall cube they are a part of. This is determined by the x value passed in. If 1, indicating it is the front face, draw following vertices.

```

    Also, no textures are used with this version. */
void TexturedCube(int x, float sizeQ) {
  beginShape(QUADS);

  switch(x) {
    case 1: // +Z "front" face
      vertex(-sizeQ, -sizeQ, sizeQ);
      vertex( sizeQ, -sizeQ, sizeQ);
      vertex( sizeQ, sizeQ, sizeQ);
      vertex(-sizeQ, sizeQ, sizeQ); break;

```

```

case 2: // -Z "back" face
vertex( sizeQ, -sizeQ, -sizeQ);
vertex(-sizeQ, -sizeQ, -sizeQ);
vertex(-sizeQ, sizeQ, -sizeQ);
vertex( sizeQ, sizeQ, -sizeQ); break;

case 3: // +Y "bottom" face
vertex(-sizeQ, sizeQ, sizeQ);
vertex( sizeQ, sizeQ, sizeQ);
vertex( sizeQ, sizeQ, -sizeQ);
vertex(-sizeQ, sizeQ, -sizeQ); break;

case 4: // -Y "top" face
vertex(-sizeQ, -sizeQ, -sizeQ);
vertex( sizeQ, -sizeQ, -sizeQ);
vertex( sizeQ, -sizeQ, sizeQ);
vertex(-sizeQ, -sizeQ, sizeQ); break;

case 5: // +X "right" face
vertex( sizeQ, -sizeQ, sizeQ);
vertex( sizeQ, -sizeQ, -sizeQ);
vertex( sizeQ, sizeQ, -sizeQ);
vertex( sizeQ, sizeQ, sizeQ); break;

case 6: // -X "left" face
vertex(-sizeQ, -sizeQ, -sizeQ);
vertex(-sizeQ, -sizeQ, sizeQ);
vertex(-sizeQ, sizeQ, sizeQ);
vertex(-sizeQ, sizeQ, -sizeQ); break;
}
endShape();
}

```

/* drawSide - This method calls the above TextureCube in a translated position based on asteroid.loc

Based upon:

<http://www.openprocessing.org/visuals/?visualID=2545> */

```

void drawSide(int argx,int argy,int argz,int x)
{
pushMatrix();
translate(argx,argy,argz);
TexturedCube(x, sizeQuad);
popMatrix();
}

```

/* render - The render method starts the sequence of by calling drawSide in succession 6 times for the 6 faces of a cube. Before this the drawing is translated to the asteroid.loc vector, checks are made to adjust the drawing if being removed or if child (colour change). The rendering is rotated by incremented amounts in the x and y axis. Finally the start variable is used to how to access the sub bands of the frequency spectrum for each face of the cube so the sides move to the sound. This also depends on the size of the cube as a whole. */

```

void render() {
pushMatrix();

translate(loc.x, loc.y, loc.z);

```

```

    /* If removing, adjust the alpha values, update the explosion item and once alpha reaches 0, adjust
    scores and set flag. */
    if(removing) {
        testExp.update(alphaValue);
        alphaValue-=0.03;
        if(alphaValue <=0) {
            if(parent) { game.playerScore+=10; } else{ game.playerScore+=100; }
            isDead = true;
        }
    }

    /* Adjust colour depending on parent/child. */
    colorMode(HSB, 1.0);
    if(parent) {
        fill(0.1, pow(1,0.1), 0.9, alphaValue);
    }
    else {
        fill(0.5, pow(1,0.1), 0.9, alphaValue);
    }

    /* if isHit, fill white and full alpha. */
    if(isHit) {
        fill(1.0, 1.0);
    }
    colorMode(RGB, 255);

    /* scale and rotate. */
    scale(0.1);
    rotateX(rotx);
    rotateY(roty);

    /* Calculate increment values based upon start size and fft size, to be used below. */
    int FFTsize = game.minimObj.thisSample.fft.avgSize();
    int end = 100, inc = 100;
    int diff = (Math.abs(start-end)) + inc;
    diff /= 100;
    r = (diff * sizeQuad)/2;
    diff *=diff;
    int incrementSize = FFTsize/diff;

    /* The following is adjusted from : http://www.openprocessing.org/visuals/?visualID=2545 */
    int i=0;
    noStroke();
    for(int x=start;x<=end;x+=inc)
    for(int y=start;y<=end;y+=inc)
    {
        drawSide(x,y,170+(int)game.minimObj.thisSample.fft.getAvg(i), 1);
        drawSide(x,y,-170-(int)game.minimObj.thisSample.fft.getAvg(i), 2);
        drawSide(x,160+(int)game.minimObj.thisSample.fft.getAvg(i),y, 3);
        drawSide(x,-160-(int)game.minimObj.thisSample.fft.getAvg(i),y, 4);
        drawSide(170+(int)game.minimObj.thisSample.fft.getAvg(i),x,y, 5);
        drawSide(-170-(int)game.minimObj.thisSample.fft.getAvg(i),x,y, 6);
        i+=incrementSize;
    }

```

```

popMatrix();
rotx+=PI/600;
roty+=PI/600;
/* The following is adjusted from : http://www.openprocessing.org/visuals/?visualID=2545 */

```

```

}

```

/* drawRadarPoint - Method called each iteration in MainGame when in-game and not adjusting settings. Used to plot a point in relation to player to illustrate where the asteroid is in comparison. Due to the dot product between the player direction and player2asteroid vector returning between 0-180 (in front/behind), to adjust a point around 360 degrees in the correct manner, need to know the direction the player is heading in etc.

The radar variables represent this. */

```

void drawRadarPoint() {
  //If moving right
  if(goRight) {
    // if increasing & at 180
    if(prevTheta > theta && rotateBy >= radians(180)) {
      radar1 = true; radar2=false;radar3=false;radar4=false;
      rotateBy = radians(180);
      rotateBy = rotateBy +(rotateBy-theta);
      rotateZ(-rotateBy);
    }
    // if decreasing & at 180
    else if(prevTheta < theta && rotateBy >= radians(180)){
      radar2 = true; radar1=false;radar3=false;radar4=false;
      rotateBy = radians(180);
      rotateBy = rotateBy +(rotateBy-theta);
      rotateZ(rotateBy);
    }
    // if decreasing & at 0
    else if(prevTheta < theta && rotateBy <= radians(0)) {
      radar3 = true; radar2=false;radar1=false;radar4=false;
      rotateBy = radians(360);
      rotateBy = rotateBy -theta;
      rotateZ(rotateBy);
    }
    // if increasing and at 0
    else if(prevTheta > theta && rotateBy <= radians(0)) {
      radar4 = true; radar2=false;radar3=false;radar1=false;
      rotateBy = radians(360);
      rotateBy = rotateBy -theta;
      rotateZ(-rotateBy);
    }
    // else central, no change
    else {
      if(radar1) {
        rotateZ(-rotateBy);
      }
      else if(radar2) {
        rotateZ(rotateBy);
      }
      else if(radar3) {
        rotateZ(-rotateBy);
      }
    }
  }
}

```

```

    }
    else if(radar4) {
        rotateZ(rotateBy);
    }
}
// If moving left
else {
    // if increasing & at 180
    if(prevTheta > theta && rotateBy >= radians(180)) {
        radar1 = true; radar2=false;radar3=false;radar4=false;
        rotateBy = radians(180);
        rotateBy = rotateBy +(rotateBy-theta);
        rotateZ(rotateBy);
    }
    // if decreasing & at 180
    else if(prevTheta < theta && rotateBy >= radians(180)){
        radar2 = true; radar1=false;radar3=false;radar4=false;
        rotateBy = radians(180);
        rotateBy = rotateBy +(rotateBy-theta);
        rotateZ(-rotateBy);
    }
    // if decreasing & at 0
    else if(prevTheta < theta && rotateBy <= radians(0)) {
        radar3 = true; radar2=false;radar1=false;radar4=false;
        rotateBy = radians(360);
        rotateBy = rotateBy -theta;
        rotateZ(-rotateBy);
    }
    // if increasing & at 0
    else if(prevTheta > theta && rotateBy <= radians(0)) {
        radar4 = true; radar2=false;radar3=false;radar1=false;
        rotateBy = radians(360);
        rotateBy = rotateBy -theta;
        rotateZ(rotateBy);
    }
    else {
        // else central, no change
        if(radar1) {
            rotateZ(rotateBy);
        }
        else if(radar2) {
            rotateZ(-rotateBy);
        }
        else if(radar3) {
            rotateZ(rotateBy);
        }
        else if(radar4) {
            rotateZ(-rotateBy);
        }
    }
}

stroke(0,255,0);
ellipse(0, -(distance/30), 2,2);
prevTheta = theta;
}
}

```

```

/* ObjectSpace Game
13/08/09

author: Ashley Morrison
class: Background
brief: This class represents the state of the main menu visualisation. It contains lists of SpiralGalaxy
objects and points representing stars. The constructor adds a set amount to each list with a random x/y
position on the screen.*/

class Background {

    /* spirals - Array list of SpiralGalaxy objects the background uses. */
    ArrayList spirals;

    /* stars - Array list of points in space to draw stars at for background on menu screen. */
    ArrayList stars;

    /* Background ctr - fills the two array lists with SpiralGalaxy & PVector values respectively. */
    Background() {
        stars = new ArrayList();
        for(int h = 0; h < 200; h++) {
            stars.add(new PVector(random(width), random(height), random(0,255)));
        }

        spirals = new ArrayList();
        for(int i = 0; i < 10; i++) {
            SpiralGalaxy3 newSpiral = new SpiralGalaxy3((int)random(300));
            spirals.add(newSpiral);
        }
    }

    /* update - Used to update the spiral galaxy objects based on whether the menu music beat is Onset. If
it is, increment the colour, set the beat to true to be accessed in the SpiralGalaxy object render method.
Adjust the twist value that determines the pattern of the spiral galaxy and finally if the number of points
making up the galaxy is less than the max(20k), increment this amount by a random value. */
    void update() {
        float randPick = random(0,spirals.size());

        // if beat
        if(titleSong.beat3.isOnset()) {
            SpiralGalaxy3 thisSpiral = (SpiralGalaxy3)spirals.get((int)randPick);

            /* beat being true will adjust the colour in SpiralGalaxy.render() */
            thisSpiral.beat=true;
            thisSpiral.colourChange=true;
            thisSpiral.etwist+=random(0.001,0.005);

            /* if NoStars less than 20k, go through galaxy angle/radius lists and add extra values. These
values will be picked up
on in the galaxy update and extra points will be drawn. */
            if(thisSpiral.stars <20000) {
                float NoAdd = random(10,200);
                for (int j=0; j< NoAdd; j++){
                    thisSpiral.angle.add(random(0,2*PI));
                    thisSpiral.radius.add(random(1,thisSpiral.Rmax));
                }
            }
        }
    }
}

```

```

        }
        thisSpiral.stars+=NoAdd;
    }
}
else {
    SpiralGalaxy3 thisSpiral = (SpiralGalaxy3)spirals.get((int)randPick);
    thisSpiral.beat=false;
}
}

/* render - Draws each SpiralGalaxy and all star points to screen based on the current state of these
values. */
void render() {
    // stars
    for(int h = 0; h < stars.size(); h++) {
        PVector tmp = (PVector)stars.get(h);
        stroke(255,tmp.z);
        point(tmp.x, tmp.y);
    }

    // SpiralGalaxies
    for(int i = 0; i < spirals.size(); i++) {
        SpiralGalaxy3 thisSpiral = (SpiralGalaxy3)spirals.get(i);
        if(titleSong.beat3.isOnset()) { thisSpiral.drawGalaxy(); }else { thisSpiral.drawGalaxy(); }
    }
}
}
}

```

```

/* ObjectSpace Game
13/08/09

```

Class Taken exactly from minim audio library examples:
<http://code.compartmental.net/minim/examples/BeatDetect/FrequencyEnergy/BeatListener.pde>

```

Instantiated in the MinimAudio class for listening to different songs/types of beat.
*/

```

```

class BeatListener implements AudioListener
{
    private BeatDetect beat;
    private AudioPlayer source;

    BeatListener(BeatDetect beat, AudioPlayer source)
    {
        this.source = source;
        this.source.addListener(this);
        this.beat = beat;
    }

    void samples(float[] samps)
    {
        beat.detect(source.mix());
    }

    void samples(float[] sampsL, float[] sampsR)

```

```

    {
        beat.detect(source.mix);
    }
}

```

```

/* ObjectSpace Game
13/08/09

```

```

    author: Ashley Morrison
    class: Explosion
    brief: This class is used by each asteroid object upon being destroyed. It is based closely upon a
point-spherical distribution sketch on OpenProcessing by 'Starkes':

```

```

http://www.openprocessing.org/visuals/?visualID=861

```

```

The explosion class constructor creates an array of Particle objects based on theta/u values which are
passed in to the Particle object ctr to calculate a position around a sphere with them. The update method
cycles through all particles and calls update and render methods on them.
*/

```

```

class Explosion

```

```

{
    /* numParticles - Number of particles to create around a sphere of radius x. */
    int numParticles = 500;

```

```

    /* radius, expansionInc - radius of sphere to distribute points and the increment for how quickly the
set of points should expand in an explosion-like manner. */
    float radius = 2, expansionInc=4;

```

```

    /* parent - Boolean value passed in from the asteroid that instantiated it. If true, explosion in 1 colour,
else another. */
    boolean parent=false;

```

```

    /* Particles - set of Particle objects to update and render. */
    Explosionparticle[] Particles = new Explosionparticle[numParticles];

```

```

    /* Explosion ctr with parent boolean. Cycle through numParticles and create a particle with location
around sphere */

```

```

    Explosion(boolean Parent)
    {
        parent=Parent;
        for(int i = 0; i < numParticles; i++)
        {
            float theta = random(0,TWO_PI);
            float u = random(-1,1);
            Particles[i] = new Explosionparticle(theta,u,radius,expansionInc);
        }
    }
}

```

```

    /* update - alpha value passed in from asteroid and further passed on to ExplosionParticle objects to
use in render() */

```

```

    void update(float alphaVal)
    {
        for(int i = 0; i < numParticles; i++)
        {

```



```

    Particles[i].update(alphaVal);
    Particles[i].render(parent);
  }
}
}

```

/* author: Ashley Morrison

class: ExplosionParticle

brief: This class is instantiated and contained within an arraylist of other ExplosionParticle objects to represent the explosion of an asteroid.

It is also based on the sketch at OpenProcessing:

<http://www.openprocessing.org/visuals/?visualID=861>

The explosionparticle class constructor creates sets up the relevant values passed in from Explosion. The update method updates alphaValue and calculates the latest x,y,z position of this particle. The expansion effect is done simply by incrementing the radius of the sphere about which the points are distributed. The expansion increment added to the radius value is itself also decremented meaning the radius expansion rate decreases over time. */

```

class Explosionparticle

```

```

{

```

```

  /* theta, u - Values used to determine the x/y/z points in update() */

```

```

  float theta, u;

```

```

  /* x,y,z - X/Y/Z values for the position of the particle during the duration of the expansion. */

```

```

  float x,y,z;

```

```

  /* tmpx,tmpy,tmpz - Previous x/y/z values for using to draw a line between old and current positions.
  */

```

```

  */

```

```

  float tmpx, tmpy, tmpz;

```

```

  /* rad, expansionInc - radius and the expansion increment variable used to expand the set of particles
  around a growing radius size. */

```

```

  float rad,expansionInc;

```

```

  /* alphaValue - Alpha value passed in from Asteroid-Explosion to fade out after x period. */

```

```

  float alphaValue = 1.0;

```

```

  /* Explosionparticle ctr - theta, u, radius and increment values passed in initially. */

```

```

  Explosionparticle(float Theta, float U, float radius, float increment)

```

```

  {

```

```

    theta = Theta;

```

```

    u = U;

```

```

    rad = radius;

```

```

    expansionInc = increment;

```

```

  }

```

```

  /* update - Changes the alpha value and recalculates the x,y and z values. */

```

```

  void update(float alphaVal)

```

```

  {

```

```

    alphaValue=alphaVal;

```

```

    expansionInc -= 0.1;

```

```

    rad+=(expansionInc);

```

```

    tmpx = x;

```

```

    tmpy = y;
    tmpz = z;

    x = rad*cos(theta)*sqrt(1-(u*u));
    y = rad*sin(theta)*sqrt(1-(u*u));
    z = u*rad;

}

/* render - Checks whether the particle is a part of a parent asteroid or not, changes colour values
accordingly. */
void render(boolean parent)
{
    pushMatrix();
    colorMode(HSB, 1.0);
    if(parent) { stroke(0.1, pow(1,0.1), 0.9,alphaValue); }else { stroke(0.5, pow(1,0.1),
0.9,alphaValue); }
    line(tmpx,tmpy,tmpz,x,y,z);
    colorMode(RGB, 255);
    popMatrix();
}
}

/* ObjectSpace Game
13/08/09

author: Ashley Morrison
class: FFTSample
brief: Instantiated in the MinimAudio class. This class creates an FFT object from the minim audio
library to extend the analysis that offers. It updates a current maximum and overall average on the FFT
over the song to be used for the Rose/Missile classes for damage dealt/pattern generated.
*/

class FFTSample {

    /* song - AudioPlayer object from the minim library, used to play a song. */
    AudioPlayer song;

    /* fft - FFT object also taken from the minim audio library for audio analysis. */
    FFT fft;

    /* currentMaximum - the current maximum peak value of the frequency spectrum. */
    float currentMaximum;

    /* average - the average frequency spec value calculated so far over the current duration of the song.
*/
    float average;

    /* counter - the value used to calculate the average in the findMax method below. */
    int counter;

    /* FFTSample - creates an fft object from minim based on the logAverages division of sub bands
noted in the background of the thesis. */
    FFTSample(AudioPlayer song2) {

```

```

    song = song2;
    if(song != null)
    {
        /* buffer size and sample rate taken from the song object which is passed in from the MinimAudio
object during instantiation. */
        fft = new FFT(song.bufferSize(), song.sampleRate());
        fft.logAverages(22, 3);
    }
}

/* update - forwards the fft sample on for analysis of the next frequency spec. */
void update() {
    fft.forward(song.mix);
}

/* findMax - Method used to calculate the current maximum peak and the overall average. */
void findMax() {
    float currentMax = 0;
    float tempTotal = 0;
    // cycle through all bands and find the highest value, also add each one to a tmp variable.
    for(int i = 0; i < fft.avgSize(); i++)
    {
        float temp = fft.getAvg(i);
        tempTotal += temp;
        if(temp > currentMax)
        {
            currentMax = temp;
        }
    }
    // find the local average by dividing the total by the size
    float localAverage = tempTotal/fft.avgSize();
    average = average * counter;

    counter++;

    // find the overall average by adding the local average to existing global average and then dividing
by the number of times this has been calculated.
    average = (average+localAverage) / counter;
    // set current maximum
    currentMaximum = (int)currentMax;
}
}

```

```

/* ObjectSpace Game
13/08/09

```

```

author: Ashley Morrison
class: GUI

```

```

brief: Uses and contains all the gui related objects as well as update methods. Constructor creates
just the menu items first then when the player hits the play button, the rest of the items are added due to

```

being based on in-game values. Two different update methods for the in-game related data and the menu screen data.

```
*/  
  
class GUI {  
  
    /* file - File object for choosing a music file to play and retrieving the filename. */  
    File file;  
  
    /* fc - file chooser object used with file to create a directory browser. */  
    JFileChooser fc;  
  
    /* weaponEnergyBar,scoreBar,healthBar - Three main bar values for the player data. Used in-game to  
    illustrate the game state from player's perspective. */  
    Slider weaponEnergyBar,scoreBar,healthBar; // player stats  
  
    /* gameData - The panel for the player data in-game. */  
    ControlP5 gameData;  
  
    /* gameSettings - The panel for the adjustable settings in-game. */  
    ControlP5 gameSettings;  
  
    /* mainmenu - The panel for the menu screen settings. */  
    ControlP5 mainmenu;  
  
    // game/visualisation settings  
    /* beatSensitivityBar, beatSensitivityBar2, noGalaxiesBar, noStarsBar, spaceRadiusBar - sliders to  
    adjust the beat sensitivity, number of galaxies, stars and the radius of the sphere about which they're  
    distributed. */  
    Slider beatSensitivityBar, beatSensitivityBar2, noGalaxiesBar, noStarsBar, spaceRadiusBar;  
  
    /* wormholeNoStarsBar, wormholeRadius - slider for the number of stars in the wormhole and the  
    radius of it.*/  
    Slider wormholeNoStarsBar, wormholeRadius;  
  
    /* galaxyNoParticlesBar, galaxyRadiusBar, asteroidSpeedBar, itemdropSpeedBar - sliders for the  
    number of particles per Pulsar object, the radius of them, the maxspeed  
    value for asteroids and the same for item drops.*/  
    Slider galaxyNoParticlesBar, galaxyRadiusBar, asteroidSpeedBar, itemdropSpeedBar;  
  
    /* difficulty,difficulty2,mode,hud,sfx,sfx2 - Radio buttons for the difficulty, game mode, hud being  
    on/off and sound effects being on/off. */  
    Radio difficulty,difficulty2,mode,hud,sfx,sfx2;  
  
    /* vol1, vol2 - Sliders to adjust the volume of the music for the menu and in-game songs. */  
    Slider vol1, vol2;  
  
    /* filename - a string variable to be used for holding the filename that is chosen by the user to play in-  
    game, passed back to Main, then MainGame and finally game.minimObj.*/  
    String filename = "";  
  
    /* GUI Ctr - instanties all the menu screen gui related items and adds them to mainmenu ControlP5  
    object. */  
    GUI(PApplet main) {
```

```

/* MAIN MENU SETTINGS */
mainmenu = new ControlP5(main);

noGalaxiesBar = mainmenu.addSlider("noGalaxiesBar",0,25,NoGalaxies,width/2-75,height-(height/4)-315,100,10);
noGalaxiesBar.setLabel("No. Galaxies");
noStarsBar = mainmenu.addSlider("NoStarsBar",0,5000,NoStars,width/2-75,height-(height/4)-300,100,10);
noStarsBar.setLabel("No. Stars");
spaceRadiusBar = mainmenu.addSlider("spaceRadiusBar",0,10000,R,width/2-75,height-(height/4)-285,100,10);
spaceRadiusBar.setLabel("Space Radius");

wormholeNoStarsBar =
mainmenu.addSlider("wormholeNoStarsBar",0,20000,wormholeNoStars,width/2-75,height-(height/4)-250,100,10);
wormholeNoStarsBar.setLabel("Wormhole No.Stars");
wormholeRadius = mainmenu.addSlider("wormholeRadius",0,5000,wormholeRmax,width/2-75,height-(height/4)-235,100,10);
wormholeRadius.setLabel("Wormhole Radius");
galaxyNoParticlesBar =
mainmenu.addSlider("galaxyNoParticlesBar",0,2000,galaxyParticleNo,width/2-75,height-(height/4)-200,100,10);
galaxyNoParticlesBar.setLabel("Galaxy Particle Count");
galaxyRadiusBar = mainmenu.addSlider("galaxyRadiusBar",0,5000,galaxyRadius,width/2-75,height-(height/4)-185,100,10);
galaxyRadiusBar.setLabel("Galaxy Radius");

difficulty = mainmenu.addRadio("radio",width/2-75,height-(height/4)-150);
difficulty.add("Normal",0);
difficulty.add("Hard",1);

mode = mainmenu.addRadio("modeRadio",width/2,height-(height/4)-150);
mode.add("Game",0);
mode.add("Vis",1);

hud = mainmenu.addRadio("hudRadio",width/2,height-(height/4)-105);
hud.add("HUD",0);
hud.add("No HUD",1);

sfx = mainmenu.addRadio("sfxRadio",width/2-75,height-(height/4)-105);
sfx.add("SFX Off",0);
sfx.add("SFX On",1);

beatSensitivityBar = mainmenu.addSlider("beatSensitivityBar",0,50000,beatSensitivity,width/2-75,height-(height/4)-60,100,10);
beatSensitivityBar.setLabel("Beat Sensitivity");
mainmenu.addButton("fileButton",10,width/2-75,height-(height/4)-25,150,20).setLabel(" Choose File (.wav or .mp3) ");

voll = mainmenu.addSlider("voll",-100,5,-16,width/2-75,height-(height/4)+20,100,10);
voll.setLabel("Volume");

mainmenu.addButton("playButton",10,width/2-50,height-(height/4)+60,80,20).setLabel(" Play");
/* MAIN MENU SETTINGS */

}

```

```

/* createGameGUI - method called after the play button above has been pressed. This instantiates the
in-game and player stats objects
and adds them to the respective controlP5 object. */
void createGameGUI(PApplet main) {

/* PLAYER SETTINGS */
gameData = new ControlP5(main);
gameData.hide();

scoreBar = gameData.addSlider("playerScore",0,20000,game.playerScore,width/2+75,height-
(height/6)+25,100,10);
scoreBar.setLabel(" Score");
scoreBar.setBroadcast(false);

weaponEnergyBar =
gameData.addSlider("playerWeaponEnergy",0,2000,game.playerWeaponEnergy,width/2+75,height-
(height/6)-5,100,10);
weaponEnergyBar.setLabel(" Energy");
weaponEnergyBar.setBroadcast(false);

healthBar = gameData.addSlider("playerHealth",0,400,game.playerHealth,width/2+75,height-
(height/6)-35,100,10);
healthBar.setLabel(" Shield");
healthBar.setBroadcast(false);
/* PLAYER SETTINGS */

/* GAME/VIS SETTINGS */
gameSettings = new ControlP5(main);
gameSettings.hide();

difficulty2 = gameSettings.addRadio("radio",width/2-75,height-(height/6)-315);
difficulty2.add("Normal",0);
difficulty2.add("Hard",1);
difficulty2.setValue(difficulty.value());

sfx2 = gameSettings.addRadio("sfxRadio2",width/2,height-(height/6)-315);
sfx2.add("SFX Off",0);
sfx2.add("SFX On",1);
sfx2.setValue(sfx.value());

beatSensitivityBar2 =
gameSettings.addSlider("beatSensitivityBar2",0,50000,beatSensitivity,width/2-75,height-(height/6)-
265,100,10);
beatSensitivityBar2.setLabel("Beat Sensitivity");
beatSensitivityBar2.setValue(beatSensitivityBar.value());

itemdropSpeedBar = gameSettings.addSlider("itemdropSpeedBar",1,20,game.maxspeedItems,width/
2-75,height-(height/6)-230,100,10);
itemdropSpeedBar.setLabel("Item Drop Max Speed");
asteroidSpeedBar =
gameSettings.addSlider("asteroidSpeedBar",1,20,game.maxspeedAsteroids,width/2-75,height-
(height/6)-215,100,10);
asteroidSpeedBar.setLabel("Asteroid Max Speed");

vol2 = gameSettings.addSlider("vol2",-100,5,-2,width/2-75,height-(height/6)-180,100,10);

```

```

    vol2.setLabel("Volume");
    /* GAME/VIS SETTINGS */

}

/* update - Updates the in-game player stats as well as the in-game settings. */
void update() {
    weaponEnergyBar.setValue(game.playerWeaponEnergy);
    scoreBar.setValue(game.playerScore);
    healthBar.setValue(game.playerHealth);

    game.gameDifficulty = (int)newGUI.difficulty2.value();
    beatSensitivity = (int)beatSensitivityBar2.value();
    game.minimObj.beat.setSensitivity(beatSensitivity);

    game.maxspeedAsteroids = asteroidSpeedBar.value();
    game.maxspeedItems = itemdropSpeedBar.value();

    game.minimObj.song.setGain(vol2.value());
}

/* update2 - Updates the main menu settings. */
void update2() {
    NoStars = (int)noStarsBar.value();
    NoGalaxies = (int)noGalaxiesBar.value();

    wormholeNoStars = (int)wormholeNoStarsBar.value();
    wormholeRmax = (int)wormholeRadius.value();

    galaxyParticleNo = (int)galaxyNoParticlesBar.value();
    galaxyRadius = (int)galaxyRadiusBar.value();

    beatSensitivity = (int)beatSensitivityBar.value();

    if(titleSong!=null) { titleSong.song.setGain(vol1.value()); }
}
}

```

```

/* ObjectSpace Game
13/08/09

```

```

author: Ashley Morrison
class: ItemDrop

```

brief: Class that represents an item drop/player stat booster in the game. Item drops are spawned from the centre of Pulsers and can be one of three types corresponding to the three types of pulsers. These three types are displayed through the colours red, blue and yellow which if retrieved by the player boost the score, weapon energy level and shield respectively.

Once spawned they use the steer method taken from Daniel Shiffman's example boid class:
http://www.shiffman.net/itp/classes/nature/week06_s09/seekarrive/Boid.pde

Using this they travel towards the centre of the wormhole which is at an arbitrary point around the player. The player has the option of attracting the item towards him/herself by way of the right mouse

button and diverting it in their direction. If the itemdrop effectively collides with the wormhole it is removed, if it collides with the player then the relevant boost will be awarded after which it's removed.

The spawning of the item drops is designated by a particular type of Onset being detected (either a kick, hat or snare) along with the type of pulser matching this onset detection being compared. Along with the ctr and steer methods are update, render and drawRadarPoint methods.

*/

```
class ItemDrop
{
    /* pos - A vector representing the position of this item. */
    PVector pos;

    /* dir - A vector representing the direction of this item. */
    PVector dir;

    /* target - A vector representing the target, whether the wormhole or player. */
    PVector target;

    /* attraction - A boolean for whether the player is currently diverting the item to themselves. */
    boolean attraction = false;

    /* isDead - A boolean for whether the item has collided either with wormhole or player and can be
    removed. */
    boolean isDead = false;

    /* maxspeed, maxforce, origMaxspeed - Floating point values for the maximum speed and force this
    item can accelerate to. The maxspeed can be altered when the player is attracting it towards them,
    hence the use of an original value for when this is no longer the case. */
    float maxspeed,maxforce,origMaxspeed;

    /* prevTheta,theta,temp,distance - All used for the drawing of the point on the radar corresponding to
    asteroid-player location discrepancy. prevTheta and theta are the difference in dot products between the
    direction of the player and vector from player-asteroid from the last update/current update. rotateBy,
    the amount to rotate around the centre point. distance, the distance to offset the point from the centre
    corresponding to actual distance calculated. */
    float prevTheta,theta,rotateBy,distance = 0;

    /* radar1,radar2,radar3,radar4, goRight - Used to determine which of 4 states the radar is in, between
    increasing/decreasing past 180 and 0 respectively. The rotation needs to be different depending on
    which direction the player is heading in etc. goRight signifies whether the player is rotating roughly
    right or left and is also needed to update the radar correctly. */
    boolean radar1,radar2,radar3,radar4, goRight=false;

    /* type - An integer to represent one of three types the item can be which dictates when/where it will
    be spawned and the colour/shape deformation as well. Either 0, 1, or 2, i.e. gold, red or blue, i.e. shield,
    score, weapon modifier. */
    int type;

    /* ItemDrop - The item drop constructor, all values passed in, including type which is determined by
    the type of onset detected and that matching the pulser type being compared. Direction updated through
    steer method in update() */
    ItemDrop(PVector Pos, PVector Target, float ms, float mf, int Type)
    {
        type = Type;
        pos = Pos;
        dir = new PVector(0,0,0);
        target = Target;
    }
}
```



```

maxspeed = ms;
origMaxspeed = ms;
maxforce = mf;
}

```

/* steer - Creates a direction vector based on current position and target position.

Method directly taken from Daniel Shiffman's example boid class : http://www.shiffman.net/itp/classes/nature/week06_s09/seekarrive/Boid.pde

with the additions of a boolean flag when the distance between asteroid-player is not greater than 0 and what to do if this is the case, i.e. awarding of points, shield or weapon bonuses if the collision is with the player. */

```

PVector steer(PVector target, boolean slowdown) {

    PVector steer = new PVector(0,0); // The steering vector
    PVector desired = PVector.sub(target,pos); // A vector pointing from the location to the target
    float d = desired.mag(); // Distance from the target is the magnitude of the vector
    // If the distance is greater than 0, calc steering (otherwise return zero vector)
    if (d > 0) {
        // Normalize desired
        desired.normalize();
        // Two options for desired vector magnitude (1 -- based on distance, 2 -- maxspeed)
        if ((slowdown) && (d < 100.0f)) desired.mult(maxspeed*(d/100.0f)); // This damping is somewhat
arbitrary
        else desired.mult(maxspeed);
        // Steering = Desired minus Velocity
        steer = PVector.sub(desired,dir);
        steer.limit(maxforce); // Limit to maximum steering force
    }

    if(d <50.0 && (attraction)) {
        isDead = true;
        if(type == 0) {
            // gold
            game.playerHealth+=10;
        }
        else if(type == 1) {
            // red
            game.playerScore+=1000;
        }
        else if(type == 2) {
            // blue
            game.playerWeaponEnergyBase+=10;
        }
        steer = new PVector(0,0);
    }
    else if(d <10.0 && !(attraction)) {
        isDead = true;
        steer = new PVector(0,0);
    }
    return steer;
}

```

/* update - The update method for calculating the distance between the item and target. If attraction is set the target is the player, if not, the wormhole. If the item is being attracted by the player, the maxspeed is adjusted as twice the original value. If not, the maxspeed is proportional to the average value calculated in FFTSample meaning the difficulty in retrieving the items fluctuated with the

```

development of the song. The direction vector is passed to the steer method and then added to the
position. dec is the average passed in from MainGame that is taken from FFTSample.*/
void update(float dec) {

    /* distance between myself and target, either player or wormhole. */
    float distance = pos.dist(target);

    /* If not attraction, moving towards wormhole, maxspeed is fft average. */
    if(!(attraction)) {
        maxspeed = dec;
        dir.add(steer(target,true));
    }
    /* if moving towards player, maxspeed becomes constant of twice the original value. */
    else if(attraction) {
        maxspeed = origMaxspeed*2;
        dir.add(steer(new PVector(0,0,0),true));
    }

    /* Move item */
    pos.add(dir);
}

/* render - Draws the item as a low detail sphere with a trail of smaller spheres behind it by offsetting
in the opposite direction of the item drop. */
void render() {
    pushMatrix();
    translate(pos.x, pos.y, pos.z);

    /* Check type and change stroke/fill values accordingly between gold, red and blue. */
    if(type == 0) {
        stroke( 0.1, pow(0.5,0.1), 1-0.5, 0.15 ); // 0.15
        fill( 0.1, pow(0.5,0.1), 1-0.5, 0.15 ); // 0.15
    }
    else if(type == 1) {
        stroke( 0.01, pow(0.5,0.1), 1-0.5, 0.15 ); // 0.15
        fill( 0.01, pow(0.5,0.1), 1-0.5, 0.15 ); // 0.15
    }
    else if(type == 2) {
        stroke( 0.6, pow(0.5,0.1), 1-0.5, 0.15 ); // 0.15
        fill( 0.6, pow(0.5,0.1), 1-0.5, 0.15 ); // 0.15
    }
}

sphereDetail(5);

/* If not attracting towards player, set the size of the item to be proportional to the fft sample of a
sub band derived from the item type. Meaning if the item is moving towards the wormhole, it's size
will be fluctuating with the music and all items of type a will fluctuate in the same
manner. */
float sizeItem = 0;
if(!(attraction)) {
    sizeItem = game.minimObj.thisSample.fft.getAvg((type+1)*2)/40;
}
/* else, player attracting it, set size as 1. */
else {
    sizeItem = 1;
}

sphere(sizeItem+0.4);

```

```

    /* Draw the other smaller spheres behind the main one by offsetting in opposite direction from
    item. */
    pushMatrix();
    for(int a=0; a<5; a++) {
        PVector tmpPos = new PVector(pos.x-(dir.x*a), pos.y-(dir.y*a), pos.z-(dir.z*a));
        translate(tmpPos.x, tmpPos.y, tmpPos.z);
        sphere((sizeItem+0.4) - ((a+1)/10));
    }
    popMatrix();

    popMatrix();
}

```

`drawRadarPoint()` - Method called each iteration in `MainGame` when in-game and not adjusting settings. Used to plot a point in relation to player to illustrate where the asteroid is in comparison. Due to the dot product between the player direction and player2asteroid vector returning between 0-180 (in front/behind), to adjust a point around 360 degrees in the correct manner, need to know the direction the player is heading in etc.

```

    The radar variables represent this. */
void drawRadarPoint() {
    //If moving right
    if(goRight) {
        // if increasing & at 180
        if(prevTheta > theta && rotateBy >= radians(180)) {
            radar1 = true; radar2=false;radar3=false;radar4=false;
            rotateBy = radians(180);
            rotateBy = rotateBy +(rotateBy-theta);
            rotateZ(-rotateBy);
        }
        // if decreasing & at 180
        else if(prevTheta < theta && rotateBy >= radians(180)){
            radar2 = true; radar1=false;radar3=false;radar4=false;
            rotateBy = radians(180);
            rotateBy = rotateBy +(rotateBy-theta);
            rotateZ(rotateBy);
        }
        // if decreasing & at 0
        else if(prevTheta < theta && rotateBy <= radians(0)) {
            radar3 = true; radar2=false;radar1=false;radar4=false;
            rotateBy = radians(360);
            rotateBy = rotateBy -theta;
            rotateZ(rotateBy);
        }
        // if increasing and at 0
        else if(prevTheta > theta && rotateBy <= radians(0)) {
            radar4 = true; radar2=false;radar3=false;radar1=false;
            rotateBy = radians(360);
            rotateBy = rotateBy -theta;
            rotateZ(-rotateBy);
        }
        // else central, no change
        else {
            if(radar1) {
                rotateZ(-rotateBy);
            }
            else if(radar2) {

```

```

    rotateZ(rotateBy);
  }
  else if(radar3) {
    rotateZ(-rotateBy);
  }
  else if(radar4) {
    rotateZ(rotateBy);
  }
}
// If moving left
else {
  // if increasing & at 180
  if(prevTheta > theta && rotateBy >= radians(180)) {
    radar1 = true; radar2=false;radar3=false;radar4=false;
    rotateBy = radians(180);
    rotateBy = rotateBy +(rotateBy-theta);
    rotateZ(rotateBy);
  }
  // if decreasing & at 180
  else if(prevTheta < theta && rotateBy >= radians(180)){
    radar2 = true; radar1=false;radar3=false;radar4=false;
    rotateBy = radians(180);
    rotateBy = rotateBy +(rotateBy-theta);
    rotateZ(-rotateBy);
  }
  // if decreasing & at 0
  else if(prevTheta < theta && rotateBy <= radians(0)) {
    radar3 = true; radar2=false;radar1=false;radar4=false;
    rotateBy = radians(360);
    rotateBy = rotateBy -theta;
    rotateZ(-rotateBy);
  }
  // if increasing & at 0
  else if(prevTheta > theta && rotateBy <= radians(0)) {
    radar4 = true; radar2=false;radar3=false;radar1=false;
    rotateBy = radians(360);
    rotateBy = rotateBy -theta;
    rotateZ(rotateBy);
  }
  else {
    // else central, no change
    if(radar1) {
      rotateZ(rotateBy);
    }
    else if(radar2) {
      rotateZ(-rotateBy);
    }
    else if(radar3) {
      rotateZ(rotateBy);
    }
    else if(radar4) {
      rotateZ(-rotateBy);
    }
  }
}
stroke(255,0,255);

```

```

    ellipse(0, -(distance/30), 1,1);
    prevTheta = theta;
  }
}

```

```

/* ObjectSpace Game
13/08/09

```

```

author: Ashley Morrison
class: MainGame
brief: This is the game object that contains the game loop with lists of all asteroids, item drops,
missiles and galaxies in-game. Stars, galaxies and asteroids are spawned at varying offsets from a
sphere, the radius of which was set in Main. This class also contains an instance of MinimAudio for the
playing/analysing of the in-game song used for spawning asteroids and item drops.

```

It also contains camera rotation code for rotating around a point in 3d, HUD code for displaying the current key positions of asteroids/items etc, player scores and a rose/weapon pattern as well as in-game settings that are changeable by hitting TAB. The HUD can be hidden with SHIFT.

Lastly it uses the Traer Physics library to keep hold of a set of particle position fired in some direction that is used to map a pattern of dots to representing the player attack. The bulk of this code is found in the Rose/Missiles classes respectively. The loop checks for whether it is in setup mode or not, if setup, simple display a blank screen with the in-game settings, else cycle through all the objects in the game and update/render.

```

*/

```

```

class MainGame {

```

```

    /* minimObj - The MinimAudio instance used to create, play and analyse the song of choice for in-
game. If in-game and not setup, play.*/
    MinimAudio minimObj;

```

```

    /* onSet - Used as a flag for when there is a onset detected and if this matches with the firing of the
player weapon, 2 extra missiles are fired giving a boost to the play for synchronised efforts. */
    boolean onSet = false;

```

```

    /* pulsers - The array list of pulser objects created in the constructor with positions around the player.
Updated in the Pulsers & updatePulsers methods respectively. */
    ArrayList pulsers;

```

```

    /* asteroidList - Array list of asteroid objects to be cycled through each update and calling each
asteroids update method. */
    ArrayList asteroidList;

```

```

    /* missiles - Array list of missile objects in the game at any given moment, missiles objects have a
finite life span and are periodically removed in the update methods found in Missiles() */
    ArrayList missiles;

```

```

    /* starsAlpha - the alpha values for drawing the points of the stars around the sphere to give a greater
feeling of depth in the scene. */
    float[] starsAlpha;

```

```

/* stars - The corresponding stars array which uses the alpha values above and is an array of positions
around a sphere used to draw points to. */
float[][] stars;

/* galaxies - 2d Array of floating point values for the distribution of Pulsers around the player, at an
offset from the normal spherical distribution. */
float[][] galaxies;

/* cam - The Matrix used for camera rotation found in CamRotation() */
PMatrix3D cam;

/* gravitate, isFire - Boolean flags for whether the player is attracting an item to his/herself and
whether the player is firing the weapon. */
boolean gravitate = true, isFire = false;

/* newRose - Rose object which is used to generate patterns that are connected with the current max
and averages calculated in FFTSample that are used to display the missile objects. A preview of the
current iteration is shown on the HUD. */
Rose newRose;

/* n - int that is updated to the overall average calculated in the FFTSample object and passed to
Rose.setPoints. */
int n=1;

/* mx, my - Mouse x and Mouse Y values updated in the main loop. */
float mx,my;

/* d - The direction the player is facing. */
PVector d;

/* tmpDir - non normalized player direction vector */
PVector tmpDir;

/* pos - The position of the player at 0,0,0 */
PVector pos;

/* physics - The particle system including from the Traer Physics library: http://www.cs.princeton.edu/~traer/physics/ */
ParticleSystem physics;

/* q - A particle to be re-assigned and attached to missile objects each time one is spawned. */
Particle q;

/* closeAsteroids - An array list of the asteroids that are within a certain range of the player to be used
for drawing the closest asteroids on the radar. */
ArrayList closeAsteroids;

/* onTarget - Is the player currently closely lined up with either an asteroid or an item drop, if yes,
true. */
boolean onTarget = false;

/* onAttract - Is the player currently onTarget, holding down right mouse button and has enough
weapon energy to attract an item. */
boolean onAttract = false;

/* tmpxRot - Amount of rotation in the x axis, used in CamRotation() */
float tmpxRot = 0;

```

```

/* tmpyRot - Amount of rotation in the y axis, used in CamRotation() */
float tmpyRot = 0;

/* items - Array list of all the item drops currently running in the game. */
ArrayList items;

/* itemOK - Boolean variable periodically set to allow the onset of a beat to result in items being
spawned, to restrict how often items are spawned. */
boolean itemOK = false;

/* itemCounter - Integer used with itemOK. */
int itemCounter = 0;

/* addItem - Integer counter used to total up how many items are being added in any one onset
detection, again for limiting purposes. */
int addItem = 0;

/* newWormhole - Wormhole object that is the point of attraction for all item drops. */
Wormhole newWormhole;

/* centreWormhole - A vector showing the centre position of the wormhole, used for the exact target
item drops move towards. */
PVector centreWormhole;

/* backCol - Used for the background colour, fades to white on completion of a song. */
int backCol = 0;

/* playerScore, playerHealth, playerWeaponEnergy, playerWeaponEnergyBase - Integer score and
shield values, floating point weapon energy values, base for when
                                weapon energy adjusted. */
int playerScore = 0, playerHealth=100; float playerWeaponEnergy=1000,
playerWeaponEnergyBase=1000;

/* maxspeedAsteroids - Default value for the maxspeed asteroids can travel. Can be changed in the
settings in-game. */
float maxspeedAsteroids = 2.0;

/* maxspeeItems - Default value for the maxspeed item drops can travel at. Also editable in the in-
game settings. */
float maxspeedItems = 3.0;

/* setupGame - If the player presses TAB, settings gui appears and main game if statement is re-
directed to a blank screen and gui, otherwise, update,render
as normal. */
boolean setupGame = false;

/* gameDifficulty - An integer for accessing/updating the game difficulty, which can be changed in
the settings. */
int gameDifficulty=0;

/* wormholeCentrePulse - The pulser object drawn at the centre of the wormhole object. */
Pulser wormholeCentrePulse;

/* MainGame - The constructor for the game object.

Constructs wormhole, item list, Rose object, particle system, stars alpha value, stars, galaxy
positions, asteroidList, pulser positions, missile list,
camera matrix, minimObj & the Pulser object at the centre of the wormhole in this order. */

```

```

MainGame(PApplet Main) {

  background(0);

  newWormhole = new Wormhole();

  items = new ArrayList();
  newRose = new Rose();
  pos = new PVector(0,0,0);
  physics = new ParticleSystem();

  /* STARS ALPHA */
  starsAlpha = new float[NoStars];
  for(int h = 0; h < starsAlpha.length; h++)
  {
    float a = random(1, 255);
    starsAlpha[h] = a;
  }

  /* STARS POSITIONS
  Based upon: http://www.openprocessing.org/visuals/?visualID=2277
  */
  stars = new float[NoStars][3];
  for(int i = 0; i < stars.length; i++)
  {
    float p = random(-PI, PI);
    float t = asin(random(-1, 1));
    stars[i] = new float[] {
      2*(R * cos(t) * cos(p)),
      2*(R * cos(t) * sin(p)),
      2*(R * sin(t))
    };
  }

  /* GALAXY POSITIONS
  Based upon: http://www.openprocessing.org/visuals/?visualID=2277
  */
  galaxies = new float[NoGalaxies][3];
  for(int j = 0; j < galaxies.length; j++)
  {
    float p = random(-PI, PI);
    float t = asin(random(-1, 1));

    galaxies[j] = new float[] {
      R * cos(t) * cos(p)/2,
      random(-500, 500),
      R * sin(t)/2
    };
  }

  /* ASTEROID LIST */
  asteroidList = new ArrayList();

  /* PULSER OBJECT POSITIONS */

```



```

pulsers = new ArrayList();
for(int i = 0; i < NoGalaxies; i++) {
    Pulsar newCon = new Pulsar((int)random(0,3), false);
    pulsers.add(newCon);
}

/* MISSILE ARRAY */
missiles = new ArrayList();

/* CAMERA MATRIX */
cam = new PMatrix3D();

/* MINIM OBJECT */
minimObj = new MinimAudio(Main, false);

/* PULSER-WORMHOLE OBJECT */
wormholeCentrePulse = new Pulsar((int)random(0,3), true);
}

/* updatePulsers - Method used to update pulsers particle positions and check for beat onset detection
and spawn item drops if necessary. Called from Pulsers() method below which cycles through galaxy
array and calls this method with the current index value. */
void updatePulsers(int index) {
    float r;
    addItem = 0;

    /* Create pulser based on one in array at index value */
    Pulsar thisPulsar = (Pulsar)pulsers.get(index);

    /* cycle through this pulsers particle array and check for beat detections, always gravitate but only
spawn an item if beat onset. */
    for(int i = 0; i < thisPulsar.Z.length; i++) {
        thisPulsar.Z[i].gravitate(new PulsarParticle(thisPulsar.avgX, thisPulsar.avgY, 0, 0, 0.6, false));

        /* If this pulser item type matches with any of the three beat onsets, spawn an item. */
        if(minimObj.beat2.isKick() && thisPulsar.type==0 || minimObj.beat2.isHat() &&
thisPulsar.type==1 || minimObj.beat2.isSnare() && thisPulsar.type==2)
        {
            for(int y=0; y<20; y++) {
                thisPulsar.Z[i].gravitate( new PulsarParticle( thisPulsar.avgX, thisPulsar.avgY, 0, 0, 0.5,
false ));
            }

            /* only add set of item drops per Pulsar Array list update. */
            if(addItems == 0 && itemOK==true) {
                ItemDrop newItem = new ItemDrop(new PVector(galaxies[index][0]+thisPulsar.avgX,
galaxies[index][1]+thisPulsar.avgY, galaxies[index][2]), centreWormhole, maxspeedItems, 0.1,
thisPulsar.type);
                items.add(newItem);
                addItem++;
            }
        }

        r = float(i)/thisPulsar.Z.length;

        /* Check the type of pulser and adjust the stroke value accordingly. */

```

```

if(thisPulser.type == 0) {
  stroke( 0.1, pow(r,0.1), 1-r, 0.35 ); // 0.15
}
else if(thisPulser.type == 1) {
  stroke( 0.01, pow(r,0.1), 1-r, 0.35 ); // 0.15
}
else if(thisPulser.type == 2) {
  stroke( 0.6, pow(r,0.1), 1-r, 0.35 ); // 0.15
}

/* display the pulser's particle array*/
thisPulser.Z[i].display();
}
}

/* CamRotation - gets the amount of rotation in the x/y and rotates the current matrix by these values
before finding the direction vector and calling the camera() method with d. */
void CamRotation() {
  float r;
  /* x and y axis rotation. */
  tmpxRot += -(mouseY - height / 2.0) / height / 5;
  tmpyRot += -(mouseX - width / 2.0) / width / 5;

  /* rotation values used to offset the display of a missile based upon the rotation of the player to
keep it in front. */
  if(degrees(tmpxRot) >= 180 || degrees(tmpxRot) <= -180) {
    tmpxRot = 0;
  }
  if(degrees(tmpyRot) >= 180 || degrees(tmpyRot) <= -180) {
    tmpyRot = 0;
  }

  /* CAM ROTATION
Taken from : http://www.openprocessing.org/visuals/?visualID=2277
*/
  cam.rotateX(-(mouseY - height / 2.0) / height / 5);
  cam.rotateY(-(mouseX - width / 2.0) / width / 5);
  PVector x = cam.mult(new PVector(1, 0, 0), new PVector(0, 0, 0));
  PVector y = cam.mult(new PVector(0, 1, 0), new PVector(0, 0, 0));
  d = x.cross(y); tmpDir = new PVector(d.x, d.y, d.z); d.normalize(); d.mult(R);

  camera(pos.x, pos.y, pos.z, d.x, d.y, d.z, y.x, y.y, y.z);
  /* CAM ROTATION */
}

/* Asteroids - Method called in main game loop render() to cycle through, update, add and draw
asteroids to the game. */
void Asteroids() {
  /* ASTEROIDS */
  colorMode(RGB,255);

  /* if beat onset, add a new asteroid to the list at a point offset from around the player.
Placement of asteroids based upon: http://www.openprocessing.org/visuals/?visualID=2277
*/
  if ( minimObj.beat.isOnset() ) {
    newWormhole.beat=true;

    float p = random(-PI, PI);

```

```

float t = asin(random(-1, 1));

float tmpX = 1.5*(R * cos(t) * cos(p)) + 200;
float tmpY = 0;

/* If difficulty is hard, adjust the y value to attack the player from all angles, else distribute the
asteroids at 0 on the y axis. */
if(newGUI.difficulty2.value()==1) {tmpY = R * cos(t) * sin(p); }

float tmpZ = 1.5*(R * sin(t)) + 200;

Asteroid newAsteroid = new Asteroid(new PVector(tmpX, tmpY, tmpZ), new PVector(0,0,0),
maxspeedAsteroids, 0.2, 45, 2, -100, 0, true);
newAsteroid.god = false;
asteroidList.add(newAsteroid);
}

/* cycle through the asteroid list of asteroids closest to the player, if asteroid is larger than x,
remove. */
for(int l=0; l<closeAsteroids.size(); l++) {
Asteroid thisAsteroid = (Asteroid)closeAsteroids.get(l);
float d = PVector.dist(thisAsteroid.loc,pos);

if (d > 2000) {
closeAsteroids.remove(l);
}
}

/* Cycle through main asteroid list, calculate whether player is within a margin to be directly
facing the asteroid, add to closeAsteroids if within X, check if dead or hit player and set the removing
boolean flag which will cause an explosion and alpha values to drop. */
for(int k=0; k<asteroidList.size(); k++) {
Asteroid thisAsteroid = (Asteroid)asteroidList.get(k);
thisAsteroid.update(pos);

PVector diff = PVector.sub(thisAsteroid.loc,pos);
float theta = PVector.angleBetween(diff, tmpDir);

if(theta < 0.05) {
onTarget = true;
}

float d = PVector.dist(thisAsteroid.loc,pos);
if(d < 2000) {
closeAsteroids.add(thisAsteroid);
}

if(thisAsteroid.isDead || thisAsteroid.HitPlayer(pos, false)) {

/* if not removing already, if sfx on, play sound effect, if game mode, lose health and flash the
screen white to emphasise collision. */
if(!(thisAsteroid.removing)) {
if(newGUI.sfx2.value()==1) { explosion =
game.minimObj.minim.loadSnippet("explosion.wav"); explosion.play(); }
if(newGUI.mode.value()==0) { playerHealth-=20; }
backCol = 255;
}
}
}

```

```

        asteroidList.remove(k);
    }
}
/* ASTEROIDS */
}

/* Pulsers - Cycle through galaxy array positions, translate and call updatePulsers to detect beat
onsets, spawn items and draw accordingly. */
void Pulsers() {
    /* PULSERS */
    colorMode(HSB,1.0);
    for(int j = 0; j < galaxies.length; j++)
    {
        pushMatrix();
        translate(galaxies[j][0], galaxies[j][1], galaxies[j][2]);
        updatePulsers(j);
        popMatrix();
    }
    /* PULSERS */
}

/* Wormhole - change the wormhole rotation speed dependent on the average of the FFTSample,
increment the colour value. Translate and draw the pulser object to the centre of the wormhole. */
void Wormhole() {
    /* WORMHOLE */
    pushMatrix();
    // speed dependent on fft average.
    newWormhole.speed=minimObj.thisSample.average/5000;
    // colour fluctuates between alpha=0, alpha=0.3 etc, flashes full if beat onset.
    newWormhole.colourInc();
    colorMode(HSB,1.0);
    newWormhole.render();
    popMatrix();

    /* WORMHOLE-PULSER DRAW */
    pushMatrix();
    translate(centreWormhole.x, centreWormhole.y, centreWormhole.z);
    for(int h = 0; h < wormholeCentrePulse.Z.length; h++) {
        wormholeCentrePulse.Z[h].gravitate(new PulserParticle(wormholeCentrePulse.avgX,
wormholeCentrePulse.avgY, 0, 0, 1.6, false ));
        if(minimObj.beat2.isKick()) {
            for(int z=0; z<20; z++) {
                wormholeCentrePulse.Z[h].gravitate(new PulserParticle(wormholeCentrePulse.avgX,
wormholeCentrePulse.avgY, 0, 0, 1.5, false ));
            }
        }
        wormholeCentrePulse.Z[h].display();
    }
    popMatrix();
    colorMode(RGB,255);
    /* WORMHOLE */
}

```

/* Items - Cycle through array list of items. Same calculation for items and onTarget as for asteroids. Also check if right mouse button down and weapon energy available before diverting item with boolean flag switch. */

```
void Items() {
    /* ITEMS */
    for(int h=0; h<items.size(); h++) {
        ItemDrop thisItem = (ItemDrop)items.get(h);

        PVector diff = PVector.sub(thisItem.pos,pos);
        float theta = PVector.angleBetween(diff, tmpDir);

        /* on target*/
        if(theta < 0.05) {
            onTarget = true;

            /* attracting item */
            if(onAttract && (playerWeaponEnergy>=10)) {
                thisItem.attraction = true;
            }
        }

        /*if item has been caught by player or met with wormhole, dead, remove from list. */
        if(thisItem.isDead) {
            items.remove(h);
        }
        /* update items with fft average to be used if not in attraction. */
        thisItem.update(minimObj.thisSample.average/5);

        colorMode(HSB,1.0);
        thisItem.render();
        colorMode(RGB,255);

        // reset attraction flag, means player needs to hold down mouse, thereby depleting more energy to
        attract it all the way. */
        thisItem.attraction = false;
    }
    /* ITEMS */
}
```

/* Stars - Cycle through stars array, translate to position and draw a point using the alpha value of the same index to adjust brightness. */

```
void Stars() {
    /* STARS */
    for(int i = 0; i < stars.length; i++)
    {
        pushMatrix();
        float[] p = stars[i];
        translate(stars[i][0], stars[i][1], stars[i][2]);
        stroke(255,starsAlpha[i]);
        point(stars[i][0], stars[i][1], stars[i][2]);
        popMatrix();
    }
    /* STARS */
}
```

/* Missiles - Cycle through all current missile objects, check whether player has energy before spawning more and check if firing coincides with a beat onset and if so, fire two extra missiles for the price of one as bonus. */

```

void Missiles() {
    /* MISSILE */
    if(isFire) {
        if(playerWeaponEnergy>=10) {
            /* position of particle spawned using particle system is the position of the player. */
            q = physics.makeParticle(1, pos.x,pos.y,pos.z);
            /* direction is a value proportional to the direction the player is facing. */
            q.velocity().set(d.x/30, d.y/30, d.z/30);
            /* pass in the current number of dots (maximum of fft) and n (avg of fft) to customise the
missile. */
            Missile newMissile = new Missile(q, d, tmpxRot, tmpyRot, minimObj.currentNoDots, n);
            missiles.add(newMissile);

            /* if onset, create two extra particles, offset to the left/right of original to cause extra damage
as bonus. */
            if(onSet) {
                q = physics.makeParticle(1, pos.x,pos.y,pos.z);
                q.velocity().set(d.x/30-2, d.y/30, d.z/30);
                Missile newMissile2 = new Missile(q, d, tmpxRot, tmpyRot, minimObj.currentNoDots, n);
                missiles.add(newMissile2);

                q = physics.makeParticle(1, pos.x,pos.y,pos.z);
                q.velocity().set(d.x/30+2, d.y/30, d.z/30);
                Missile newMissile3 = new Missile(q, d, tmpxRot, tmpyRot, minimObj.currentNoDots, n);
                missiles.add(newMissile3);
            }

            /* move missiles along. */
            physics.tick();
        }

        /* decrement player weapon energy if player has minimum. */
        if(playerWeaponEnergy >=20) {
            playerWeaponEnergy-=20;
        }
    }

    /* Cycle through current missiles, update positions, remove if dead and remove if greater than 20
total. */
    for(int l=0; l<missiles.size(); l++) {
        Missile thisMissile = (Missile)missiles.get(l);
        thisMissile.update();

        if(thisMissile.dead()) {
            physics.removeParticle(thisMissile.p);
            missiles.remove(l);
        }

        if(missiles.size() > 20) {
            missiles.remove(0);
        }
    }
    physics.tick();
    /* MISSILE */
}

```

```

/* HUD - draws crosshair, rose pattern and radar points for asteroids, item drops and the wormhole. */
void HUD() {
    camera();
    if(onTarget) {
        stroke(0, 255, 0);
    }
    else {
        stroke(255);
    }

    /* CROSS-HAIR */

    line(width / 2 - 9, height / 2 - 0, width / 2 + 8, height / 2 + 0);
    line(width / 2 - 0, height / 2 - 9, width / 2 + 0, height / 2 + 8);

    if(minimObj.beat2.isKick()) { stroke(0,0,255); fill(0,0,0);ellipse(width/2,height/2,25,25); }
    /* TARGETTING */

    /* ROSE DISPLAY */
    n = (int)minimObj.thisSample.average+5;
    pushMatrix();
    translate(width / 2, height-(height/6));
    fill(0,255);
    stroke(255);
    ellipse(0, 0, 80, 80);
    ellipse(0, 0, 75, 75);
    newRose.setDayglowColor(n+newRose.theta/6);
    newRose.setPoints(50, n, minimObj.currentNoDots, 0);
    popMatrix();
    /* ROSE DISPLAY */

    /* BASIC RADAR ELLIPSE */
    fill(0,0,0,255);
    float offSetX = width/2-155;
    float offSetY = height-(height/6);

    translate(offSetX, offSetY);

    smooth();
    stroke(255);
    fill(0,255);
    ellipse(0, 0, 150, 150);
    ellipse(0, 0, 155, 155);
    stroke(255,0,0);
    ellipse(0, 0, 1,1);
    /* BASIC RADAR ELLIPSE */

    /* WORMHOLE DRAW */
    if(tmpDir != null) {
        PVector tmpWormhole = centreWormhole;
        PVector diff = PVector.sub(tmpWormhole,pos);
        // angle between the direction of the player to wormhole and the direction of the player.

```

```

newWormhole.theta = PVector.angleBetween(diff, tmpDir);
//distance between the wormhole and player.
newWormhole.distance = tmpWormhole.dist(pos);

pushMatrix();
  if(mx>width/2) { // moving right
    newWormhole.goRight = true;
    newWormhole.drawRadarPoint();
  }
  else if(mx <width/2) { // moving left
    newWormhole.goRight = false;
    newWormhole.drawRadarPoint();
  }
  else { // central
    newWormhole.drawRadarPoint();
  }
popMatrix();
/* WORMHOLE DRAW */

```

```

/* ITEM DRAW */
for(int a=0; a< items.size(); a++) {
  ItemDrop thisItem = (ItemDrop)items.get(a);

  PVector diff2 = PVector.sub(thisItem.pos,pos);

  //angle between direction to item and direction of player.
  thisItem.theta = PVector.angleBetween(diff2, tmpDir);
  //distance between item and player.
  thisItem.distance = thisItem.pos.dist(pos);

  if(thisItem.distance < 1000) {
    pushMatrix();
    if(mx>width/2) { // moving right
      thisItem.goRight = true;
      thisItem.drawRadarPoint();
    }
    else if(mx <width/2) { // moving left
      thisItem.goRight = false;
      thisItem.drawRadarPoint();
    }
    else { // central
      thisItem.drawRadarPoint();
    }
    popMatrix();
  }
}
/* ITEM DRAW */

```

```

/* ASTEROID DRAW */
for(int b=0; b< closeAsteroids.size(); b++) {
  Asteroid thisAsteroid = (Asteroid)closeAsteroids.get(b);

  PVector diff2 = PVector.sub(thisAsteroid.loc,pos);

```



```

// angle between asteroid-player direction and player direction.
thisAsteroid.theta = PVector.angleBetween(diff2, tmpDir);
thisAsteroid.distance = thisAsteroid.loc.dist(pos);

pushMatrix();
pushMatrix();
if(mx>width/2) { // moving right
  thisAsteroid.goRight = true;
  thisAsteroid.drawRadarPoint();
}
else if(mx <width/2) {
  thisAsteroid.goRight = false;
  thisAsteroid.drawRadarPoint();
}
else {
  thisAsteroid.drawRadarPoint();
}
popMatrix();
popMatrix();
}
/* ASTEROID DRAW */
}
}

/* render - main game loop. Determines whether in setup mode or not and what to draw otherwise.
Updates gui values, determines when to change to game over state and in what fashion. Also
replenishes weapon energy, increments item counter to limit number spawned, updates the audio object
and mouse positions along with calling all the object specific methods above. */
void render() {

  // if in setup mode, display blank screen
  if(setupGame) {
    background(0);
  }

  // set wormhole position.
  centreWormhole = new PVector(newWormhole.mx+newWormhole.centre.x,
newWormhole.centre.y, newWormhole.my+newWormhole.centre.z);
  newGUI.update();

  // if there is a beat detection onset in the frequency that matches a kick sound, onset is true. Used
with missiles firing/sync.
  if(minimObj.beat2.isKick()) { onSet = true; }

  // if in game mode
  if(!(setupGame)) {
    newGUI.gameData.show();

    camera();

    if(minimObj.song.isPlaying() && playerHealth>0) {backCol = 0;}
    if(!game.minimObj.song.isPlaying()) { game.backCol+=5; if(game.backCol>=255)
{ gameOver=true;inGame=false; newGUI.gameData.hide(); } }
    if(playerHealth == 0) { game.minimObj.song.pause(); game.backCol+=3; if(game.backCol>=255)
{ gameOver=true;inGame=false; newGUI.gameData.hide(); } }

```

```

background(backCol);

closeAsteroids = new ArrayList();
itemCounter++;

// replenish weapon energy
if(playerWeaponEnergy < playerWeaponEnergyBase) {
    playerWeaponEnergy+=1;
}

// increment item counter, reset periodically.
itemOK = false;
if(itemCounter> 100) {
    itemCounter = 0;
    itemOK = true;
}

// update audio
minimObj.update();

colorMode(HSB,1.0);

mx = mouseX;
my = mouseY;

/* MAIN GAME OBJECT CALLS */
CamRotation();
Asteroids();
Pulsers();
Wormhole();
Items();
Stars();

if(onAttract) {
    if(playerWeaponEnergy >=1.5) {
        playerWeaponEnergy-=1.5;
    }
}

Missiles();

if(newGUI.mode.value()==0) {
    HUD();
}
else {
    newGUI.gameData.hide();
}
/* MAIN GAME OBJECT CALLS */
}

/* IF NOT GAME MODE - CHECK WHETHER SETUP IS FLAGGED, IF SO, DISPLAY GUI */

/* TARGETTING */
camera();
if(inGame && setupGame) { newGUI.gameSettings.show(); newGUI.gameData.hide();}

```

```

    fill(255);
    stroke(255);

    /* RESET FLAGS */
    onSet = false;
    onTarget = false;
    isFire = false;
}

}

/* ObjectSpace Game
13/08/09

author: Ashley Morrison
class: MainMenu
brief: This is a class that draws text to the screen depending on what state the system is in, either title
or game over.
*/

class MainMenu {

    MainMenu() {
        // alternative fonts: Ethnocentric Distant Galaxy Good Times Induction Mael Nasalization Neon
        Lights SF Movie Poster Times New Roman
        textFont(createFont("Ethnocentric", 32));
        textAlign(CENTER);
    }

    void render() {

        /* if main menu, display the title text */
        if(mainMenu) {
            textSize(40);
            fill(255);
            text("OBJECTS IN SPACE", width/2, height-(height-75));
        }

        /* if game over, display message, score and instructions. */
        if(gameOver) {
            textSize(40);
            fill(255);
            text("GAME OVER", width/2, height/2-200);

            textSize(30);
            fill(255);
            text("SCORE: " + str(game.playerScore), width/2, height/2);

            textSize(20);
            fill(255);
            text("PRESS ENTER TO RETURN TO MAIN MENU", width/2, height/2+200);
        }
    }
}
}

```

```

/* ObjectSpace Game
13/08/09

author: Ashley Morrison
class: MinimAudio
brief: Encapsulates all the audio related objects and method calls that the system uses. Contains
minim library objects: Minim, AudioPlayer FFT, BeatDetect and instantiations of the BeatListener
class. Uses three different beat listeners for the menu screen song, the in-game song energy
detection(asteroids) and frequency detection(pulsers). Sets two main values, the overall average
frequency value and a latest maximum peak value.
*/

class MinimAudio {

    /* minim - Main Minim audio library class object. */
    Minim minim;

    /* song - Minim library object for loading in/playing songs. */
    AudioPlayer song;

    /* thisSample - instantiation of the class which encapsulates and uses the minim library FFT class. */
    FFTSample thisSample;

    /* bl,beat, bl2,beat2, bl3,beat3 - Beat detection and listener objects for the three varieties of beat
detection used. */
    BeatListener bl; BeatDetect beat; BeatListener bl2; BeatDetect beat2; BeatListener bl3; BeatDetect
beat3;

    /* currentNoDots - set in the update method to be proportional to the current maximum peak value. */
    int currentNoDots;

    /* sLength - song length in milliseconds. */
    int sLength;

    /* tempFactor - used to return the fraction of the song completed which in turn is used to blend the
pattern of the wormhole from a starting value to an end design. */
    float tempFactor;

    /* MinimAudio Ctr - Takes in the PApplet to use for constructing the Minim object. Also a boolean
value to determine whether the state is currently in the title screen or not. */
    MinimAudio(PApplet main, boolean titleScreen) {
        minim = new Minim(main);

        /* if a file hasn't been chosen, use this default. */
        if(filename.equals("")) { filename = "kavinsky.mp3"; }

        /* if title screen, set up the song and beat detection with the filename set in Main. */
        if(titleScreen) {
            song = minim.loadFile(filename2, 4096); // 4096
            beat3 = new BeatDetect();
            beat3.setSensitivity(0);
            bl3 = new BeatListener(beat3, song);
        }
        /* else use the file chosen or the default set above. */
        else if(!titleScreen) {
            song = minim.loadFile(filename, 4096); // 4096

```

```

}

/* create instance of FFTSample. */
thisSample = new FFTSample(song);

/* create beat detection for missile/audio sync */
beat = new BeatDetect();
println(beatSensitivity);
beat.setSensitivity(beatSensitivity);
bl = new BeatListener(beat, song);

/*setup beat detection for pulsers - uses frequency energy values. */
beat2 = new BeatDetect(song.bufferSize(), song.sampleRate());
beat2.setSensitivity(500);
bl2 = new BeatListener(beat2,song);

sLength = song.length();

if(titleScreen) {
    song.play();
    song.loop();
}
else {
    song.play();
    /* if mode is visualisation, loop the song. */
    if(newGUI.mode.value()==1) { song.loop(); }
}
}

/* update - updates the FFT which forwards the analysis of the spectrum values. Recalculates the
factor by which to adjust the wormhole pattern.
Also calls findMax to calculate the maximum peak and overall average values. */
void update() {
    thisSample.update();
    tempFactor = (float)millis() / sLength;

    thisSample.findMax();
    // set number of dots for rose pattern
    currentNoDots = (int)thisSample.currentMaximum/5;
}
}

/* ObjectSpace Game
13/08/09

author: Ashley Morrison
class: Missile
brief: Missile object which takes the pattern generated by the Rose class for display and the position
returned by the particle in the Traer particle system as it's position upon being generated and fired in
the direction the player is facing. Contains an update method which checks for collisions and renders
the pattern to a new point in space based on a particle position. Each missiles has it's own particle
object which is updated by the "physics.tick()" call in MainGame. Also has a timer value that is
decremented resulting in missiles being removed after x time if they haven't hit anything.
*/

class Missile {

```

```

/* r - radius of missile object. */
float r;

/* timer - Timer value decremented over time resulting in time span before removal. */
float timer;

/* p - This missiles particle object, get updated in MainGame through physics.tick call. */
Particle p;

/* rotx, roty, rotz - rotational values for displaying the pattern in front of the player. */
float rotx,roty,rotz;

/* maximumSpecValue - the value taken from the FFTSample for currentMaximum when this
missiles was created. */
int maximumSpecValue;

/* averageSpecValue - the value taken from the FFTSample for the overall average when this missile
was created. */
int averageSpecValue;

/* Missile - constructor - passes in the particle object from MainGame as well as rotational values and
the currentMaximum/average FFT values. */
Missile(Particle P, PVector Vel, float rotX, float rotY, int Max, int Avg) {
    rotx = rotX;
    roty = rotY;
    p = P;
    r = 40.0;
    timer = 100.0;
    rotz = 0;
    maximumSpecValue = Max;
    averageSpecValue = Avg;
}

/* update - Method to update location, increments the z rotation, checks for collisions, reduces timer
and renders to screen if still alive. */
void update() {
    rotz +=0.1;
    if(rotz>360) {
        rotz = 0;
    }

    collision();
    timer -= 1.0;
    render();
}

/* collision - Gets the particle position, compares with asteroid positions in the same way the asteroid
collision detection does using Daniel Shiffman's example collision code:
http://www.shiffman.net/itp/classes/nature/week06_s09/seekarrive/Boid.pde */
boolean collision() {
    PVector tmpPos = new PVector(p.position().x(), p.position().y(), p.position().z());

    for(int k=0; k<game.asteroidList.size(); k++) {
        Asteroid thisAsteroid = (Asteroid)game.asteroidList.get(k);
        float d = PVector.dist(tmpPos,thisAsteroid.loc);

```

```

float sumR = thisAsteroid.r + (r*.97/2);

if (d < sumR && thisAsteroid.god == false) {
    thisAsteroid.asteroidHealth -=maximumSpecValue;

    thisAsteroid.isHit = true;
    // kills missile on impact
    timer=0;

    // if health less/equal to 0, asteroid removed, if parent, splitAsteroid called to create two new
ones.
    if(thisAsteroid.asteroidHealth <= 0) {
        if(thisAsteroid.parent && (game.gameDifficulty==1) ) {
            thisAsteroid.splitAsteroid();
        }

        timer = 0;
        // removing boolean set to cause explosion and reduce alpha
        thisAsteroid.removing = true;
        return true;
    }
}

return false;
}

/* render - Method to display missile */
void render() {
    stroke(0,255,0);
    pushMatrix();

    /* translate to position of the particle p */
    translate(p.position().x(), p.position().y(), p.position().z());
    rotateX(rotx);
    rotateY(roty);

    /* draw the missile by calling the rose.setPoints method with relavant values. */
    game.newRose.setPoints(r, averageSpecValue, maximumSpecValue, rotz);
    popMatrix();
}

/* dead - Check if timer is less/equal to 0, if so, return true */
boolean dead() {
    if (timer <= 0.0) {
        return true;
    } else {
        return false;
    }
}
}

/* ObjectSpace Game
13/08/09

```

author: Ashley Morrison

class: Pulsar

brief: This class is heavily based upon the OpenProcessing example 'Gravity Swarm' by Claudio Gonzales: <http://www.openprocessing.org/visuals/?visualID=2363> The Swarm.pde file is now the Pulsar class and the particle.pde is now the PulsarParticle class. Besides an initial call to gravitate the particles around a central point in the ctr, each pulsar has it's array of PulsarParticles updated/displayed through calls made in the MainGame.updatePulsars() method depending on beat onsets. It has also been adjusted for OpenGL, and each Pulsar object is distributed in 3d space around a position offset from a sphere.

Also, the initial distribution of the PulsarParticles is extended to be around a sphere as this influences the type of pattern to be made subsequently. This distribution around a circle uses a 2d version of: <http://www.openprocessing.org/visuals/?visualID=861> by 'Starkes'. The point of attraction is also designated as the centre of the circle.

*/

```
class Pulsar {
```

```
    /* Z - Array of PulsarParticles, size set in the title screen. */  
    PulsarParticle[] Z = new PulsarParticle[galaxyParticleNo];
```

```
    /* pos - Positional vector of the Pulsar. */  
    PVector pos;
```

```
    /* avgX, avgY - x,y value to gravitate around. */  
    float avgX = 0;  
    float avgY = 0;
```

```
    /* type - The type of the PulsarParticle, randomly chosen at the MainGame setup to be one of three  
types used in conjunction with the item drops and beat detection. */  
    int type;
```

```
    /* Pulsar - Ctr method, passes in an integer for the type and a boolean designating whether this Pulsar  
is a generic one or the one used with the centre of the wormhole. */  
    Pulsar(int Type, boolean wormhole) {
```

```
        type = Type;
```

```
        /* if wormhole variety, set particle size to much larger */  
        if(wormhole) { Z = new PulsarParticle[2000]; }
```

```
        /* calculate the x,y positions of a sphere and call PulsarParticle ctr with the theta/u values to be  
used. */
```

```
        for(int i = 0; i < Z.length; i++) {  
            float theta = random(0,TWO_PI);  
            float u = random(-1,1);
```

```
            Z[i] = new PulsarParticle( theta, u, 0, 0, 2, true );  
            avgX+=Z[i].x;  
            avgY+=Z[i].y;  
        }
```

```
        avgX = avgX/Z.length;  
        avgY = avgY/Z.length;  
        /* set position of Pulsar */  
        pos = new PVector(avgX, avgY);
```



```

    /* cause an initial gravitation towards centre point. */
    for(int i = 0; i < Z.length; i++) {
        for(int y=0; y<20; y++) {
            Z[i].gravitate( new PulserParticle( avgX, avgY, 0, 0, 0.6, false ) );
        }
        /* display particles in array. */
        Z[i].display();
    }
}
}
}

```

```

/* ObjectSpace Game
13/08/09

```

```

author: Ashley Morrison
class: PulserParticle
brief: The particle class to be used with the Pulser objects. Contain x,y,z values and previous x,y,z
values as well as magnitude, angle and mass floating point values for use in pulling them towards a
position in a way analogous to the pull of gravity. If this is the first time instantiation, calculate the x/y
positions around a point on a circle. Else x/y = theta/u.
*/

```

```

class PulserParticle {

    /* x,y,z,px,py,pz - Current and previous x,y,z values. */
    float x; float y; float z; float px; float py; float pz;

    /* magnitude - floating point value for the magnitude of the pull. */
    float magnitude;

    /* angle - floating point value for the angle of attraction. */
    float angle;

    /* mass - floating point value for the mass of the particle which will determine the magnitude of the
force. */
    float mass;

    /* theta - theta value to be passed in from PulserParticle. */
    float theta;

    /* u - u value to be passed in from PulserParticle. */
    float u;

    /* radius - radius of the distribution of the particles, set in menu screen. */
    float radius = galaxyRadius; // 750

    /* PulserParticle - passed in x/y values, magnitude, angle and mass. */
    PulserParticle( float dx, float dy, float V, float A, float M, boolean startup) {

        if(startup) {
            theta = dx;
            u = dy;

            /* Uses circular distribution of particles taken from: http://www.openprocessing.org/visuals/?visualID=861 except just for x/y. */
            x = radius*cos(theta)*sqrt(1-(u*u));
            y = radius*sin(theta)*sqrt(1-(u*u));

```

```

    px = radius*cos(theta)*sqrt(1-(u*u));
    py = radius*sin(theta)*sqrt(1-(u*u));
  }
  else {
    x = dx;
    y = dy;

    px = dx;
    py = dy;
  }

  magnitude = V;
  angle = A;
  mass = M;
}

/* gravitate - Takes in a new PulsarParticle object to be attracted to which has the position of being in
the centre of the circle. Uses equations for calculating the magnitude and angle values taken from:
http://www.openprocessing.org/visuals/?visualID=2363*/
void gravitate( PulsarParticle Z ) {
  float F, mX, mY, A;
  if( sq( x - Z.x ) + sq( y - Z.y ) != 0 ) {
    F = mass * Z.mass;
    mX = ( mass * x + Z.mass * Z.x ) / ( mass + Z.mass );
    mY = ( mass * y + Z.mass * Z.y ) / ( mass + Z.mass );
    A = findAngle( mX - x, mY - y );

    mX = F * cos(A);
    mY = F * sin(A);

    mX += magnitude * cos(angle);
    mY += magnitude * sin(angle);

    magnitude = sqrt( sq(mX) + sq(mY) );
    angle = findAngle( mX, mY );
  }
}

/* display - decreases the magnitude value to slowly bring the particles to a stop over time. updates
the x,y values based on the magnitude/angle values calculated in gravitate() and draws a line between
the current and old x/y positions before updating the old ones.*/
void display() {
  magnitude *= 0.825;

  // update latest values with angle of direction and decremented magnitude.
  x += magnitude * cos(angle);
  y += magnitude * sin(angle);

  // draw to screen
  pushMatrix();
  line(px,py,x,y);
  popMatrix();

  // update old values.
  px = x;
  py = y;
}

```

```

}

/* findAngle - Takes in the x/y values for the Particle to be attracted to. This method is unaltered from
the original example:
http://www.openprocessing.org/visuals/?visualID=2363*/
float findAngle( float x, float y ) {
  float theta;
  if(x == 0) {
    if(y > 0) {
      theta = HALF_PI;
    }
    else if(y < 0) {
      theta = 3*HALF_PI;
    }
    else {
      theta = 0;
    }
  }
  else {
    theta = atan( y / x );
    if( ( x < 0 ) && ( y >= 0 ) ) { theta += PI; }
    if( ( x < 0 ) && ( y < 0 ) ) { theta -= PI; }
  }
  return theta;
}

```

```

/* ObjectSpace Game
13/08/09

```

```

author: Ashley Morrison
class: Rose

```

```

brief: This class is a modification and re-use of Jim Bumgardner's 'Rose Display' OpenProcessing
sketch: http://www.openprocessing.org/visuals/?visualID=1555

```

It's also the sketch that most inspired the possibilities of using audio to fuel visualisations for this project. The key value for the equation that is documented in the background section of the thesis is a value n which determines the number of petal like shapes or curves the pattern generates. This rose generation now takes in a value n that is connected with the FFTSample class and the average of those frequency values. It also passes in a number of points to be used for drawing the pattern which is the current maximum peak value, hence in a moment of explosion in the audio, the rose generation will jump in complexity.

```

*/

class Rose {

  /* n - set in MainGame to be equal to the average+5. */
  int n;

  /* theta - used to calculate the radius, colour and curve points */
  float theta;

  /* sizePoint - how big should the points be displayed as. */
  float sizePoint;

```

```

/* rr,gg,bb - the red, green and blue values selected in the setDayglowColor() method, left from the
original sketch. */
int rr, gg, bb;

/* Rose - ctr */
Rose() {

}

/* setPoints - Takes in a radius value to determine the size of the pattern drawn, a new n value, the
number of dots to draw and a rotational value. */
void setPoints(float radius,int newN, int NoDots, float rot) {
    sizePoint = 2;
    n = newN;

    float rad = radius*.97/2;
    float cx = 0;
    float cy = 0;

    pushMatrix();
    rotateZ(rot);
    /* for the total number of dots to be drawn this time, calculate a colour, radius and x,y positions for
NoDots and draw as ellipses. */
    for (int i = 0; i <= NoDots; ++i)
    {
        pushMatrix();
        theta = i*PI*2 / NoDots;
        setDayglowColor(n+theta/6);
        float r = rad * sin(n*theta);
        float px = cx + cos(theta)*r;
        float py = cy + sin(theta)*r;
        ellipse(px,py,2,2);
        popMatrix();
    }
    popMatrix();
}

/* setDayglowColor - takes in a hue value which is also proportional to the n value having been
passed in from setPoints() */
void setDayglowColor(float myHue) {
    float ph = sin(millis()*.0001);

    rr = (int) (sin(myHue) * 127 + 128);
    gg = (int) (sin(myHue + (2*ph) * PI/3) * 127 + 128);
    bb = (int) (sin(myHue + (4*ph) * PI/3) * 127 + 128);
    fill(rr,gg,bb);
    stroke(rr,gg,bb);
}
}
}

```

```

/* ObjectSpace Game
13/08/09

```

```

author: Ashley Morrison
class: SpiralGalaxy

```

brief: SpiralGalaxy class modified slightly from Philippe Guglielmetti's 'Spiral Galaxy' OpenProcessing sketch: <http://www.openprocessing.org/visuals/?visualID=699>
 This class is used for the menu visualisation and contains modifications the the display of the points as well as allowing the number of points to be added to. The result on the menu screen is a set of spiral patterns that develop with some synchronisation to the music, with more points being added over time as well as changes in colour etc.

```

*/

class SpiralGalaxy3 {

  /* colour - the colour of the spiral galaxy to be used. */
  float colour = random(1);

  /* stars - the current number of stars, incremented over time with the audio. */
  int stars=0;

  /* Rmax - galaxy radius */
  int Rmax=300;

  /* speed - a random rotational speed. */
  float speed=random(0.001, 0.005);

  // stars follow elliptic orbits around the center
  /* eratio - ellipse ratio */
  float eratio=random(0.2,1);
  /* etwist - twisting factor (orbit axes depend on radius) */
  float etwist=random(0,8.0/Rmax);

  /* angle - Array list of angle values. */
  ArrayList angle = new ArrayList();

  /* radius - Array list of radius values. */
  ArrayList radius = new ArrayList();

  /* angleArray - array of angle values. */
  float []angleArray;

  /* radiusArray - array of radius values. */
  float []radiusArray;

  /* cx, cy - centre x and centre y values. */
  float cx; float cy;

  /* beat - Boolean flag for whether the corresponding beat has been detected. */
  boolean beat=false;

  /* colourChange - Boolean flag for whether it is time increment the colour. Changed in Background
  class. */
  boolean colourChange=false;

  /* pos - Position of the spiral galaxy in 2d on the screen. */
  PVector pos;

  /* SpiralGalaxy - Constructor, takes in a radius size. Initialises the star angle/radius array values to be
  used in determining point positions in render() */
  SpiralGalaxy3(int rmax){
    pos = new PVector(random(-width/3,width/3), random(-height/3,height/3));
  }
}

```

```

Rmax=rmax;
// begin in the center
cx = width/2;
cy = height/2;

// init stars
for (int i=0; i< stars; i++){
  angle.add(random(0,2*PI));
  radius.add(random(1,Rmax));
}
colorMode(RGB,255);
}

/* updateStarCount - Convert between the array and array lists to adjust the number of stars to draw
per galaxy. */
void updateStarCount() {
  /* UPDATING SIZE */
  int len = angle.size();
  angleArray=new float[len];
  radiusArray=new float[len];

  Float[] fa1 = new Float[len];
  angle.toArray(fa1);

  Float[] fa2 = new Float[len];
  radius.toArray(fa2);

  // re-fill arrays with new values from adjusted array lists, angle and radius.
  for (int h = 0; h < len; h++)
  {
    angleArray[h] = fa1[h];
    radiusArray[h] = fa2[h];
  }
  /* UPDATING SIZE */
}

/* drawGalaxy - cycles through angle and radius values to calculate latest x,y values around centre
point. Slightly modified to adjust for star count changes and colour changes.
http://www.openprocessing.org/visuals/?visualID=699 */
void drawGalaxy(){
  float strokeCol;

  colorMode(HSB,1);
  float r,a,x,y,b,s,c,xx,yy,dd;

  updateStarCount();

  pushMatrix();
  translate(pos.x, pos.y);
  for (int i=0; i< stars; i++){
    r=radiusArray[i];
    a=angleArray[i]+speed*(Rmax/r)*3.0; // increment angle
    angle.set(i,a);

    /* Calculation of new point positions taken entirely from Guglielmetti's sketch. */
    x=r*sin(a);
    y=r*eratio*cos(a);
    b=r*etwist;

```

```

s=sin(b);
c=cos(b);
xx=cx + s*x + c*y; // a bit of trigo
yy=cy + c*x - s*y;
/* Calculation of new point positions taken entirely from Guglielmetti's sketch. */

strokeCol = float(i)/stars;
if(beat) {
  stroke( colour, pow(strokeCol,0.1), 1-strokeCol, 0.4 );
}
else {
  stroke( colour, pow(strokeCol,0.1), 1-strokeCol, 0.35 );
}
// draw point and extend in the z axis by an amount proportional to the radius value.
point(xx, yy, -r/2);
}
popMatrix();

strokeWeight(1);
colorMode(RGB,255);
if(colourChange) {
  colour+=random(0.02);
  if( colour > 1 ) {
    colour = colour%1;
  }
  colourChange=false;
}
}
}

/* ObjectSpace Game
13/08/09

author: Ashley Morrison
class: Wormhole
brief: Wormhole class modified from Philippe Guglielmetti's 'Spiral Galaxy' OpenProcessing sketch:
http://www.openprocessing.org/visuals/?visualID=699

This class is used for the wormhole feature in-game that item drops are drawn towards. The changes
made are setting the centre point, pulsing the colour of object drawn, altering the appearance by not
drawing X points that are close to the centre to give more of a black hole effect, changing the
ratio/twist values determining the pattern drawn when there is a beat and by a factor of how far along
the song is. The wormhole position is also drawn to the HUD to relay it's position in relation to the
player.
*/

class Wormhole {
  /* speed - a random rotational speed. */
  float speed=0.0001;

  // stars follow elliptic orbits around the center
  /* eratio - ellipse ratio */
  float eratio=0.1;

```

```

/* etwist - twisting factor (orbit axes depend on radius) */
float etwist=0.005;//8.0/wormholeRmax; // twisting factor (orbit axes depend on radius)

/* angleArray - array of angle values. */
float []angle=new float[wormholeNoStars];

/* radiusArray - array of radius values. */
float []radius=new float[wormholeNoStars];

/* cx, cy - centre x and centre y values. */
float cx; float cy;

/* x,y values to translate the wormhole by */
float mx,my;

/* centre - A Vector representing the centre position of the wormhole points. */
PVector centre;

/* prevTheta,theta,temp,distance - All used for the drawing of the point on the radar corresponding to
asteroid-player location discrepancy. prevTheta and theta are the difference in dot products between the
direction of the player and vector from player-asteroid from the last update/current update. rotateBy,
the amount to rotate around the centre point. distance, the distance to offset the point from the centre
corresponding to actual distance calculated. */
float prevTheta,theta,rotateBy,distance = 0;

/* radar1,radar2,radar3,radar4, goRight - Used to determine which of 4 states the radar is in, between
increasing/decreasing past 180 and 0 respectively. The rotation needs to be different depending on
which direction the player is heading in etc. goRight signifies whether the player is rotating roughly
right or left and is also needed to update the radar correctly. */
boolean radar1,radar2,radar3,radar4, goRight=false;

/* beat - Boolean flag for whether the corresponding beat has been detected. */
boolean beat=false;

/* hole - floating point value to determine the size of the black hole that is incremented with a beat
detected. */
float hole = 10;

/* colour, increasing, decreasing - A colour value in the hsb range and two boolean values determining
whether the colour is being incremented or decremented as it phases between almost invisible and full
brightness when a beat is detected. */
float colour=0; boolean increasing=true, decreasing=false;

/* Wormhole - Constructor for the Wormhole class. */
Wormhole(){

/* random position around player. */
mx = random(-1000,1000);
my = random(-1000,1000);
speed=speed/frameRate;
centre = new PVector(0,0);

// begin in the center
cx = width/2;
cy = height/2;

// init angle/radius values.
for (int i=0; i< wormholeNoStars; i++){

```



```

    angle[i]= random(0,2*PI);
    radius[i]=random(1,wormholeRmax);
}

// used in calculating centre point based on smallest radius value.
float minimum=10000;
int index=0;
float tmpRadius=0;

// cycles through all radius values finding the smallest and sets index to relevant i value to be used in
calculateCentre()
float xx,yy;
for (int i =0; i< wormholeNoStars; i++){
    float r=radius[i];

    tmpRadius = r;
    // index set here
    if(tmpRadius < minimum) { minimum=tmpRadius; index=i; }
}

calculateCentre(index);
}

/* calculateCentre - uses the index found above for the value in the radius array that was smallest to
access this in working out the x/y values and setting centre accordingly. */
void calculateCentre(int index) {
    float xx,yy;

    // index used here to get radius and angle values for working out the x/y
    float r=radius[index];
    float a=angle[index]+speed; // increment angle

    float x = r*sin(a);
    float y= r*cos(a);
    float b=r*etwist;
    float s=sin(b);
    float c=cos(b);

    xx = cx+s*x+c*y;
    yy = cy+c*x-s*y;

    // centre point set here.
    centre = new PVector(xx,yy,-r/2);
}

/* colourInc - Used to set a colour value for drawing the points, flips between increasing and
decreasing the value to create a slow pulsing effect and changes to full brightness on a beat detection.
Uses hsb scale, maximum value of 0.4 before decreasing to near 0. */
void colourInc() {
    colorMode(RGB,255);

    if(increasing) {
        colour+=random(0.001);
    }
    else if(decreasing) {
        colour-=random(0.001);
    }
}

```

```

float diff = (Math.abs(colour-0.4));

// if approx 0.4 and decreasing, switch flags
if(diff>0.38 && decreasing) { increasing=true; decreasing=false; }

// if almost 0 and increasing, switch flags
if(diff<0.002 && increasing) { increasing=false; decreasing=true; }
}

/* render - cycles through angle and radius values to calculate latest x,y values around centre point.
Slightly modified to adjust for changes in the pattern and black hole size.
http://www.openprocessing.org/visuals/?visualID=699 */
void render(){

float xx,yy;

pushMatrix();
// translate to mx/my position
translate(mx,0,my);

for (int i =0; i< wormholeNoStars; i++){
float r=radius[i];
float a=angle[i]+speed; // increment angle
angle[i]=a;

/* Calculation of new point positions taken entirely from Guglielmetti's sketch. */
float x = r*sin(a);
float y= r*eratio*cos(a);
float b=r*etwist;
float s=sin(b);
float c=cos(b);
xx = cx+s*x+c*y;
yy = cy+c*x-s*y;
/* Calculation of new point positions taken entirely from Guglielmetti's sketch. */

if(r > hole) {
if(!beat) {
if(r>255) {
stroke(colour,255);
}
else {
stroke(colour,255-r);
}
}
}
// else beat=true, change ratio,twist and hole sizes + increase brightness
else {
hole+=0.0001;
eratio = 1.2 * game.minimObj.tempFactor + 0.1;
etwist = 0.1 * game.minimObj.tempFactor + 0.005;
stroke(colour+0.1,255);
}

// draw point and extend in the z axis by an amount proportional to the radius value.
point(xx,yy,-r/2);
}
}
popMatrix();
beat=false;

```

```
}
```

/* drawRadarPoint - Method called each iteration in MainGame when in-game and not adjusting settings. Used to plot a point in relation to player to illustrate where the asteroid is in comparison. Due to the dot product between the player direction and player2asteroid vector returning between 0-180 (in front/behind), to adjust a point around 360 degrees in the correct manner, need to know the direction the player is heading in etc.

```
        The radar variables represent this. */
void drawRadarPoint() {
    //If moving right
    if(goRight) {
        // if increasing & at 180
        if(prevTheta > theta && rotateBy >= radians(180)) {
            radar1 = true; radar2=false;radar3=false;radar4=false;
            rotateBy = radians(180);
            rotateBy = rotateBy +(rotateBy-theta);
            rotateZ(-rotateBy);
        }
        // if decreasing & at 180
        else if(prevTheta < theta && rotateBy >= radians(180)){
            radar2 = true; radar1=false;radar3=false;radar4=false;
            rotateBy = radians(180);
            rotateBy = rotateBy +(rotateBy-theta);
            rotateZ(rotateBy);
        }
        // if decreasing & at 0
        else if(prevTheta < theta && rotateBy <= radians(0)) {
            radar3 = true; radar2=false;radar1=false;radar4=false;
            rotateBy = radians(360);
            rotateBy = rotateBy -theta;
            rotateZ(rotateBy);
        }
        // if increasing and at 0
        else if(prevTheta > theta && rotateBy <= radians(0)) {
            radar4 = true; radar2=false;radar3=false;radar1=false;
            rotateBy = radians(360);
            rotateBy = rotateBy -theta;
            rotateZ(-rotateBy);
        }
        // else central, no change
        else {
            if(radar1) {
                rotateZ(-rotateBy);
            }
            else if(radar2) {
                rotateZ(rotateBy);
            }
            else if(radar3) {
                rotateZ(-rotateBy);
            }
            else if(radar4) {
                rotateZ(rotateBy);
            }
        }
    }
    // If moving left
```

```

else {
  // if increasing & at 180
  if(prevTheta > theta && rotateBy >= radians(180)) {
    radar1 = true; radar2=false;radar3=false;radar4=false;
    rotateBy = radians(180);
    rotateBy = rotateBy +(rotateBy-theta);
    rotateZ(rotateBy);
  }
  // if decreasing & at 180
  else if(prevTheta < theta && rotateBy >= radians(180)){
    radar2 = true; radar1=false;radar3=false;radar4=false;
    rotateBy = radians(180);
    rotateBy = rotateBy +(rotateBy-theta);
    rotateZ(-rotateBy);
  }
  // if decreasing & at 0
  else if(prevTheta < theta && rotateBy <= radians(0)) {
    radar3 = true; radar2=false;radar1=false;radar4=false;
    rotateBy = radians(360);
    rotateBy = rotateBy -theta;
    rotateZ(-rotateBy);
  }
  // if increasing & at 0
  else if(prevTheta > theta && rotateBy <= radians(0)) {
    radar4 = true; radar2=false;radar3=false;radar1=false;
    rotateBy = radians(360);
    rotateBy = rotateBy -theta;
    rotateZ(rotateBy);
  }
  else {
    // else central, no change
    if(radar1) {
      rotateZ(rotateBy);
    }
    else if(radar2) {
      rotateZ(-rotateBy);
    }
    else if(radar3) {
      rotateZ(rotateBy);
    }
    else if(radar4) {
      rotateZ(-rotateBy);
    }
  }
}

stroke(0,0,255);
ellipse(0, -(distance/30), 2,2);
prevTheta = theta;
}
}

```