CHAUCER IN MIDDLE ENGLISH USING ALLEGRO AND ALOGG

A Senior Study
by
Michael C. Rodefer

Major: Computer Science/Mathematics

Maryville College
Fall, 2009

Date Approved ________________________, by ______________________

Faculty Supervisor

Date Approved ________________________, by ______________________

Editor
Abstract

The motivation behind this thesis was to create a custom audio player that could play the works of Chaucer in Middle English (narrated by Dr. Sam Overstreet). The solution was to use a games library (Allegro), a sound library (Alogg), and Win32 API for the GUI. After creating a structure, and repeated extension to add more features, the end result is a highly customized program that is easily extendable, and could be modulated to play other works with relative ease.
Contents

Acknowledgements iii

1 Introduction 1

2 The Foundation 2
   2.1 Sound ................................................. 2
   2.2 Libraries .............................................. 4
      2.2.1 Playback ............................................ 6
      2.2.2 Input .............................................. 8

3 The Interface 10
   3.1 Graphical Demands ................................. 10
   3.2 Schema .............................................. 13

4 Conclusion 18

Bibliography 20
Acknowledgements

I’d like to thank my mother, who always believes in me even when I don’t. I’d like to thank my father, for working so hard all his life. I’d like to thank Dr. Overstreet and Dr. Plaut for advising me on this project. I’d like to thank the entire Math and Computer Science department for sticking with me all these years. Special thanks to Mr. Powell, just for inspiring me. Allegro is provided under a giftware license. Alogg is provided under a BSD license.
Chapter 1

Introduction

The goal of my thesis was to create a standalone program that replayed Dr. Overstreet’s Middle English reading of Chaucer’s *Book of the Duchess* and *Canterbury Tales*. There were various goals set forth during this time. The first order of business was simply to figure out what to do, full stop. At the beginning of the project, the available data was a combination of sound files that had already been formatted into wma (Windows Media Player) format. The files had been marked at every verse with Sony Sound Forge. After that, different components had to be acquired to create the program. Many design decisions had to be made (and altered) to fit the specifications of the project. The original format of the sound files was soon found to be insufficient to meet the goals of the project. Additionally, the current program written by Dr. Overstreet, while perfectly functional, was not portable and relied on many conditions that were just not met by (or were impossible to meet) on most computers. In the end, a myriad of problems had to be solved. These fell into two distinct types: foundational and graphical.
Chapter 2

The Foundation

2.1 Sound

One of the first major design decisions dealt with was the sound file type. As stated, the current file type being used for the online program was wma. Wma proved to be unsuitable for this project for several reasons.

Firstly, the wma library is closed source [3]. This means that the types of programs that can play wma are severely limited. A closed source library is a library that is not publicly available for free use; therefore, there is no way to make wma files play on any arbitrary type of player. There are reverse engineered libraries that are able to play this type of file, but the licenses for these libraries are also restrictive.

Another problem with wma is that there are few file types that allow built-in time stamping and marking, so changing from wma required finding a substitute for this feature. This feature was essential for the creation of this program, but is a rare feature among sound file types. For example, wav files, the simplest type of sound file, lack this feature, and leave no possibility of implementing it. Mp3 files, another commonly used file type, also lack this feature. No other common type of sound file can duplicate this functionality of wma files.
CHAPTER 2. THE FOUNDATION

The last problem with wma files is a consequence of their closed source nature. The specifications of wma files were not publicly disclosed. This makes wmas useless for this sort of project, even neglecting their closed source nature. Without knowing the precise specifications of a wma file, it is impossible to write a program that can randomly access the internal workings of a wma file without relying on some sort of external program (like Windows Media Player) that knows the secret specifications. This was unacceptable.

The best way to solve the first problem and last problem was to choose a new type of sound file; one that was open source had to be chosen for this project. The wav sound file type was far too large to use. The wma file type, again, is not open source, and cannot be used for the project due to compatibility problems and unknown formatting. This consideration was irritating because it forced a choice of another type of sound file, and a significant shift in the focus of the project. Mp3 is also a patented sound file type, and was thus eliminated from the file type candidates. The best selection was made simply by looking over the types of sound files that are open source, and choosing from among them the one that was most suitable. The second problem, that of time stamps, is addressed next.

The most viable sound file type turned out to be the ogg format. Ogg is a “fully open, non-proprietary, patent-and-royalty-free, general-purpose compressed audio format” [12]. It also is more compressed than most other formats are, so it does not cause any sort of memory problems like that which wav causes. It also did not present any of the problems of the wma format. All of the specifications of ogg files are known to the public; the compression of ogg files is also suitable. The ogg format seemed like the prefect candidate for the project; however, the ogg format has no way to preserve the time stamps, which are needed to select different lines within Chaucer’s works.

The solution to this required Sony Sound Forge. This program was used originally by Dr. Overstreet to time stamp the sound files and label the different lines within
CHAPTER 2. THE FOUNDATION

Chaucer’s works. The usefulness of this seemed rather questionable though; the only format that could even save these markers innately was wma. This was completely unsuitable for the project; wma files had already been rejected for use. There was no turning back, so an alternate solution needed to be found. Eventually, an obscure option that allowed the track information (the verse numbers) to be exported into a text file was located. Most of these files were not human readable and therefore unusable, but the last option allowed for exporting as a track listing in a plain text format. This information turned out to be the key to retaining the track information outside the sound file.

Sadly, this information was a bit messy, and needed some hand editing to provide a workable, computer readable source. The files underwent various modifications via a simple text editor. A myriad of different kinds of superfluous annotation were removed. Some additional unnecessary data was also contained within the track listings, and had to be removed before the files could be processed efficiently. For example, extra punctuation had been stored in the text file, and was eventually removed. These processes were repeated several times over the eleven different track listing files. After all was said and done, the track listings contained no data except the line numbers and the times in hours, minutes, and seconds. The information was now in an easily parsable format, but how to process it was still not clear.

2.2 Libraries

With the sound file and track listing situation now under control, another primary design decision was necessary. The basis of the project needed to be determined: the language, and the libraries. A lot of libraries were looked at; primarily, something powerful that could handle multimedia was required. The search for libraries narrowed to game oriented libraries, due to their close association with multimedia. Eventually,
Allegro was chosen due to its well established position in the free computer gaming community in multimedia rich games like *M.U.G.E.N.*, the everything vs. everything fighting game, and *Icy Tower*, a game that involves jumping up a tower as quickly as possible [7]. It was also one of the few free libraries with a license that allowed the user to sell his or her final product; it is licensed as giftware, which is not restrictive in the least. Choosing Allegro led to C++ being chosen as the programming language. Allegro was originally intended to provide a GUI; more on this later.

A suitable sound library was now needed; this turned out to be more tricky than expected. The first hurdle was not much of one, right? Well, finding a useful library with a weak license turned out to be a time consuming process. Most libraries that were capable of playing ogg files lacked necessary features. Most of these libraries are rather unassuming, but two in particular stood out. Unfortunately, the two libraries both had the same name: Alogg. One of the libraries had a license that was too restrictive to be used in this project. The other library had a different sort of problem; finding the library was nigh impossible. It was finally found using the Wayback Machine; the Wayback Machine is named after the machine in The Rocky and Bullwinkle Show [6]. Anyway, using this archive allowed the downloading of an ancient (in terms of software) package of Alogg distributed by Javier González in 2002 [5]. This library was released under the BDS license, which is “extremely simple and very liberal” [4]. This seemed like a good solution, except the source code had virtually no documentation.

Choosing the Alogg library made choosing a compiler simple. It actually removed the ability to choose completely, due to its quirky nature. The Alogg library steadfastly refused to compile under complier except Microsoft Visual Studio. The library itself threw around 100 errors when used with Dev-C++. It did something similar with Borland C++. The errors themselves were irreparable. The compiler in Dev-C++ was as sardonic as a machine could hope to be, calmly dumping errors onto the
screen about missing certain header files; this might have made sense, if the header files weren’t included in the project. So, Microsoft’s compiler and environment were chosen because they worked properly with all these accessories.

2.2.1 Playback

With the sound library in place and the platform chosen, it was time to make something operable out of those edited track listings. The idea that seemed most reasonable was to convert all of the times into milliseconds; this would allow direct interaction with the Alogg library and expedite the development of the program. This theory worked well in practice; it was an utter success. The program that did this conversion only took a few parameters. Admittedly, it did require the number of lines in the entire passage to properly convert the track listing into code. There were a few bugs with this, but later on, error checking was added that would detect any defect in line numbering, such as multiples of the same number or gaps between numbers. This allowed for detection of a purposely missing line 480 in The Book of the Duchess, a rejected line that is not considered authentic, without prior knowledge of this discrepancy by the programmer. After these things were accounted for, the track listings were suitably converted into thousands of switch statements that contained only line numbers and referenced millisecond times that were in the millions. This solution does not look beautiful, but is fast in practice (O(1)), due to the low amount of operations required to call a single switch statement, and the lack of file operations.

The next issue was getting the sounds to play back. Playing back recordings of Dr. Overstreet’s talking does not seem like a complicated problem, even if it is in Middle English. Unfortunately, there are quite a few things that can go wrong on a computer, even if it seems like a simple playback issue (even a cassette could do that, after all). It was not simple, however, and often the file played back for far too long, or hung abruptly and kept repeating an obnoxious noise that would often accelerate
CHAPTER 2. THE FOUNDATION

and continue for a very long time. Alogg was plagued by strange polling issues, and trying to get them resolved was a major challenge that eventually involved hacking up Alogg internally.

The first issue caused by Alogg’s broken polling was quite disturbing; the resultant sounds issued were problematic. They mostly involved playing Dr. Overstreet’s reading correctly at the beginning, and going strong unto the end, at which point it would start playing the last word repeatedly.

In the end, changing the way the program handled termination of ogg playback corrected this awful problem. Forcing the file to close after playback created another odd problem: the file would always close either too early or too late. Sometimes, this worked out fine, and Dr. Overstreet’s reading terminated at the proper time; often though, the reading continued to get confusing messages that caused it to suddenly jerk and cease playing before a verse was completed. At other times, it missed its goal after coddling through a verse and failed to stop until it was too late. The source of this problem was due mostly to buffering.

The buffer in Alogg had some strange design feature (or flaw). The problem was not too difficult to correct when the logic behind it became clear. Alogg was last updated in 2002, and back then processors were not quite as good as ones today. As a relic of even older times, the program was compensating by buffering chunks that had to be at least 4KB. This was to reduce polling and not overuse the CPU. This “feature” made the previous problem possible. The solution to this was elementary: all of the code in the Alogg library that restricted the minimum buffer size had to be removed. This involved fishing around the source files looking for instructions that were limiting it; three different ones were identified and removed. Once this was done, the buffer was set appropriately to be 1/1000th of the previous buffer size. This rectified the random polling problem, but left the recordings always stopping predictably too early. Forcing them to play for an extra 300 milliseconds made the
readings play properly consistently.

2.2.2 Input

With most of the major design issues mostly cleared up, and the ogg reading mechanism implemented, it was time to test the program. The only thing needed to make this happen was a function to read in keys. One of the downsides of the Allegro gaming library is that it is not compatible with standard input and output; this means that all cin >> statements would not even compile. This was a big nuisance when attempting to test sound playback. Before a method to do this was discovered, the information to test playback was just hard coded. This does defeat the purpose of human input, so an alternative had to be developed soon.

The solution to this was to use Allegro’s built in key maps to set certain bindings to respond to key presses. Admittedly, this is not one of the most spectacular accomplishments during the duration of this project; it was rather convenient though, because it ensured that the only buttons which mattered were the enter buttons and the number buttons.

This in turn made error testing in this program simple. All of the error trapping routines were typical and unremarkable. The only decisions that had to be made with them was with corner cases. For example, the behavior when a line was requested over the boundaries had to be defined. The program would just prompt for the user to re-input the line numbers in this case. Error trapping was also needed to handle the spurious poetic line that had been deliberately omitted by the editors of The Book of the Duchess. The program at this point just substituted a suitable number in for line 480 – the non-existent line. Error messages for it do exist, but were not incorporated into the program yet. Another error, attempting to read from line 32 to line 29 or something else of that nature, also refused to process.

All of these processes culminated to allow the program to function properly. It
was able to replay accurately any line to any other line in The Book of the Duchess and The Canterbury Tales without a hitch. The input method did not allow invalid input; it ignores most of it, and throws errors on the rest. The ogg file format was efficient and tiny, and with a little work the program would likely fit on a CD. This eventually led to the alpha version of the software, a screenshot of which is seen in Figure 2.1.

The program was in good shape so far. It was able to play back fine; however, it was not very beautiful. While a program like this was standard as command line programs go, a more interesting interface was required for a product that was intended to be commercial.
Chapter 3

The Interface

3.1 Graphical Demands

Even with so much progress, one problem that needed to be corrected was the GUI (graphical user interface). In the current state, the program had none. The program lacked a beautiful new set of feathers, and the road to this was fraught with peril.

The problems with the GUI began with just trying to find a platform for it. The default GUI seemed like an acceptable option at the onset. It does work; the Allegro GUI is fully functional. The only problem with the Allegro GUI is how ugly it looks; it looks just slightly better than the command prompt. Maybe it could be modified though; this should not be a problem. There are a lot of packages that are free that are able to skin the Allegro GUI and transform how it looks.

The search for a GUI was on. The search itself seems to come with a bit of inauspiciousness. Most web sites, in a typical Allegro manner (Allegro is old), were broken or abandoned. Most of the GUIs on these web sites suffered the same fate as the sites themselves. Ill maintained graphics packages for Allegro clutter the web. A large number of them were explored, and some of the best ones were linked to on the
Eventually, the BGUI2 was tested. BGUI2 can make the standard interface look much nicer [11]. To the author’s dismay, BGUI2 quickly became a time wasting sink. BGUI2’s biggest problems were its conflicts. They could not be overcome. Many different methods were attempted. The BGUI2 was compiled by itself. The debug settings were changed. The debug settings on the Alogg library were changed. Both libraries were compiled as multi-threaded DLLS. They were even compiled together into one library. None of these methods lead to success. It continued to be evasive. No matter what was done, using both BGUI2 and Alogg together resulted in a program hang, followed by a crash, whenever a sound file attempted to play.

Hope was not lost: the GUI problem was resolved in the second half of thesis. There were more snags that had to be cut before that was possible, however.

Moving into 352 the program still was without a suitable GUI, so more Allegro based GUIs were tested. It became clear that they were not going to work in this project. This is a sad weakness that the Allegro library has; it does not provide any sort of more modern menu system. This is not a weakness that most people would worry about using Allegro; it is just designed as a games engine, after all; it was not created primarily for the purpose of writing other applications. This meant that every Allegro GUI (like the aforementioned BGUI2, another GUI called Adime, and one called CGUI) was unsuitable [8]. None of these GUIs looked good; sadly, all of them had a very DOS look to them that was not proper for this project.

After searching for a while I finally decided to use Win32 API. It was finally selected for use in the final project; however, there were certainly a good number of challenges in dealing with it.

One of the first problems with Win32 API was compatibility with the Allegro engine. At first, there seemed to be no problem, until tests were attempted by adding Win32 API code into the project. This did not go over so well; in fact, it made the
entire project refuse to compile (a tragically frequent problem in creating this thesis). The workaround to this problem was not so easy to come by.

Here is some background on the cause of the problem. The Win32 API defines a type called BITMAP. The same type is defined in the Allegro library. The end result of this is that the Allegro library and Win32 API will naturally not work together.

The solution to this came from a small tutorial [2]. The way to solve the problem was to add a new header called <winalleg.h> and remove the standard header used for Win32 API <windows.h>. This fixed most of the problems, and also allowed using Allegro’s bitmap type [2].

Branching out from this topic, it again appeared that there would be some way to use another IDE besides Visual Studio; this did not pan out. The main reason that this would not work is that the files for the Alleg library that worked were precompiled; they had to be placed in the directory to be detected by Visual Studio [1]. Oddly enough, there was no apparent way to get them to work with a different environment (like Dev C++) even with the source code, so this idea was scrapped.

Win32 API still seemed clunky, so more effort was made looking for a more logical replacement. The forms that were built into Visual Studio seemed like a good option. Like many other things in this project, they proved incompatible. Since Win32 API had proven to at least function properly with the Allegro library, it was decided that even if it was a bit complicated (and difficult to edit) that it would be used anyway. This was despite the fact that Visual C++ had no direct way of working with Win32 API resource files (.rc extension) [9].

The problem of creating a GUI interface, which also resulted in the creation of dialogs (a new problem; the Win32 name for graphical windows), was solved by a program called RadASM. One of its features, ResEd Resource Script Editor, was crucial to quickly creating the interface [10]. Though it is possible to create it by hand, it is hard to visualize exactly what a Win32 dialog will look like. That is where
the resource editor came in handy; it allowed for the creation of dialog menus on the fly.

Unfortunately, the style of the dialog boxes was a bit confusing; a lot of manual editing was still required for them to work right. It was also difficult to find a way to link together the dialog box with the application.

That being said, the resource editor was effective. It was irritating that the Express Edition of Visual Studio was incapable of handling resource editing. The end result was that both applications were required to make the program. Eventually the resource file was in acceptable condition. It needed a resource header, but a decent one was automatically generated by ResEd. So, at this point, there was a rather crude dialog window.

This was a good thing, and an important point in the project. The only big thing left to do once both the interface and the program were completed was to link them together into one big program.

Doing this required using dialog procedures. Using dialog procedures with C++ is not as flexible as C style ones, so the program was tweaked a little to make it compatible with C. The only thing that needed to be done was to move the locations of some of the variables (C does not allow variables to be declared in the middle of a program; they must be declared on the top). The other thing that was required was to add something to replace Boolean functions, but this was easy: integers were used.

### 3.2 Schema

The first revision of the program; see Figure 3.1. This is the first version that had a graphical interface, which was formed with three different areas that allowed the input of integers. Two of those areas were designed to take line numbers, and the third one was designed to take a different number for either *The Book of the Duchess*
or one of the ten fragments (groupings) of *The Canterbury Tales*, and play it.

This structure was a little unwieldy, and not that graphically pleasing, so eventually it was decided to put a drop down menu into the code. This allows the user to select whatever they want.

The structure of the final project is as follows; this is a basic description of all the more important design aspects, neglecting the minutiae:

1. Eleven functions for the various books. They are responsible for calling the play functions for individual books, and the play parameters ("book", start, end).

2. Eleven tables that list every single line of the eleven books. These tables return integer values of the position of the line in milliseconds, or the time stamp of the final line if a giant number is typed in (recent design decision; this caused an error before).

3. The ogg play function, that can take the various times from the table functions and play the opened file specified from inside the book function. This function is responsible for polling the file to make sure it plays the appropriate amount of time. It uses a modified version of the Alogg function that allows smaller buffer sizes (1/1000th
of normal) for more precision. The polling has a -300 millisecond offset to correct a consistent mistiming error.

4. The primary interface (in Win32 API), that consists of:
   i. The initialization case, which controls the creation of the graphical interface.
   ii. The command function, which selects the appropriate book function, if the input is reasonable; otherwise, it throws errors. (Example: if the first line is after the last line).
   iii. The close function that also kills the entire program (including the Allegro part).

5. The main function. It has been extended several times to perform various tasks. In the beginning, it displayed all the instructions. It was eventually made invisible so that the graphical interface would be the only thing on display. It was finally altered to draw a picture and serve as a background mat for the main program. It sets up a lock function on itself as well calling the dialog box.

6. The dialog box has a programmatic component outside of the main .c file, located in the resource file and in its header. This names and determines the layout of the graphical program.

7. The lock function that detects if someone is trying to place the background over the graphical program. It kills sound playback on mouse click, then throws the focus back to the WinAPI program.

8. The program has a function (using WinAPI) that detects if the ESC key is pressed. This will abort sound playback.

9. The callback locking function employs the Allegro timer. The current state of the sound file (if it is playing or not) is stored in a special “volatile” integer. This allows the program to recognize that the value will frequently change as an interrupt, and reduces the chance of freezing.

10. Besides the aforementioned <winalleg.h> library being used in place of the
standard library for Win32, a special preprocessor instruction was required to make the program function properly. This instruction

#define ALLEGRO_NO_MAGIC_MAIN

in essence allows the program to run without Allegro doing strange things to the window that would otherwise interfere with the GUI.

11. An outside component of the program is used to process the data files extracted from the original wmas Dr. Overstreet created. This program writes another program that makes switch statements for all the line numbers, and returns the times.

12. The undocumented Alogg library made by Javier González is required for this program to operate. Besides hacking the buffer, information about how alogg_play_ex_ogg works is needed to use it properly. For the record, the parameters are ALOGG_OGG (the sound file), buffer len (written like 2<<2), volume (max is 255), pan (127 is in the center of speakers), speed (1000 is normal), and loop (1 is loop, 0 is not).

13. A relatively unknown command called set_display_switch_mode

(SWITCH_BACKGROUND). Without this command, the program is unable to play music with the Allegro window out of focus. Without it, the GUI is completely useless. A picture of the final version is seen in Figure (3.2).
Figure 3.2: The Final Version of the Program
Chapter 4

Conclusion

The biggest thing to learn from this project is how some hard-to-find, or unknown, commands usually make life a lot easier. The SWITCH\_BACKGROUND command is a good example of this; without this command, the program would not run. The use of a volatile int ironically helped stabilize the program. Hacking the Alogg library to allow smaller buffer sizes made the files play for the correct amount of time. The entire Alogg library is only sparsely documented inside itself; there is no external reference. The usefulness of being able to learn about odd, obscure commands should never be underestimated. The results of this project also show the utility of mixing different programming languages and libraries together. It would not have been possible to efficiently create anything like this without the different functionality each tool brings to the table. The complete project requires three programming languages (C, Allegro, and Win32 API).

Moving forward, the sister project to this would be the next likely step; the Windows program is finished. The sister project would be an online implementation of this program. It would require the implementation of a database to track users who logged in, and quick streaming and the ability to play from any point within the file used. Some advice to the person who attempts this:
CHAPTER 4. CONCLUSION

1. Use a file type like ogg (or just ogg itself). File types with no license are invaluable in this type of project.

2. Find an efficient way to randomly access the sound files. Taking up a lot of space on the server would be fine; do not use a slow method to do this.

3. The sound files themselves are still huge; make sure to just stream the parts requested.

4. Do not rely on client side software any more than necessary; it may not work or even exist.

Leaving the web site project behind may not be a bad thing; it will allow another person the opportunity to experience the rare aspects of this project. The requirement to learn how to make so many different parts and build something useful makes this project memorable, and its completion an achievement.
Bibliography


