Virtual Karaoke System

Gunnar Steinn Magnússon
Katrín Atladóttir

Kgs. Lyngby 2005

Thesis – 2005-82

1. ABSTRACT

In this thesis we have designed and programmed a computer game. It is a karaoke game, based on the popular PlayStation2 game, SingStar.

Our thesis is a construction project with multiple threads, sound input and output, graphics and animation which made it pretty hard to design. We tried to follow conventions in software engineering, such as design patterns and object orientation to simplify the design.

In the game we use Digital Signal Processing as a basis for comparison of original vocals to a player, singing in a microphone in real time.

Since we wanted the game to be approachable for most people, we programmed it to work on normal home computers, using Windows and with common microphones.

Our result is a fully working game, meeting our objectives. Currently it has only one song, but is expandable to have more songs.

The code for the game and sound analysis and binaries for running the game can be found on the CD that comes with the report.

2. PREFACE

This master thesis has been written for the Computer Systems Engineering (CSE) department at the Technical University of Denmark (DTU), which is a sub department of the Informatics and Mathematical Modelling (IMM) department. The study has been carried out in the period between February 2005 and October 2005 by Gunnar Steinn Magnússon and Katrín Atladóttir.

The work of the thesis has been supervised by professor Lars Kai Hansen who is a part of the Intelligent Signal Processing group in the IMM department.

3. ACKNOWLEDGMENTS

First of all we like to thank Lars Kai Hansen for agreeing on being our supervisor in this project and for the help, support and letting us be as independent as we were.

We would also like to thank the great singer and friend Ylfa for helping us with the vocals and great moral support, Óli for the Matlab help, our families for having faith in us and moral support, Haji Abbas, our friends for help and comments, and Viðar for keeping Katrín happy.

4. CONTENTS

4.1. Table of contents

4.2. List of figures

5. INTRODUCTION

5.1. Introduction

The idea for our theses is based on the PlayStation2[1] game SingStar[2]. It is a karaoke game developed and published by Sony Computer Entertainment[3].

SingStar is a kind of game where you sing with well known tunes and try to sing similar to the original vocals. It recognizes the tone of your voice, so the closer you sing to the original tune, the more points you get.

We think it's a great game to play with your friends for a fun night so we wanted to make it PC compatible and be able to select the songs ourselves. So we thought it would be great to make a similar game where we can be in charge of the song selection and it would be a fun way to learn more about Digital Signal Processing (DSP) and making graphics in OpenGL[4].

Since we don't have much background in DSP we had to do a lot of reading and it made the project a lot harder for us to work on.

We wanted it to be aimed for regular PC users so we just use a rather cheap microphone (one like people use for Skype[5] for example) and a regular home PC.

5.2. Objectives

We wanted to create a fully functional computer game. What we want to be able to do in our game is to add a song we select and then be able to play it and compare our singing voice to the original singing voice and the rate it with points. The points are added to high score which is saved on the disk for later

play. The game is supposed to have a functional menu system.

6. The Game Design

The game is designed to be a full game with a menu, high score and rules. Since designing a large game can be very complex with lot of objects and threads we tried to use simple designs and use design patterns where appropriate. We used Head First Design Patterns[6] as our guide and when we mention a design pattern in this chapter, it can be found in that book.

Since we were two working on this project we had to find a way to prevent conflicts in working with the files. We decided to use CVS[7] (Concurrent Versions System) and the client we used for our Windows machines is TortoiseCVS[8], an open source CVS client.

When using CVS, the latest version of each file is kept in a server along with all changes made to the files in the past, so if a conflict occurred we could always see what had been changed and find the correct version of the file. Another profit of using CVS is that our project was constantly backed up, so we never had to worry about loosing all the work we had done.

We decided to use OpenGL as our graphical environment since we have experience in it.

We used code from NeHe Productions[9] as our base to set up our OpenGL environment and code from the course 02563 Virtual Reality Systems[10], taught by Niels Jørgen Christensen, which provides a few helper classes we used, such as vector calculations.

The overall design of our game loop was inspired by the game loop in Tricks of the Windows Game Programming Gurus[11].

A class diagram of the overall design can be found in Figure 1. Following is a description of each part of the design.

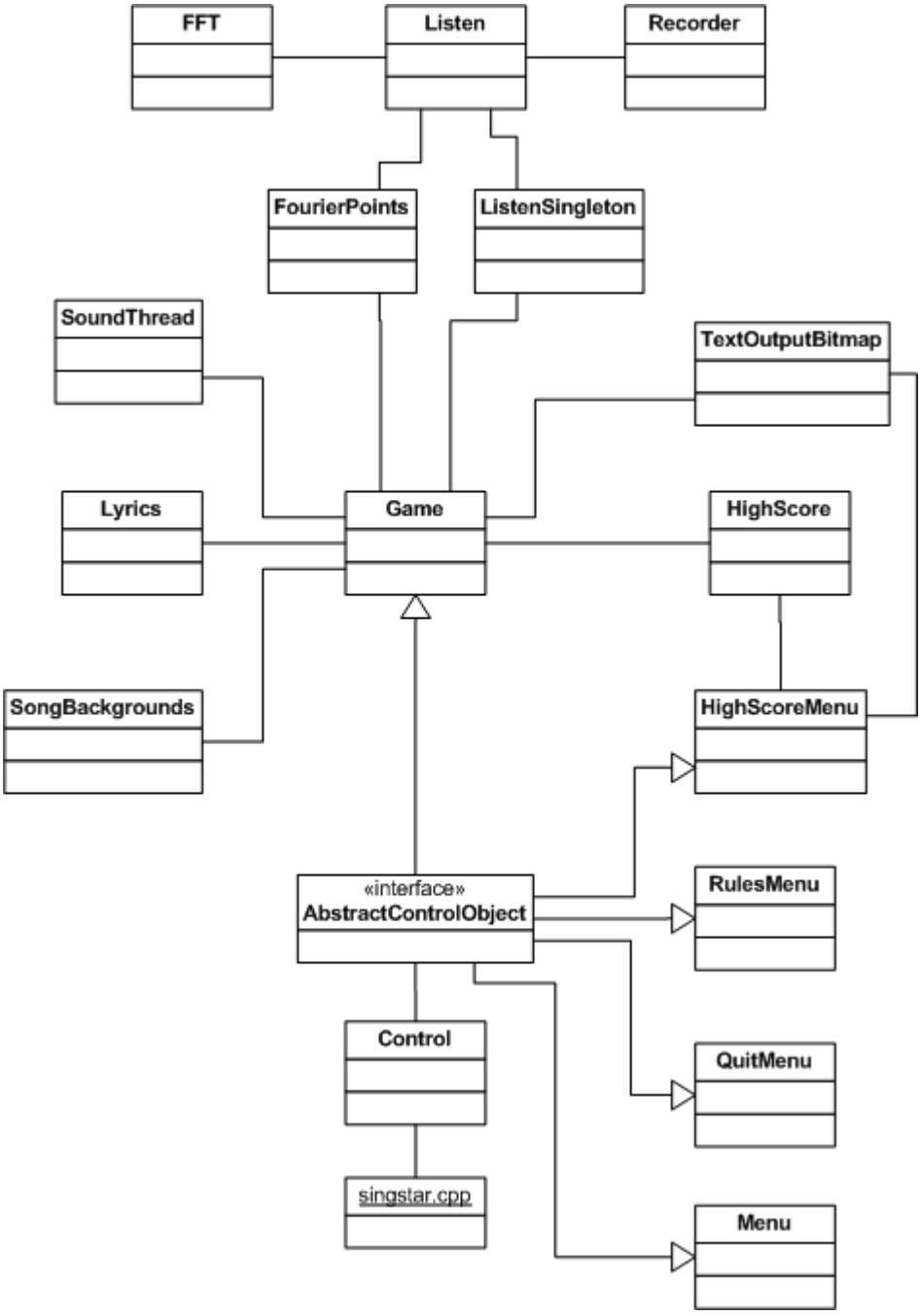

Figure 1: Class diagram of the game

singstar.cpp

Singstar.cpp is the main entry point of the application. In it, OpenGL and Windows are initialized and the main game window is created.

Next, all helper classes are initialized, such as the high score is loaded, text bitmaps are loaded, the microphone listener is started, the thread that controls the song playing is started and finally, Control is initialized.

When Control is initialized, the menu system is in turn loaded, which loads all graphics and sounds associated with it.

The main game loop is next. Following is a pseudo code for the loop:

```
while (not exiting)
      check and handle Windows messages
      deltaTime = calculate time since last frame
      keys = what keys have changed since last frame
      // Tell game that we are drawing another frame
      // so it can move objects, handle keyboard etc.
      control->step(deltaTime, keys)
      // display current game status on screen
      control->display()
end while
```

The Control object makes sure that the right object (some menu or the main game) gets the *step* and *display* calls.

When the program has been notified that it should terminate, it notifies all the same classes that it initialized, that they should release all resources and exit.

For example it stops the song from playing and waits until the thread has finished.

Control class

The Control class is basically an abstraction to simplify how the main game loop notifies the right classes using the variation of the Command Design Pattern.

Every main game part, like high score menu, main menu, the main game etc. inherits from the AbstractControlObject which defines how someone can control class with methods such as *init, start, display* and *step*.

The Control class keeps an instance of all game objects (main menu, game, rules menu, high score menu, quit menu) and keeps track of what part is active and acts as a middle man between the main loop and the active object. The active object can also notify the Control when it wants to switch to another game part (for example when the song finishes).

TextOutputBitmap

The TextOutputBitmap class is a class to display text on the screen using a bitmapped font. The font is created where each character is put on an even grid in the ASCII as seen in Figure 2.

Figure 2: Part of the bitmapped font TextOutputBitmap

parses

TextOutputBitmap loads the font by reading the image and for each character, calculate where on the image it is and load it into its own OpenGL call list.

Other classes can then use it to print to the screen by writing:

```
TextOutputBitmap::glPrint(x,y, “The text”);
```

The printing function also has an option for scaling, centering on the screen and using multiple fonts.

The *glPrint* function uses the *glCallLists* function in OpenGL. *glCallLists* takes as a parameter an array of OpenGL call lists and draws them all consequently. We can use that function, by storing the call lists in ASCII order, call it only with the string we are supposed to display.

HighScore class

The HighScore class keeps track of the high score. Since it is used in many places in the game and all have to use the same instance, we used the Singleton Design Pattern.

With the Singleton Design Pattern no one can create an instance of the class, it has to request an instance from a static method in the class and it is guaranteed

that every time an instance is requested it returns the same instance.

This makes sure that the program does not have inconsistent data.

The class has methods for loading score from disk, write to disk and adding score to the list.

The high score list is sorted with one round of bubble sort[12] each time a number is added.

Menu classes

The menu consists of four classes, Menu, HighScoreMenu, RulesMenu and QuitMenu, and they all basically work in the same way.

They all inherit from AbstractControlObject so Control can use it. All start by loading the graphics needed for the menu. They keep track of which sub menu is chosen, can display the right graphics according to the sub menu and can handle keyboard input and switch to other menus in response to that.

The exception to this is HighScoreMenu which asks the HighScore class for the high score and displays it on the menu using the TextOutputBitmap class which is discussed later in this report.

Game class

The Game class is where all the game activity is controlled. It combines and controls different parts of the game, such as lyrics, microphone input, the song playing and displays them on the screen.

When the game is started it initializes all the appropriate variables and classes, such as score, the graphs, the lyrics etc.

When the game is running it is in charge of letting other parts of the game, update and display.

When a game finishes, all parts of the game are stopped, the score is compared to the high score list using the HighScore class and a back button appears so the user can get back to the main menu.

Some of the more specific things, when the game is running, are discussed in the following chapters.

Progress Bar

For the user to see his progress, a progress bar is displayed on the screen. The progress bar is created by overlaying an image onto the background. Only a part of the image, representing the percentage of song played, is displayed which results in a good looking progress bar.

Drawing the tone scale

For guidance, thin lines indicating the tones are drawn behind the microphone/original-vocal graphs. Frequency can be calculated from tone number with the formula:

$$hertz = 6.875 * 2^{\frac{3+tone}{12}}$$

We then convert the frequencies to pixels on the screen by scaling them down with a constant and display the nearest tone every 30 pixels on the screen. Both a line and the tone name is displayed.

Lyrics

While a song is playing in the game, the lyrics are displayed on the bottom of the screen and a marker is placed where in the line the player is.

We used Audacity to find out in which millisecond of the song each line started and ended.

We added these times and the lyrics to a text file which the game could read.

The Lyrics class is pretty simple. It reads the text file with the lyrics information and keeps it in a *vector* and it allows other classes to retrieve lines based on the current time in the song.

For example can we call:

```
vector<SongLine *> lines = lyr.getLine2(20000, 3);
```

to get the next three lines from the 20th second in the song in a *vector*.

The Game class uses the Lyrics class to display the lyrics on the screen. Each time it draws to the screen, it asks the lyrics class for the next three lines and displays them one after another.

To indicate to the user where in the line the song is currently, a small ball bounces on top of it.

To figure where to make the ball bounce, we calculated the length of the line in milliseconds, the difference between current time and the beginning of the line in milliseconds and by that could calculate the percentage of the line that had finished and placed the ball accordingly.

This method is not perfect, but works good enough. In the few cases the ball was misplaced by too far, we cut the line into two lines.

To add a little life to it, we also made the ball bounce a little by adding a *sinus* factor, determined by the number of milliseconds since last frame, to the height.

Background Images

To make the game more alive and fun, we added background images with simple transitions, known from programs such as Microsoft Powerpoint[13].

We found a few pictures from a Dolly Parton fan site[14] and edited them to fit our game.

The SongBackgrounds class manages them in the game. It has support for multiple images and multiple transition types that can easily be added.

It starts by loading the graphics from disk and into OpenGL textures. Each frame, the current transition advances a little and then displays the result.

When a transition finishes, it selects a new one, and after a predetermined time, starts it.

We currently have two types of transitions, one where a picture slides from one side over the other picture, and one where a picture pushes the existing one out.

The transition speed is determined by the time one frame takes, so it moves on the same speed for different computers.

Drawing microphone/original vocal graphs

To display the graphs representing the players voice and the original voice we created a class, FourierPoints, that allows us to create a buffer of some tones, add tones and then retrieve the last predefined number of tones added.

To accomplish this we use a dynamic array as a circular buffer and keep track of

where that last tone was added. Then indexes for each tone can be found using modulus on the size.

The class is also capable of reading prerecorded tones from a text file, a feature we use for the original voice which is pre-calculated as seen in chapter 8.

The Game class uses the FourierPoints class both for the microphone input and original vocals.

Every 100 ms it adds a new microphone tone and displays the last 20 tones on the screen. It also displays, in another color, the last 20 and next 30 tones from the original vocals for reference for the player.

SoundThread class

The SoundThread is responsible of playing a song. It is able to loop a song (needed for example in the menu), stop a song, get play status and notify any class interested when a song has finished.

Since the SoundThread has to be in its own thread it has to be thread safe to make sure we don't get any deadlocks or memory leaks because of unexpected order of actions.

To solve these problems we made the state chart in Figure 3.

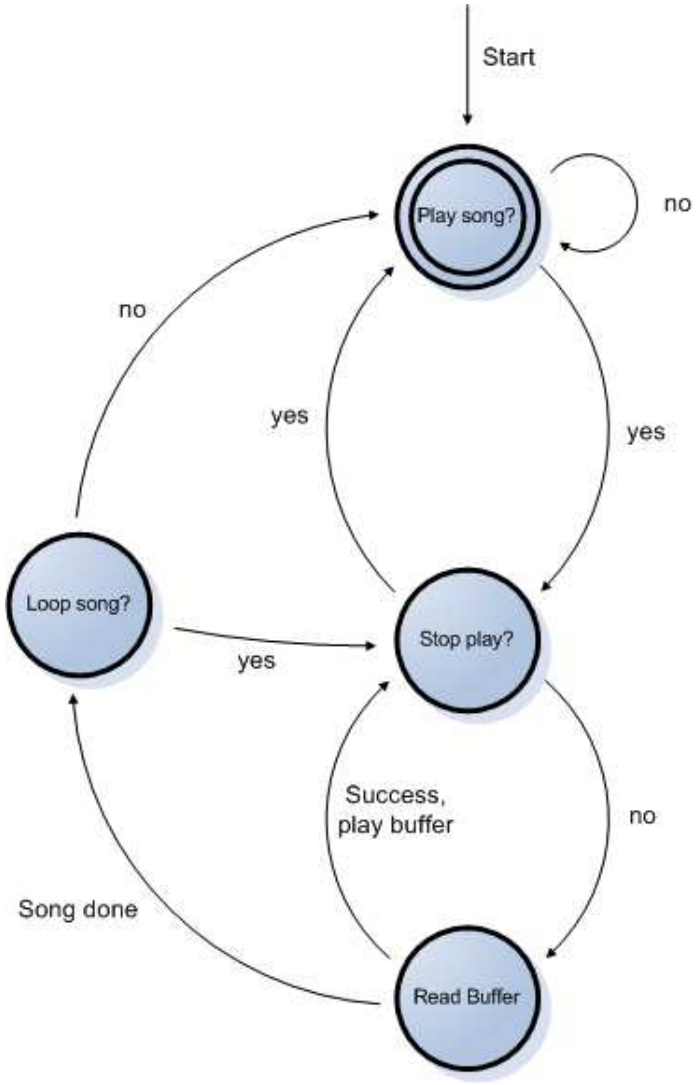

Figure 3: Figure 3: State chart for song playing in SoundThread class. It supports play, stop and looping songs

The state chart shows the main function in SoundThread.

To play or stop a song, other threads can call public functions that first lock status variables using Semaphores, set the new status and then release the Semaphore.

The song thread, then in the states *"Play song"* and *"Stop play"*, locks the same status variables with semaphores, checks what their status is and then releases the Semaphore. This way, other threads can simultaneously control the SoundThread without having a deadlock. The drawback is that the control is not instant, we have to wait until the song thread reads the command (usually only a few tens of milliseconds).

All classes can register to be notified when a song has finished. This for example is convenient for the main game part to know when it should change to the after-game state. These notifications are implemented by using the Observer Design Pattern. All classes that need to register for notification, inherit from the SoundThreadInterface interface in which they have to implement a notify method which is called when a song finishes.

Using *vector* we can have multiple classes registering for notifications although we don't currently use that feature.

The SoundThread class uses the Singleton Design Pattern since we have to use it in a few places and only want to be playing one sound at a time.

7. The Sound Processing

7.1. Introduction

A large part of the project involved manipulation of sounds and all of the sound processing was performed with two programs, Audacity[15] and Matlab[16].

Audacity is a mature multi-platform Open Source sound editor which allowed us easily to perform noise-reduction and other common sound editing.

Matlab is a well known technical computing application that allowed us to perform more specific sound processing like Fourier transforms.

7.2. The Problem

We need to select a song, isolate the vocals, analyze the frequencies in the vocals and write the analysis to a file.

While the program is playing we need to do the same kind of analysis on a microphone-input and compare that to the original vocals.

7.3. Analyzing the song

Fast Fourier transform

We wanted to compare the original vocals of the song to the microphone input based on tone of the two sounds. A tone is a specific frequency, for example middle C is 261.6 Hz.

One way of converting a sound to frequencies is using a Fourier transform. Fourier transform is a way of converting a function to a trigonometric series where each part in the sum represents a specific frequency. That is, we can use

it to convert a sound to a format where we can see the volume behind each frequency.

The original Fourier transform is a continuous function that is not well suited for computer calculations, therefore Discrete Fourier transform (DFT) is used in computer calculations.

DFT is defined by the formula:

$$f_j = \sum_{k=0}^{n-1} x_k e^{-\frac{2\pi i}{n} jk} \qquad j = 0, \ldots, n-1.$$

Calculating DFT directly takes $O(n^2)$ arithmetic operations and is therefore not realistic for real-time computation.

Some variations of the Discrete Fourier transform exist and the one we use is called Fast Fourier transform (FFT). Fast Fourier transform calculates in O(n*log(n)) arithmetic operations and is an effective algorithm to calculate Discrete Fourier transform.

Matlab has the FFT algorithm built in and we used that for our calculation.

Testing the Fast Fourier transform

We created a test program in Matlab that ran the built in FFT function on 100ms windows on the songs and found the most powerful frequency and graphed it. When using this algorithm on the original song, it is obvious that instruments affect the results too much. These results can be seen on Figure 4 which shows how the most powerful frequency changes in a selected time period in the song. Since the tones are not constant enough, when they clearly

are, when listening to the song, we can not use this as a basis for our calculations.

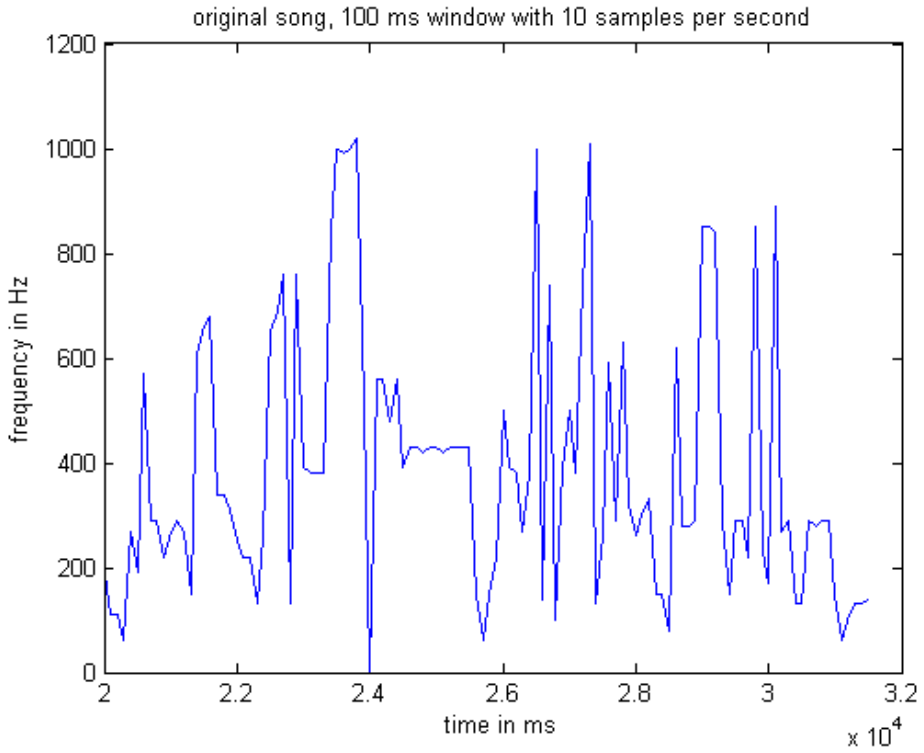

Figure 4: The most powerful frequency, calculated with FFT, on an original track with vocals and instruments, same time period as in Figure 8-12. We used 100ms non-overlapping windows. The graph has too many spikes and rapid changes to use.

The optimal solution would be to get the original vocal track without the instruments and run our algorithm on that, but that was not a possibility for us. Instead we had our friend, who is a good singer, mimic the singer and recorded her voice.

We used Audacity to record the voice, applied built in noise-reduction filter and

synchronized the timing with the original song.

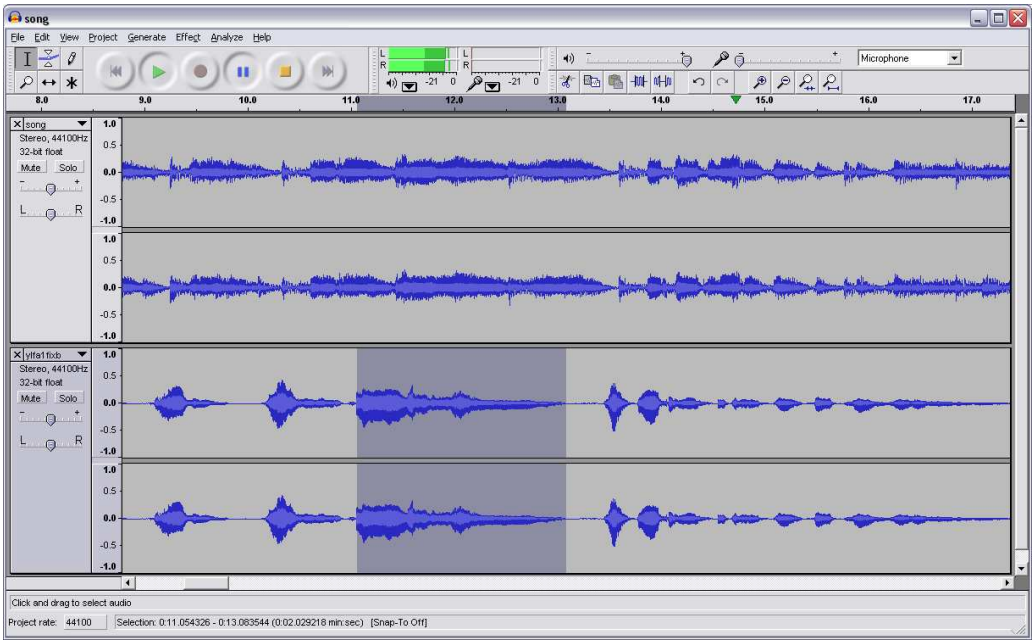

Figure 5 - Vocals were edited in Audacity which is an open source audio editor project.

We got better results running our algorithm on this recording than the original tune as can be seen in Figure 6.

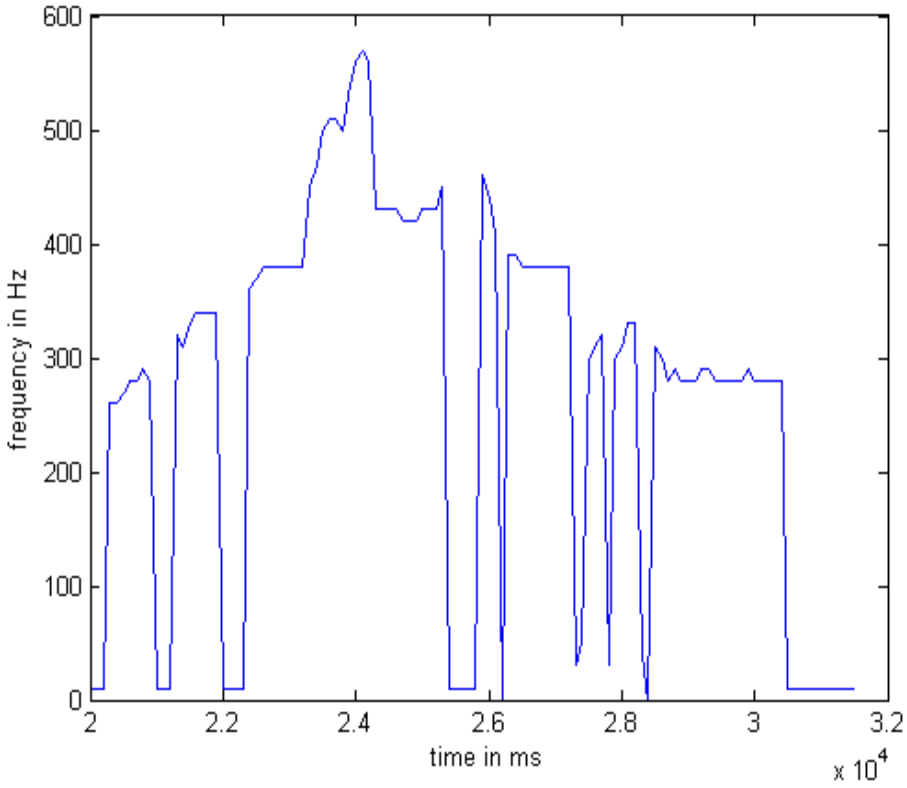

Figure 6: The most powerful frequency, calculated with FFT, on an recorded vocal, same time period as in Figure 8-12. We used 100ms non-overlapping windows. The graph looks much better than figure 4 for using as a basis for our calculations.

Although the graph in figure 6 looks much better than the graph in figure 4, there are some unexpected spikes which we assume are due to a cheap microphone and bad recording facilities. Therefore, to minimize this effect, we had our friend sing the song three times and ran our algorithm on each recording. Then finally, we calculated the average of the results except if one of

the recording had a period with a large spike, we discarded that part.

This resulted in almost no spikes.

Comparing two voices

To convince ourselves that this method of comparing two voices would work, we recorded two voices singing the song and compared the power of each frequency at a few places in the recordings.

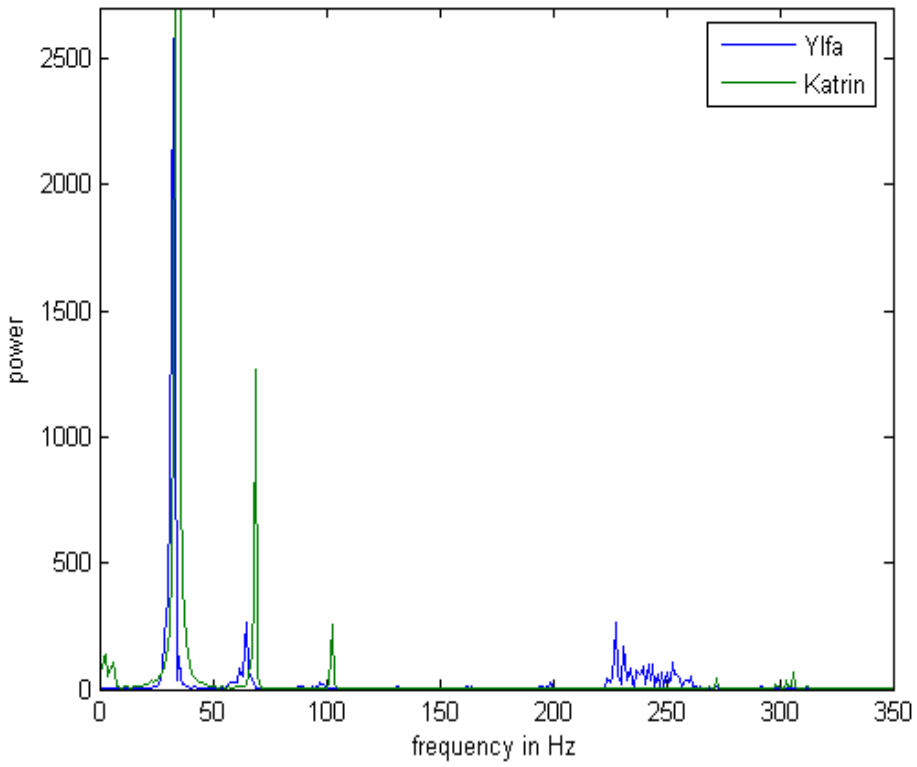

Figure 7: Comparing power of frequencies of two recordings. Both have a maximum at around 35 Hz which means our algorithm works for these two vocals.

Figure 7 shows a comparison of the the power of each frequency of two sound samples from two persons. Although they are not exactly the same, they both have the first 3 spikes in the harmonics in the same positions (small deviation because they are not singing the exact same tone). The first spike is by far the highest one in both recordings. Since we use the highest spike in our calculations these two recordings would have matched.

Dividing the song

To apply Fourier transform on a sound, we need to divide it into smaller parts, or windows, and apply the algorithm on each window.

Choosing how large windows should be and how often per second we should take a sample depends on a few factors:

Large windows could result in inaccurate results if the tone changes faster than the window, small windows on the other hand will not give the transform enough data to work with.

If a sample is taken very often per second, we require higher CPU power while the game runs, also, since we are using Windows timing function which has a resolution of 10 ms we can't take more than 100 samples per second.

Also, if we take a few samples per second, the player sees too few reference points and his results show up so late that it's hard to react to them.

To decide the windows and samples per second we did experiments on various inputs and plotted the results.

Figure 8 – 12 show results from the same time period on the same audio file.

First we tried using many small windows which results in figure 8.

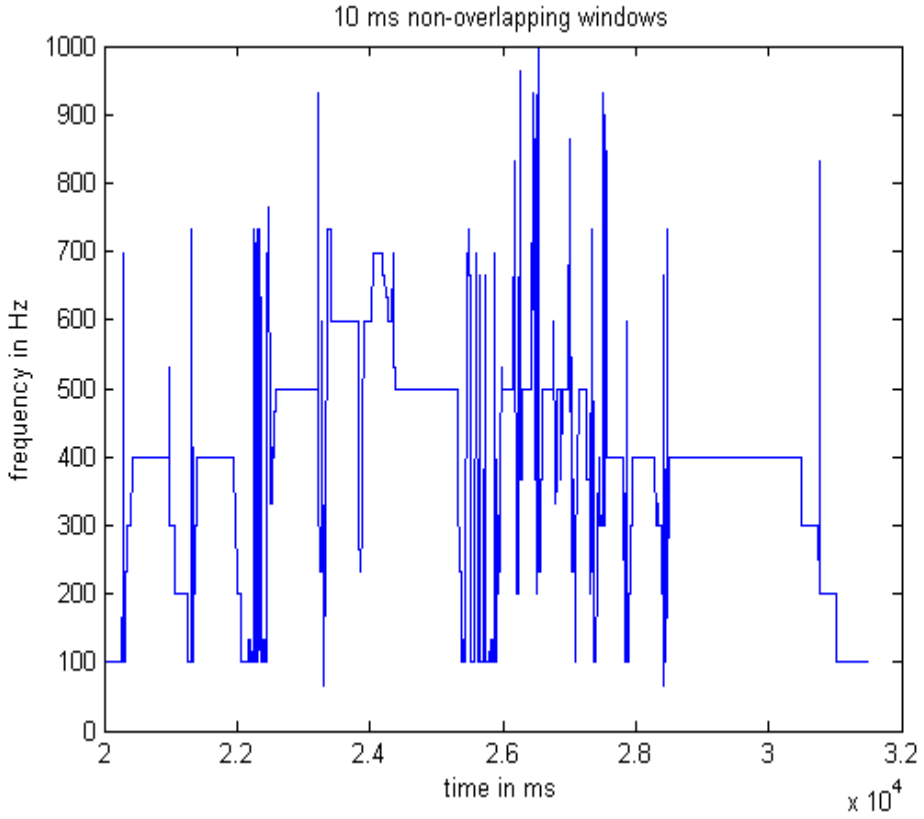

Figure 8: Most powerful frequency, calculated with FFT with 10 ms windows and with 100 samples per second.

The windows are so small that FFT does not have enough data to work with. This can be seen from rapid changes in the most powerful frequency.

We then tried a little larger windows. Figure 9 shows 20 ms non-overlapping windows with 50 samples per second.

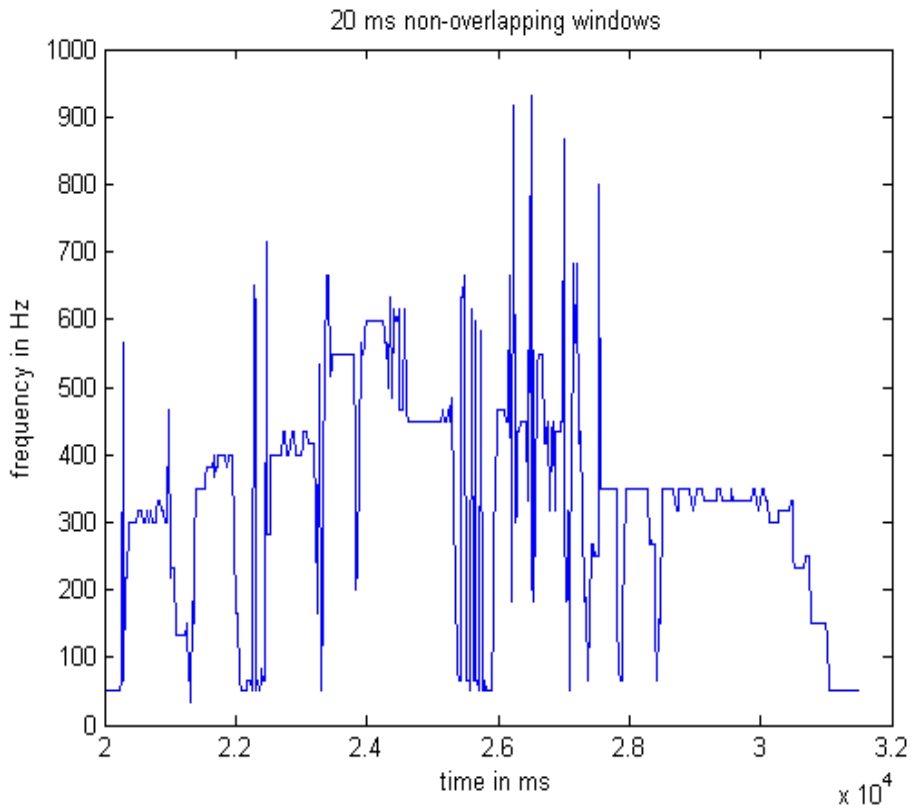

Figure 9: Most powerful frequency, calculated with FFT with 20 ms windows and with 50 samples per second.

The windows are still too small which can be seen by the rapid changes in the graph.

Next we tried to enlarge the windows to 100 ms overlapping with 50 samples

per second.

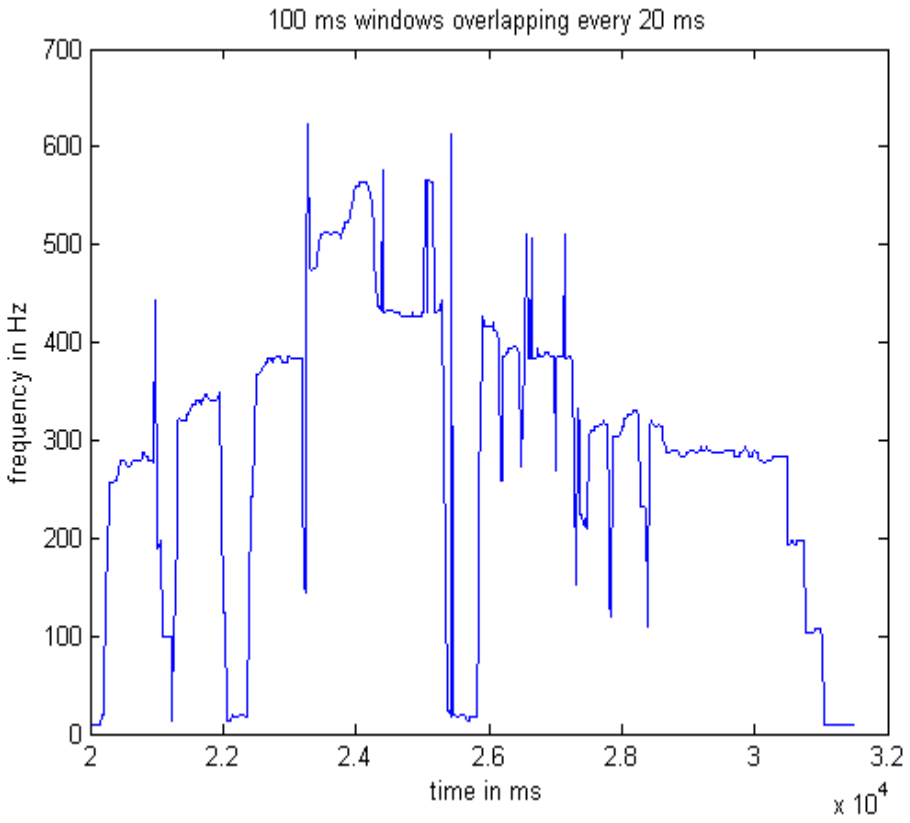

Figure 10: Most powerful frequency, calculated with FFT with 100 ms overlapping windows and with 50 samples per second.

The graph shows much better results than the graphs in figures 8 and 9 but there are still some spikes.

We then tried taking fewer samples. We calculated 20 samples per second.

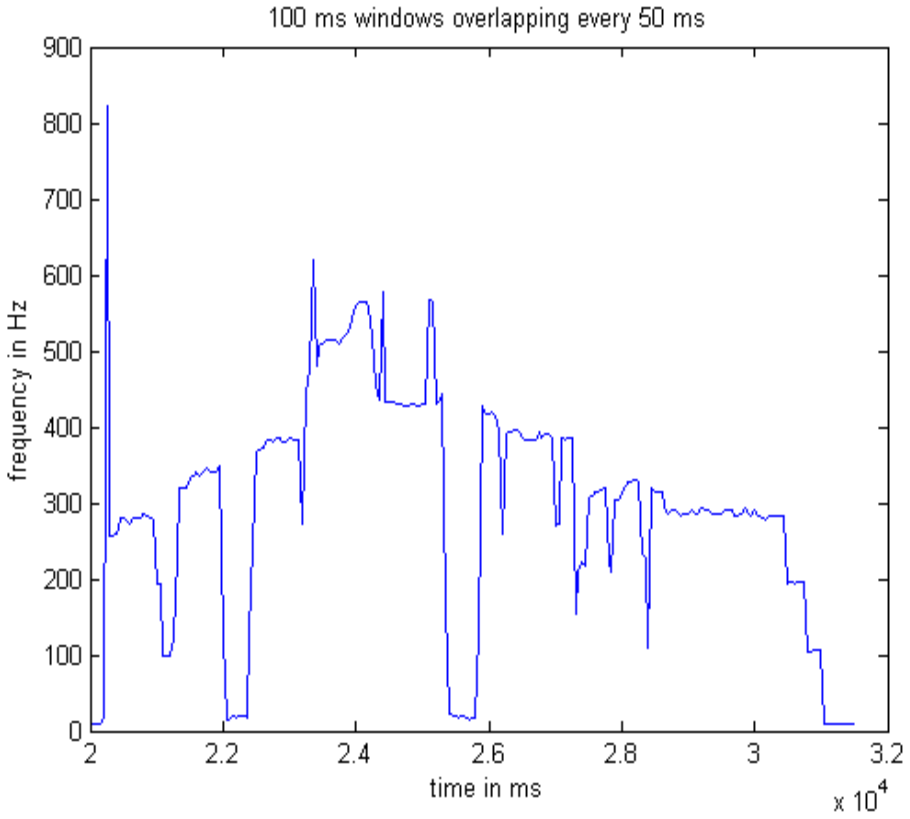

Figure 11: Most powerful frequency, calculated with FFT with 100 ms windows and with 20 samples per second.

The graph is starting to look nice, but still has a few spikes in it we would like to get rid of.

Finally we tested using 100 ms windows with 10 samples per second.

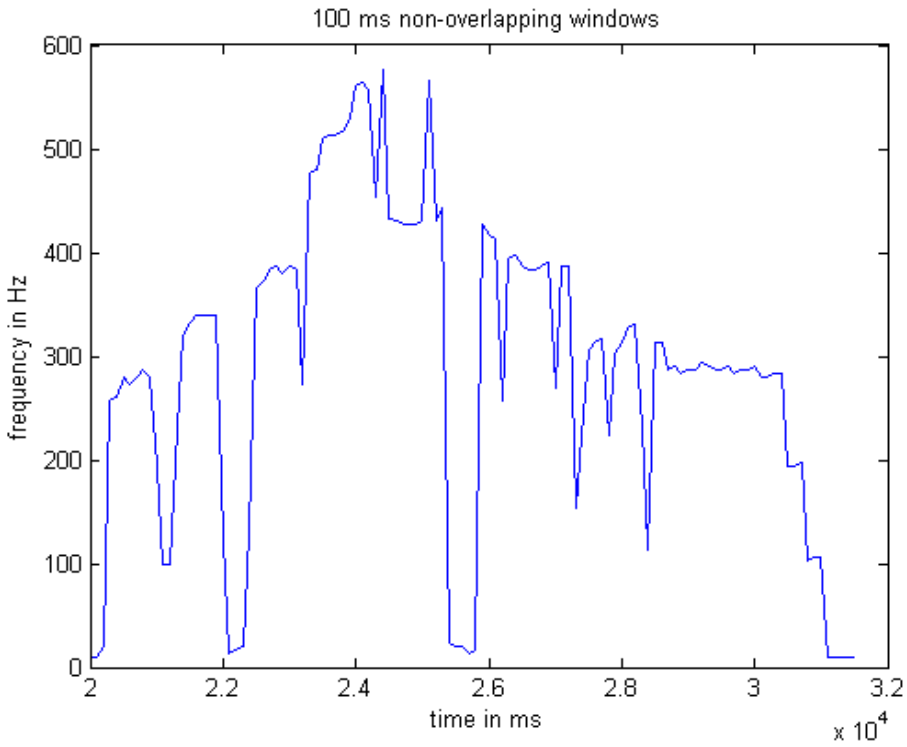

Figure 12: FFT with 100 ms windows and 10 samples per second

Here the graph is looking nice and when listening to the song it looks correct.

When comparing the graphs, it is obvious that the graph in figure 12 gives the best results. Therefore, we decided to use those settings for the calculations in our program.

The Matlab code

To calculate the most powerful frequency for each frame for the sound files we

created a Matlab function that can take in an arbitrary number of sound files, window size in milliseconds and how many times per seconds it should sample.

The function uses a built in FFT function in Matlab. We use the output for each window to calculate the power of each frequency and then store the most powerful one.

When we have processed all input files, we calculate their average (disregarding measurements with large spikes) and write the results to file which can be used in the game.

We also used the output to plot various graphs used in the report.

Our Matlab code can be found on the CD.

7.4. Microphone input

For the microphone input we used code from Reliable Software[17]. They offer a free code to sample input from a microphone and run Fourier transform on it. Since the code was exactly what we needed, we decided to use it.

The code sets up a recorder on the default sound input in Windows and it samples to a buffer. When the buffer fills up, the Listen class is notified and it calculates a Fourier transform on the input and stores it so the Game class can fetch it when needed.

The algorithm used to calculate the Fourier transform is the Cooley-Tukey algorithm[18].

7.5. Comparing The Tune

When comparing the singing voice to the original voice, we check every 100

milliseconds for the values in both and compare them.

For timing we use the Windows function GetTickCount()[19] which has a time resolution of 10ms. The error is small and does not accumulate over time.

The result of Fourier transform is a frequency so we start by converting it to a tone number.

Tones can be calculated with[20]:

$$tone = \frac{12 * \ln(hertz)}{\ln(6.875)} - 3$$

Then we get a tone number, for example middle C is tone 60. Since we never hit the tone exactly, this formula returns a floating point number which we can use to calculate the scoring.

We then calculate the difference between the tone of reference and microphone and use the graph in figure 13 as a reference for the scoring.

If the difference between them is more than a full octave, we change them to be on the same octave. This allows people to sing in their natural voice and not having to force their singing to mimic the original one.

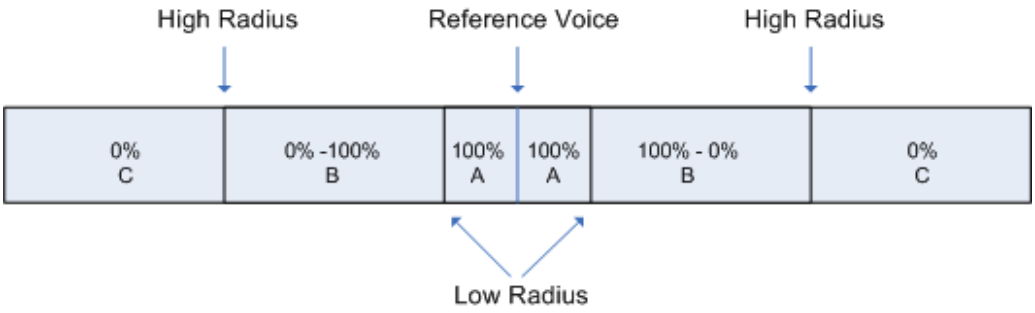

Figure 13: How the score is calculated. If a vocal falls in zone A it gets 100%, zone B is linear from 0-100% depending on how close to A it is and zone C gives 0%

We have defined two constants, LowRadius and HighRadius, which we use to determine a percent which we then multiply with the number of milliseconds since last comparison.

With this method we could easily add a difficulty feature where we could add Easy, Medium and Hard mode just by changing the LowRadius and HighRadius values.

8. TESTING

8.1. Testing

We tried to test our program as thoroughly as we could. We ran it on different computers, ran it for extensive periods, used different microphones and basically tried to crash our program.

- We found out that the game runs on equal speeds on different computers, from a 3 year old laptop to a brand new gaming machine. This includes song speed, the microphone/original voice graphs and all animation.
- When the program started it used around 61MB of RAM but after playtime of 15 minutes it had grown to 130MB.
  After searching for and cleaning memory leaks we reduced the memory usage to around 63MB after 15 minutes of play. There still are some memory leaks we haven't located but they are very small.

8.2. Beta Testing

When we thought our program was good enough to be tested by other people than ourselves, we started beta testing.

We invited some friends with different backgrounds and skills in working with computers over.

A 25 year old girl who is studying to become an actress. She only uses computers for browsing the Internet and write school projects..

A 25 year old girl who is an animator. She uses computers at work and is very familiar with them.

A 26 year old boy who is a DTU student. He uses computers for programming and all kinds of processing.

We had them start the game on their own and think out loud through all the menus and the game. Then we wrote down all their feedbacks and used them to improve our game.

Two them were not sure how the microphone/original-vocal graph worked at first but in general they were very satisfied with the game.

9. CONCLUSION

When starting this project we made goals for ourselves, those goals were to make a whole, fully functional computer game, with same objectives as SingStar.

While trying to achieve these goals we hit some bumps. For example we had no experience with DSP or sound processing, before starting the project. We learned that gathering information from the Internet and by reading books and papers is very important in a project like this. We were also lucky to have friends at DTU, studying in other fields than us, we could ask about diverse problems.

After about 8 months of work (including Easter and Summer vacations) we finished our game, we are happy with the outcome and feel we have achieved the goal we set for ourselves in the beginning. We feel that we have learned a lot from this project (which at some times seemed unsolvable) and are very happy with the outcome.

Figure 14 - The game in action. Red lines represent the original voice and green line the microphone input.

9.1. Further work

What is possible to do to make our project a better game?

Given more time and more experience in this field we could have added more features to our project and fixed some of the problems, for example:

- When calculating the highest frequency of both the original song and the players singing, we get a few spikes in the results. Part of the reason is probably due to the cheap microphone we used. Also we would like to investigate if we could have ran the sounds through some filters that would have minimized this.
- Currently, although the scoring allows you to sing in your natural octave even though the original song is in another, the displayed graph does not show it in the same octave. Fixing this would be first on our fixing list.
- All the text that is drawn with TextOutputBitmap is mono spaced and does therefore not look natural. Fixing this would not be that easy since we could not calculate easily where each letter is positioned.
- Making it possible to play over the Internet
- Make a library of songs, downloadable from the Internet
- Make users being able to add songs on their own.
- It might be fun to have videos or even a webcam from the user running in the background instead of the pictures currently running.

10. REFERENCES

1. Playstation2 - http://www.playstation.com/
2. SingStar - http://www.singstargame.com/
3. Sony Computer Entertainment – http://sony.com
4. OpenGL – http://opengl.org
5. Skype – http://skype.com
6. Freeman, Eric and Freeman, Elisabeth – *Head First Design Patterns* – First edition, O'Reilly, 2004
7. CVS – http://www.nongnu.org/cvs/
8. TortoiseCVS - http://www.tortoisecvs.org/
9. NeHe Productions - http://nehe.gamedev.net/
10. Virtual Reality Systems - http://www2.imm.dtu.dk/courses/02563/
11. Lamothe, Andre – Tricks of the Windows Game Programming Gurus – Sams, 1999
12. Sedgewick – *Algorithm in C++ Part 1-4* – p:277-278 - Third edition, Addison-Wesley, 1999
13. Microsoft Powerpoint - http://www.microsoft.com/powerpoint
14. Dolly Parton On-Line - http://www.dollyon-line.com/download/wallpaper.shtml
15. Audacity - http://audacity.sourceforge.net/

16. Matlab - http://mathworks.com/

17. Reliable Software - http://www.relisoft.com/freeware/recorder.html

18. Cooley-Tukey FFT algorithm - http://en.wikipedia.org/wiki/Cooley-Tukey_FFT_algorithm

19. GetTickCount - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/sysinfo/base/gettickcount.asp

20. Where Math meets Music - http://www.musicmasterworks.com/WhereMathMeetsMusic.html