University of Plymouth

PEARL https://pearl.plymouth.ac.uk

The Plymouth Student Scientist - Volume 06 - 2013 The Plymouth Student Scientist - Volume 6, No. 2 - 2013

2013

# An audio analysis framework for XNA developers

Hoult, S.

Hoult, S. (2013) 'An audio analysis framework for XNA developers', The Plymouth Student Scientist, 6(2), p. 310-327.
http://hdl.handle.net/10026.1/14042

The Plymouth Student Scientist
University of Plymouth

*All content in PEARL is protected by copyright law. Author manuscripts are made available in accordance with publisher policies. Please cite only the published version using the details provided on the item record or document. In the absence of an open licence (e.g. Creative Commons), permissions for further reuse of content should be sought from the publisher or author.*

# An audio analysis framework for XNA developers

Stephen Hoult

*Project Advisor: Serafim Rodrigues, School of Computing and Mathematics, Faculty of Science and Technology, Plymouth University, Drake Circus, Plymouth, PL4 8AA*

**Abstract**

To those that are not specialists in the area, audio analysis can seem a daunting subject. This is particularly the case when creating software which draws upon the subject. There are many frameworks available which aid in the development of audio-driven software, yet few that cater to those with a limited knowledge of audio analysis. The aim of this project was to develop an audio analysis framework, specifically for implementation into C# XNA projects, which will enable developers with little to no understanding of audio analysis to develop audio-driven games. The result of this project has seen the completion of a simple, yet fully functional and well documented audio analysis framework; one that that does not require an extensive prior knowledge to use, and that's easily implementable into any XNA project. The implications of this solution lead towards further development, generalisation, and refinement of the final framework; so that XNA developers of the future are provided with a simple alternative to the complex and unforgiving existing audio analysis frameworks.

## Background

This project began by delving into the audio-synchronous aspects of existing popular games, to gain understanding towards the capabilities that a framework would be expected to provide. Next, research into the development of both pre-analysis and real-time analysis techniques lead to the decision towards manipulating XNA's existing classes, to extract visualisation data from currently playing audio; which ultimately became the basis of the final real-time audio analysis framework.

### *Audio Analysis Concept Design*

XNA Visualisation data provides a logarithmic scale of frequencies ranging from 20 Hz to 20 kHz (MSDN). Rose, J (2002) explains that hearing is logarithmic; this means that a shift of only a few Hertz (Hz) at a low frequency can sound the same as a shift of many hundreds of Hz at a high frequency. This equates to the medium frequency band lying at around 1 kHz, as opposed to a linear midpoint of 10 kHz. Thus, when creating my audio analysis class I shall split the Visualisation data array into low, medium and high frequency bands accordingly, using Rose, J (2002)'s work as a reference:

*Low: 20 – 600 Hz -* This includes extreme bass ranging between 20 and 100 Hz, mid-bass ranging between 100 and 300 Hz, and finally low midrange ranging between 300-600Hz. The low frequency band contains an average of the lower and upper bass amplitudes, which will be extremely important for tracking rhythm.

*Mid: 600 Hz – 2.4 kHz* – This includes the frequencies critical for identifying dialog and harmonics which identify one instrument from another. The mid band can be used to monitor human speech samples.

*High: 2.4 kHz – 20 kHz* – This band ranges from low to top end highs. The high band, with emphasis on 2.4 kHz to 9.6 kHz, primarily conveys energy, life, and brightness to the audio.

To create each frequency band I will sample the current amplitude of each frequency included from the visualisation data array, then divide it by the number of frequencies sampled to create a single average amplitude result. This result can then be used to calculate dynamic variables.

Knowing that the visualisation data array is logarithmic I will calculate the number of frequencies to be sampled for each band using the following equation posted by Rm09 (2010):

**10^(arrayPosition * 0.01171875 + 1.30103) where arrayPosition = VisualisationData[0..255]**

This equation uses log 10 to return the Hz value for the specified position. Using this formula I have calculated the range of frequencies that must be sampled from the visualisation data array to create each band:

*Low:* 20 – 600 Hz
10^(**0** * 0.01171875 + 1.30103) = 20.00 Hz
10^(**126** * 0.01171875 + 1.30103) = 599.22 Hz

*Mid:* 600 Hz – 2.4 kHz
10^(**127** * 0.01171875 + 1.30103) = 615.61 Hz
10^(**178** * 0.01171875 + 1.30103) = 2437.62 Hz

*High:* 2.4 kHz – 20 kHz
10^(**179** * 0.01171875 + 1.30103) = 2504.29 Hz
10^(**255** * 0.01171875 + 1.30103) = 19467.54 Hz

Note: Quick math check: 127+52+77 = 256. This is the correct total number of frequencies (0 - 255).

For each frequency band the sum of the amplitudes included is calculated. The sum is then divided by the range to create the average amplitude value. E.g. for the mid frequency band the sum of the amplitudes of frequencies 127 to 178 is calculated, then divided by the range 52.

## Using the code

The following guide details only the recommended setup and use of the Audio Analysis Class (AAC)**.** Developers should feel free to stray from these procedures should they feel comfortable to do so.

### *Setup (Using Visual Studio 2010)*

*Importing the Audio Analysis Class (AAC)*

1. Create a new XNA 4.0 project. (Skip this step if you plan to use an existing XNA 4.0 Project).
2. Right click on your XNA 4.0 project -> Add -> Existing Item… ->
3. Navigate to your copy of the *AudioAnalysisXNAClass.cs*
4. Add it to your XNA 4.0 project.
5. Using the solution explorer double click on the newly added *AudioAnalysisXNAClass.cs* to open it and view its code. On line 8, change the namespace from “FinalYearProject” to your projects namespace.
6. The AAC is now successfully added to your project!

*Initialising the Audio Analysis Class (AAC)*

1. If you made a new XNA 4.0 project, open the Game1.cs using the solution explorer by double clicking on it. If you’re using an existing XNA 4.0 project, open the equivalent class. (This class will contain XNA’s game loop: Initialize(), LoadContent(), UnloadContent(), Update(GameTime gameTime), and Draw(GameTime gameTime).
2. Instantiate AudioAnalysisXNAClass at the top of this class, see figure 1, line 14, for details.

```
CollisionDetection.cs   Sound.cs   AudioAnalysisXNAClass.cs*   Game1.cs*
FinalYearProject.Game1   audioAnalysis
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using Microsoft.Xna.Framework;
5  using Microsoft.Xna.Framework.Graphics;
6  using Microsoft.Xna.Framework.Input;
7  using Microsoft.Xna.Framework.Media;
8
9
10 namespace FinalYearProject
11 {
12     public class Game1 : Microsoft.Xna.Framework.Game
13     {
14         AudioAnalysisXNAClass audioAnalysis;
15
16         GraphicsDeviceManager graphics;
17         SpriteBatch spriteBatch;
```

**Figure 1**: Instantiate the AudioAnalysisXNAClass

3. Inside the Initialize() method, initialise the AudioAnalysisXNAClass object, see figure 2, line 114, for details. (This will call the AudioAnalysisXNAClass class's constructor, enabling visualisation data).

```
CollisionDetection.cs   Sound.cs   AudioAnalysisXNAClass.cs*   Game1.cs*
FinalYearProject.Game1   Initialize()
109         /// and initialize them as well.
110         /// </summary>
111         protected override void Initialize()
112         {
113             //Initalise the AudioAnalysisXNAClass object.
114             audioAnalysis = new AudioAnalysisXNAClass();
115
116             player1 = new Player(); // Initialize the player class :)
117             myInput = new Input(); // Initialize the input class
118             mySound = new Sound(); //Initialize the sound class
```

**Figure 2:** Initialise the AudioAnalysisXNAClass

4. The AAC is now successfully initialised into your project.

*Updating the Audio Analysis Class (AAC)*

It's recommended that the AAC's visualisation data array and frequency bands are updated only once per game loop. This way, the visualisation data will stay sufficiently synchronised with the music. While calling the AAC's update functions more than once per frame will, e.g. directly before each AAC "get" function call, will provide greater synchronisation. It will also impact negatively upon performance. Therefore, caution is urged when calling the update functions more than once per game loop.

1. Inside Update(GameTime gameTime), call the AAC's update methods in the following order:
    a. audioAnalysis.Update(); - This must be called first. This method updates the visualisation data array; extracting the amplitude for 256

frequencies, from audio currently being played by XNA's MediaPlayer class. If no audio is currently playing the amplitude for each frequency will be set to 0.

b. audioAnalysis.updateAverageLowFrequencyData(); - This method updates the average amplitude for the low frequency band, using the updated version of the visualisation data array.
c. audioAnalysis.updateAverageLowFrequencyData();- This method updates the average amplitude for the mid frequency band, using the updated version of the visualisation data array.
d. audioAnalysis.updateAverageLowFrequencyData();- This method updates the average amplitude for the high frequency band, using the updated version of the visualisation data array.

2. See figure 3 for an example of how to correctly update the AAC.

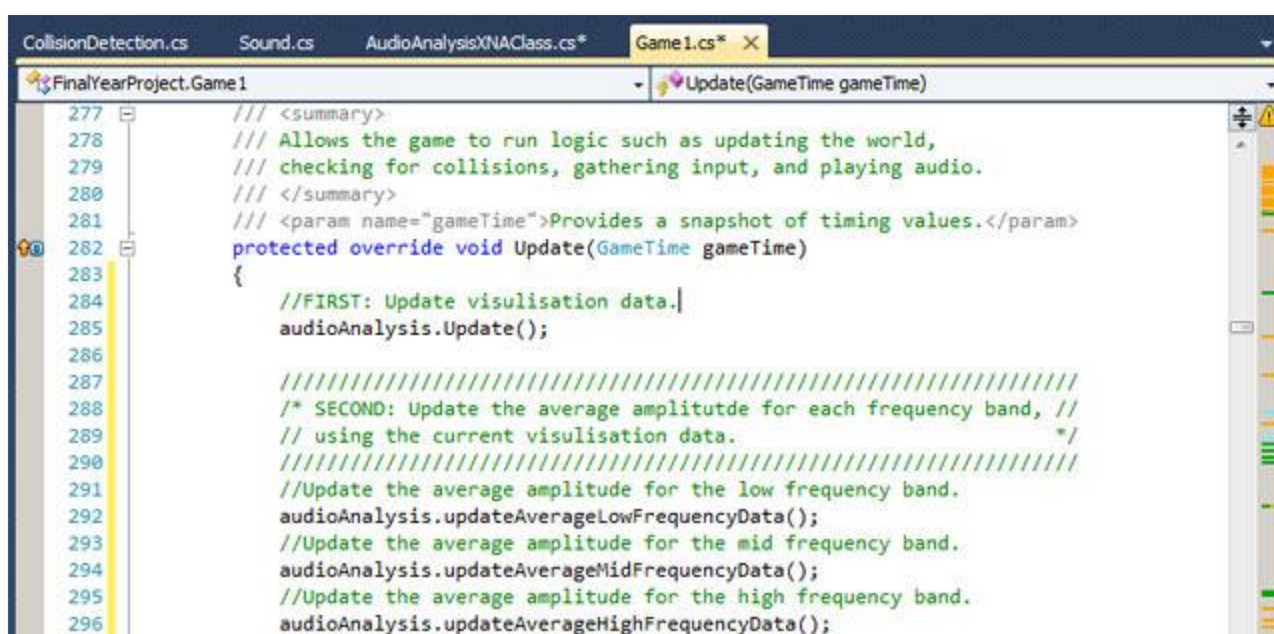


**Figure 3:** How to correctly update the AudioAnalysisXNAClass

3. If you have followed the above steps correctly the AAC is now successfully being updated once per game loop!

*Getters and Setters*

The Audio Analysis Class (AAC) contains a getter and setter method for each of its private variables; lowFrBandAverage, midFrBandAverage, highFrBandAverage.

*Set Methods*

The set methods can be used should you wish to manually override the update functions, and specify your own average amplitude values. See figure 4 for an example of this.

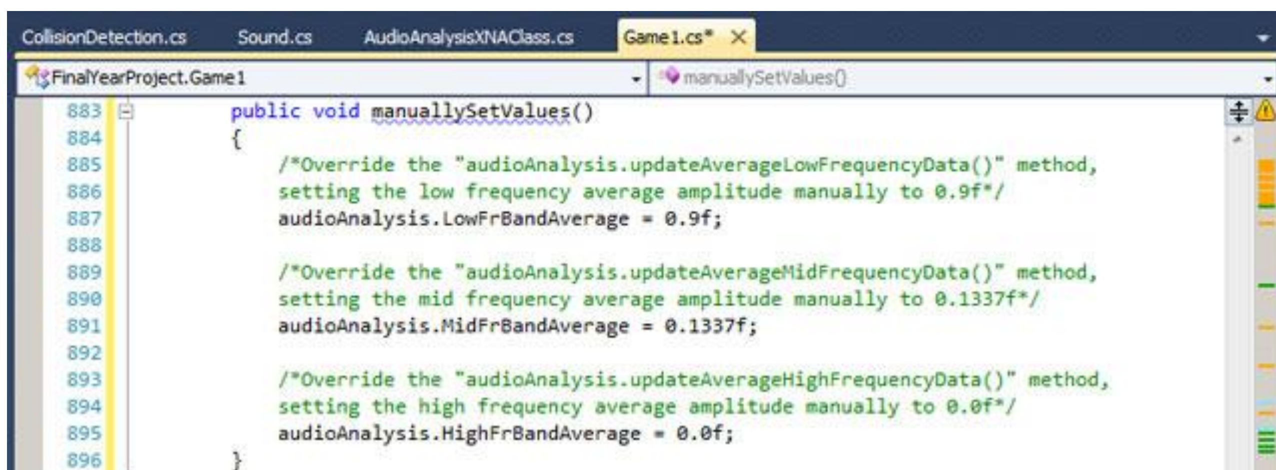


**Figure 4:** How to manually set the average amplitude value for each frequency band.

However, each these manual values will be replaced next time the low, mid and high frequency average update method is called, with the current low, mid and high averages according to the audio.

*Get Methods*

The get methods can be used should you wish to check the value that the variables contain. The following code snippet manually sets the values of the average amplitude variables each game loop, using the code from figure 4, before ‘getting’ and printing the manually set values to the console, see figure 5 for details.

```
protected override void Update(GameTime gameTime)
{
//FIRST: Update visulisation data.
audioAnalysis.Update();

//////////////////////////////////////////////////////////////////////
/* SECOND: Update the average amplitutde for each frequency band, //
// using the current visulisation data.                          */
//////////////////////////////////////////////////////////////////////
//Update the average amplitude for the low frequency band.
audioAnalysis.updateAverageLowFrequencyData();

//Update the average amplitude for the mid frequency band.
audioAnalysis.updateAverageMidFrequencyData();

//Update the average amplitude for the high frequency band.
audioAnalysis.updateAverageHighFrequencyData();

manuallySetValues(); //Manually set all of the average frequency values
getAverageAmplitudes(); //Print all of the average frequency values to the console.
}
```

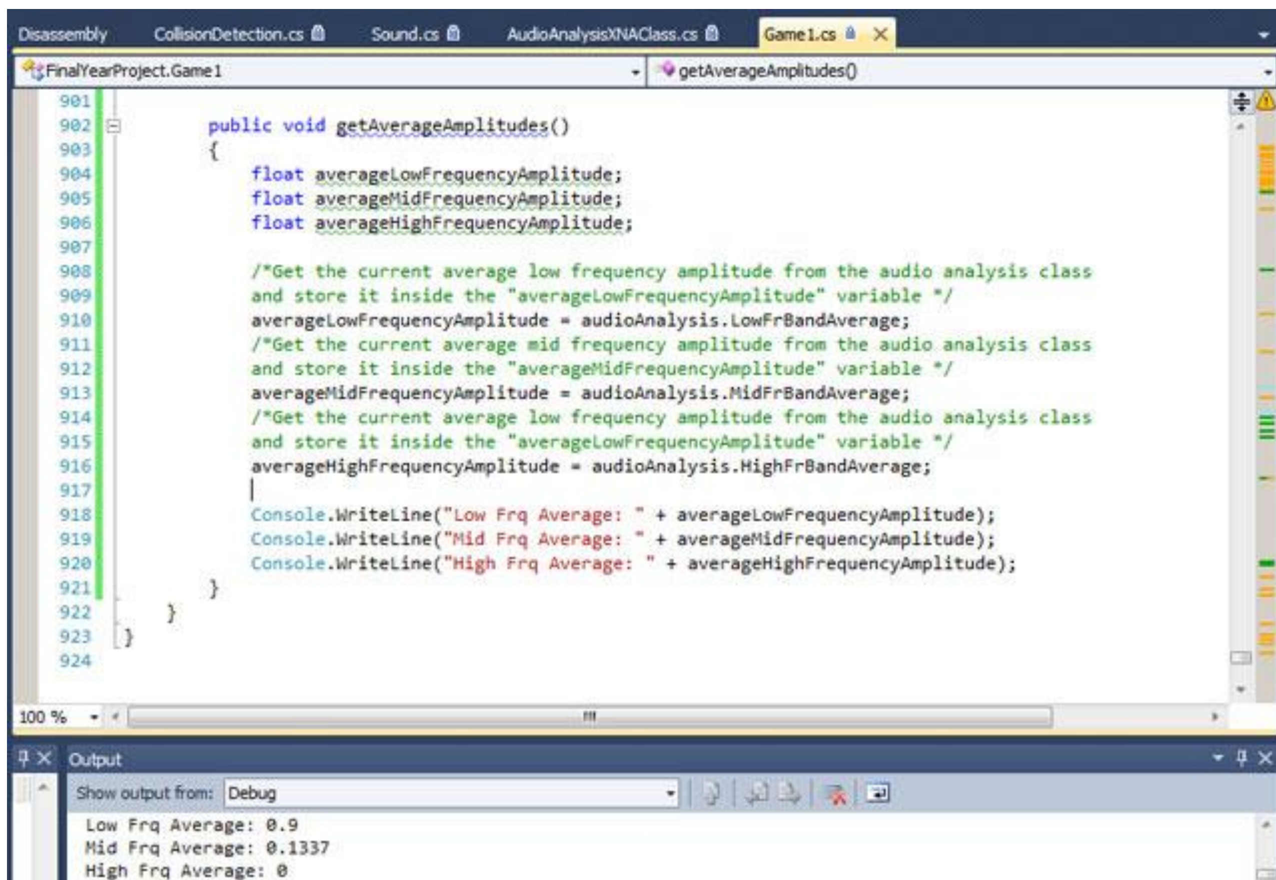


**Figure 5:** Values set in figure 4, returned and printed to the console. (See bottom left.)

Finally the get methods can also be used to create extra methods should they be required. The creation of extra methods is an option I've provided for more experienced users of the audio analysis framework. Of course, they could also directly implement new methods into the audio analysis class itself. The creation of new methods is much down to the creativity and ingenuity of the individual developers. Figure 6 shows a very simple example of the creation of a new method, using the get method to compare the average amplitude of the mid frequency band against a min float value, to return a min or max double value.

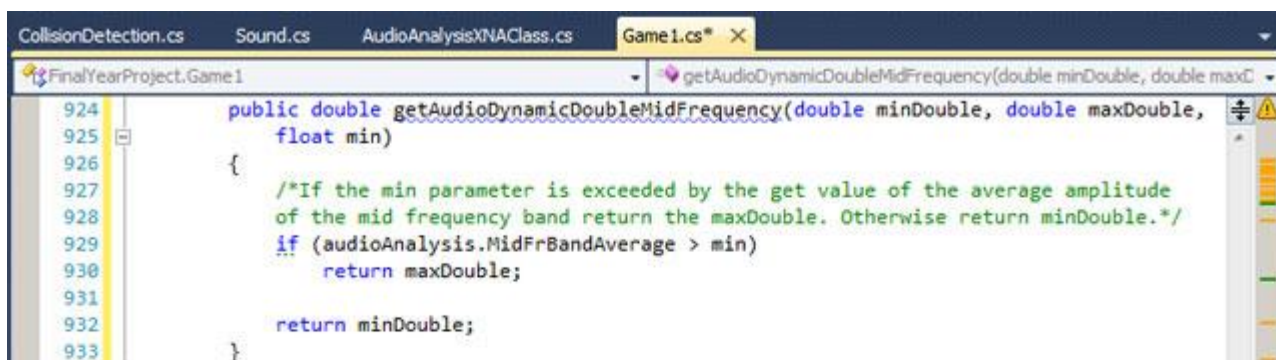


**Figure 6:** Creation of a new method to return a double.

**Overview of the Audio Analysis Classes Functionality**

The following section will provide an example of how one of the methods from each function set could be used within an XNA project. The remaining methods from each function set will not be demonstrated, as they work in exactly the same manor, except that they will calculate the return value using a different frequency band average. The frequency band average used to calculate a return value is specified in the methods name. For example the method “getBool_2StepLowFrq” will calculate the return value using the low frequency band average value, as specified by the ‘LowFrq’ at the end of the methods name. Whilst the “getBool_2StepMidFrq” will calculate the return value using the mid frequency band average value, as specified by the ‘MidFrq’ at the end of the methods name. Finally, the “getBool_2StepHighFrq” will calculate the return value using the high frequency band average value, as specified by the ‘HighFrq’ at the end of the methods name.

It should also be noted that the frequency band specified within the method name will to some degree determine the possible uses of the returned variable. For example using the low frequency band to calculate return values would be best for variables that control rhythm. Whilst using the mid frequency band would be best for variables to be associated with human speech samples within the audio, or perhaps the main synth melody. Finally using the high frequency band would be best for variables that convey energy or brightness. However, in the end, it is down to the individual developer’s creativity and knowledge of these 3 frequency bands which will ultimately dictate how they are used.

Finally the value of each frequency band average will always be between 0.0f and 1.0f. Specifying a value that is outside these boundaries will result in some return values becoming unobtainable.

*Function Set 1 Example: 2Step Return an Audio-dynamic C# Boolean Variable*

This method is used to return a Boolean variable using the low frequency band. If the low frequency band average of an audio clip is greater than 0.49f (an arbitrary value) this method is will set the Boolean variable “shoot” to true. This variable could then be used as a trigger for enemy units to fire their weapons when it is set to true. Otherwise, if the audios average low frequency band is less than 0.49f false will be returned, and enemies will cease their fire.

```
bool shoot = audioAnalysis.getBool_2StepLowFrq(0.49f);
```

*Function Set 2 Example: 2Step Return an Audio-dynamic C# Integer Variable*

This method is used to return a choice of two integer variables using the low frequency band. If the low frequency band average of an audio clip is greater than 0.54f, this method will set the integer variable gameIntensity1Or6 to 6. Otherwise, if this criterion is not met, gameIntensity1Or6 will be set to 1. This variable could be used as a trigger, spawning a different enemy type when the value of gameIntensity1Or6 is set to 6, than when it is set to 1.

```
int gameIntensity1Or6 = audioAnalysis.getInt_2StepLowFrq(1, 6, 0.54f);
```

*Function Set 3 Example: 6Step Return an Audio-dynamic C# Integer Variable*

This method is used to return integer values between 1 and 6 using the low frequency band. If the low frequency band average of an audio clip is less than 0.1f, the minInt value of 1 will be returned. If the frequency band average is greater than 0.1f, but also less than or equal to 0.2f, 2 will be returned. If the frequency band average is greater than 0.2f, but also less than or equal to 0.3f, 3 will be returned. If the frequency band average is greater than 0.3f, but also less than or equal to 0.4f, 4 will be returned. If the frequency band average is greater than 0.4f, but also less than or equal to 0.54f, 5 will be returned. Finally if the frequency band average is greater than 0.54f 6, the maximum value, will be returned.

Each time a value is returned gameIntensity1To6's value will be set to that value. Once again this variable could be used as a trigger, perhaps to spawn different types of enemies.

The only difference from the previous method being that this method has the ability to spawn 6 different types of enemies, instead of just 2.

```
int gameIntensity1To6 = audioAnalysis.getInt_6StepLowFrq(1, 6, 0.1f, 0.2f, 0.3f, 0.4f,
0.54f);
```

*Function Set 4 Example – 2Step Return an Audio-dynamic C# Float Variable*
This method is used to return a choice of two float variables using the low frequency band. If the low frequency band average of an audio clip is greater than 0.45f, this method will set the float variable enemyMoveSpeed to 4.5f. Otherwise, if this criterion is not met, enemyMoveSpeed will be set to 0.9f. This variable could be used to regulate enemy move speed. So that enemies move slowly when the music is less intense, and faster when it comes more so.

```
float enemyMoveSpeed = audioAnalysis.getFloat_2StepLowFrq(0.9f, 4.5f, 0.45f);
```

*Function Set 5 Example – 6Step Return an Audio-dynamic C# Float Variable*
This method is used to return float values between 0.75f and 2.0f using the mid frequency band. If the mid frequency band average of an audio clip is less than 0.2f, 0.75f will be returned. If the frequency band average is greater than 0.2f, but also less than or equal to 0.3f, 1.96f will be returned. If the frequency band average is greater than 0.3f, but also less than or equal to 0.4f, 3.75f will be returned. If the frequency band average is greater than 0.4f, but also less than or equal to 0.5f, 5.55f will be returned. If the frequency band average is greater than 0.5f, but also less than or equal to 0.64f, 7.34f will be returned. Finally if the frequency band average is greater than 0.64f, 9.1337f - the maximum value, will be returned.

The returned float is stored in the enemyAnimation variable, which could for example be used to regulate the scale of enemy animations.

```
Float enemyAnimation = audioAnalysis.getFloat_6StepMidFrq( 0.175f, 9.1337f, 0.2f,
0.3f, 0.4f,
0.5f, 0.64f);
```

*Function Set 6 – 2Step Return an Audio-dynamic C# String Variable*

This method is used to return one of two Strings using the high frequency band. The highFrqMonitor String variable will be set to contain "High frequencies active" if the high frequency average is above 0.0f. Otherwise, it will return "No highs". This variable could be used to monitor the state of the high frequency average, perhaps to display to the user when an audio sample currently contains no high frequencies.

```
String highFrqMonitor = audioAnalysis.getString_2StepHighFrq( "No highs",
        "High frequencies active", 0.0f);
```

*Function Set 7 Example – 6Step Return an Audio-dynamic C# String Variable*
This method is used to return one of six Strings using the mid frequency band. If the mid frequency band average of an audio clip is less than 0.1f, "Very low" will be returned. If the frequency band average is greater than 0.1f, but also less than or equal to 0.2f, "Low" will be returned. If the frequency band average is greater than 0.2f, but also less than or equal to 0.3f, "Low-Medium" will be returned. If the frequency band average is greater than 0.3f, but also less than or equal to 0.4f, "Medium" will be returned. If the frequency band average is greater than 0.4f, but also less than or equal to 0.54f, "High" will be returned. Finally if the frequency band average is greater than 0.54f "Very High" will be returned.

Similar to the previous method, the midFrqMonitor variable could perhaps be used to display to the user the intensity of the mid frequency band.

```
String midFrqMonitor = audioAnalysis.getString_6StepMidFrq( "Very low", "Low",
"Low-Medium",
"Medium", "High", "Very High", 0.1f, 0.2f, 0.3f, 0.4f, 0.54f);
```

*Function Set 8 Example – 3Step Return an Audio-dynamic XNA Vector2 Variable*
This method is used to return one of three Vector2 variables using the low frequency band. If the low frequency band average of an audio clip is less than 0.4f, minParticleTrailVelocity will be returned. If the frequency band average is greater than 0.4f, but also less than or equal to 0.54f, midParticleTrailVelocity will be returned. Finally if the frequency band average is greater than 0.54f maxParticleTrailVelocity will be returned.

The currentParticleTrailVelocity variable could perhaps be used regulate the velocity at which particles travel for a trail effect, using one of the three velocities provided as parameters.

```
//Example particle trail velocities:
Vector2 minParticleTrailVelocity = new Vector2(0.7f, 0.7f);
Vector2 midParticleTrailVelocity = new Vector2(2.0f, 2.0f);
Vector2 maxParticleTrailVelocity = new Vector2(4.0f, 4.0f);

Vector2 currentParticleTrailVelocity = audioAnalysis.getVector2_3StepLowFrq(
minParticleTrailVelocity, midParticleTrailVelocity, maxParticleTrailVelocity, 0.4f,
0.54f);
```

*Function Set 9 Example – 6Step Return an Audio-dynamic XNA Vector2 Variable*
This method is used to return one of six Vector2 variables using the mid frequency band. If the mid frequency band average of an audio clip is less than 0.1f, verySmallParticleExplosionVelocity will be returned. If the frequency band average is greater than 0.1f, but also less than or equal to 0.2f, smallParticleExplosionVelocity will be returned. If the frequency band average is greater than 0.2f, but also less than or equal to 0.3f, mediumParticleExplosionVelocity will be returned. If the frequency band average is greater than 0.3f, but also less than or equal to 0.4f, bigParticleExplosionVelocity will be returned. If the frequency band average is greater than 0.4f, but also less than or equal to 0.54f, veryBigParticleExplosionVelocity will be returned. Finally if the frequency band average is greater than 0.54f massiveParticleExplosionVelocity will be returned.

The currentParticleExplosionVelocity variable could be used to regulate the velocity at which particles travel for explosions, using one of the six velocities provided as parameters.

```
// Example particle explosion velocities:
Vector2 verySmallParticleExplosionVelocity = new Vector2(0.5f, 0.5f);
Vector2 smallParticleExplosionVelocity = new Vector2(1.5f, 1.5f);
Vector2 mediumParticleExplosionVelocity = new Vector2(5.0f, 5.0f);
Vector2 bigParticleExplosionVelocity = new Vector2(10.0f, 10.0f);
Vector2 veryBigParticleExplosionVelocity = new Vector2(15.0f, 15.0f);
Vector2 massiveParticleExplosionVelocity = new Vector2(25.0f, 25.0f);

Vector2 currentParticleExplosionVelocity = audioAnalysis.getVector2_6StepMidFrq(
verySmallParticleExplosionVelocity, smallParticleExplosionVelocity,
mediumParticleExplosionVelocity, bigParticleExplosionVelocity,
bigParticleExplosionVelocity, massiveParticleExplosionVelocity, 0.1f, 0.2f, 0.3f, 0.4f,
0.54f);
```

*Function Set 10 Example – 3Step Return an Audio-dynamic XNA TimeSpan Variable*
This method is used to return one of three TimeSpan variables using the low frequency band. If the low frequency band average of an audio clip is less than 0.4f, the maxFireRateInterval will be converted into a TimeSpan using the TimeSpan.FromSeconds(maxFireRateInterval), then returned. If the frequency band average is greater than 0.4f, but also less than or equal to 0.54f, the midFireRateInterval will be converted into a TimeSpan using the TimeSpan.FromSeconds(midFireRateInterval), then returned. Finally if the frequency band average is greater than 0.54f the minFireRateInterval will be converted into a TimeSpan using the TimeSpan.FromSeconds(minFireRateInterval), then returned.

The timeSpanBetweenShots variable could perhaps be used regulate time between firing a projectile. For Example, when the low frequency band average is below 0.4f projectiles will almost stop being fired at all, according to the maxFireRateInterval float that was used to setup the returned TimeSpan. Whilst when the low frequency band average is greater than 0.54f projectiles will be fired extremely quickly, according to the minFireRateInterval float that was used to setup the returned TimeSpan. While when the low frequency band average is between 0.1f and 0.7f

projectiles will be fired at one of the other TimeSpan rates depending on the exact current average.

```
//Example fire rate intervals, as floats:
float minFireRateInterval = 0.05f;
float midFireRateInterval = 0.15f;
float maxFireRateInterval = 10000f; //This basically turns off shooting.

TimeSpan timeSpanBetweenShots = audioAnalysis.getTimeSpan_3StepLowFrq(
minFireRateInterval, midFireRateInterval, maxFireRateInterval, 0.4f, 0.54f);
```

*Function Set 11 Example – 6Step Return an Audio-dynamic XNA TimeSpan Variable*
This method is used to return one of six TimeSpan variables using the low frequency band. If the low frequency band average of an audio clip is less than 0.1f, the slowestInterval will be converted into a TimeSpan using the TimeSpan.FromSeconds(slowestInterval), then returned. If the frequency band average is greater than 0.1f, but also less than or equal to 0.2f, the slowInterval will be converted into a TimeSpan using the TimeSpan.FromSeconds(slowInterval), then returned. If the frequency band average is greater than 0.2f, but also less than or equal to 0.3f, the mediumInterval will be converted into a TimeSpan using the TimeSpan.FromSeconds(mediumInterval), then returned. If the frequency band average is greater than 0.3f, but also less than or equal to 0.4f, the fastInterval will be converted into a TimeSpan using the TimeSpan.FromSeconds(fastInterval), then returned. If the frequency band average is greater than 0.4f, but also less than or equal to 0.7f, the veryFastInterval will be converted into a TimeSpan using the TimeSpan.FromSeconds(veryFastInterval), then returned. Finally if the frequency band average is greater than 0.7f the extremelyFastInterval will be converted into a TimeSpan using the TimeSpan.FromSeconds(extremelyFastInterval), then returned.

The timeSpanBetweenSpawning variable could perhaps be used regulate time between the spawning of enemies. For Example, when the low frequency band average is below 0.1f enemies will almost stop spawning at all, according to the slowestInterval float that was used to setup the returned TimeSpan. Whilst when the low frequency band average is greater than 0.7f enemies will spawn extremely quickly, according to the extremelyFastInterval float that was used to setup the returned TimeSpan. While when the low frequency band average is between 0.1f and 0.7f enemies will spawn at one of the other TimeSpan rates depending on the exact current average.

```
float extremelyFastInterval = 0.05f;
float veryFastInterval = 0.2f;
float fastInterval = 0.3f;
float mediumInterval = 0.5f;
float slowInterval = 0.9f;
float slowestInterval = 10000f; //This basically stops spawning.

TimeSpan timeSpanBetweenSpawning =
audioAnalysis.getTimeSpan_6StepLowFrq(
extremelyFastInterval, veryFastInterval, fastInterval, mediumInterval,
```

```
slowInterval, slowestInterval, 0.1f, 0.2f, 0.3f, 0.4f, 0.7f);
```

*Function Set 12 Example – 3Step Return an Audio-dynamic XNA Color Variable*
This method is used to return one of three Color variables using the low frequency band. If the low frequency band average of an audio clip is less than 0.4f, colour_LowResponce will be returned. If the frequency band average is greater than 0.4f, but also less than or equal to 0.54f, colour_MediumResponce will be returned. Finally if the frequency band average is greater than 0.54f colour_HighResponce will be returned.

The currentColour variable could perhaps be used regulate the colour of particle trails or explosions.

```
Color colour_LowResponce = new Color(1.0f, 0.0f, 0.0f); // Red
Color colour_MediumResponce = new Color(0.0f, 1.0f, 0.0f); // Green
Color colour_HighResponce = new Color(0.0f, 0.0f, 1.0f); // Blue
Color currentColour = audioAnalysis.getColour_3StepLowFrq(
colour_LowResponce,
    colour_MediumResponce, colour_HighResponce, 0.4f, 0.54f);
```

*Function Set 13 Example – 6Step Return an Audio-dynamic XNA Color Variable*
This method is used to return one of six Color variables using the high frequency band. If the high frequency band average of an audio clip is less than 0.2f, colour_MinResponce will be returned. If the frequency band average is greater than 0.2f, but also less than or equal to 0.25f, colour_LowMediumResponce will be returned. If the frequency band average is greater than 0.25f, but also less than or equal to 0.3f, colour_MediumResponce will be returned. If the frequency band average is greater than 0.3f, but also less than or equal to 0.35f, colour_HighMediumResponce will be returned. If the frequency band average is greater than 0.35f, but also less than or equal to 0.7f, colour_HighResponce will be returned. Finally if the frequency band average is greater than 0.7f colour_MaxResponce will be returned.

The currentColour variable could be used to regulate the background colour of the project. So when the high frequency average is a very large value, the background colour will change to a light grey. Whilst when it is a low value, it will remain black or dark grey. This will create background colour strobe effect – flashing the background colour a light grey on high frequency spikes (such as hi-hat hits).

```
Color colour_MinResponce = new Color(0, 0, 0); // Black
Color colour_LowMediumResponce = new Color(3, 3, 3); // Very very dark grey
Color colour_MediumResponce = new Color(6, 6, 6); // Very dark grey
Color colour_HighMediumResponce = new Color(9, 9, 9); // dark grey
Color colour_HighResponce = new Color(12, 12, 12); // grey
Color colour_MaxResponce = new Color(40, 40, 40); // light grey

Color currentColour = audioAnalysis.getColour_6StepHighFrq(
colour_MinResponce,
```

```
colour_LowMediumResponce, colour_MediumResponce,
colour_HighMediumResponce,
colour_HighResponce, colour_MaxResponce,
0.2f, 0.25f, 0.3f, 0.35f, 0.7f);
```

*Function Set 14 Example – 3Step Return an Audio-dynamic XNA Color[] Array*
This method is used to return one of three Color arrays, Color[], using the low frequency band. If the low frequency band average of an audio clip is less than 0.4f, audioVisColourArray1 will be returned. If the frequency band average is greater than 0.4f, but also less than or equal to 0.54f, audioVisColourArray2 will be returned. Finally if the frequency band average is greater than 0.54f audioVisColourArray3 will be returned.

The currentColourArray could be used regulate the colours of multiple items that work in synchronization / in colour combinations – and provides a more efficient means of returning colour combinations that simply calling the getColour method multiple times.

```
//initalise colour arrays for audio analysis use:
audioVisColourArray1 = new Color[2]; //array of 2 colours
audioVisColourArray2 = new Color[2]; //array of 2 colours
audioVisColourArray3 = new Color[2]; //array of 2 colours

//Colours for a low responce
audioVisColourArray1[0] = new Color(0, 153, 153);
audioVisColourArray1[1] = new Color(153, 153, 0);
//Colours for a medium responce
audioVisColourArray2[0] = new Color(152, 237, 0);
audioVisColourArray2[1] = new Color(0, 237, 152);
//Colours for a high responce
audioVisColourArray3[0] = new Color(208, 0, 110);
audioVisColourArray3[1] = new Color(110, 0, 208);

//A new array large enough to hold largest colour combination returned by the AAC
(2).
Color[] currentColourArray = new Color[2];

//Get the currentColourArray from using the AAC.
Color[] currentColourArray = audioAnalysis.getColourArray_3StepLowFrq(
audioVisColourArray1, audioVisColourArray2, audioVisColourArray3, 0.4f, 0.54f);
```

**Audio-Driven Prototype Game**
Alongside this framework a 2D 'audio-driven' game was created, see figures 7, 8 and 9. This game implements the audio analysis framework demonstrating some of its possible capabilities. This prototype game has been uploaded to git hub as an XNA executable, along with the full source code.

To download an exe of the game click here!
https://github.com/SteveTKD64/DissertationAudioAnalysisProject/archive/master.zip
To download the game source code click here!
https://github.com/SteveTKD64/AACVSSource/archive/master.zip

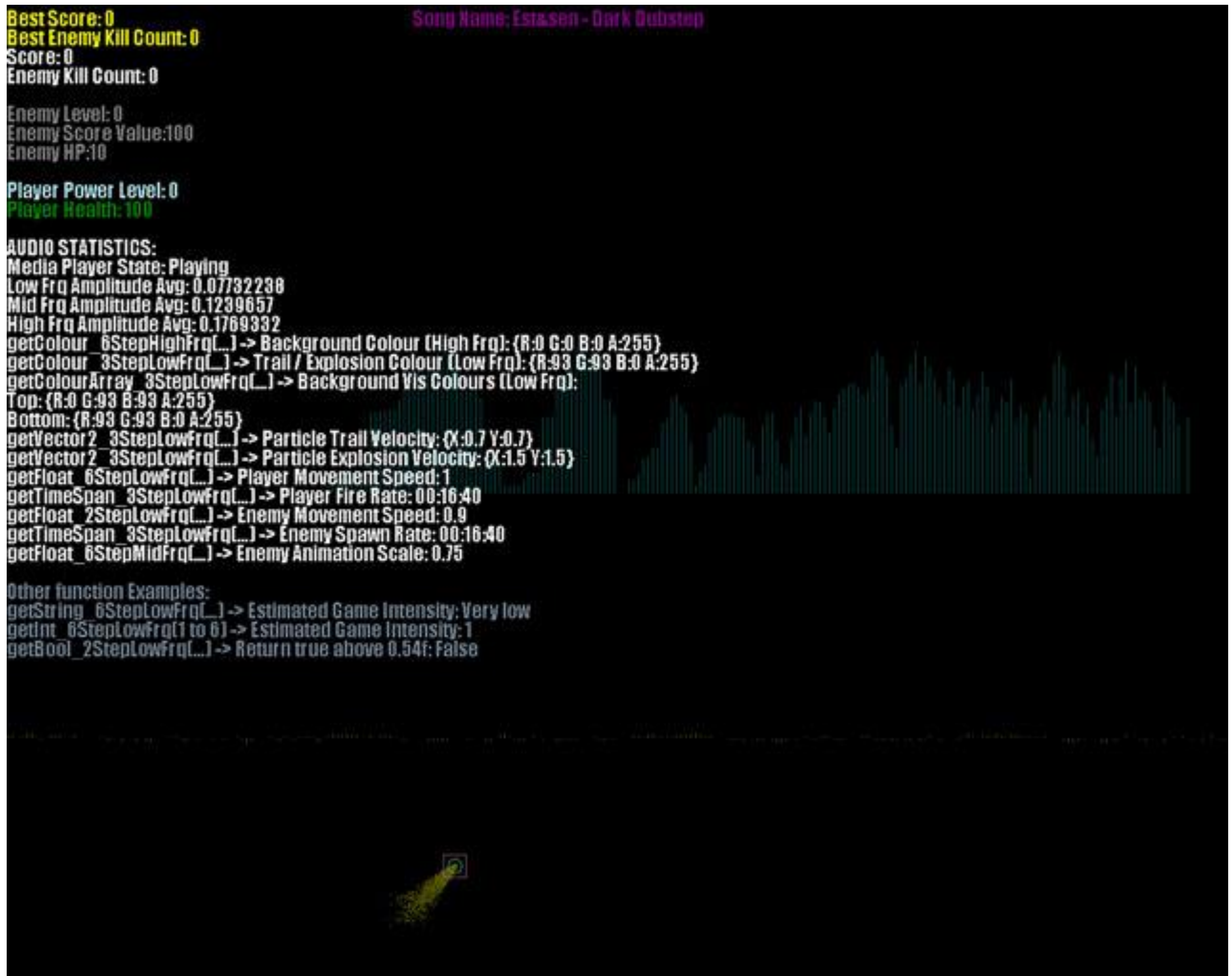


**Figure 7**: "Very low" gameplay intensity - low frequency band average amplitude of 0.077

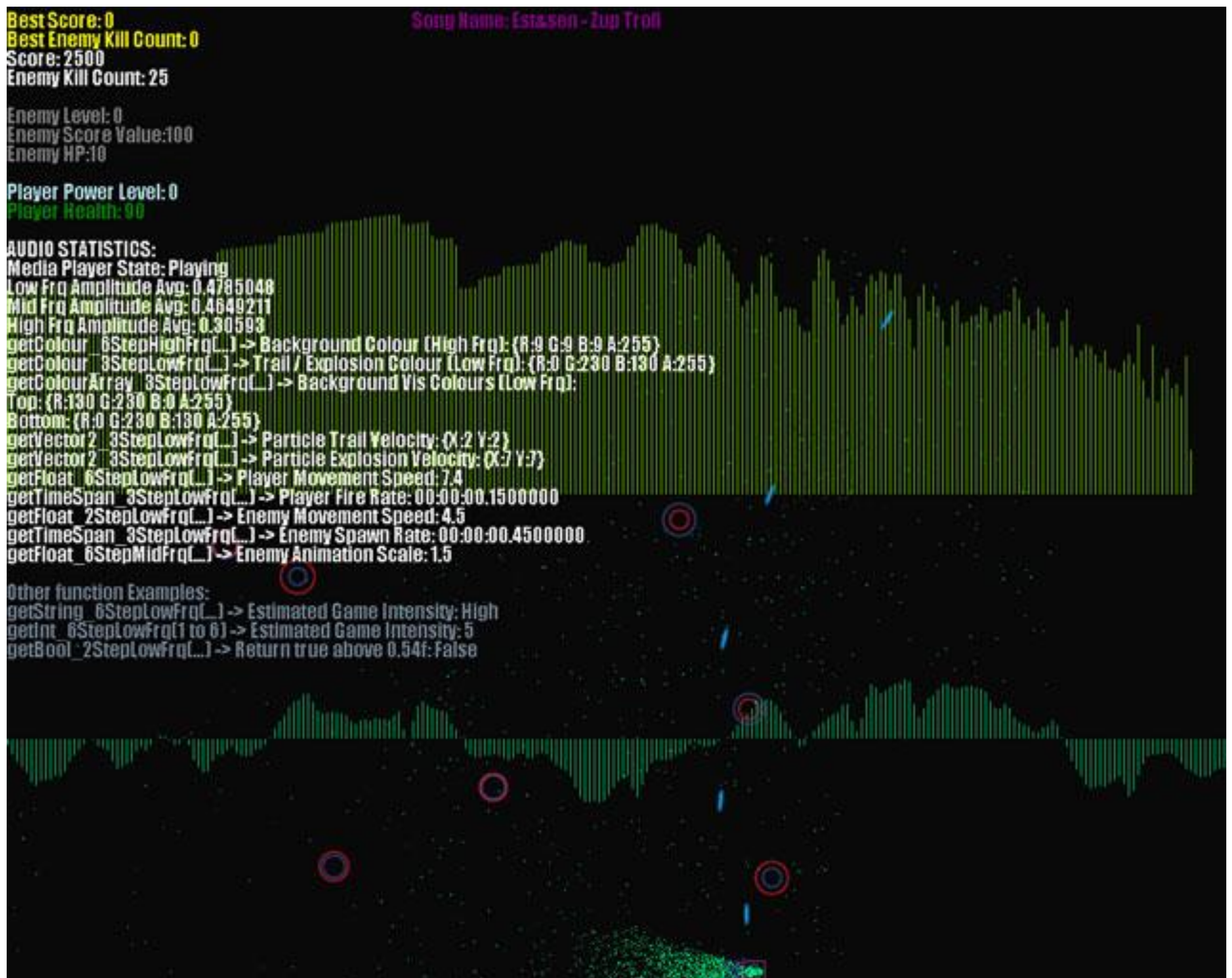


**Figure 8:** "High" gameplay intensity - low frequency band average amplitude of 0.478

**Figure 9**: "Insane" gameplay intensity - low frequency average amplitude of 0.542

## Future Considerations

Knowing what I now know, there are some major changes that I would consider making to the audio analysis class. When creating the audio analysis class, I considered carefully the effects of functionality vs. performance. Since this framework is designed for the use of developers, who may be building applications already significantly impact upon performance, it's important that my own methods are of the lowest impact possible. Considering this, I decided to trade a small amount of functionality and maintainability, to produce two sets of functions for each return data type. This way, should developers only wish for a simple 'on or off' type return of two specified values, inflicting minimum impact to their program, the 2Step functions provide optimal performance. On the other hand, the 6Step functions provide some extra functionality, at the expense of slightly higher impact, but providing a smoother curve over which values are returned, and allowing the developer to specify six different values for return.

If I am to continue the audio analysis class in the future, I will attempt to analyse the impact that both these methods cause. If the difference is relatively small I will consider scrapping both, in place of a 10Step method. My reasoning behind this is that a 10Step method will likely provide the maximum functionality a developer could

need from a single method, over a smooth return curve. At the same time, as with the 6Step method, it's also possible to use it as a 2, 3, 4, etc. step method, by manipulating the parameters in such a way that makes them out of bounds (out of bounds being outside the normalised amplitude array of 0 to 1). For example a 10Step function can be converted into an XStep function easily. If X is the number of parameters you wish the function to choose from, starting from string1 and going to string10, then you must enter X-1 parameters within bounds the for calculation, starting with the float max parameter, and going to the float min parameter.

## Acknowledgements

Music created by Andreas Estensen (Est&Sen). Please click here to see his facebook page for more!
https://www.facebook.com/estandsen

## References

MSDN - 'MediaPlayer.GetVisualizationData Method':
http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.media.mediaplayer.getvisualizationdata.aspx (Accessed 20/04/2013)

Rm09 (2010) – 'Understanding VisualizationData':
http://xboxforums.create.msdn.com/forums/t/22707.aspx (Accessed 20/04/2013)

Rose, J (2002) – 'Range Finder':
http://www.dplay.com/tutorial/bands/index.html (Accessed: 10/02/2013)