Analysis of the CS249 Style through the Creation of Photomosaics

Nicholas Dovidio

Spring 2009

Abstract

Mathematical artwork employs mathematics to create pieces of artwork. In one such method, photomosaics, images or other objects are used to approximate a target image. In effect the target image is simulated by the tiling images. To create such pieces of artwork a large system of linear equations is solved. In this project we create photomosaics by solving that system using Ilog's Cplex software package. To implement this code efficiently and accurately we used the style presented in CS249. This involved many complexities such as writing a wrapper around the Cplex interface so that it conformed to our framework's specifications. Another major difficulty encountered was working in the CS249 methodology in Java. In this paper we first discuss and show our results concerning photomosaics. Once the application domain has been presented and understood, we then discuss how the CS249 style enabled us to write better software.

Introduction

Professor Robert Bosch of Oberlin College has created a variety of photomosaics using different mathematical formulas. These include the traditional photomosaics, where smaller photos are combined to approximate a target image. He has also written papers on first using a stippled image to approximate a target image. Then the dots from the stippled image are connected by the shortest possible path. He calls this traveling salesman artwork. Additionally he used complete sets of dominoes to approximate a target image. These various forms of artwork have a constant theme, solving a large linear system of equations. As an undergraduate myself, Gavin Taylor (Now a PhD candidate Duke University) and Assistant Professor Tim Chartier (Davidson College) created a variety of different pieces of software to recreate and refine Professor Bosch's work. In this project we revisit one of those pieces of software, specifically the photomosaic creator, and restructure it in the style of CS249. Additionally we will rewrite the software to use Ilog's Cplex instead of NEOS to solve the linear system of equations.

By rewriting the code in the CS249 we have created a better piece of software. By better we mean a more robust piece of software with fewer bugs. The software is also more expandable, with this basis intact it would be much simpler to incorporate code that solves the other two mentioned mosaic problems. In this project we will discuss the use of the CS249 style and specifically analyze how it

performs in Java. For example, the limited power of Java generics and the lack of operator overloading can make the Java implementation of the CS249 methods more complicated and cluttered.

This paper is broken up into two main sections. The first is an explanation of the mathematics behind photomosaics. Here we also present our created images and some analysis of them. After that section is concluded the reader will have a firm grasp of the problem's domain and the problem we are attempting to solve. The remainder of the paper is a discussion of the CS249 style as implemented in this project. The topics we will focus on will be on framework design, interface design, exception handling, and value types amongst others. Throughout these discussions we will specifically focus on the dissimilarities between Java and C++

The Photomosaic Problem

The photomosaic problem stated abstractly is to take a set of images, known as tiling images, and then put them on a canvas in such a way that they approximate a target image. These images may be used more than once. To create the target image the tiled images are laid out in a way to minimize their distance from the target image. Below we show such an example.

Dr. Michael Mossinghoff approximated by the Davidson College Staff

To describe this problem more formally we must first describe linear programming. By understanding the application domain of a linear programming problem, we will better be able to explain our interface decisions in our implementation later.

Linear Programming Overview

Linear programming is a form of optimization where all of the constraints are

linear (Chartier 63). More specifically, a function known as the objective function is maximized or minimized according to some constraints. The objective function and the constraints are made up of decision variables whose values are what must be chosen.

The Photomosaic Problem

Now that the underlying ideas behind linear programming have been presented, we will discuss the constraints of the linear system we are attempting to solve (Bosch). Recall that our goal is to use our tiling images to best approximate a target image, while satisfying the constraints.

The notation used here is as follows: We will have k tiling images and use each image at most m times. We will define the set of tiling images to be F and let f be a specific image from this set. We will break up our target image into fixed sized blocks. For simplicity we will have the same number of blocks in each row and column. This scheme mostly preserves the image's orientation since we only use all horizontal or all vertical images in our tiling set. We define $t_{i,j}$ to be the image intensity of the block at row i and column j in the target image. The range for i and j will be $0 \leq i \leq floor\left(\sqrt{(k*m)}\right)$ and $0 \leq j \leq floor\left(\sqrt{(k*m)}\right)$. We define this upper bound as R.

With this notation we can now formalize the problem. First our decision variables will be of the form $x_{f,i,j}$. This means that for each grid block we have a decision variable for each image in our set. This decision variable will be binary and will be true when we use image f to approximate block $t_{i,j}$ else it will be zero. We then have some constraints on our decision variables. The first is that we cannot use each image more than m times and we cannot use an image a negative number of times. More specifically: $0 \leq \sum_{i=1}^{R} \sum_{j=1}^{R} x_{f,i,j} \leq m$. We also have a grid constraint, each block must have exactly one image inside of it. This corresponds that for each block: $\sum_{f=1}^{k} x_{f,i,j} = 1$. Lastly we have our objective function. Recall the goal of this function is to minimize the difference in detail between the approximate image and the target image. Mathematically this is: $Minimize: \sum_{f=1}^{k} \sum_{i=1}^{R} \sum_{j=1}^{R} Cost\left(x_{f,i,j}, t_{i,j}\right)$. The cost function is an attribute that we experiment with to allow users to produce different photomosaics. Typically it is defined as the two norm (Bosch). Now that we have formally described the problem, we present a few different results. We experiment with different cost functions, images, and number of uses of images.

Experiments in Cost Function

The choice of a cost function can have dramatic effects on the resulting image. In all of our experiments we used cost functions of the following form: First divide each image f into four quadrants. Then average the pixel values of those

quadrants (Taylor). Similarly for each block $t_{i,j}$ divide the region into four quadrants and average the pixel values. Then the cost of placing an image f at $t_{i,j}$ is computed as follows: First compute the difference between the corresponding quadrants. Then compute the corresponding norm of this value. In the euclidean norm case this is: $\sqrt{\left(\sum_{q=1}^{4}\left(\left|\left(f_q-t_{i,j,q}\right)\right|\right)^2\right)}$ where q is an image coordinate. Note we include absolute values here because it will become important when using other norms. Below we show examples of how using different norms changes the resulting image:

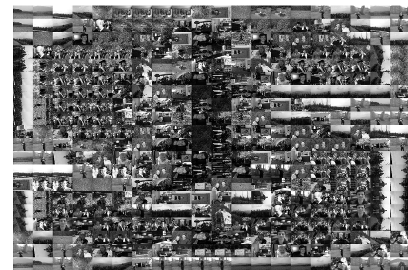
Graduation Picture using the 1, 2, and 3 norms.

The above mosaics are made from made from a small tiling set where the various images were used multiple times. The end result is a tiled image that used 400 pictures. Disappointingly these results are not a good representation of the target image. This suggests that we either need a better objective function, better set of tiling images or more images. In the next section we explore change the number of times we use an image. Another possible objective function considered was scaling the cost function so that predicting the interior points is much more important than predicting the edges. When we discuss our code implementation, we will see how this would be an easy change.

Using More Images

Another attribute to our mosaic is m , the maximum never of times we may use a

specific image. By increasing the number of uses we can increase the resolution of our photomosaic. Below we experiment by changing the number of images used in our approximation of the Stanford tree.



Above we employed a small image set to approximate the Stanford logo. We chose to use the images a maximum of one time, two times, three times, and four times respectively. Clearly as we increase the number of images used, our corresponding accuracy also increases. We observe that even with a large number of uses, the image resolution appears distorted. This behavior occurs because the tiling images and the target image have a different aspect ratio.

Concluding Image Analysis
We have successfully created our photomosaics and have two attributes that we can alter to produce a better image, namely the cost function and the number of images used. By exploring these parameters and changing the tiling set we can create a variety of photomosaics. Through the rest of the paper the design is analyzed in the context of the CS249 style. We will see how our implementation will allow for easy modifications and improvements.

CS249: SOS Methodology

The backbone of CS249 is based on SOS: source code representation, outside-in development and short-cycle development (Cheriton 14). During this project we successfully followed these tenets to guide our development. In the following sections we will discuss how they improved the overall software product and focus on how they help reduce the slice of the programming pyramid a programmer needs understand to complete his work (Cheriton 9). These ideas have influenced our design in many areas including naming, exceptions and

template use. We discuss and analyze the decisions arising from the CS249 style and also focus on how Java alters those decisions.

Source Code Representation

In a software program the software is the truth, not some diagram and not some associated documentation (Cheriton). The goal of the CS249 style is to make the source code readable to reduce bugs and improve the ability to evolve the software. To do this objects are based off of the application domain and the objects are strongly typed. By basing our objects in this manner we reduce the number of names and concepts that a programmer will need to know (Cheriton 563). Further, CS249 argues for an attribute only interface. In such an interface attributes can be read and possibly written (Cheriton 59). This helps keep a separation of state and processing. We implement these ideas throughout the project but they are clearest in two cases. The first is our design of the Cplex wrapper module and the second is the use of our value types.

The Linear Programming Solver

As mentioned previously, one of the goals of this project was to write a wrapper around Ilog's Cplex solver. This software package is an excellent linear solver, but many of the terms and mechanisms are not compatible with the CS249 style. To create the appropriate interface we looked at a typical linear system of equations such as the one below (Chartier):

$$20 \leq 8B + 1V$$
$$17 \geq 6B + 1V$$
$$Maximize(B)$$

This system contains only a few key entity types. First there are decision variables. These can then be combined into expressions. An expression can either be combined with another expression to create a constraint or the expression can be used as the objective function. Lastly, the combination of these parts creates our resulting system of equations. We defined our classes and interfaces to directly model the application domain (Cheriton 667). Specifically our wrapper around Cplex has four classes: Variable, Expression, Constraint, and Linear_Solver_Factory. Each of these corresponds to a specific part of a linear system, note that Linear_Solver_Factory is a factory object that creates the linear system components. In a future release this class would act as a directory, allowing users to query about all variables, constraints, and expressions it has created.

Before even reading the individual classes, a user will expect certain behavior from them purely based on their name. This helps to limit the slice that a programmer needs to learn before he can effectively use the module (Cheriton 9). By analyzing each individual class the programmer can further understand the structure and realize it corresponds directly to the application domain. For example, one method of the Linear_Solver_Factory is:

```
public Variable variableNew(Variable_Type type,
        int min, int max, String name)
```

By looking at the method name and call signature we can understand that this method creates a new decision variable, of a certain type, with a certain range

and name. In contrast the Cplex documentation has over 550 pages of information. This cannot be quickly and easily understood and may deter possible users from purchasing the software package. Admittedly our wrapper removes some of Cplex's functionality, which may be the reason to purchase the software, but we believe most users simply want to setup and solve large systems of linear equations and not be concerned with the underlying methods.

Value Types in Java

In CS249 we discuss the use of value types which are abstract collections that have a very specific set of operations (Cheriton 620). At the framework level we have classes such as Ordinal, Nominal and Interval. Each of these have a consistent set operations that is used throughout the framework to help limit the slice of the software a developer needs to know. We then inherit and instantiate classes from these base classes. For example, instead of individually modeling each and every picture we may instead use a PictureCount value type which inherits from Ordinal. We use such a value type because it makes the code more readable and provides type isolation. For example based on the name alone the programmer would expect PictureCount to always be positive and to be comparable to other PictureCount values. By having our code correspond to expected behavior we improve readability and help the programmer understand the class's purpose and function faster. Also the programmer will already know the framework meaning of Ordinal and understand what operations are allowed on PictureCount. Lastly, this provides type isolation so we do not have someone adding PictureCount values to AirplaneCount values. The previously mentioned ideas worked extremely well in C++. But the limits on Java generics and lack of operator overloading forced us to make some modifications to these ideas.

To illustrate the problems with Java value types we will investigate implementing the Ordinal class. In the CS249 framework this class has the full set of relational operators. The first problem with Java generics is that they cannot take in a primitive class such as int or double. Instead Java requires the generic to be based off of some class such as Integer (Cheriton 541). This seems to be unnecessary and will result in performance deterioration because now we have to ask the Ordinal's RepType for its value. This involves two function calls. Another performance hit comes because we cannot inline function calls in Java, while the C++ version of Ordinal can inline all function calls. In effect using value types in Java is expensive, while it is nearly free in C++.

Since Java does not have operator overloading for the relational operators, it uses the comparable interface instead. This means that our base type must also implement the comparable interface. This does not initially seem like a problem until one examines the comparable interface. To implement the comparable interface one must override:

```
public int compareTo(Object o) throws ClassCastException
```

This could be a relic from when Java did not have generics and such interfaces were implemented in terms of passing around Objects (Cheriton 613). In our implementation we also define:

```
public int compareTo(Ordinal<RepType> o) throws ClassCastException
```

This more specific method will be called when our Ordinal class is being used. But by having our Ordinal compareTo method have the Object version we begin to loose the compile time type checking we desire. Instead we now have run time checking with exceptions. Further Java does not keep track of the RepType at run-time, allowing two different instances of Ordinal to be compared without any errors!

To put this concretely the following is compilable Java code assuming the client catches the ClassCastException:

```
PictureCount p = new PictureCount(5);
String testString = "B";
System.out.println(p.compareTo(testString));
```

This undermines one of the main points of using user defined value types, compile time checking. We also mention that a similar problem was encountered when implementing the Nominal class and overriding the equals method.

The value type methodology does not translate well into Java. As mentioned this occurs for two main reasons, the first is performance. By forcing everything to go through an extra level of indirection using these value types is much slower than the equivalent C++ version. Also the compareTo interface is a poor substitute for operator overloading and pushes off what should be compile time checks to run time checks. Perhaps a better and future implementation would totally ignore the standard Java equals and compareTo interfaces and create a new set of interfaces. The disadvantage of this approach is that it would require programmers learning a larger slice and not conform with expected Java behavior. It should be noted that the Java approach is better than the traditional C #define syntax. While we do not get compile time checking, we at least get some run time checking, which can help make debugging a problem easier. In contrast, C would give us no hint that we were doing something wrong.

Outside-In Development

Another key to the CS249 methodology is outside-in development. In this approach use scenarios are first developed, where a use scenario is “an application, pattern or setting for using the software that you propose to develop” (Cheriton 23). In effect we analyze how the software will be used and have that motivate our interface design. In our project we knew that our main goal was to create and solve photomosaics. To accomplish this task, we recognized that there were a few key separate modules. The first module was analyzing the tiled and target images. The next was solving the system of linear equations. The last was analyzing a solved linear system and creating the resulting tiled image. Each of these is a separate problem that required a separate interface. Since developing the linear solver was the most complicated piece of software, we choose to focus on how outside-in development aided in this area.

Outside-In Development and Solving Linear Equations

At the heart of the photomosaic problem is solving a large system of linear equations. Hence our use scenarios were all systems of linear equations where

we knew the solution ahead of time. We observed that solving these small systems would require similar operations to solving the photomosaic system.

We used the following scenario as the basis for our tests. Suppose that by law a ship must have a life vest or lifeboat capacity for every person on board. The ship's captain would like to maximize the number of lifeboats used because of their comfort. But the lifeboats take up more room than life jackets and hence a combination of lifeboats and life vest must be used. As a specific example we have the following (Chartier):

$$\textit{People Constraint} \;\; 20 \leq 8B + 1V$$
$$\textit{Volume Constraint} \;\; 17 \geq 6B + 1V$$
$$\textit{Maximize}(B)$$

In this example there are 20 people on board. Each lifeboat can hold 8 individuals and each life vest can hold 1. The total volume allotted for the life vests and lifeboats is 17. Each lifeboat takes up 6 units and each life vest takes up 1 unit. By having a specific scenario in mind we can write up some code that we expect to solve the problem. Below we show the final code that will solve this problem:

```
Variable boats = factory.variableNew(Linear_Solver_Factory.Variable_Type.INT,
0, 100, "Boat var");
Variable vests = factory.variableNew(Linear_Solver_Factory.Variable_Type.INT,
0, 100, "Vests var");
Expression peopleOnBoard = factory.expressionNew("Capacity");
Expression volume = factory.expressionNew("Volume");
Expression objective = factory.expressionNew("Objective");
peopleOnBoard.termNew(8, boats);
peopleOnBoard.termNew(1, vests);
volume.termNew(6, boats);
volume.termNew(1, vests);
objective.termNew(1, boats);
factory.constraintNew(20, peopleOnBoard,
Linear_Solver_Factory.Constraint_Type.LE);
factory.constraintNew(17, volume, Linear_Solver_Factory.Constraint_Type.GE);
factory.objectiveFunctionIs(Linear_Solver_Factory.Solution_Type.MAXIMIZE,
        objective);
```

This code is very readable and corresponds directly to the stated problem. The original test code, not shown, looked nearly identical. By developing simple use scenarios such as above, we could refine and debug the linear solver wrapper before using it on the more complicated photomosaic problem.

Short Cycle Development

Another of the main tenets of CS249 is short cycle development, which views software as an iterative process. Here each development cycle takes an existing piece of software and refines it into a better and better product. During these iterations features may be removed, redefined, and updated. This approach ensures that software developers have frequent feedback, flexibility in choosing

a release date, and clear goals (Cheriton 35).

Standard Development Cycle

Before we discuss how the short cycle aided our development let us analyze a typical long cycle. In such a development process the cycle would have been the full ten weeks of the term. It may have been 3 weeks of planning, 4 weeks of coding, and three weeks of refining. The problem with this methodology is that it lacks flexibility. Flexibility is needed in a software projects because unexpected problems always occur. For example, I spent over a week working on the makefile. This normally easy task was very difficult because I had to link to the Cplex library in a very specific manner. This occurred because Stanford only has a limited number of license keys for Cplex.

Iterative Software Design

In our design we developed the the three different portions of the program separately. These were the tiling and target image analysis, solving the linear system, and producing the resulting image. In effect these were separate modules that each could have been tackled by a different programmer. In this subsection we will analyze how the linear solver portion evolved and changed during each iteration.

The linear solver classes went through three major cycles. The first cycle entailed getting the software functional and allowing it to pass the basic use scenarios. During this cycle we attempted to stay strict to the attribute only interface design and other design principles discussed in the course. The next cycle focused on interface design and refactoring. We realized that some of our interfaces were unnecessarily complex. Further we realized that certain types of constraints such as range constraints on decision variables were difficult to implement. We refined the Variable class to make such types of constraints on variables easier. We also replaced “verby” type words such as “add” in expression with “termNew” (Cheriton 90). We thought about using “sum” instead of termNew but felt that in our context this was clearer.

The last iteration we made involved optimizing exceptions. Almost every Cplex method throws some type of exception. Initially we passed these all through to the client and required them to handle them. As an optimization we now handle these exceptions inside of the linear solver classes. This decision was chosen to help encapsulate the underlying linear solver functionality. Additionally, the client user has no knowledge of what to do with a Cplex exception. Now the only exceptions that a client user will have are related to parsing his input. Such errors the user can understand and find informative. Once this refined exception handling was in place, we implemented the logging mechanism as described in class. This logging mechanism is based on different levels of severity and could allow users to filter the log based on such attributes. In our simplistic implementation when an exception occurs we have the log create a new log entry. When such an entry is made, a message is printed to the standard error as suggested in CS249A assignment 3. These error messages can be used by a developer to help debug the software.

Product Release Point

In my original project proposal I discussed rewriting three different mosaic problems in the CS249 style. In my schedule I first planned on writing and solving the general photomosaic problem. Once this was completed and my linear solver classes fully debugged I was then going to focus on the other two problems. By keeping a short cycle I was able to observe that this schedule was overly ambitious. For example, it took much longer to refine and complete the linear solver interface than I originally intended. The short cycle meant that after each refinement I could release and still have a finished product, just without all of the features I originally wanted (Cheriton 35). This flexibility is crucial in industry when you need to be able to adapt to your competitor's release dates.

Short Cycle Development Synopsis

The use of short cycle development assured that we had a final project by our deadline. But it also allowed us to iteratively refine the program up until the release date in a controlled fashion. While short cycle development focuses on development of about a week in length, a standard development cycle may have been the full ten weeks of the term. The long cycle loses the flexibility of the short cycle and fails to respond when unexpected complexities come up.

Other Topics

In the next few subsections we briefly touch on topics and problems encountered during the development process. We mention how SOS and the underlying framework helped us through these problems. We choose to analyze operator overloading, functors, the prior code, and client interface.

Preexisting Framework

One of the difficulties of this project was beginning without a solid framework. This forced the framework to be developed and refined before even tackling the photomosaic problem. Further this was a distraction throughout the project. A programmer would like to ideally have the framework in place when he begins the project. This would allow him to focus on the new features that he is implementing (Cheriton 682).

Client Interface

In the client's initial implementation it simply took in information from the command line. We did not feel this was adequate and added a simple GUI to read in this information from the user. This implementation takes in the file names and other such information to create our photomosaic. Currently the photomosaic problem is fixed at start and calls the appropriate interface methods to create the linear system. This is another example of iterative software development. We improved on the command line version and now have a graphical and more user friendly version.

Functors

In an effort to keep state and processing separate we make use of functors. In our implementation we use functors to implement the visitor pattern, the functor

object will visit objects in a collection and perform the appropriate action (Cheriton 151). In C++ implementing these is very easy and syntactically pretty, we simply override the () operator and have it take in an object type. Since Java does not allow for the overloading of (), we had to implement a method to take its place. But the effect is the same.

The visitor pattern is most apparent when we are performing actions on all of the tiled images. Such an action could be as simple as printing their brightness information to a file. The approach presented in some textbooks is to have the individual class define a print routine. This method is inferior because print can have many meanings. For example does image.print() mean to print the image as a graphical representation or a series of doubles representing pixel intensities? We use both such meanings inside of this program. The functor gives us the flexibility to perform the appropriate one. Additionally, the functor can store state such as destination which is related to the printing and not to the image object

File Wrappers and Java's Limitations

This project relied on constant access to the file system. A common bug found in programming is forgetting to close a file when the last reference leaves. In C++ this problem can be avoided by having a wrapper around the file class and then overriding the destructor to close the file when the wrapper is being destroyed. Disappointingly there is no equivalent solution in Java. This is another example of where the simplicity of Java can hurt the programmer.

Java and Memory Management

Java memory management is based around garbage collection. In this vain the programmer is far removed from memory management and cannot keep track of his own memory. As a further problem when Java runs out of memory it throws an OutOfMemoryError. Errors in Java differ from exception. Once an error has occurred, the program has already reached catastrophic failure and cannot recover. When this error occurs “the garbage collector has already tried its best to free memory by reclaiming space from any objects that are no longer strongly referenced” (Shirazi). This is horrible because we cannot implement any type of recovery mechanism such as running in a safe mode (Cheriton 711). We know and understand our program and recognize that some references could be removed. The memory management problem is further complicated because there is no way of keeping track of how much memory we have used. This prevents us from implementing an on low memory type notification.

Java's memory management system, or lack thereof, seriously degraded the quality of the software we created. We wanted to use as many images as possible, so that we could produce the best mosaic. The end effect is that we would keep increasing the number of images used until we hit the OutOfMemoryError, and then knew we had hit our maximum.

Analysis of the Prior Code

As mentioned previously while at Davidson College Gavin Taylor, Dr. Chartier,

and myself wrote three different mosaic pieces of software. The specific piece of software that this project is based on was originally written by Taylor and created photomosaics using the NEOS linear solver. The new piece of software in the CS249 style is better than the prior piece in many ways. The first is that it uses object oriented programming techniques to make the code more understandable. For example we use functors to separate processing from state. In contrast Taylor's version was procedural in design. Additionally we have written a wrapper around all of Cplex which can be used to solve many different linear systems and not just the photomosaic problem. Our design also shows effective and specific uses of exceptions. In contrast the other version had exceptions in many different places without any formal consistency. This lack of standard resulted in confusing and convoluted code. In conclusion our version is superior because of its extendibility and ease of use.

Synopsis and Concluding Remarks

In conclusion, by following the CS249 methodology we were able to produce a complete and functional piece of software. We were also able to do this within the assignment's time frame. By following the CS249 principles we were able to develop efficiently. Lastly, we have offered some analysis of using the CS249 principles in Java. We have seen that while Java prevents us from using the full CS249 framework, we can still implement many of the underlying ideas of CS249.

Appendix

Running the Software

Running and executing the program involves a few steps all at the terminal on a Stanford Unix computer such as the pods. When running the programs follow the associated directions.

1. Type: setenv ILOG_LICENSE_FILE "/usr/sweet/etc/ilog/access.ilm"
2. Type make photo_setup. This will allow you to convert a directory to BW images, analyze those images, and analyze a target image.
3. Type: make photo_solve. This will analyze the information created above and create a solution to the linear system of equations.
4. Type: make photo_setup. Use this program again to analyze the linear solver solution and create the tiled image.

The separation of the program into two distinct modules was performed for two reasons. The first is that we cannot upload 100s of megabytes of pictures to our AFS space. The second is that Stanford has a limited number of keys for Ilog's Cplex and we do not want to hog them while doing some of our time consuming operations.

Acknowledgment

The original photomosaic piece of software was written by Gavin Taylor. I have since rewritten almost all of his entire code with the exception of the tiled image data gathering methods. I would like to thank him for providing his software as a

base for my work on this project and the other two mosaic software pieces I have written.

Works Referenced

Bosch, Robert. *Opt Art*. www.oberlin.edu/math/faculty/bosch/optart_survey.pdf. Access Date: 3/8/2008.

Chartier, Tim. *Mathematical Modeling 210 Lecture Notes Spring 2006.*

Cheriton, David. *Object-Oriented Programming from a Modeling and Simulation Perspective*. http://www.stanford.edu/class/cs249a/. Access Date: 3/8/2008.

Taylor, Gavin. Math221 Photomosaic Code. Davidson College.

ILOG CPLEX 9.0 User's Manual. eaton.math.rpi.edu/cplex90html/pdf/usrcplex.pdf. Access Date: 3/8/2005.
Taylor, Gavin. Referenced his original code from Mathematical Modeling 210. Spring 2005.

Flanagan, David. *O'Reilly Book Excerpts: Java in a Nutshell, 5th Edition. Generic Types, Part*. http://www.onjava.com/pub/a/onjava/excerpt/javaian5_chap04/index1.html. Access Date: 3/8/2008.

Shirazi, Jack Catching OutOfMemoryErrors to Preserve Monitoring and Server Processes. http://www.onjava.com/pub/a/onjava/2001/08/22/optimization.html. Access Date: March 14, 2008

Sun Developer Network http://java.sun.com/. Access Date: 3/8/2008.